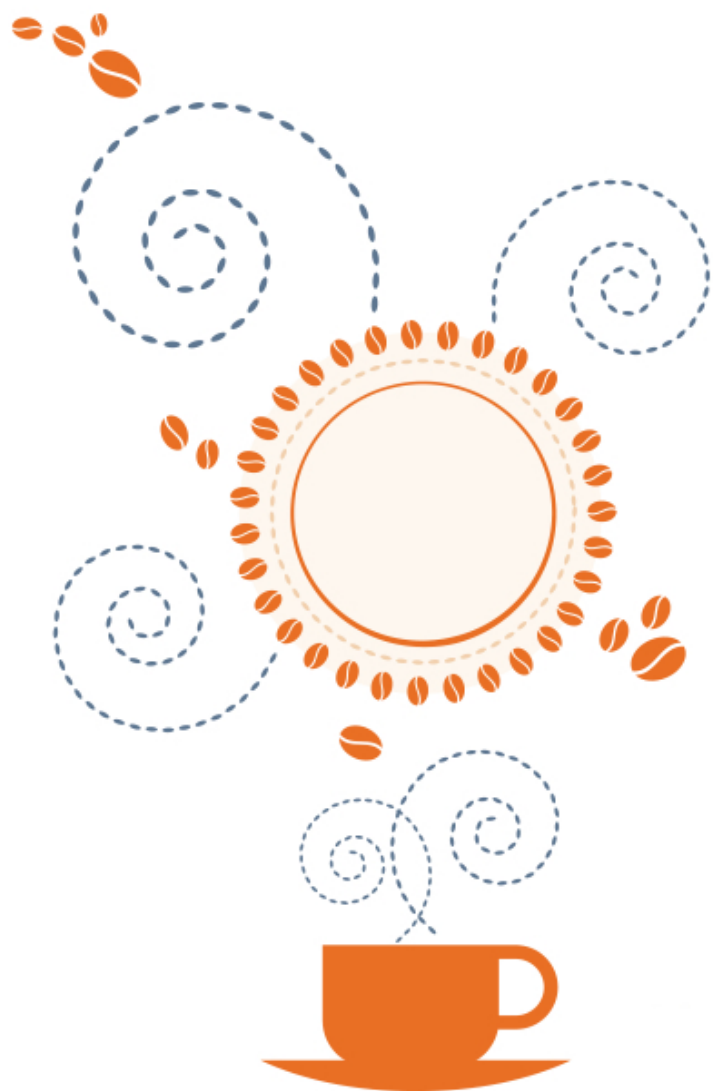


JavaFX

Interfaces com qualidade para aplicações desktop



Casa do
Código

BRUNO OLIVEIRA

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

Agradecimentos

Deus colocou em minha vida pessoas fantásticas, que me ajudam, aconselham e lutam comigo pelas minhas conquistas e também durante minhas derrotas. Primeiramente, meu agradecimento é ao Nosso Criador, Deus. Sem Ele, nossa vida não tem sentido, não tem luta, não tem felicidade. Ele é e sempre será o motivo para seguir em frente, no caminho que Ele prepara para mim a cada instante. Agradeço também à minha família, que são meus cúmplices em todos os momentos, de mãos dadas, ajudando-me a trilhar o caminho. À minha mãe, meu pai e meu irmão, que me ajudaram com ensinamentos de como enfrentar a vida de cabeça erguida, lutando pelos nossos objetivos, e mostrando que a dedicação é ponto de prioridade para nossas conquistas. À minha sogra, que me fez crescer pessoalmente, mostrando a batalha de perseverar e não desistir daqueles que amamos, minha segunda mãe. E à minha namorada, a mulher da minha vida, que enfrenta as barreiras e obstáculos com muito vigor e felicidade, chorando, sorrindo, esbravejando, aconselhando, sendo feliz... Agradeço a cada conversa, a cada momento de companheirismo, compreensão e muito amor que passamos juntos. Agradeço a Deus por me dar de presente você. Eu te amo Lais Renata! Agradeço aos amigos que me incentivaram a continuar lutando pelos meus sonhos, de forma especial cito alguns nomes como Bruno Souza, Rogério Rymsza e William Siqueira, e carinhosamente cito Paulo Silveira e Adriano Almeida, da Casa do Código, com sua paciência, e me fazendo acreditar em mim mesmo, com grande responsabilidade, e que poderia concluir este livro. E por fim, agradeço a você leitor, pelo carinho e pela vontade de fazer deste livro, seu guia de JavaFX. Espero que seja uma leitura agradável e divertida!

Sobre o autor

Conheci o Java há 3 anos, em meu curso técnico em informática com foco em programação. Desde então desenvolvo softwares com esta linguagem, e pretendo sempre crescer com ela. Hoje, sou estudante de Análise e Desenvolvimento de Sistemas, e também sou estagiário desenvolvedor Java/Android. Além disso, sou entusiasta JavaFX e, sempre que posso, desenvolvo softwares com esta plataforma, para aprender novas “artimanhas”. Tenho um carinho especial por desenvolvimento de jogos Mobile, em Android, ainda pretendo trabalhar com isto. Recentemente, fui autor de um artigo da revista MundoJ, sobre JavaFX, na edição de Março/Abril (edição 58).

Sumário

1	Apresentação	1
1.1	Um pouco da história do JavaFX	1
1.2	Por que estudar JavaFX?	2
1.3	O que veremos no livro?	2
1.4	Organização do livro	3
2	Começando com JavaFX	5
2.1	Criando um novo projeto com JavaFX	5
2.2	Desenvolvendo o formulário de login	9
2.3	Dando o primeiro toque de requinte	13
3	GolFX - Nossa loja de artigos esportivos	15
3.1	Entendendo o problema	15
3.2	Por onde começar?	16
3.3	Utilizando boas práticas de programação	16
3.4	Vitrine de produtos	20
4	Mais telas da nossa aplicação	29
4.1	Exibição de item escolhido	29
4.2	Carrinho de compras	33
4.3	Excluindo produtos	34
5	Primeiro toque de requinte - CSS	37
5.1	A importância do CSS	37
5.2	Criação de arquivo CSS	38
5.3	Alterando cursores	39
5.4	Efeito Hover	40

6	Segundo toque de requinte - Effects	43
6.1	Uma breve introdução	43
6.2	Sombreamento externo	43
6.3	Sombreamento interno	47
6.4	Reflexão d'água	49
7	Terceiro toque de requinte - Transitions e Timelines	53
7.1	O que são Transitions e Timelines?	53
7.2	Fade Transition	53
7.3	Scale Transition	55
7.4	Parallel e Sequential Transitions	57
7.5	Timeline	60
8	JavaFX vs Swing	63
8.1	Entendendo javax.swing	63
8.2	JavaFX dentro do Swing?	66
9	Mais componentes JavaFX	69
9.1	Accordion	69
9.2	HBox e VBox	70
9.3	GridPane para linhas e colunas	71
9.4	Um HTML editor pronto para você usar	72
9.5	HyperLinks	74
9.6	A famosa e temida barra de progresso	75
9.7	Visualizando árvores com TreeView	76
9.8	WebView para renderizar HTML	78
9.9	Popup Controls	79
9.10	Gráficos para dar mais vida	82
9.11	Audio e Video	87
9.12	Shapes para o controle fino	89
10	JavaFX Scene Builder	93
10.1	Conhecendo a ferramenta	93
10.2	Library Panel	94
10.3	Hierarchy Panel	95

10.4	Content Panel	96
10.5	Inspector Panel	97
10.6	Inserindo CSS	103
10.7	Classe Application	104
10.8	Classe Initializable	106
11	Executando tarefas concorrentemente	109
11.1	Iniciando por Threads	109
11.2	Conhecendo Tasks	111
11.3	Implementando Task no sistema	112
12	Utilizando WebServices de forma simples	117
12.1	Chamando Serviços Remotos via Web	118
12.2	Twitter API e Twitter4j	118
12.3	Autenticando usuário no Twitter Developers	119
12.4	Criando uma lista de Tweets	121
	Bibliografia	125

CAPÍTULO 1

Apresentação

1.1 UM POUCO DA HISTÓRIA DO JAVAFX

Tudo começou há aproximadamente 7 anos, com um projeto inicial de um desenvolvedor chamado Chris Oliver, com a intenção de criar uma linguagem cujos recursos seriam extremamente avançados em interface gráfica e, ao mesmo tempo, fáceis de implementar. Parecia ser uma tarefa muito difícil, mas ele resolveu iniciar, dando origem ao projeto F3. A Sun Microsystems gostou da proposta de Chris e resolveu comprar sua ideia, fazendo a linguagem passar a ser chamada de JavaFX Script, uma linguagem um tanto semelhante ao JavaScript, ou seja, o código não era oficialmente Java.

A primeira versão do JavaFX Script saiu em maio de 2007, em uma conferência da JavaOne. Os planos eram audaciosos: em pouco tempo, elevar o JavaFX para Desktop e Browser, e futuramente para dispositivos móveis.

Com o passar do tempo, os desenvolvedores JavaFX Script tomaram um susto: a linguagem seria descontinuada, em 2010.

Porém, em outubro de 2011, a Oracle, que então havia adquirido a Sun Microsystems, lançou a versão 2.0 do JavaFX, com uma grande novidade: o código seria totalmente Java! Desde então, o JavaFX cresce no mercado a níveis muito altos. Ele utiliza o conceito RIA (*Rich Internet Application*), tornando aplicações Desktop com qualidade gráfica altíssima e conceitos de programação eficazes, o que o fez ser uma saída para as aplicações Swing, do Java, cujo gráfico deixava a desejar.

Há pouco tempo, a Oracle anunciou que o JavaFX será totalmente open-source, além de uma possível edição para implementação em iOS e Android. Aguardamos ansiosamente pela confirmação! Ao mesmo tempo, há esforços independentes para trazer o JavaFX ao mobile, como a RoboVM:

<http://blog.robovm.org/2013/05/robovm-002-released.html>

1.2 POR QUE ESTUDAR JAVA FX?

O JavaFX possui várias razões de ser utilizado efetivamente: organização de código, manutenção rápida e descomplicada e o principal motivo, qualidade gráfica para uma área onde os recursos de programação são limitados.

Sim, é possível criar aplicações Desktop com qualidade gráfica avançada, com conceitos CSS e belos efeitos visuais! Sinceramente, acho que apenas isto já é um grande incentivo para os desenvolvedores Desktop conhecerem e estudarem mais sobre o JavaFX.

O meu objetivo com este livro é incentivar o uso desta plataforma de aplicações Desktop e mostrar que este tipo de aplicação não está acabando, como é exposto por muitos nomes da área. Pelo contrário, cada vez mais prova-se a qualidade gráfica do JavaFX, comparando-se a grandes aplicações Web.

Além deste livro, cito dois grandes locais para estudo do JavaFX. O primeiro é o livro de Carl Dea: [2], muito bom para estudar códigos básicos, introduzindo a plataforma aos novos desenvolvedores, e também, na minha opinião, o principal blog de tutoriais de JavaFX: [1], aqui encontram-se diversos códigos e tutoriais para implementação em seus projetos, uma excelente ferramenta para estudos.

1.3 O QUE VEREMOS NO LIVRO?

Você verá durante nosso percurso aplicações efetivas, simplificadas e objetivas, cuja prática levará ao conhecimento de diversas características da plataforma. Para tranquilizar, saiba que sua sintaxe ainda é Java (ufa!), apesar de certos códigos que podem surpreender o desenvolvedor no primeiro momento.

Mas acredite, esta surpresa será um trunfo do JavaFX. Você aprenderá, também, sobre como é dimensionada a questão do famoso MVC (Model - View - Controller), e perceberá que é muito mais prático do que as antigas aplicações Swing, facilitando a comunicação de camadas e simplificando sua interpretação.

1.4 ORGANIZAÇÃO DO LIVRO

Iniciaremos este livro com a instalação e configuração do JavaFX e a criação da primeira aplicação simples, um formulário de login, para conhecer o básico da plataforma.

Nos capítulos 3 e 4, criaremos uma pequena aplicação utilizando conceitos simples da plataforma. Esta aplicação será um sistema de gerenciamento de uma loja de artigos esportivos, no qual usaremos recursos gráficos avançados para dar maior riqueza à aplicação.

Nos capítulos 5, 6 e 7, falaremos sobre efeitos visuais e folha de estilos permitidos pelo JavaFX, utilizando-os nas próprias telas do projeto realizado. São muitas as possibilidades!

O capítulo 8 mostrará um pouco da relação de amor e ódio do JavaFX com o Swing, e até como trabalhar com os dois simultaneamente.

No capítulo 9 você vai conhecer componentes mais ricos e ter ideias de como poderá incrementar sua aplicação com eles.

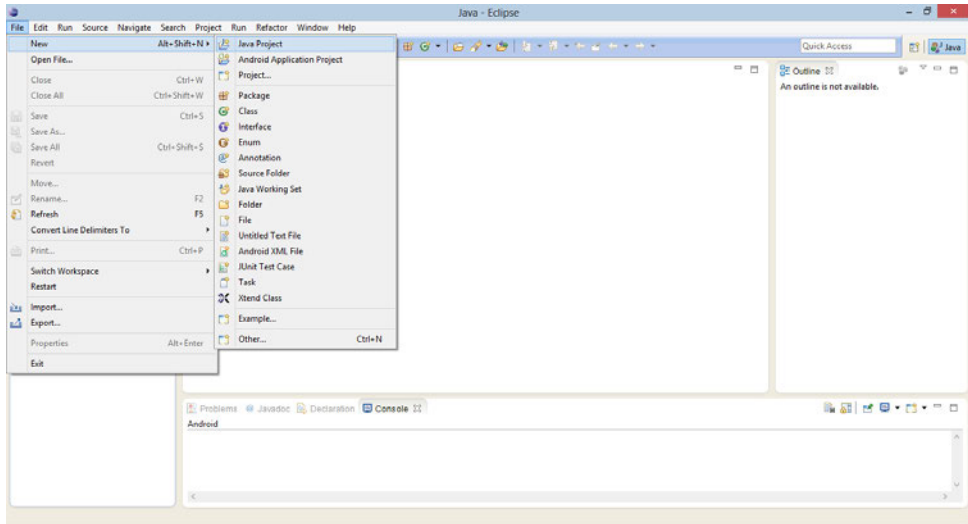
O capítulo 10 mostrará uma forma mais simples de criar e organizar os componentes na tela, com um mecanismo drag and drop.

CAPÍTULO 2

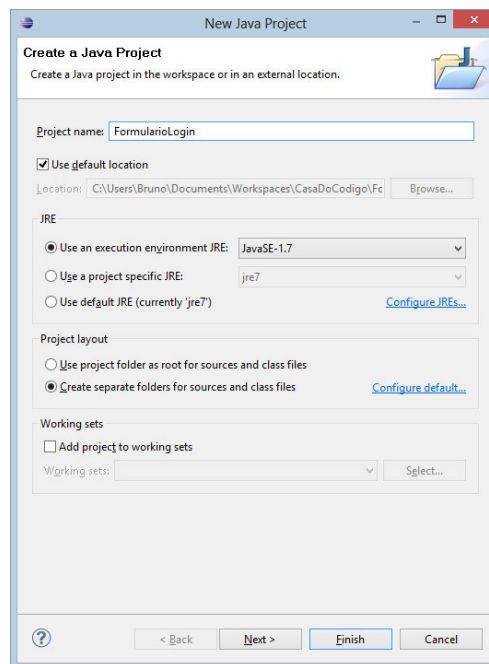
Começando com JavaFX

2.1 CRIANDO UM NOVO PROJETO COM JAVA FX

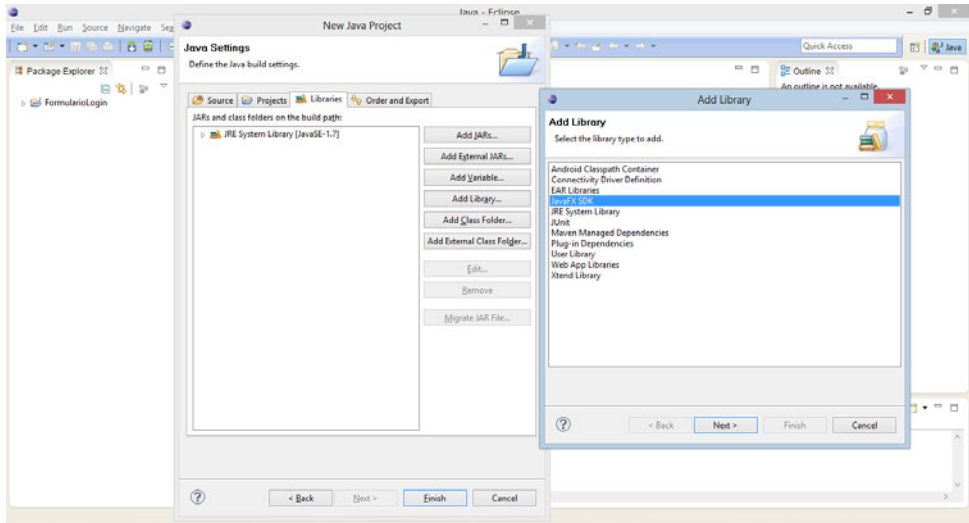
Vamos começar criando um pequeno formulário de login e senha, cuja lógica é simples: um campo de texto simples para o campo “login”, um campo de texto oculto para o campo “senha” e dois botões, um para “entrar” e outro para “sair”. Primeiro, abra o Eclipse, então clique na aba *File*, depois em *New* e, por fim, escolha a opção *Java Project*.



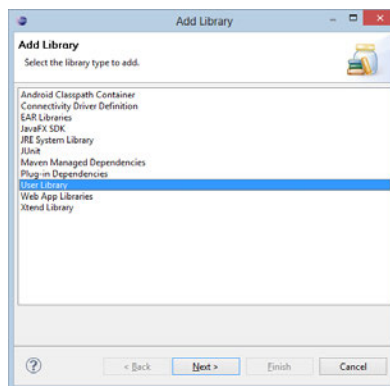
Na tela a seguir, apenas dê o nome do seu projeto. Para este fim, darei o nome de `FormularioLogin`. Pode clicar em *Next*.



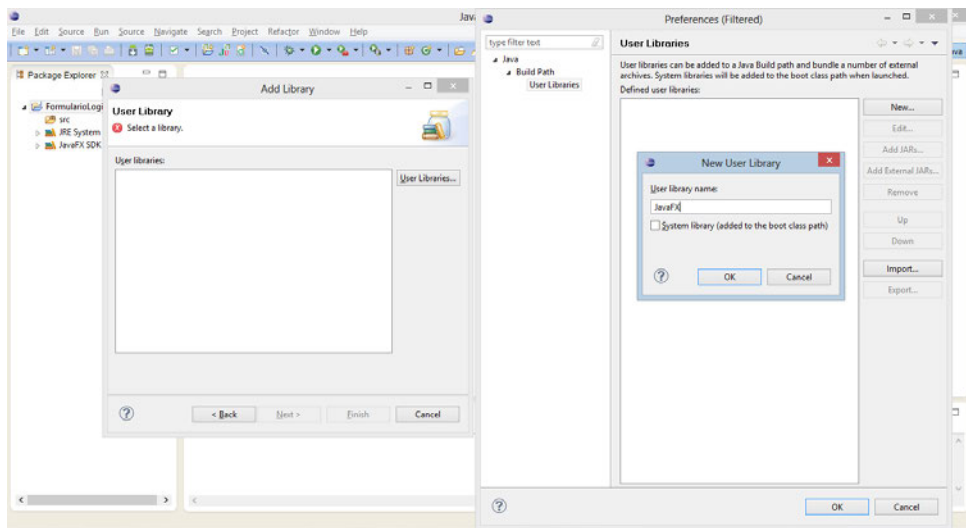
Na próxima tela, precisa-se escolher a biblioteca do JavaFX SDK e, então, clicar em *Finish*. Caso tenha baixado e instalado o Eclipse Juno, provavelmente já está tudo configurado e basta adicioná-la ao projeto, conforme a figura abaixo:



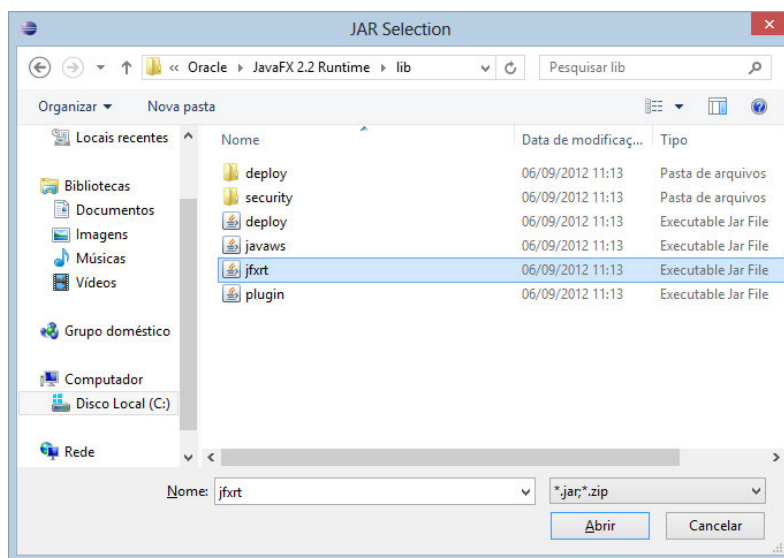
Caso seu Eclipse não esteja listando essa biblioteca, vamos criar uma nova definição. Clique em *Add Library*, e clique em *User Library*:



Surgirá uma tela para adicionar uma nova biblioteca. Então clique em *User Libraries...* e, na tela que surgir, clique em *New*. Coloque o nome da biblioteca de JavaFX, e dê *OK*.



Após a biblioteca criada, precisamos adicionar o jar do JavaFX. Para isto, clique em *Add External JARs...* e procure pelo `jfxrt.jar`. No Windows, ele se encontra em um local como `C:/Program Files/Oracle/JavaFX 2.2 Runtime/lib/jfxrt.jar`, no Mac e no Linux, algo como `/Contents/Home/jre/lib/jfxrt.jar`. Nada que um locate `jfxrt.jar` não resolva.



Dê *OK* e *Finish*, na tela de adicionar biblioteca.

Agora já podemos iniciar nosso código!

2.2 DESENVOLVENDO O FORMULÁRIO DE LOGIN

Criaremos uma classe que será responsável pela visualização (View, do padrão MVC) e também pelo controle das informações (Controller). Esta classe se estende de uma `Application`, pertencente ao JavaFX. Com isto, teremos que sobrescrever o método `start(Stage stage)`, vindo desta classe, automaticamente. Este método é onde desenvolveremos nossa lógica e criaremos nossos componentes. Ele será muito útil, também, para chamarmos outros formulários da nossa aplicação, código que veremos mais tarde. Daremos o nome de `LoginApp`. Nossa classe inicialmente ficará assim:

```
import javafx.application.Application;
import javafx.stage.Stage;

public class LoginApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {

    }
}
```

Primeiramente, vamos criar um painel onde se localizarão os componentes da tela. Neste exemplo, utilizaremos o `AnchorPane`, por dar total liberdade na localização de seus componentes. Após instanciar um novo painel, daremos seu tamanho, com o método `setPrefSize(double prefWidth, double prefHeight)`, cujo primeiro parâmetro é sua largura, e o segundo, sua altura.

```
AnchorPane pane = new AnchorPane();
pane.setPrefSize(400, 300);
```

Então, precisamos criar uma cena para fazer acontecer nosso formulário. Nela, passaremos o painel, que será a parte principal (ou total) da tela.

```
Scene scene = new Scene(pane);
```

Por fim, devemos indicar qual cena será usada no nosso `Stage`, que seria a tela propriamente dita. Esta `Stage` é passada no próprio método `start`, e a indicação da cena é passada pelo método `setScene(Scene scene)`.

```
stage.setScene(scene);
```

Agora, precisamos abrir o `Stage`, com o método `show()`.

```
stage.show();
```

E para indicar o ponto de execução da classe, precisamos do método `main(String[] args)`, utilizando o método `launch(String[] args)`, que vem da classe `Application`.

```
public static void main(String[] args) {  
    launch(args);  
}
```

Execute seu código! Veremos uma tela em branco, e não queremos ver isso, não é?

Começaremos a criar e adicionar nossos componentes. Primeiro, criaremos um campo de texto simples para ser nosso espaço para “login”. Este componente chama-se `TextField`. Criaremos o campo, e usaremos o método `setPromptText(String value)`, para dar-lhe um texto inicial. Este texto some sempre que seu foco é adquirido.

```
TextField txLogin = new TextField();  
txLogin.setPromptText("Digite aqui seu login");
```

Agora, criaremos um campo de texto oculto para senha. Este componente chama-se `PasswordField`. Usaremos, também, o método `setPromptText(String value)`, para dar o valor inicial.

```
PasswordField txSenha = new PasswordField();  
txSenha.setPromptText("Digite aqui sua senha");
```

E criaremos os dois botões para “entrar” e “sair”. O componente de botão é o `Button`. Na sua construção, passamos o texto do botão.

```
Button btEntrar = new Button("Entrar");  
Button btSair = new Button("Sair");
```

Agora, precisamos adicionar todos os componentes para o painel. Para isto, utilizamos o método `getChildren().addAll(Node... elements)`, passando como parâmetro todos os componentes.

```
pane.getChildren().addAll(txLogin, txSenha, btEntrar, btSair);
```

Todos os componentes visuais são filhos de `Node`, no JavaFX, seguindo o padrão *Composite*, que será explicado mais tarde. Seria o equivalente ao `Component` do Swing.

Nossa classe ficará assim:

```
public class LoginApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        AnchorPane pane = new AnchorPane();
        pane.setPrefSize(400, 300);
        TextField txLogin = new TextField();
        txLogin.setPromptText("Digite aqui seu login");
        PasswordField txSenha = new PasswordField();
        txSenha.setPromptText("Digite aqui sua senha");
        Button btEntrar = new Button("Entrar");
        Button btSair = new Button("Sair");
        pane.getChildren().addAll(txLogin, txSenha, btEntrar, btSair);
        Scene scene = new Scene(pane);
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Ao executar, veremos que os componentes estão desordenados. Para isto, devemos ajustar o local de cada `Node`, utilizando os métodos `setLayoutX(double value)` e `setLayoutY(double value)`. Estes métodos ajustam as coordenadas X e Y, em relação à tela. Utilizaremos um pequeno cálculo matemático para descobrir o centro da tela. Subtrairemos a largura da tela pela largura do componente e dividiremos por 2.

```
txLogin.setLayoutX((pane.getWidth() - txLogin.getWidth()) / 2);
txLogin.setLayoutY(50);
/* Repete este código para os outros componentes... */
```

OBSERVAÇÃO:

`pane.setPrefSize(400, 300)` = Indica a largura e altura do painel principal. `Node.getWidth()` = Retorna a largura do componente.

Digite estes códigos após a abertura da tela (`stage.show()`), pois só então teremos o valor correto da largura dos componentes. Antes disso, o valor será `-1.0`.

Nossa classe completa já começou a ficar grandinha:

```
public class LoginApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        AnchorPane pane = new AnchorPane();
        pane.setPrefSize(400, 300);
        TextField txLogin = new TextField();
        txLogin.setPromptText("Digite aqui seu login");
        PasswordField txSenha = new PasswordField();
        txSenha.setPromptText("Digite aqui sua senha");
        Button btEntrar = new Button("Entrar");
        Button btSair = new Button("Sair");
        pane.getChildren().addAll(txLogin, txSenha, btEntrar, btSair);
        Scene scene = new Scene(pane);
        stage.setScene(scene);
        stage.show();

        txLogin.setLayoutX((pane.getWidth() - txLogin.getWidth()) / 2);
        txLogin.setLayoutY(50);
        txSenha.setLayoutX((pane.getWidth() - txSenha.getWidth()) / 2);
        txSenha.setLayoutY(100);
        btEntrar.setLayoutX(
            (pane.getWidth() - btEntrar.getWidth()) / 2);
        btEntrar.setLayoutY(150);
        btSair.setLayoutX((pane.getWidth() - btSair.getWidth()) / 2);
        btSair.setLayoutY(200);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

```
}  
}
```

2.3 DANDO O PRIMEIRO TOQUE DE REQUINTE

No JavaFX, podemos utilizar CSS para dar toques de requinte para a aplicação, deixando-a visualmente mais agradável aos olhos do usuário final. Este CSS é próprio da plataforma, tendo algumas sintaxes diferentes da que já conhecemos.

Para começarmos a entender a força da interface gráfica do JavaFX, utilizaremos um pequeno código CSS para melhorar a aparência do painel principal. Não se preocupe com o código, inicialmente, teremos um capítulo completo apenas sobre o CSS mais à frente.

```
/* Nosso primeiro toque de requinte... */  
pane.setStyle("-fx-background-color: linear-gradient(  
    from 0% 0% to 100% 100%, blue 0%, silver 100%);");
```

Agora, execute a aplicação e veja sua qualidade gráfica, com um plano de fundo gradiente, nas cores azul e prata.

OBSERVAÇÃO

Este CSS é um pouco avançado, e nele pode-se usar dois estilos: `linear-gradient` e `radial-gradient`. Pode-se, também, usar cores sólidas, sendo que, nesse caso, simplesmente utiliza-se o nome da cor, após o item `-fx-background-color`.

Finalizamos o nosso primeiro formulário feito com JavaFX. A partir daqui, entenderemos conceitos e padrões utilizados por esta ferramenta. Mas espero ter animado você, leitor, com este pequeno experimento. Aplicações interessantes serão feitas neste livro, e a nossa intenção é de abrir a sua criatividade para criar aplicações de excelente qualidade gráfica, dando seus próprios “toques de requinte” (acostume-se com esta expressão).

CAPÍTULO 3

GolFX - Nossa loja de artigos esportivos

3.1 ENTENDENDO O PROBLEMA

Uma vez familiarizado com os códigos do JavaFX, já temos desenvoltura para implementar um sistema por completo.

Criaremos um sistema para a GolFX, uma empresa fictícia do ramo de vendas de material esportivo. O grande problema da companhia é que a divulgação de seus produtos e serviços é feita de um jeito um tanto quanto “antiquado”: o famoso “boca a boca”, e as vendas andam caindo, pois este tipo de divulgação não é mais eficaz para os dias atuais.

Por este motivo, esta empresa solicitou um sistema que mostre todos seus serviços e produtos, de maneira elegante e objetiva. Quem sabe, até utilizar um tema esportivo no design.

Pensando nestes fatores, utilizaremos o JavaFX, pela sua qualidade gráfica em

aplicações Desktop, e sua facilidade no padrão de projeto (MVC). Mas, desta vez, como já estamos habituados ao estilo de codificação, usaremos boas práticas de programação para deixar o código o mais limpo possível.

3.2 POR ONDE COMEÇAR?

O nosso projeto será simples. Será baseado em quatro telas com as devidas funcionalidades:

- O primeiro é o formulário de login e senha, que já temos pronto, por sinal. Apenas faremos as funções de `logar()` e `fecharAplicacao()`. Aproveitaremos, também, para dar uma melhorada no código, usando boas práticas de programação.
- O segundo será o formulário de vitrine, que exibirá uma lista dos produtos vendidos pela GolFX. Poderemos escolher um item da lista para ver maiores detalhes.
- O terceiro formulário será a exibição detalhada (que, aqui, não será muito detalhada) do item escolhido anteriormente. Haverá a opção de adicionar ao carrinho de compras.
- E, por fim, o quarto formulário será o carrinho de compras, onde terá uma lista dos itens escolhidos pelo usuário, tendo a possibilidade de excluir um item da lista, voltar à vitrine para a escolha de outros itens, e a confirmação da compra.

Ao longo do caminho, aprenderemos alguns conceitos do JavaFX, e usaremos os principais componentes desta ferramenta, para ficarmos craques nas suas utilizações e sairmos por aí programando efetivamente.

Vamos iniciar alterando o código do nosso primeiro formulário.

3.3 UTILIZANDO BOAS PRÁTICAS DE PROGRAMAÇÃO

Quando se trata de JavaFX, podemos utilizar um padrão de código semelhante ao padrão da IDE *Netbeans*, onde há uma divisão da inicialização dos componentes e a inicialização das ações dos componentes. Particularmente, acho esta uma divisão muito interessante, para dar maior visibilidade ao código.

Para isto, criaremos nossos componentes como variáveis globais, e então, eles serão instanciados e configurados em um método chamado `initComponents()`:

```
public class LoginApp extends Application {  
  
    private AnchorPane pane;  
    private TextField txLogin;  
    private PasswordField txSenha;  
    private Button btEntrar, btSair;  
    private static Stage stage;  
  
    /* Demais códigos já implantados... */  
}
```

Antes, precisamos indicar a variável `stage`, conforme o `Stage` do método `start`, além de criar o seu getter. Faremos isto em todas as `Applications`, para podermos encerrar a tela quando necessitarmos.

```
@Override  
public void start(Stage stage) throws Exception {  
    /* Demais códigos */  
    LoginApp.stage = stage;  
}  
  
public static Stage getStage() {  
    return stage;  
}
```

E o nosso método `initComponents()` ficará assim:

```
private void initComponents() {  
    pane = new AnchorPane();  
    pane.setPrefSize(400, 300);  
    pane.setStyle("-fx-background-color: linear-gradient(  
        from 0% 0% to 100% 100%, blue 0%, silver 100%);");  
    txLogin = new TextField();  
    txLogin.setPromptText("Digite seu login...");  
    /* Outros códigos de inicialização e configuração de componentes */  
}
```

Podemos, também, criar um método para iniciar as coordenadas dos componentes, que chamaremos de `initLayout()`:

```
private void initLayout() {
    txLogin.setLayoutX((pane.getWidth() - txLogin.getWidth()) / 2);
    txLogin.setLayoutY(50);
    /* Demais códigos de inicialização das coordenadas */
}
```

Criaremos o método `initListeners()`, no qual ficarão as ações dos componentes. No nosso caso, precisamos de ações nos dois botões. Começaremos pelo botão de sair, que é muito simples.

```
private void initListeners() {
    btSair.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            fecharAplicacao();
        }
    });
}

private void fecharAplicacao() {
    System.exit(0);
}
```

O método `setOnAction(EventHandler<ActionEvent> value)` indica a ação do botão. Ele recebe como argumento um `EventHandler<ActionEvent>()`. Nesse caso, optamos por usar uma classe anônima, que implementa essa interface e possui um método `handle(ActionEvent event)`, onde ficarão os códigos de ação. Algo muito semelhante a um `ActionListener` do Swing.

Logo abaixo da ação do botão de sair, faremos a ação do botão de entrar. A ação perguntará se o login é igual a *admin* e a senha é igual a *casadocodigo*. Se sim, entra para a próxima tela, se não, mostra uma mensagem de erro.

```
btEntrar.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        logar();
    }
});

private void logar() {
```

```
if (txLogin.getText().equals("admin") &&
    txSenha.getText().equals("casadocodigo")) {
    // TODO Abrir a tela VitrineApp
} else {
    JOptionPane.showMessageDialog(null, "Login e/ou senha
        inválidos", "Erro", JOptionPane.ERROR_MESSAGE);
}
}
```

Ainda não temos a tela `VitrineApp`, por isso deixamos um `TODO` indicando que ali será onde abriremos essa nova tela.

Outro ponto importante a ser citado é o uso do `JOptionPane` no código. Ou seja, é totalmente possível utilizar componentes Swing no JavaFX. Mais à frente, teremos um capítulo exclusivo para tratar disso.

Veja que o código tornou-se muito mais visível para o desenvolvedor. Fica muito claro quando vemos o nosso método `start`:

```
@Override
public void start(Stage stage) throws Exception {
    initComponents();
    initListeners();
    Scene scene = new Scene(pane);
    stage.setScene(scene);
    // Remove a opção de maximizar a tela
    stage.setResizable(false);
    // Dá um título para a tela
    stage.setTitle("Login - GolFX");
    stage.show();
    initLayout();
    LoginApp.stage = stage;
}
```

A partir de agora, utilizaremos sempre este padrão para facilitar nossa visualização e melhorar o entendimento do próprio código.

Podemos rodar nossa aplicação e verificar como está ficando:

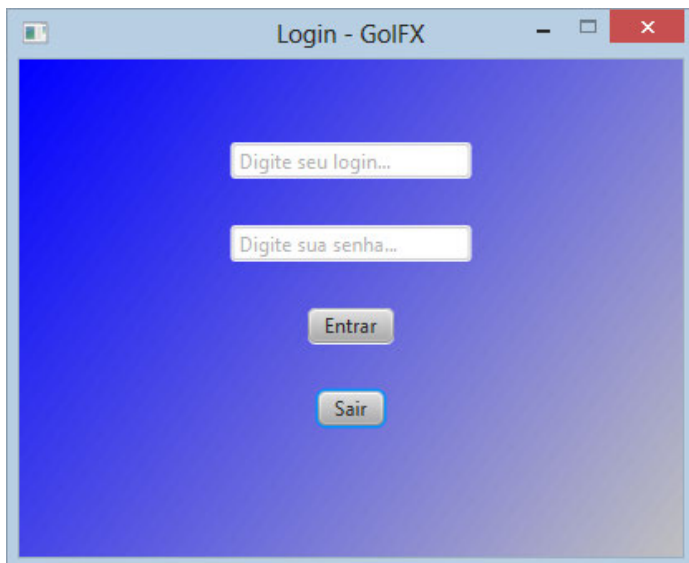


Figura 3.1: Tela de Login

Vamos então para a segunda tela, a vitrine de produtos!

3.4 VITRINE DE PRODUTOS

Esta tela, como já esclarecido, será responsável por conter todos os produtos da empresa. Será simples: terá apenas um campo de texto para filtrar os dados exibidos e uma tabela que exibirá a lista de produtos, contendo o nome e o preço dos mesmos. Então, nosso primeiro passo é criar a tela e suas configurações iniciais. Não irei repetir o código, apenas direi os principais passos.

- Crie uma classe `VitrineApp` que estenda de uma `Application`;
- Crie o método `start` e o método `main`;

Quando tratamos de criação de tabelas, utilizamos os componentes `TableView` para a tabela em si e `TableColumn` para cada coluna da tabela. Porém, este processo é um pouco extenso, mostraremos passo a passo.

Primeiro, precisamos criar uma classe interna da classe `VitrineApp` para interpretação dos dados que serão exibidos na tabela. Esta classe usa o padrão *Property*, do JavaFX, onde se indicam as propriedades (*getter* e *setter*) de tipos primitivos.

Criaremos a classe como pública, iniciaremos as variáveis de propriedades e criaremos nosso construtor:

```
public class ItensProperty {  
    private SimpleStringProperty produto;  
    private SimpleDoubleProperty preco;  
  
    public ItensProperty(String produto, double preco) {  
        this.produto = new SimpleStringProperty(produto);  
        this.preco = new SimpleDoubleProperty(preco);  
    }  
}
```

Este conceito de Properties é uma recomendação do JavaFX. O conceito é semelhante ao Model, apenas mudando o fato de que usaremos tipos Property, ao invés de primitivos (um exemplo: `SimpleDoubleProperty`, ao invés de um simples `double`). Vale lembrar que este padrão será bastante utilizado em registros de `TableView`, ou seja, para outros conceitos (MVC e DAO, por exemplo, caso esteja usando Hibernate/JPA) os conceitos de Model serão normais, e não desta forma com Property.

Por fim, precisamos criar os *getters* e *setters* das propriedades. É importante ter atenção, pois o processo é um pouco diferente do habitual.

```
public String getProduto() {  
    return produto.get();  
}  
  
public void setProduto(String produto) {  
    this.produto.set(produto);  
}  
  
public double getPreco() {  
    return preco.get();  
}  
  
public void setPreco(double preco) {  
    this.preco.set(preco);  
}
```

Para o armazenamento dos produtos, criaremos três classes, uma *Model* e duas *Controllers* (padrão MVC): uma para armazenar um produto, outra para gerenciar a

lista de produtos cadastrados, e outra para gerenciar os produtos selecionados para o carrinho. Chamaremos de `Produto`, `Vitrine` e `Carrinho`.

O padrão dessas classes são aqueles já conhecidos, com *getters* e *setters*. Na classe `Produto`, teremos dois atributos: `String produto` e `double preco`.

```
public class Produto {
    private String produto;
    private double preco;

    public Produto(String produto, double preco) {
        this.produto = produto;
        this.preco = preco;
    }

    // Getters e Setters
}
```

Já na classe `Carrinho`, o padrão é baseado em um *Controller* para gerenciamento da lista de produtos escolhidos para o carrinho. Teremos apenas um atributo que representará esta lista. Podemos adicionar novos produtos e retornar a lista total.

```
public class Carrinho {

    private static List<Produto> produtos = new ArrayList<Produto>();
```

Para o método para adicionar novos produtos, utilizaremos a opção de adicionar diversos produtos ao mesmo tempo, ou apenas um, se preferir.

```
public void addProdutos(Produto... ps) {
    for (Produto p : ps)
        produtos.add(p);
}
```

Por fim, criaremos o método para retornar a nossa lista de produtos.

```
public List<Produto> getProdutos() {
    return produtos;
}
```

Crie agora a classe `Vitrine`, idêntica a classe `Carrinho`. Ela representará todos os produtos do nosso catálogo. Mais tarde ela será modificada.

Mostraremos agora um esboço da tela de vitrine, para entendermos como será o andamento e também entender a função de cada componente, apesar de serem poucos.

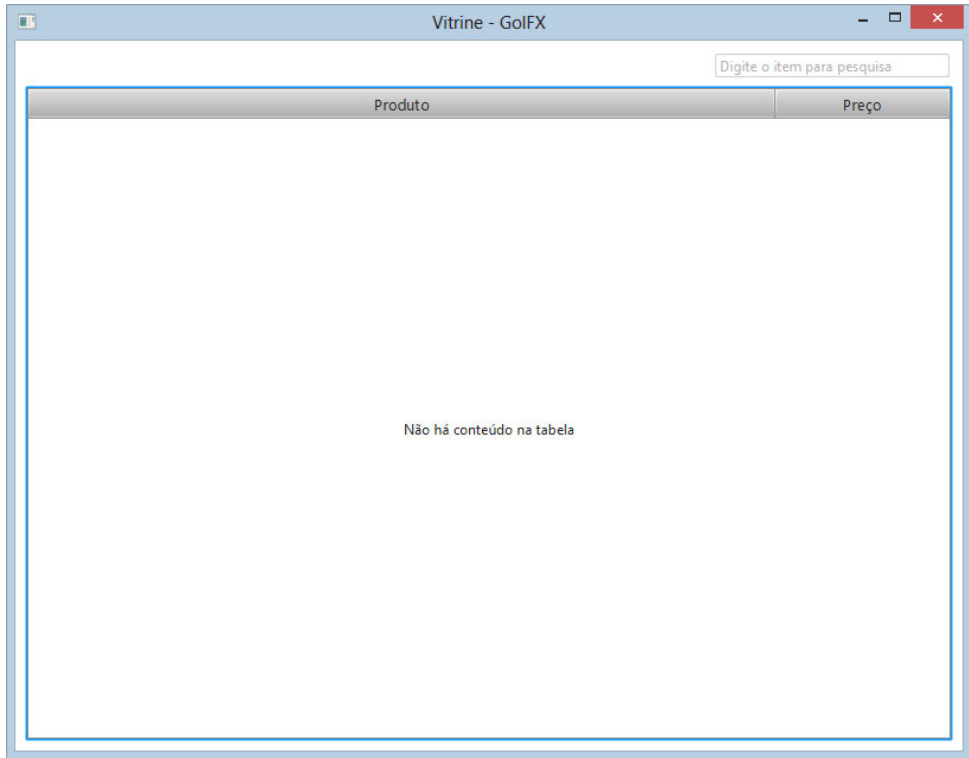


Figura 3.2: Esboço da tela de vitrine

Voltando à classe `VitrineApp`, após a finalização da classe interna `ItemProperty`, indicaremos que ela será a parametrização da `TableView` e `TableColumn`, na qual o último necessita, também, indicar qual o tipo de dado será passado para a coluna. Aproveitaremos para criar todos os componentes e também uma `ObservableList<ItemProperty>`, que é mais um padrão JavaFX, cuja função é semelhante a uma `ArrayList`, porém utiliza a forma de propriedades.

```
private AnchorPane pane;  
private TextField txPesquisa;  
private TableView<ItemProperty> tbVitrine;  
private TableColumn<ItemProperty, String> columnProduto;
```

```
private TableColumn<ItensProperty, Double> columnPreco;
private static ObservableList<ItensProperty> listItens = FXCollections
    .observableArrayList();
private static Carrinho carrinho;
```

Vamos criar o nosso `initComponents()` para iniciar nossos componentes e o carrinho, deixaremos à sua escolha para dar seus próprios toques de requinte.

```
pane = new AnchorPane();
pane.setPrefSize(800, 600);

txPesquisa = new TextField();
txPesquisa.setPromptText("Digite o item para pesquisa");

tbVitrine = new TableView<ItensProperty>();
tbVitrine.setPrefSize(780, 550);

columnProduto = new TableColumn<ItensProperty, String>();
columnPreco = new TableColumn<ItensProperty, Double>();
tbVitrine.getColumns().addAll(columnProduto, columnPreco);
pane.getChildren().addAll(txPesquisa, tbVitrine);

carrinho = new Carrinho();
```

Precisamos indicar qual campo da nossa classe interna estará em cada coluna, para isso usamos o método `setCellValueFactory`, passando uma classe `PropertyValueFactory`, para indicar o campo.

```
columnProduto.setCellValueFactory(
    new PropertyValueFactory<ItensProperty, String>("produto"));
columnPreco.setCellValueFactory(
    new PropertyValueFactory<ItensProperty, Double>("preco"));
```

Precisamos indicar a lista de `ItensProperty` para a exibição na tabela. Para isto, usaremos o nosso `ObservableList`. Colocarei este código em um método chamado `initItens()`.

```
private void initItens() {
    Vitrine v = new Vitrine();
    v.addProdutos(new Produto("Bola Topper", 15.00), new Produto(
        "Luvas Umbro", 9.00), new Produto("Camisa Esportiva", 40.00),
        new Produto("Chuteira Nike Mercurial", 199.00), new Produto(
```

```

        "Caneleira Topper", 10.00));
    for (Produto p : v.getProdutos())
        listItens.add(new ItensProperty(p.getProduto(), p.getPreco()));
}

```

E na inicialização, indicamos que esta lista, a `listItens`, será usada pela nossa tabela:

```
tbVitrine.setItems(listItens);
```

Aproveite este momento para resolver o `TODO` da tela `LoginApp`, colocando a chamada da tela de vitrine, assim já poderemos ver o resultado inicial.

```

private void logar() {
    if (txLogin.getText().equals("admin") &&
        txSenha.getText().equals("casadocodigo")) {
        try {
            new VitrineApp().start(new Stage());
            LoginApp.getStage().close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else {
        JOptionPane.showMessageDialog(null, "Login e/ou senha
            inválidos", "Erro", JOptionPane.ERROR_MESSAGE);
    }
}

```

[section Colocando funcionalidades na tela de vitrine]

Começaremos, então, a colocar as funcionalidades na tela `VitrineApp`. Crie um método chamado `findItems()`, retornando uma `ObservableList<ItensProperty>`, no qual iteramos sobre nossa lista de produtos e pesquisamos aqueles que possui no `getProduto()` o valor digitado no campo de pesquisa:

```

private ObservableList<ItensProperty> findItems() {
    ObservableList<ItensProperty> itensEncontrados = FXCollections
        .observableArrayList();
    for (ItensProperty itens : listItens) {
        if (itens.getProduto().contains(txPesquisa.getText())) {
            itensEncontrados.add(itens);
        }
    }
}

```

```
    }  
    }  
    return itensEncontrados;  
}
```

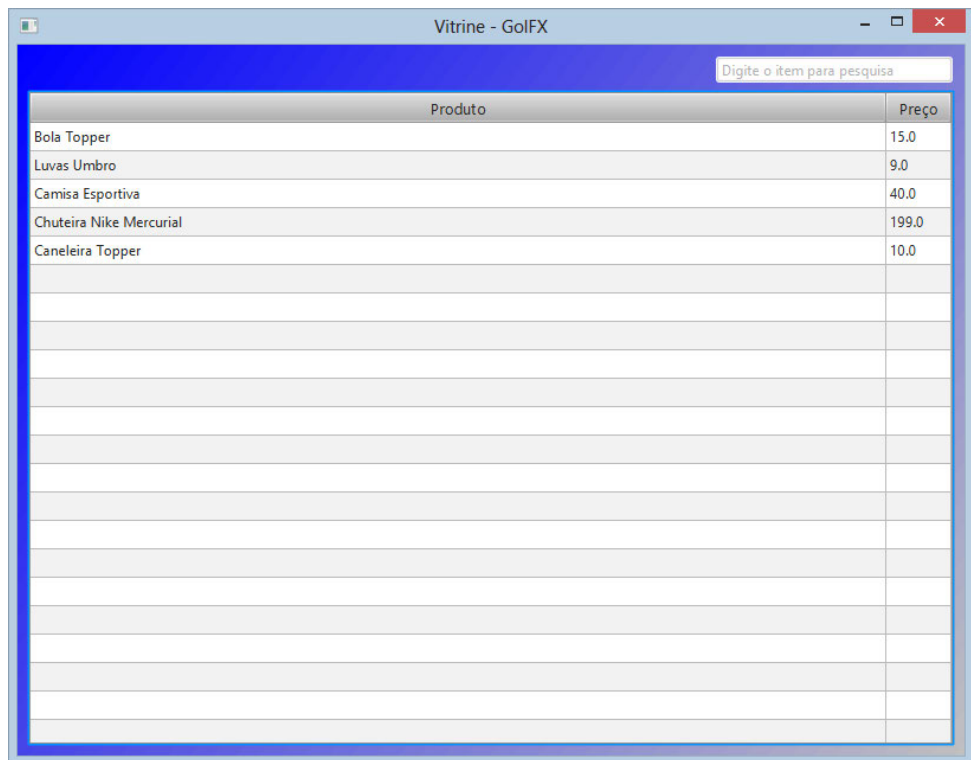
E para finalizar nossa tela, precisamos indicar a funcionalidade do campo de texto de pesquisa. Colocaremos no nosso `initListeners()` um callback para isso, através de uma classe anônima:

```
txPesquisa.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        if (!txPesquisa.getText().equals("")) {  
            tbVitrine.setItems(findItems());  
        } else {  
            tbVitrine.setItems(listItens);  
        }  
    }  
});
```

Quando o usuário digitar algo para pesquisa, a lista exibirá apenas os itens que tenham na descrição do produto o valor digitado. Se o usuário não digitar nada, então volta a exibição normal de todos os itens. Este método será chamado apertando o botão `ENTER` do teclado, quando o foco estiver no campo de texto.

Monte do seu jeito e do seu gosto, para deixar a vitrine com a sua cara. Mas lembre-se que teremos um capítulo apenas para tratar de recursos visuais.

Rode a aplicação e veja a tela de vitrine:



Produto	Preço
Bola Topper	15.0
Luvas Umbro	9.0
Camisa Esportiva	40.0
Chuteira Nike Mercurial	199.0
Caneleira Topper	10.0

Figura 3.3: Vitrine

CAPÍTULO 4

Mais telas da nossa aplicação

4.1 EXIBIÇÃO DE ITEM ESCOLHIDO

Caminhando para a próxima Application, temos a tela de exibição do item escolhido na tabela de produtos, que será chamada `ItemApp`. Usaremos o mesmo padrão da tela anterior para criação, apenas ajustaremos o tamanho para 600x400.

Inicie os componentes indicados abaixo e crie também uma variável privada estática e seu getter e setter para receber o produto selecionado da tela `VitrineApp`:

```
public class ItemApp extends Application {  
    private AnchorPane pane;  
    private ImageView imgItem;  
    private Label lbPreco, lbDescricao;  
    private Button btAddCarrinho;  
    private static Stage stage;  
    private static Produto produto;  
  
    @Override
```

```

public void start(Stage stage) throws Exception {
    pane = new AnchorPane();
    pane.setPrefSize(600, 400);
    /* Demais códigos de inicialização da tela e componentes */
}

public static Stage getStage() {
    return stage;
}

public static Produto getProduto() {
    return produto;
}

public static void setProduto(Produto produto) {
    ItemApp.produto = produto;
}
}

```

O único componente que ainda não vimos é o `ImageView`; ela recebe uma `Image`, que por sua vez recebe uma `String` com o caminho da imagem.

```

imgItem = new ImageView(new Image(
    "http://www.sportcenterlopes.com.br/images/" +
    "250_topper_campo_2009replic.jpg"));

```

Ajuste também as coordenadas de cada componente, utilizando os métodos `setLayoutX(double value)` e `setLayoutY(double value)`.

Criaremos uma forma de surgir as imagens conforme o item escolhido. Para isto, crie uma *array de Strings* com 5 imagens buscadas da internet, e crie também outra variável privada estática para receber o índice do item selecionado na tabela.

```

private static int index;
private static String[] images = {
    "http://www.sportcenterlopes.com.br/images/" +
    "250_topper_campo_2009replic.jpg",
    "http://1.bp.blogspot.com/_H8uGs8K8kaY/TLZTXR8nIgI/" +
    "AAAAAAAF_0/BvpxdqGF4PE/s1600/luva_umbro.png",
    "http://bing2.mlstatic.com/camisa-nike-active-importada-manga-" +
    "longa-esportiva-vermelha_MLB-F-199843960_1391.jpg",
    "http://www.showtenis.com.br/images/_product/979/979112/" +

```



```

        "chuteira-nike-mercurial-glide-3-fg-campo--199fd9.jpg",
        "http://www.katy.com.br/imagens/produtos/original/" +
        "caneleira-topper-training-difusion-13340619502673137.jpg" };

/* Demais códigos... */

public static int getIndex() {
    return index;
}

public static void setIndex(int index) {
    ItemApp.index = index;
}

```

Agora, precisamos retornar à `VitrineApp` para indicar o clique da tabela, para chamarmos nossa tela de exibição do item.

Adicionaremos um *listener* para a `tbVitrine`, indicando primeiramente os dois campos estáticos do `ItemApp` e, depois, chamando seu formulário.

```

tbVitrine.getSelectionModel().selectedItemProperty()
    .addListener(new ChangeListener<ItemProperty>() {
        @Override
        public void changed(
            ObservableValue<? extends ItemProperty> value,
            ItemProperty oldItem, ItemProperty newItem) {

            /*
             * Indicando os valores de produto e
             * index para ItemApp
             */
            ItemApp.setProduto(new Produto(newItem.getProduto(),
                newItem.getPreco()));
            ItemApp.setIndex(tbVitrine.getSelectionModel()
                .getSelectedIndex());

            /* Chamando o formulário de exibição de item */
            try {
                new ItemApp().start(new Stage());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });

```

```
    }  
  });
```

Quando clicarmos em um item da tabela de produtos, o formulário de exibição de item será chamado, com o valor do produto selecionado. E a variável `index` receberá o índice do item clicado na tabela da vitrine.

Quando exibir a imagem, identifique a array com o índice `index`.

```
imgItem = new ImageView(new Image(images[index]));
```

Mais uma tela pronta, faltando apenas a funcionalidade de adicionar ao carrinho, que será feita no próximo tópico. Rode sua aplicação!

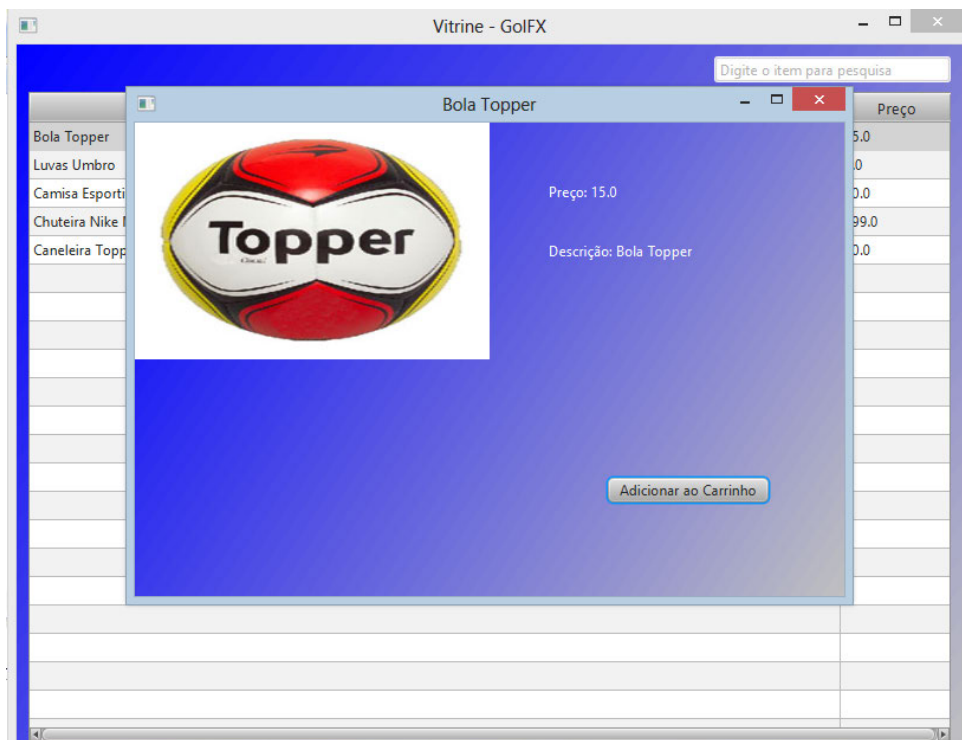


Figura 4.1: Item Bola Topper selecionado

4.2 CARRINHO DE COMPRAS

Chamaremos esta próxima tela de `CarrinhoApp`. Este formulário necessitará também de uma tabela, baseada na mesma classe interna da tela `VitrineApp`. Nela, ficará armazenada a lista de produtos selecionados pelo cliente, com o botão “*Adicionar ao carrinho*” da tela `ItemApp`. Na nossa `CarrinhoApp`, estarão os seguintes componentes:

```
private AnchorPane pane;
private TableView<ItensProperty> tbCarrinho;
private TableColumn<ItensProperty, String> columnProduto;
private TableColumn<ItensProperty, Double> columnPreco;
private Button btExcluirItem, btVoltarVitrine, btConfirmarCompra;
private static ObservableList<ItensProperty> listItens;
```

Dimensione seus componentes ao seu gosto, dando seu próprio toque de requinte, e aproveite para copiar a mesma classe interna `ItensProperty`, da classe `VitrineApp`. Criaremos, também, um método para iniciar os itens da nossa tabela.

```
private void initItens() {
    for (Produto p : VitrineApp.getCarrinho().getProdutos())
        listItens.add(new ItensProperty(p.getProduto(), p.getPreco()));
}
```

Este código funcionará sem problemas, porém nosso carrinho ainda não tem nenhum produto. Voltaremos à nossa classe `ItemApp` para adicionar a funcionalidade do botão *Adicionar ao carrinho*.

```
btAddCarrinho.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent arg0) {
        VitrineApp.getCarrinho().addProdutos(produto);
        try {
            new CarrinhoApp().start(new Stage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
});
```

Agora sim! Já conseguimos adicionar produtos ao nosso carrinho de compras, faça o teste!

4.3 EXCLUINDO PRODUTOS

Voltaremos à classe `Carrinho` para adicionarmos o método `removeProduto(Produto p)` para remoção do produto, ao clicar no botão “*Excluir Item*”.

```
public void removeProduto(Produto p) {
    Iterator<Produto> itProduto = produtos.iterator();
    while (itProduto.hasNext()) {
        Produto produto = itProduto.next();
        if (produto.getProduto().equals(p.getProduto())
            && produto.getPreco() == p.getPreco()) {
            itProduto.remove();
        }
    }
}
```

Usaremos o método `removeProduto`, da classe `Carrinho`, e também removeremos da tabela.

```
btExcluirItem.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent arg0) {
        VitrineApp.carrinho.removeProduto(new Produto(tbCarrinho
            .getSelectionModel().getSelectedItem().getProduto(),
            tbCarrinho.getSelectionModel().getSelectedItem()
                .getPreco()));
        tbCarrinho.getItems().remove(
            tbCarrinho.getSelectionModel().getSelectedItem());
    }
});
```

Em seguida, criaremos a função de voltar à tela principal. Este método apenas fechará a tela do carrinho, e também fechará a tela do item selecionado, deixando aberto apenas a tela da vitrine. No caso, criamos a variável `stage` na tela `ItemApp` de forma pública estática, o mesmo na tela `CarrinhoApp`, porém apenas privada estática.

```
btVoltarVitrine.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent arg0) {
```

```

        CarrinhoApp.getStage().close();
        ItemApp.getStage().close();
    }
});

```

E, para finalizar nossa tela, criaremos a funcionalidade do botão *Confirmar Compra*. Aqui entraremos com o banco de dados, adicionando estes produtos para um novo registro de compra, com os dados do cliente etc. Utilizaremos um exemplo mais simples, utilizando uma `Thread` para uma contagem de 5 segundos para confirmação da compra.

```

btConfirmarCompra.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        Thread thread = new Thread() {
            public void run() {
                try {
                    sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                JOptionPane.showMessageDialog(null,
                    "Compra realizada com sucesso!");
                Platform.runLater(new Runnable() {
                    @Override
                    public void run() {
                        CarrinhoApp.getStage().close();
                        ItemApp.getStage().close();
                    }
                });
            }
        };
        thread.start();
    }
});

```

Com isso, temos a finalização do layout básico e das funcionalidades do nosso projeto GolFX!

Podemos rodar nossa aplicação por completo e já vemos o resultado de nossos códigos e funcionalidades.

O gerente da empresa já está bastante contente com o trabalho feito, mas ainda está achando a parte visual um tanto básica demais. Afinal, erguemos a moral do JavaFX de tal modo que aparentava ser mais do que isso. E realmente é! No próximo capítulo, começaremos a dar toques de requinte impressionantes ao nível de uma aplicação Desktop, o que fará toda a diferença para a satisfação do nosso cliente.

CAPÍTULO 5

Primeiro toque de requinte - CSS

5.1 A IMPORTÂNCIA DO CSS

Antes de sairmos por aí desenvolvendo arquivos CSS para implementação nas telas, devemos pensar no *porquê* do seu uso, e também pensar em *quando* devemos utilizá-lo.

Para dar um grande toque de requinte às aplicações Desktop, os desenvolvedores necessitam de *libraries* ou *LookAndFeels* para personalização de cores ou efeitos em componentes e, em sua grande maioria, não é possível a configuração completa da personalização, limitando o programador a utilizar uma *skin* fixa, com determinadas cores.

O CSS é uma linguagem de estilo muito utilizada em aplicações Web, pois isola a parte de personalização gráfica dos códigos de regras de negócio. Com o JavaFX, existe a possibilidade do uso desta linguagem de forma eficaz e organizada, abrindo os horizontes de desenvolvedores Desktop para um visual agradável e elegante.

Para utilizarmos o CSS e suas propriedades em nossos componentes, temos duas maneiras:

- Criando um arquivo CSS e identificando cada *StyleSheet* em seus componentes;
- Utilizando as propriedades CSS em cada componente, pelo método `setStyle(String value)`, presente na classe `Node`.

Na prática, em nosso último capítulo, utilizamos um pouco de CSS, através da segunda opção citada acima, no componente *pane* (`AnchorPane`).

```
/* Efeito gradiente azul e prata em um painel */
pane.setStyle("-fx-background-color: linear-gradient(
    from 0% 0% to 100% 100%, blue 0%, silver 100%);");
```

Porém, utilizar esta forma pode se tornar desgastante e desorganizado. Por isso, mostraremos a primeira maneira, mais utilizada, para identificação de *StyleSheets* com CSS: a criação de um arquivo CSS.

5.2 CRIAÇÃO DE ARQUIVO CSS

Para criarmos nosso primeiro arquivo CSS, podemos utilizar o próprio bloco de notas do Windows. Então, abra o bloco de notas para identificarmos efeitos visuais para alguns componentes da classe `LoginApp`.

Primeiro, indicaremos o efeito gradiente do `pane` no nosso arquivo CSS. Para isso, utilizamos um ID para identificar a classe do `pane`.

```
.pane {
    -fx-background-color: linear-gradient(from 0% 0% to
        100% 100%, blue 0%, silver 100%);
}
```

Agora, salve este arquivo com o nome `login.css` no mesmo pacote que se localiza as classes do nosso projeto, ou seja, no diretório `src`.

Então, identificaremos por código a *StyleSheet* e a *StyleClass*. Primeiro, usaremos a `scene` para passarmos a *StyleSheet*.

```
@Override
public void start(Stage stage) throws Exception {
    /* Demais códigos... */
    Scene scene = new Scene(pane);
    scene.getStylesheets().add("login.css");
    /* Demais códigos seguintes... */
}
```


Apenas adicionamos o nome do arquivo CSS, como a nossa Stylesheet. Precisamos identificar a classe responsável pelo efeito visual no componente `pane`. A sintaxe é bem parecida.

```
private void initComponents() {  
    pane = new AnchorPane();  
    pane.setPrefSize(400, 300);  
    pane.getStyleClass().add("pane");  
    /* Demais códigos... */  
}
```

Podemos excluir o código do `setStyle` mais acima, pois daremos o mesmo efeito. Rode a aplicação e veja o resultado, que será idêntico ao anterior.

5.3 ALTERANDO CURSORES

Com o CSS do JavaFX, também podemos alterar o cursor exibido em um determinado componente. Trocaremos o cursor nos componentes `Button` da nossa tela de login, para o cursor `Hand`, que seria uma mão, simulando um clique.

Para isso, utilizaremos a definição de estilo `-fx-cursor`.

```
/* Estilo do pane */  
  
.btEntrar {  
    -fx-cursor: hand;  
}  
  
.btSair {  
    -fx-cursor: hand;  
}
```

Por fim, identificaremos em código as classes de cada botão.

```
private void initComponents() {  
    /* Demais códigos... */  
    btEntrar = new Button("Entrar");  
    btEntrar.getStyleClass().add("btEntrar");  
    btSair = new Button("Sair");  
    btSair.getStyleClass().add("btSair");  
    /* Demais códigos seguintes... */  
}
```

Quando executarmos a aplicação, veremos o efeito de cursor quando passarmos o mouse sobre os botões.



Figura 5.1: Efeito de cursor em botões

5.4 EFEITO HOVER

Bolaremos o efeito de troca de cores dos botões ao passar o mouse por cima do componente, conhecido como *efeito hover*. Para tal, usaremos *selectors* em classes do CSS, os quais irão modificar a cor de fundo e do texto dos botões. Veremos o seu uso:

```
.btEntrar:hover {  
    -fx-background-color: green;  
    -fx-text-fill: white;  
}  
  
.btSair:hover {  
    -fx-background-color: red;  
    -fx-text-fill: white;  
}
```

É necessário que a expressão *hover* esteja junto com o nome da classe e os “dois pontos (:)”, sem espaços. No nosso exemplo, ao passar o mouse do botão *Entrar*, a cor de fundo ficará verde e o texto do botão ficará branco.



Figura 5.2: Efeito Hover no botão btEntrar

E ao passar o mouse no botão *Sair*, a cor de fundo ficará vermelho e o texto do botão ficará branco.



Figura 5.3: Efeito Hover no botão btSair

O nosso primeiro toque de requinte está dado: utilização de estilos CSS em formulários, com arquivos CSS.

Há uma documentação grande da Oracle sobre o CSS do JavaFX, que possui diferenças do CSS normal. O nome desse documento é *JavaFX CSS Reference Guide* e é bastante completo:

<http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>

Então, estudem-no e criem suas próprias folhas de estilo CSS nos demais formulários do projeto, não esquecendo da identificação em código.

CAPÍTULO 6

Segundo toque de requinte - Effects

6.1 UMA BREVE INTRODUÇÃO

Visual Effects no JavaFX é o grande ápice da plataforma, pois eleva a parte gráfica a níveis incríveis para aplicações Desktop.

Veremos alguns exemplos de efeitos visuais na tela de “vitrine” e de “item escolhido”, porém saiba que existem mais efeitos. Neste capítulo, mostraremos 3 efeitos interessantes, a reflexão d’água, o sombreamento interno e o sombreamento externo.

6.2 SOMBREAMENTO EXTERNO

Aplicaremos o efeito de sombreamento externo no campo de texto de pesquisa, na tela de “vitrine”. A classe responsável por este efeito é a `DropShadow`. Existem diversas variações de tons, nuances e outros itens para este efeito, então aproveitem para testá-lo e explorá-lo de diversas formas. Para inserirmos efeitos visuais em componentes, usamos o método `setEffect(Effect effect)`, que está presente em todos os `Nodes`. E todos os efeitos herdam da classe `Effect`.

```
private void initComponents() {  
    /* Demais códigos... */  
    txPesquisa = new TextField();  
    txPesquisa.setEffect(new DropShadow());  
    /* Demais códigos seguintes... */  
}
```

Podemos simplesmente instanciar uma nova `DropShadow` dentro do método `setEffect`, e o efeito já será realizado.

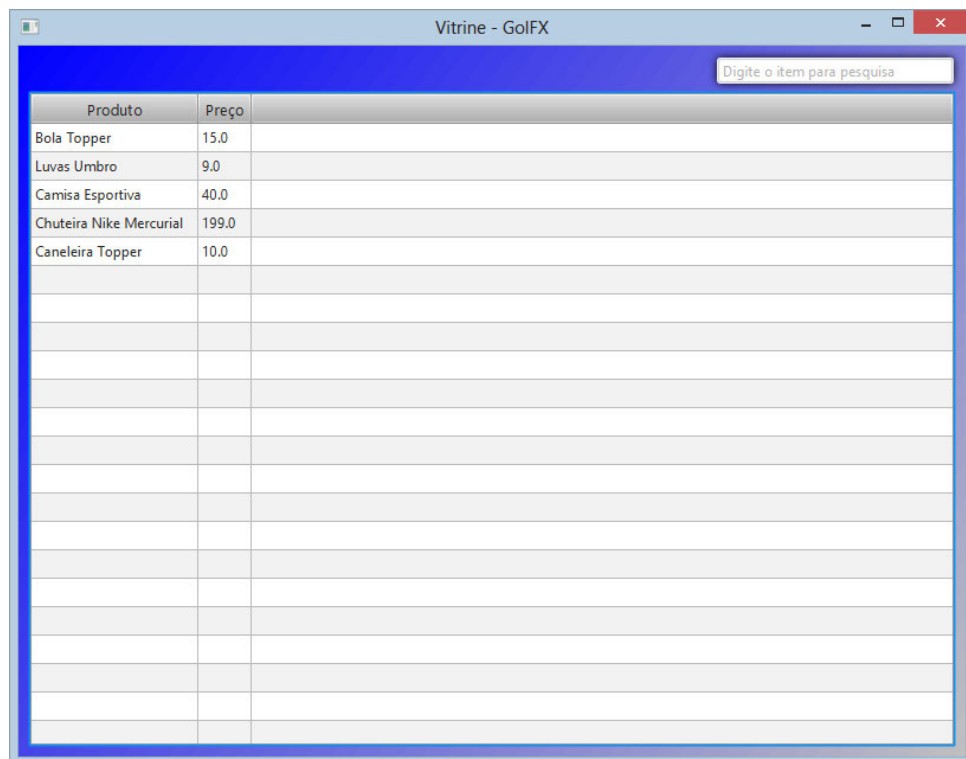


Figura 6.1: Efeito `DropShadow`

Vendo pela primeira vez, tem-se a sensação de que a mudança foi inútil, porém se prestarmos atenção veremos um sombreamento em torno do componente `txPesquisa`, de forma bem simples, tornando-se quase imperceptível.

Podemos aumentar o grau de força do sombreamento, usando o método `setSpread(double value)`, da `DropShadow`. Para isso, precisamos realizar uma

pequena mudança na forma de passar o nosso Effect: iremos primeiramente criar uma nova DropShadow, utilizar o método citado, e depois passar como parâmetro para o método `setEffect`.

```
private void initComponents() {  
    /* Demais códigos... */  
    txPesquisa = new TextField();  
    DropShadow ds = new DropShadow();  
    ds.setSpread(0.5);  
    txPesquisa.setEffect(ds);  
    /* Demais códigos seguintes... */  
}
```

Veremos agora que o sombreamento se tornou mais forte e perceptível.

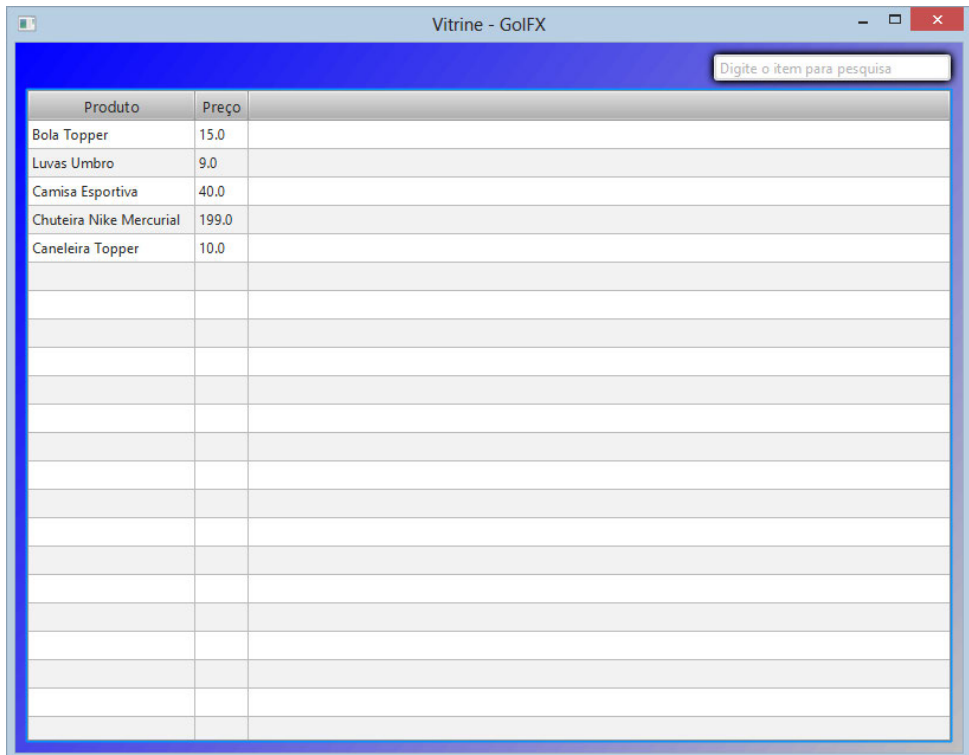


Figura 6.2: Efeito DropShadow intensificado

Podemos também alterar a cor do sombreamento, utilizando o método `setColor(Color color)` da `DropShadow`. Podemos passar cores prontas do JavaFX, que se derivam da classe `Color`, ou também, podemos passar a cor em formato hexadecimal, semelhante em aplicações Web.

```
private void initComponents() {  
    /* Demais códigos... */  
    txPesquisa = new TextField();  
    DropShadow ds = new DropShadow();  
    ds.setSpread(0.5);  
    ds.setColor(Color.RED);  
    // ds.setColor(Color.web("#FF0000"));  
    // O código acima também daria um tom avermelhado ao sombreamento  
    txPesquisa.setEffect(ds);  
    /* Demais códigos seguintes... */  
}
```

Veremos o sombreamento avermelhado na prática.

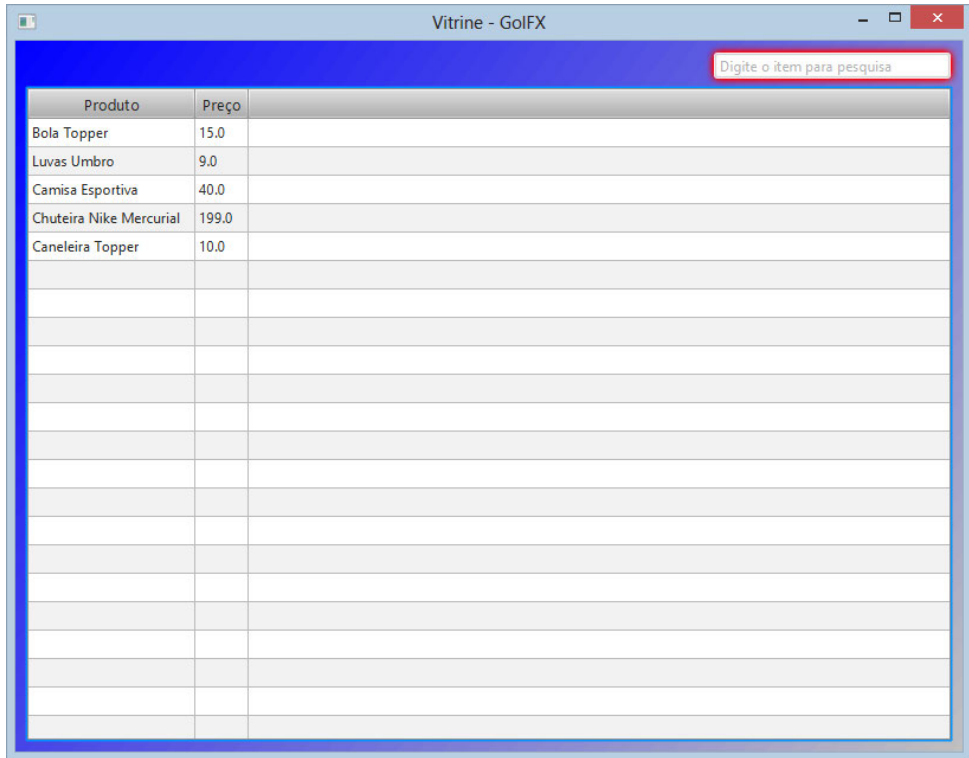


Figura 6.3: Efeito DropShadow avermelhado

Este efeito é agradável em ImageViews no formato PNG. Ao passar o mouse sobre o item (Efeito Hover), ele contorna a imagem com o sombreamento avermelhado, dando um toque de requinte muito interessante, ou também em campos de texto (`TextField`) com fundo branco.

6.3 SOMBREAMENTO INTERNO

Mostraremos agora um outro tipo de sombreamento: o sombreamento interno. A classe responsável por este efeito é a `InnerShadow`, e dá um efeito interessante principalmente em Buttons. O código é semelhante ao da `DropShadow`. Usaremos para dar um sombreamento no botão *Adicionar ao Carrinho*, da tela `ItemApp`.

```
private void initComponents() {  
    /* Demais códigos... */  
}
```

```
btAddCarrinho = new Button("Adicionar ao Carrinho");  
btAddCarrinho.setEffect(new InnerShadow());  
/* Demais códigos seguintes... */  
}
```

Veremos agora o resultado do sombreamento interno básico.



Figura 6.4: Efeito InnerShadow

Podemos também mudar a cor do sombreamento utilizando o método `setColor(Color color)` da `InnerShadow`.

```
private void initComponents() {  
    /* Demais códigos... */  
    btAddCarrinho = new Button("Adicionar ao Carrinho");  
    InnerShadow is = new InnerShadow();  
    is.setColor(Color.RED);  
    btAddCarrinho.setEffect(is);  
    /* Demais códigos seguintes... */  
}
```

Agora, podemos ver o sombreamento interno customizado, com a cor vermelha.



Figura 6.5: Efeito InnerShadow avermelhado

6.4 REFLEXÃO D'ÁGUA

Este efeito, em minha opinião, é o mais interessante. Ele cria um efeito de reflexão d'água em um componente, e qualquer mudança do componente, seja de cor ou texto, também é refletido automaticamente.

Usaremos este efeito na imagem do item escolhido, na tela `ItemApp`, o que ocasionará um efeito surpreendente de imagem reflexiva. O seu uso é semelhante ao uso básico da `InnerShadow` e `DropShadow` — basta instanciar uma nova reflexão, que é a classe `Reflection`.

```
private void initComponents() {  
    /* Demais códigos... */  
    imgItem = new ImageView(new Image(images[index]));  
    imgItem.setFitWidth(300);  
}
```

```
imgItem.setFitHeight(200);  
imgItem.setEffect(new Reflection());  
/* Demais códigos seguintes... */  
}
```

Com este simples código, vemos um efeito elegante de reflexão d'água na `ImageView imgItem`, o mesmo vale para qualquer imagem dos itens escolhidos na vitrine.



Figura 6.6: Efeito Reflection

IMPORTANTE:

Todos os *Effects* do JavaFX derivam-se do package `javafx.scene.effect`, e não do package `com.sun.scenario.effect`.

E assim vemos alguns *Visual Effects* permitidos pelo JavaFX, que dão efeitos gráficos elegantes, tornando a aplicação Desktop mais rica e dinâmica para o usuário.

A Oracle disponibiliza um conteúdo explicativo com códigos de cada Visual Effect, e sua implementação:

http://docs.oracle.com/javafx/2/visual_effects/jfxpub-visual_effects.htm

CAPÍTULO 7

Terceiro toque de requinte - Transitions e Timelines

7.1 O QUE SÃO TRANSITIONS E TIMELINES?

Transitions e Timelines são efeitos visuais em JavaFX para troca de valores de propriedades de forma dinâmica e elegante, dando efeitos de transição. Por exemplo, podemos mudar a posição de um componente de um lado para o outro da tela, como se tivesse se arrastando, em um tempo determinado pelo desenvolvedor. Outro exemplo: podemos dar um efeito Fade In/Fade Out em um componente, transformando-o em opaco e visível em um tempo estabelecido programaticamente.

7.2 FADE TRANSITION

Começaremos demonstrando na prática sobre estes efeitos com o segundo exemplo dado na seção anterior: pegaremos a imagem da tela `ItemApp` e daremos

um efeito *Fade In/Fade Out*, ao executar a tela. Daremos um tempo estabelecido de 3 segundos, para enxergar o efeito de forma clara. Para isto, usaremos a classe `FadeTransition`. Criaremos um método chamado `initTransition()` para ser iniciado no método `start`. Vejamos a sintaxe:

```
public void start(Stage stage) throws Exception {
    /* Demais códigos... */
    initComponents();
    initListeners();
    initTransition();
    /* Demais códigos seguintes... */
}

private void initTransition() {
    FadeTransition transition = new FadeTransition(
        Duration.millis(3000), imgItem);
    transition.setFromValue(0.0);
    transition.setToValue(1.0);
    transition.play();
}
```

Estudando sua API, veremos cada parte:

- Instanciamos um novo `FadeTransition`, passando como parâmetro dois valores: a quantidade de tempo que acontecerá a transição (no nosso caso, utilizamos a ajuda da classe `Duration`, determinando 3000 milissegundos, ou seja, 3 segundos) e qual o componente que sofrerá a ação do efeito;
- Determinamos o grau de opacidade inicial, com o método `setFromValue(double value)`, lembre-se que este grau varia de 0.0 (invisível) a 1.0 (visível);
- Determinamos o grau de opacidade final, com o método `setToValue(double value)`;
- Executamos a transição, com o método `play()`.

O resultado final veremos ao clicar em um item da tabela da vitrine, na classe `VitrineApp`. Ao executar a tela `ItemApp`, veremos o efeito *Fade In/Fade Out* executando por 3 segundos. Na imagem seguinte, veremos este efeito em um tempo próximo à 1,5 segundos, onde a opacidade é média (aproximadamente 0.5).

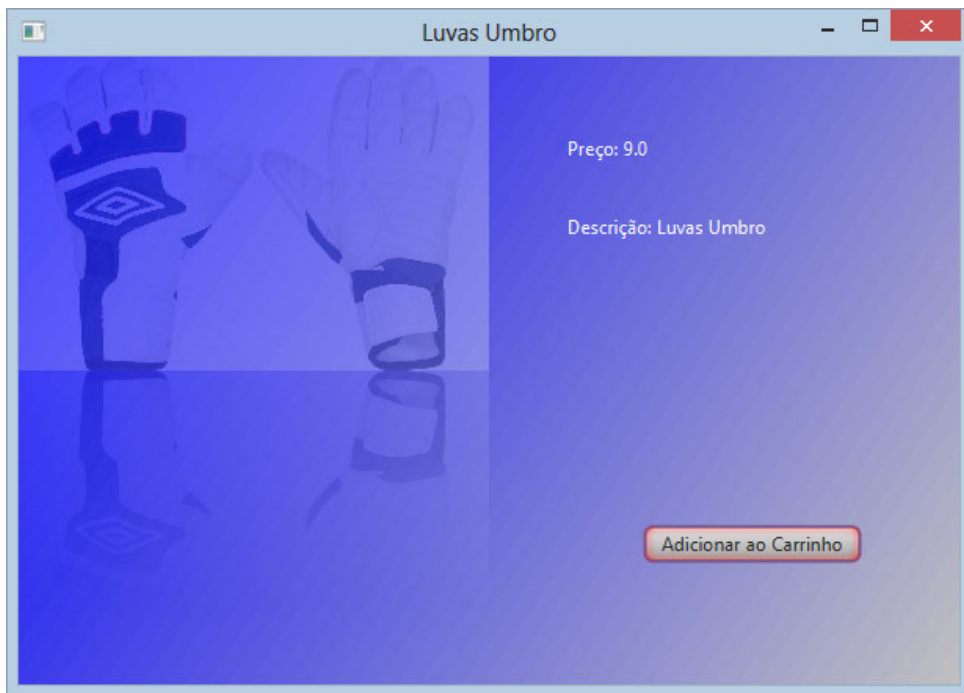


Figura 7.1: Efeito Fade Transition

Podemos indicar para este `FadeTransition` uma repetição no sentido inverso, usando o método `setAutoReverse(boolean value)`. E também usar o método `setCycleCount(int value)` para determinar quantas vezes esta transição se repetirá. Se usarmos como parâmetro o valor `Timeline.INDEFINITE`, o efeito se repetirá infinitamente, até a tela fechar ou a aplicação encerrar.

7.3 SCALE TRANSITION

Outro efeito bastante utilizado é o de escala, com o qual podemos aumentar ou diminuir o tamanho de um componente em um determinado espaço de tempo. A sintaxe é parecida com a da `FadeTransition`, mudando apenas os métodos de indicação de valores das propriedades. Vejamos um exemplo com o botão *Adicionar ao Carrinho* da tela `ItemApp`. Ao passar o mouse sobre o botão, o efeito é acionado, aumentando o tamanho do botão para 50% a mais, e, ao retirar o mouse do botão, o efeito é acionado de forma contrária, reduzindo o tamanho para o tamanho normal. Como

estas ações são do botão, colocaremos no método `initListeners()`, responsável pelas ações dos componentes.

```
private void initListeners() {
    btAddCarrinho.setOnMouseEntered(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            ScaleTransition transition = new ScaleTransition(
                Duration.millis(2000), btAddCarrinho);
            transition.setToX(1.5);
            transition.setToY(1.5);
            transition.play();
        }
    });

    btAddCarrinho.setOnMouseExited(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            ScaleTransition transition = new ScaleTransition(
                Duration.millis(2000), btAddCarrinho);
            transition.setToX(1.0);
            transition.setToY(1.0);
            transition.play();
        }
    });
    /* Demais códigos... */
}
```

O método `setToX(double value)` altera a largura do componente, e o método `setToY(double value)` altera sua altura.

A imagem a seguir contém o efeito no ponto 1 segundo, ao passar o mouse sobre o botão.



Figura 7.2: Efeito Scale Transition

Também podemos utilizar os métodos `setCycleCount(int value)` e `setAutoReverse(boolean value)` nesta transição.

7.4 PARALLEL E SEQUENTIAL TRANSITIONS

Podemos indicar transições para serem executadas paralelamente, uma ao mesmo tempo da outra, ou sequencialmente, uma após a outra. Para isto, usamos as classes `ParallelTransition` e `SequentialTransition`. Daremos um exemplo com cada uma, através dos dois efeitos criados anteriormente. Mas, para isto, precisamos indicar a `ScaleTransition` do `Button btAddCarrinho` na execução da tela, e não mais com efeito `Hover` (ao passar o mouse).

Tiraremos o código do `OnMouseEntered` e `OnMouseExited` do `btAddCarrinho` para o exemplo seguinte. Primeiro, trabalharemos um exemplo com transições paralelas. Veremos que não se pode utilizar o método `play()` em cada `Transition`, e sim, na `ParallelTransition`.

```
private void initTransition() {  
    FadeTransition fTransition = new FadeTransition(  
        Duration.millis(2000), imgItem);  
    fTransition.setFromValue(0.0);  
    fTransition.setToValue(1.0);  
    ScaleTransition sTransition = new ScaleTransition(  
        Duration.millis(2000), btAddCarrinho);  
    sTransition.setToX(1.5);  
    sTransition.setToY(1.5);  
    sTransition.setAutoReverse(true);  
    ParallelTransition pTransition = new ParallelTransition();  
    pTransition.getChildren().addAll(fTransition, sTransition);  
    pTransition.play();  
}
```

Percebe-se que há um efeito acontecendo paralelamente ao outro, podendo fazer isto com quantos Effects desejarmos, apenas adicionando através do método `getChildren().addAll(Transition... transitions)`, da `ParallelTransition`. Na imagem a seguir, veremos os efeitos acontecendo paralelamente, aproximadamente no ponto 1 segundo.

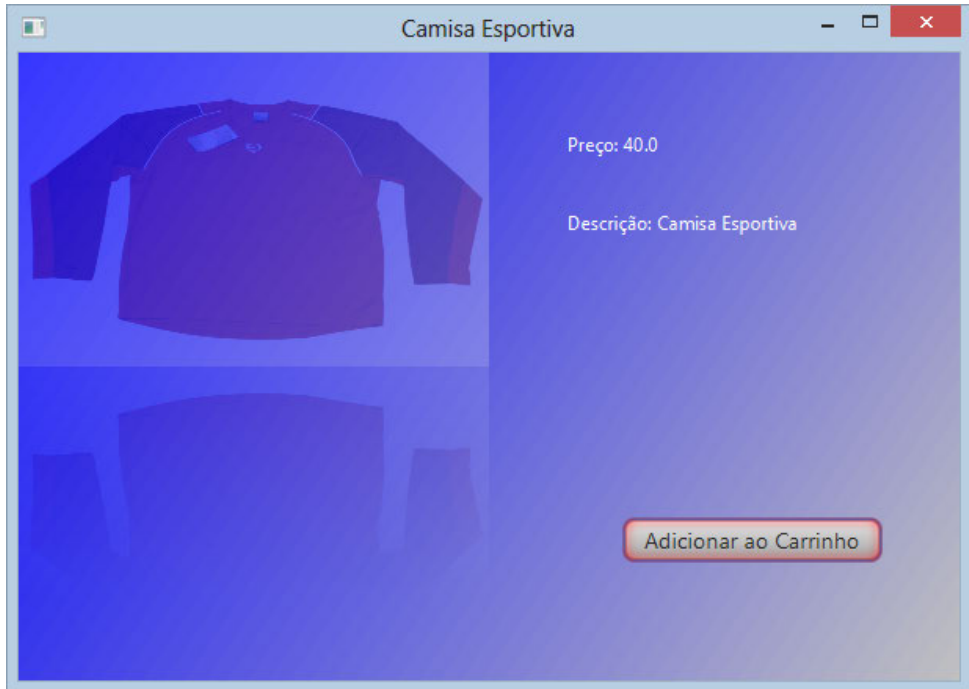


Figura 7.3: Transições paralelas

Para criarmos transições sequenciais (`SequentialTransition`), a sintaxe é semelhante. O efeito causado é de transições acontecendo uma após a outra.

```
private void initTransition() {  
    /* Criar FadeTransition e ScaleTransition do exemplo anterior... */  
    SequentialTransition seqTransition = new SequentialTransition();  
    seqTransition.getChildren().addAll(fTransition, sTransition);  
    seqTransition.play();  
}
```

Com este código, o efeito Fade In/Fade Out do `imgItem` acontece primeiro, e posteriormente o efeito de escala do `btAddCarrinho` acontece.



Figura 7.4: Transições Sequenciais

7.5 TIMELINE

Com os *Timelines*, podemos modificar propriedades de componentes em um determinado espaço de tempo. Com eles, podemos simular efeitos semelhantes aos de *Transitions*, com a vantagem de podermos modificar as propriedades, como texto, cor, escala, opacidade etc.

No nosso exemplo, colocaremos uma *Timeline* no componente `imgItem`, com um efeito Fade In/Fade Out em um tempo indeterminado. Para isto, modificaremos a propriedade de opacidade (`opacityProperty`) do componente.

```
private void initTimeline() {
    Timeline timeline = new Timeline();
    KeyValue kv = new KeyValue(imgItem.opacityProperty(), 0.0);
    KeyFrame kf = new KeyFrame(Duration.millis(2000), kv);
    timeline.getKeyFrames().add(kf);
    timeline.setCycleCount(Timeline.INDEFINITE);
}
```

```
    timeline.setAutoReverse(true);  
    timeline.play();  
}
```

Para a criação de Timelines, precisamos do auxílio de duas classes: `KeyValue` e `KeyFrame`. A `KeyValue` é responsável por identificar qual propriedade de componente será modificada (`imgItem.opacityProperty()`) e quanto será seu novo valor, após o efeito (o.o). Já a `KeyFrame` identifica o tempo de execução do efeito (`Duration.millis(2000)`) e quais `KeyValues` serão ocorridas (kv), podemos indicar diversas `KeyValues` para o mesmo `KeyFrame`.

Identificamos a `KeyFrame` que será realizada no `Timeline`, com o método `getKeyFrames().add(KeyFrame kf)`. Os dois métodos posteriores identificam que o ciclo do `Timeline` acontecerá infinitamente e que seu processo acontecerá também na forma reversa. Por fim, o executamos, com o método `play()`.

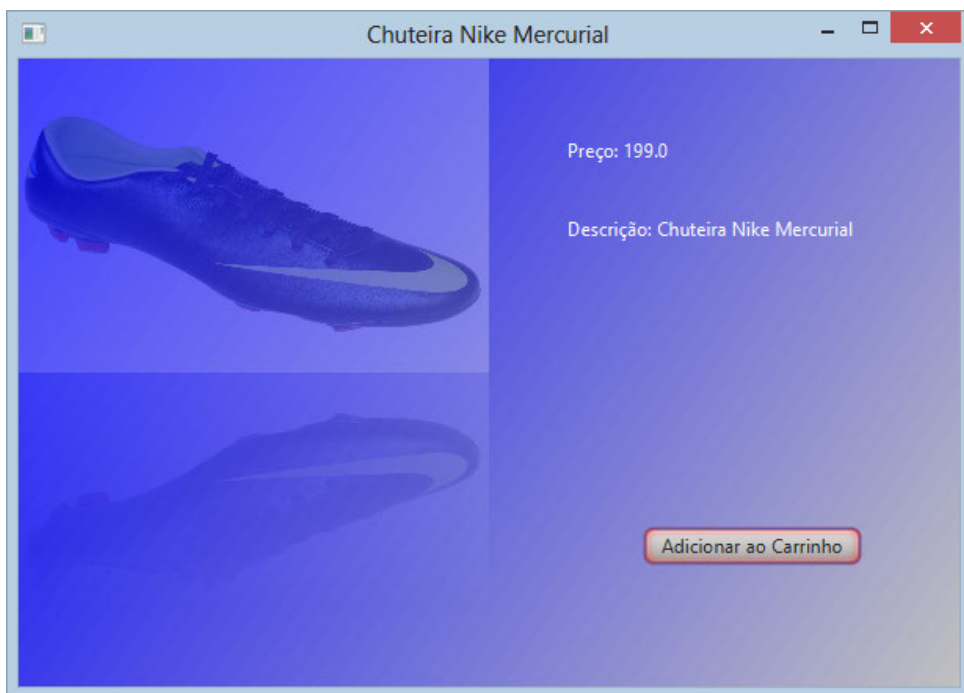


Figura 7.5: Timeline

A Oracle disponibilizou uma página com uso de Transitions e Timelines para aprendizado:

<http://docs.oracle.com/javafx/2/animations/basics.htm>.

CAPÍTULO 8

JavaFX vs Swing

É bem provável que você já conheça Swing, ou ao menos tenha aprendido o básico quando começou a desenvolver com Java. Por que JavaFX e não Swing?

8.1 ENTENDENDO JAVAX.SWING

Swing é um kit de ferramentas de componentes utilizados em Java (*Widget Toolkit*). Ele surgiu no Java 1.2 como uma extensão do Java, oferecendo componentes de alto nível e procurando renderizá-los por conta própria, sem delegar esta tarefa ao sistema operacional, o que era revolucionário na época. Porém, devido a este fato, sua performance era bem menor, comparado a outras APIs gráficas, e consumia mais memória RAM.

Ele veio como uma forma de aprimorar o AWT (*Abstract Window Toolkit*), API gráfica das primeiras versões do Java, a qual dependia de código nativo da plataforma onde rodava. Este fator tornou o AWT obsoleto, pois dependendo do sistema operacional utilizado pelo desenvolvedor, os componentes se comportavam de uma

maneira diferente.

O Swing não necessitava de código nativo da plataforma e ainda possuía uma gama de *skins* a serem utilizados, chamados de *Look and Feel*. Todos os componentes Swing faziam parte do *package* `javax.swing`, e continham a letra J em seu início, como o `JButton`, o `JTextField`, o `JPanel`, etc. Apesar de saber que a grande maioria dos leitores já ouviram falar ou já “devoraram” o Swing, vamos mostrar um pequeno exemplo de uma aplicação com esta API.

No nosso exemplo, temos apenas uma tela e um texto estático (`JLabel`) com o texto “Casa do Código!”. Para isto utilizaremos três componentes:

- `JFrame`: É a tela em si, ela deve ser estendida (`extends`) na sua classe;
- `JPanel`: É o painel onde colocaremos os componentes da tela;
- `JLabel`: É o componente com texto estático (seria o `Label`, do JavaFX, que estamos aprendendo).

Veremos como ficaria esta simples tela:

```
public class TelaComSwing extends JFrame {
    private JPanel pane;
    private JLabel lbMensagem;

    public TelaComSwing() {
        super("Nossa tela com Swing!");
        initComponents();
    }

    private void initComponents() {
        pane = new JPanel();
        lbMensagem = new JLabel("Casa do Código!");
        pane.add(lbMensagem);
        getContentPane().add(pane, BorderLayout.CENTER);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new TelaComSwing();
    }
}
```

```
}  
}
```

Explicando sucintamente o código, o nosso construtor indica uma superclasse (`JFrame`), na qual o parâmetro passado é o texto da barra de título da tela.

Criamos um método `initComponents()`, semelhante ao que fazemos no JavaFX para inicialização de componentes. Neste método, iniciamos os dois componentes `pane` e `lbMensagem`, depois adicionamos o `JLabel` no painel.

O método `getContentPane()` indica o painel principal da tela, então adicionamos o painel ao painel principal, colocando-o no centro, com o parâmetro `BorderLayout.CENTER`. Depois, indicamos que o fechamento de tela será definitivo, com o método `setDefaultCloseOperation`, passando como parâmetro o `JFrame.EXIT_ON_CLOSE`.

O método `pack()`, cria e renderiza cada componente na tela ajustando seus tamanhos. O `setVisible`, passando o valor `true`, abre a tela. Por fim, o método principal `main` cria a tela e executa-a.

Veremos que a interface é um tanto precária, porém na época de surgimento do Swing, era revolucionária, pois sua interface parecia muito com a do próprio sistema operacional:

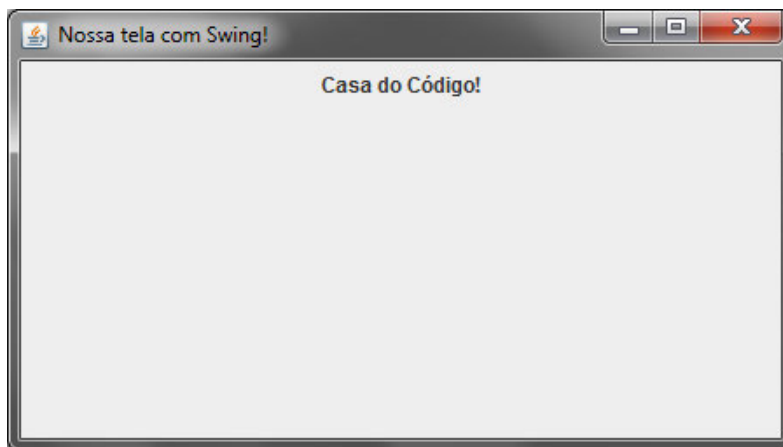


Figura 8.1: Tela com Swing

Com o passar do tempo, em versões do Java subsequentes, o Swing foi sendo melhorado, os conceitos de listeners foram inseridos, e o relacionamento com fra-

meworks conceituadas também foi incorporado. Porém, o grande empecilho visto pelos desenvolvedores era sua fraca qualidade gráfica, padrão entre aplicações Desktop.

Se implementarmos a mesma lógica da tela para o JavaFX, veremos a estrutura com algumas semelhanças e diferenças. Colocaremos aqui a mesma tela feita em Swing para o JavaFX, para que você possa ver a diferença de código e também de visual gráfico.

```
public class TelaComJavaFX extends Application {
    private AnchorPane pane;
    private Label lbMensagem;

    @Override
    public void start(Stage stage) throws Exception {
        pane = new AnchorPane();
        Scene scene = new Scene(pane);
        stage.setScene(pane);
        stage.setTitle("Nossa tela com JavaFX!");
        stage.show();
        initComponents();
    }

    private void initComponents() {
        lbMensagem = new Label("Casa do Código!");
        pane.getChildren().add(lbMensagem);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

O resultado da mesma tela feita em Swing para o JavaFX é esta:

8.2 JAVAFX DENTRO DO SWING?

Com o surgimento do JavaFX, em sua versão 2.0, a interface se tornou eficiente e elegante, porém, ainda existiam diversos programadores habituados com os códigos de aplicações Swing. Então, a Oracle resolveu incorporar a possibilidade de criar

componentes JavaFX dentro de um `JFrame`, por exemplo, através do componente `JFXPanel`.

A criação da classe envolve conceitos `JFrame` e `Application`, pois criamos a tela baseada em `JFrame`, e criamos os componentes do JavaFX, dentro do `JFXPanel`, passando uma nova `Scene`. Veremos a classe a seguir:

```
public class TelaComSwing extends JFrame {

    private static void initComponents() {
        JFrame frame = new JFrame("Tela com Swing + JavaFX!");
        final JFXPanel fxPanel = new JFXPanel();
        frame.add(fxPanel);
        frame.setSize(300, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                initFX(fxPanel);
            }
        });
    }

    private static void initFX(JFXPanel fxPanel) {
        Scene scene = createScene();
        fxPanel.setScene(scene);
    }

    private static Scene createScene() {
        AnchorPane root = new AnchorPane();
        Scene scene = new Scene(root);
        Label lbMensagem = new Label("Casa do Código!");
        root.getChildren().add(lbMensagem);
        return scene;
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                initComponents();
            }
        });
    }
}
```

```
        }  
    });  
}  
  
}
```

À primeira vista, este código pode parecer complexo. Mas não é.

O que fizemos foram dois processos: primeiro, criamos o método `initComponents()`, com a inicialização da `JFrame` e da `JFXPanel`, onde ficarão os componentes JavaFX. De novidade, há apenas o método `setSize(double width, double height)` para dimensionar a tela para 300x200. Segundo, criamos uma nova `Scene`, semelhante ao que fazíamos na classe `Application`: criamos um painel principal (`AnchorPane`) e o texto estático (`Label`), adicionando-o ao pane, que por sua vez é adicionado à scene. Este método de inicialização dos componentes JavaFX é feito dentro do `initComponents()`, porém a `Thread` da `Application` é diferente — portanto, precisamos usar `Platform.runLater(Runnable runnable)`, com a chamada do método dentro. Por fim, no método `main`, chamamos o método `initComponents()`, dentro de uma `SwingUtilities` para identificar a `Thread` padrão.

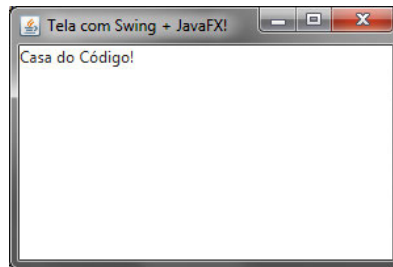


Figura 8.2: Tela com JavaFX dentro de Swing

CAPÍTULO 9

Mais componentes JavaFX

Neste capítulo, veremos diversos componentes inéditos do JavaFX. Muitos deles não possuem alternativa nativa no swing. O JavaFX realmente vem com uma biblioteca bem mais ampla de componentes visuais.

Primeiro, veremos os novos Containers existentes nesta plataforma. Containers são regiões onde se fixam componentes, como o `AnchorPane`, que utilizávamos para adicionar componentes.

9.1 ACCORDION

O primeiro Container que mostraremos é o `Accordion`, que é um conjunto de painel deslizantes, em forma de sanfona. Estes painéis são `TitledPanels`, ou seja, painéis com título, e podem ser minimizados e restaurados, dando um efeito sanfona ao Container. O uso do `Accordion` é simples: precisamos indicar novos `TitledPanels` e seus conteúdos, para então indicá-los dentro do `Accordion`.

```
private void initAccordion() {
    Accordion accordion = new Accordion();
    TitledPane tPane1 = new TitledPane("Primeiro Paine1",
        new Label("Primeiro painel aberto!"));
    TitledPane tPane2 = new TitledPane("Segundo Paine1",
        new Label("Segundo painel aberto!"));
    TitledPane tPane3 = new TitledPane("Terceiro Paine1",
        new Label("Terceiro painel aberto!"));
    accordion.getPanes().addAll(tPane1, tPane2, tPane3);
    /*
        Indicar Accordion em um painel principal,
        AnchorPane por exemplo...
    */
}
```

Veremos o resultado na imagem a seguir:

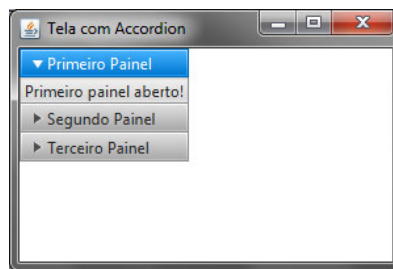


Figura 9.1: Accordion

9.2 HBox E VBox

Os próximos Containers que veremos são o HBox e o VBox, que são caixas horizontais e verticais para adicionar componentes de forma linear (um ao lado do outro). Seu uso é bastante simples também: criamos nossos componentes livremente, e adicionamos ao Box, e automaticamente haverá o ajuste um ao lado do outro ou um embaixo do outro. Veremos o uso dos dois Containers:

```
private void initHBoxAndVBox() {
    HBox hBox = new HBox();
    Label lbHBox1 = new Label("Estamos... ");
    Label lbHBox2 = new Label("na caixa... ");
}
```



```

Label lbHBox3 = new Label("horizontal!!!");
hBox.getChildren().addAll(lbHBox1, lbHBox2, lbHBox3);
VBox vBox = new VBox();
vBox.setLayoutY(30);
Label lbVBox1 = new Label("Agora estamos...");
Label lbVBox2 = new Label("na caixa...");
Label lbVBox3 = new Label("vertical!!!");
vBox.getChildren().addAll(lbVBox1, lbVBox2, lbVBox3);
/* Indicar HBox e VBox em um painel principal... */
}

```

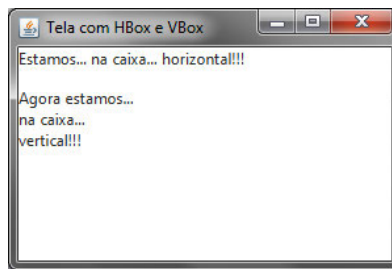


Figura 9.2: HBox + VBox

9.3 GRIDPANE PARA LINHAS E COLUNAS

O último Container que mostraremos é o GridPane, que alinha os componentes em linhas e colunas. Ele é bem interessante para organização de telas de forma exata, como uma tela de login.

```

private void initGridPane() {
    GridPane gPane = new GridPane();
    gPane.setHgap(10);
    gPane.setVgap(10);
    Label lbLogin = new Label("Login:");
    TextField txLogin = new TextField();
    Label lbSenha = new Label("Senha:");
    PasswordField txSenha = new PasswordField();
    Button btEntrar = new Button("Entrar");
    Button btSair = new Button("Sair");
    gPane.addRow(0, lbLogin, txLogin);
    gPane.addRow(1, lbSenha, txSenha);
}

```

```
gPane.addRow(2, btEntrar, btSair);  
/* Indicar GridPane em um painel principal... */  
}
```

Ao criarmos o `GridPane`, indicamos um espaçamento horizontal e vertical entre os componentes, com os métodos `setHgap(double value)` e `setVgap(double value)`. Depois, criamos todos os componentes necessários para a tela de login, e, por fim, adicionamos cada componente em linhas, com o método `addRow(int row, Node... nodes)`. Podíamos também adicionar cada componente em uma posição, com o método `add(Node node, int row, int column)`, e passaríamos como parâmetro o componente (node), o número da linha (row) e o número da coluna (column).



Figura 9.3: Grid Pane

Existem diversos outros Containers, com funções distintas, cabe ao desenvolvedor conhecer suas variações e formas de uso. Pratiquem!

9.4 UM HTMLEDITOR PRONTO PARA VOCÊ USAR

Estamos habituados a trabalhar com diversos controles (*controls*), que fazem input de dados. Um exemplo é o `TextField`, no qual podemos inserir um valor, e também retornar o mesmo. Este tipo de componente é o mais utilizado em aplicações, em sua grande maioria são `Buttons`, `TextFields` etc. Porém, existem componentes novos comparados ao Swing.

O primeiro que veremos é o HTML Editor, que é um editor de HTML 5 para reprodução de textos. Para usá-lo da forma mais simples, basta instanciá-lo e indicar em um painel principal.

```
private void initHTMLEditor() {  
    HTMLEditor editor = new HTMLEditor();  
    /* Indicar HTMLEditor em um painel principal... */  
}
```

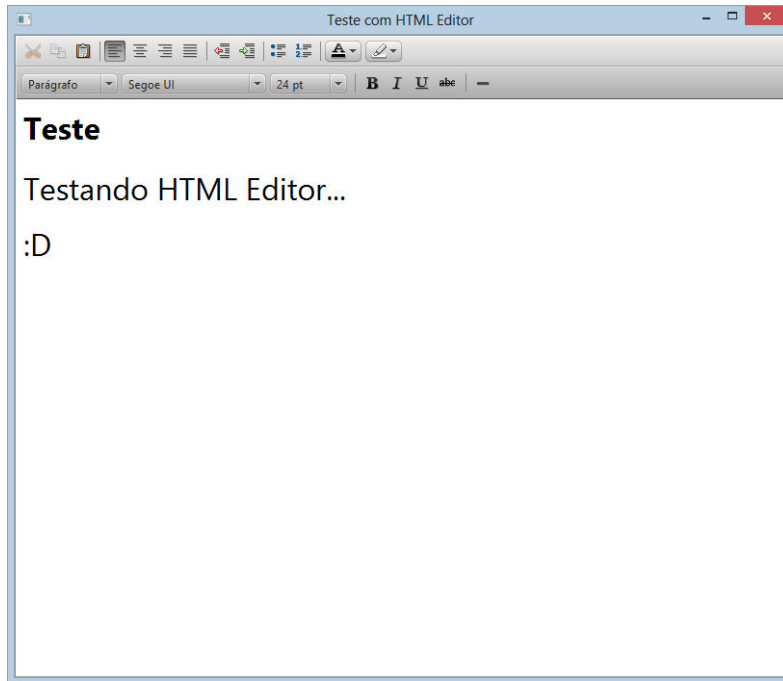


Figura 9.4: HTML Editor

Com o HTML Editor, você pode abrir e salvar arquivos em diversos formatos, utilizando juntamente um `FileChooser`.

Podemos, também, abrir um arquivo `.html` e ser visualizado como uma página de web pelo HTML Editor.

Para isto, utilizamos os métodos `getHtmlText()` e `setHtmlText(String value)`.

Para testar, crie um `Button` que chame o seguinte método, dentro do evento de ação do botão (`setOnAction(new EventHandler<ActionEvent>() { ... })`):

```
private void setText(HTMLEditor editor) {  
    FileChooser fileChooser = new FileChooser();
```

```

// Indica um filtro de extensão de arquivos
FileChooser.ExtensionFilter extFilter =
    new FileChooser.ExtensionFilter(
        "HTML files (*.html)", "*.html");
fileChooser.getExtensionFilters().add(extFilter);
// Abre uma caixa de diálogo para abertura de arquivos
File file = fileChooser.showOpenDialog(null);
if (file != null) {
    StringBuilder stringBuffer = new StringBuilder();
    BufferedReader bufferedReader = null;
    try {
        bufferedReader = new BufferedReader
            (new FileReader(file));
        String text;
        while ((text = bufferedReader.readLine()) != null) {
            stringBuffer.append(text);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            bufferedReader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    editor.setHtmlText(stringBuffer.toString());
}
}

```

Agora, ao clicar no botão criado, escolha um arquivo no formato .html em seu computador, e veja o resultado mostrado no HTML Editor.

9.5 HYPERLINKS

O segundo componente de controle é o Hyperlink, que é um link para uma determinada ação, como abertura de tela, arquivo, site, comando.

```

private void initHyperlink() {
    Hyperlink link = new Hyperlink("Clique aqui...");
    link.setOnAction(new EventHandler<ActionEvent>() {

```

```

@Override
public void handle(ActionEvent event) {
    try {
        Desktop.getDesktop().browse(new URI(
            "http://www.casadocodigo.com.br"));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
});
/* Indicar Hyperlink em um painel principal... */
}

```

Criamos uma ação ao clicar no Hyperlink, com a qual ele abre o site da Casa do Código pelo seu navegador padrão, utilizando a API do Java AWT.

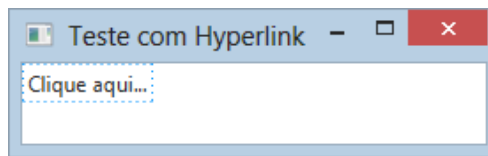


Figura 9.5: Hyperlink

9.6 A FAMOSA E TEMIDA BARRA DE PROGRESSO

O terceiro Control é o Progress Indicator, que é uma barra de progresso com porcentagem. Para usá-lo, precisamos utilizar também uma `Task`, que cria uma tarefa síncrona para funcionar como contador para o progresso do componente.

```

private void initProgressIndicator() {
    ProgressIndicator progress = new ProgressIndicator(0.0);
    Task<Void> task = new Task<Void>() {
        @Override
        protected Void call() throws Exception {
            final int max = 10;
            for (int i = 1; i <= max; i++) {
                updateProgress(i, max);
            }
            try {
                Thread.sleep(1000);
            }
        }
    };
    task.execute();
}

```

```

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    JOptionPane.showMessageDialog(
        null, "Progresso concluído!");
    return null;
}
};
progress.progressProperty().bind(task.progressProperty());
new Thread(task).start();
/* Indicar ProgressIndicator em um painel principal... */
}

```

A nossa Task faz uma iteração de 1 a 10, ligando o progresso da mesma ao progresso do ProgressIndicator, através do método `bind`. Iniciamos a Task como uma Thread, passando como parâmetro e iniciando. O método `updateProgress` é da Task, para indicar seu progress, passando o numero onde está (i) e o máximo (max), utilizando a `Thread.sleep(int value)` para interromper a continuação da execução por 1 segundo (1000 milissegundos). Por fim, ao encerrar a iteração, criamos uma caixa de diálogo mostrando que o progresso foi concluído.

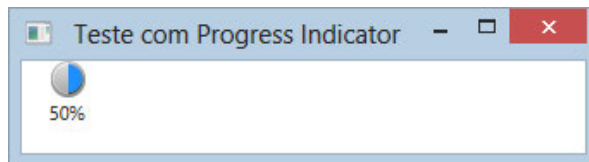


Figura 9.6: Progress Indicator

9.7 VISUALIZANDO ÁRVORES COM TREEVIEW

O quarto Control é o `TreeView`, que é um painel com uma árvore de dados. Cada item é um `TreeItem`, que precisa ter um tipo de dado como parametrização, assim como o `TreeView`. No nosso exemplo, será `String`, pois o valor de cada item será um pequeno texto. Para sua manipulação, basta utilizarmos o método `getChildren().add(TreeItem item)` ou `getChildren().addAll(TreeItem... items)`. Podemos colocar item

dentro de item. Por fim, indicamos o `TreeItem` primário com o método `setRoot(TreeItem item)`, do `TreeView`.

```
private void initTreeView() {
    TreeView<String> treeView = new TreeView<String>();
    TreeItem<String> item1 = new TreeItem<String>("Tópico 1");
    TreeItem<String> subitem1 = new TreeItem<String>("Tópico 1.1");
    TreeItem<String> subitem1_1 =
        new TreeItem<String>("Tópico 1.1.1");
    TreeItem<String> subitem1_2 =
        new TreeItem<String>("Tópico 1.1.2");
    TreeItem<String> subitem2 =
        new TreeItem<String>("Tópico 1.2");
    TreeItem<String> subitem3 =
        new TreeItem<String>("Tópico 1.3");
    TreeItem<String> subitem3_1 =
        new TreeItem<String>("Tópico 1.3.1");

    item1.getChildren().addAll(subitem1, subitem2, subitem3);
    subitem1.getChildren().addAll(subitem1_1, subitem1_2);
    subitem3.getChildren().add(subitem3_1);
    treeView.setRoot(item1);
    /* Indicar TreeView em um painel principal... */
}
```

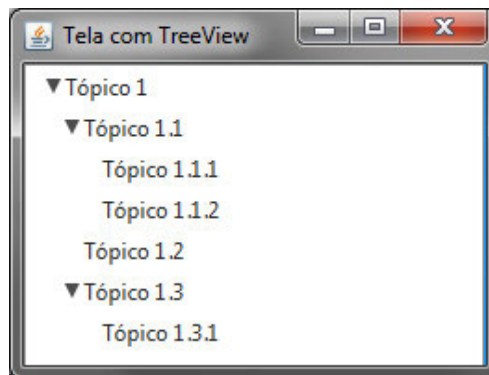


Figura 9.7: Tree View

É totalmente possível substituir a `TableView` da vitrine de produtos do nosso projeto `GolFX` por uma `TreeView`, para visualização de produtos em categorias e subcategorias.

Seria um grande desafio para você, leitor!

9.8 WEBVIEW PARA RENDERIZAR HTML

E o último `Control` sobre o qual falaremos é o `WebView`, que é um painel para visualização de um conteúdo HTML (site). Seu uso é muito simples: basta utilizarmos o método `getEngine().load(String value)`, passando o link do site que deseja abrir, como parâmetro. Podemos também executar `Document's` ou comandos `JavaScript` no site.

```
private void initWebView() {  
    WebView webView = new WebView();  
    webView.getEngine().load("http://www.casadocodigo.com.br/");  
    /* Indicar WebView em um painel principal... */  
}
```




Figura 9.8: Web View

9.9 POPUP CONTROLS

Popup Controls são componentes de popup, como menu de contexto e *tooltip*. Veremos sobre estes dois componentes e entendermos sobre suas funcionalidades.

Primeiro, o `ContextMenu` que é o menu de contexto quando apertamos o botão direito sobre um determinado componente e temos opções relacionadas ou não relacionadas àquele item. Veremos o código:

```
private void initContextMenu() {  
    final ListView<String> listView = new ListView<String>(  
        FXCollections.observableArrayList("Primeiro item",  
                                           "Segundo item", "Terceiro item"));  
    ContextMenu contextMenu = new ContextMenu();  
    MenuItem item1 = new MenuItem("Mostrar valor");  
    item1.setOnAction(new EventHandler<ActionEvent>() {
```

```

@Override
public void handle(ActionEvent event) {
    JOptionPane.showMessageDialog(null, listView
        .getSelectionModel().getSelectedItem());
}
});
MenuItem item2 = new MenuItem("Determinar valor nulo");
item2.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        listView.getItems()
            .set(listView.getSelectionModel()
                .getSelectedIndex(), "null");
    }
});
contextMenu.getItems().addAll(item1, item2);
listView.setContextMenu(contextMenu);
/* Indicar ListView em um painel principal... */
}

```

Criamos uma lista (`ListView`) para representar o componente onde terá ação do `ContextMenu`. Colocamos três valores, através de uma `ObservableList<String>`, com o `FXCollections.observableArrayList(String... values)`. Depois, criamos o `ContextMenu`. Após sua criação, criamos dois `MenuItem`s, representando os itens do menu de contexto. O primeiro item (Mostrar valor) pegará o conteúdo da linha escolhida pelo usuário na lista e o mostra com `JOptionPane`. O segundo item (Determinar valor nulo) pegará também o conteúdo da linha escolhida e o mudará para “null”. Então, precisamos indicar os itens do menu de contexto, com o método `getItems().addAll(MenuItem... items)`, e por fim identificar o `ContextMenu` da nossa lista, com o método `setContextMenu(ContextMenu contextMenu)`.

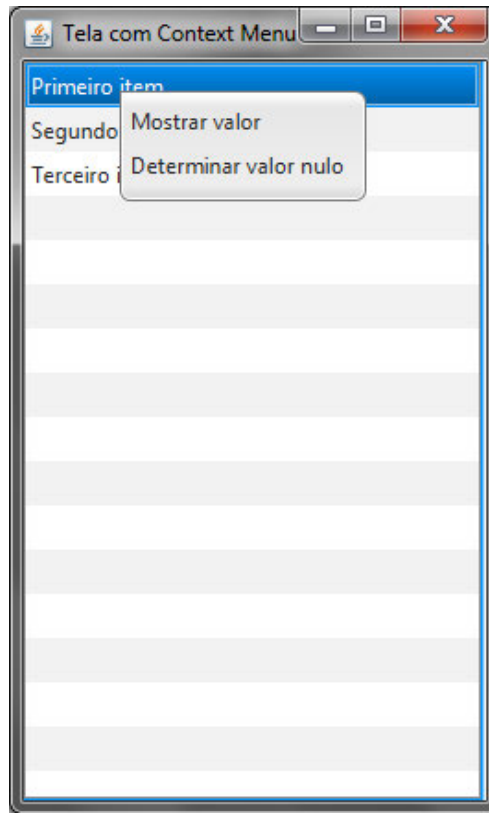


Figura 9.9: Context Menu

E o segundo Popup Control é o Tooltip, uma nota de componente, sobre o qual ao passar o mouse o Tooltip exibe uma determinada informação, ligada ao componente. O seu uso é através de um método estático chamado `install`, no qual passamos como parâmetro o Node que receberá o Tooltip e um novo Tooltip com a mensagem que será exibida ao passar o mouse sobre o Node.

```
private void initTooltip() {  
    Label lbNome = new Label("Nome: ");  
    TextField txNome = new TextField();  
    txNome.setLayoutX(50);  
    Tooltip.install(txNome,  
        new Tooltip("Digite aqui seu nome completo..."));  
    /* Indicar Label e TextField em um painel principal... */  
}
```

Ao passar o mouse sobre o componente txNome, veremos o Tooltip.

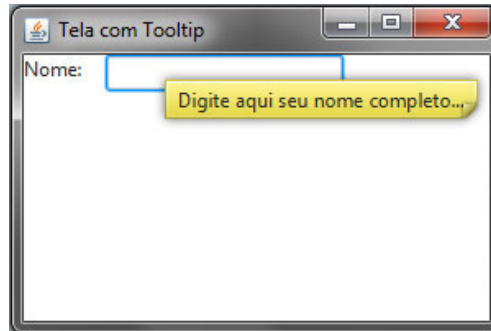


Figura 9.10: Tooltip

9.10 GRÁFICOS PARA DAR MAIS VIDA

Charts são componentes para gráficos de dados. Certamente, este tipo de componente foi o mais útil criado pela Oracle para esta plataforma. Para a criação de gráficos, eram necessários frameworks como o JFreeChart. Com os Charts, isso tornou-se muito prático.

Veremos três casos: o gráfico de barras, o gráfico de linhas e de pizza.

Vamos criar um `BarChart`, o nosso gráfico de barras. Para isto, necessitamos de duas outras classes auxiliares: o `XYChart.Data` e o `XYChart.Series`. O primeiro será para a interpretação de cada dado, e o segundo é o conjunto de dados a ser inserido no gráfico.

```
private void initBarChart() {
    BarChart<String, Number> chartLinguagens =
        new BarChart<String, Number>(
            new CategoryAxis(), new NumberAxis());
    chartLinguagens.setCategoryGap(20);

    chartLinguagens.
        setTitle("Ranking de Linguagens de Programação Mar/2013");
    XYChart.Data<String, Number> dataJava =
        new XYChart.Data<String, Number>("Java", 18.156);
    XYChart.Data<String, Number> dataC =
        new XYChart.Data<String, Number>("C", 17.141);
```

```
XYChart.Data<String, Number> dataObjectiveC =
    new XYChart.Data<String, Number>("Objective-C", 10.230);
XYChart.Data<String, Number> dataCPlus =
    new XYChart.Data<String, Number>("C++", 9.115);
XYChart.Data<String, Number> dataCSharp =
    new XYChart.Data<String, Number>("C#", 6.597);
XYChart.Series<String, Number> seriesData =
    new XYChart.Series<String, Number>();
seriesData.setName("Porcentagem (%)");
seriesData.getData().addAll(dataJava, dataC, dataObjectiveC,
    dataCPlus, dataCSharp);
chartLinguagens.getData().add(seriesData);
/* Indicar o BarChart em um painel principal... */
}
```

Explicando o código: o `BarChart` necessita de duas parametrizações, uma para indicar o tipo de dados recebido na coordenada X, e outra na coordenada Y. No nosso exemplo, faremos um gráfico mostrando o Ranking de Linguagens de Programação, realizado em março de 2013, pela *Tiobe*. Este exemplo será utilizado nos três tipos de gráficos que citaremos. A coordenada X é a linguagem de programação (ou seja, `String`), e a coordenada Y é a porcentagem de usuários desta linguagem (ou seja, `Number`). Como parâmetro, passamos um indicador das coordenadas, se for `String`, então utilizamos o `CategoryAxis`, se for `Number`, utilizamos o `NumberAxis`. Primeiro, indicamos um espaçamento entre cada dado, com o método `setCategoryGap(double value)`, depois, damos um título ao gráfico com o método `setTitle(String value)`. Então, começamos a criar os dados, utilizando a classe `XYChart.Data`, com a mesma parametrização do `BarChart`. Ela recebe dois parâmetros: o nome da linguagem (coordenada X) e o valor da porcentagem (coordenada Y). Depois, criamos o conjunto de dados, o `XYChart.Series`, com a mesma parametrização do `BarChart` também. Utilizamos o `setName(String value)` para indicar o nome do conjunto, e depois, utilizamos o método `getData().addAll(Data... datas)` para adicionar todos os dados do conjunto. Por fim, indicamos a série de dados, com o método `getData().add(Series series)` do `BarChart`. Veja o resultado:

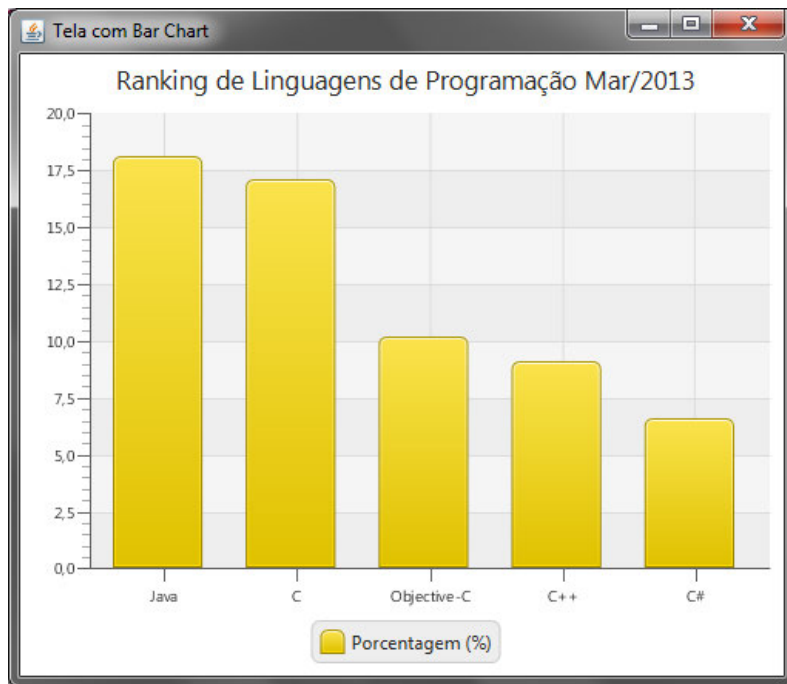


Figura 9.11: Bar Chart

Agora, mostraremos uma `LineChart`, o gráfico de linhas. Para utilizá-lo, assim como o `BarChart`, precisamos das classes `XYChart.Data` e `XYChart.Series`. Como dito anteriormente, utilizaremos o mesmo conjunto de dados do exemplo anterior.

```
private void initLineChart() {
    LineChart<Number, Number> chartLinguagens =
        new LineChart<Number, Number>(
            new NumberAxis(2008, 2013, 5), new NumberAxis());
    chartLinguagens
        .setTitle("Ranking de Linguagens de Programação Mar/2013");
    XYChart.Series<Number, Number> serieJava =
        new XYChart.Series<Number, Number>();
    serieJava.setName("Java");
    serieJava.getData().addAll(new XYChart.Data<Number, Number>(2008, 1),
        new XYChart.Data<Number, Number>(2013, 1));
    XYChart.Series<Number, Number> serieC =
        new XYChart.Series<Number, Number>();
    serieC.setName("C");
}
```

```
serieC.getData().addAll(new XYChart.Data<Number, Number>(2008, 2),
    new XYChart.Data<Number, Number>(2013, 2));
XYChart.Series<Number, Number> serieObjectiveC =
    new XYChart.Series<Number, Number>();
serieObjectiveC.setName("Objective-C");
serieObjectiveC.getData().addAll(
    new XYChart.Data<Number, Number>(2008, 45),
    new XYChart.Data<Number, Number>(2013, 3));
XYChart.Series<Number, Number> serieCPlus =
    new XYChart.Series<Number, Number>();
serieCPlus.setName("C++");
serieCPlus.getData().addAll(
    new XYChart.Data<Number, Number>(2008, 5),
    new XYChart.Data<Number, Number>(2013, 4));
XYChart.Series<Number, Number> serieCSharp =
    new XYChart.Series<Number, Number>();
serieCSharp.setName("C#");
serieCSharp.getData().addAll(
    new XYChart.Data<Number, Number>(2008, 8),
    new XYChart.Data<Number, Number>(2013, 5));
chartLinguagens.getData().addAll(serieJava, serieC, serieObjectiveC,
    serieCPlus, serieCSharp);
/* Indicar o LineChart em um painel principal... */
}
```

Se olharmos a sintaxe, não veremos surpresas. Os pontos importantes e distintos a serem ressaltados são:

- Este gráfico mostrará a variação de posições das 5 linguagens, entre 2008 e 2013. Veremos que quanto mais alta a posição, mais o gráfico desce, pois o número 1 é o menor número;
- Para indicarmos uma variação de números fixa, basta passarmos 3 parâmetros na `NumberAxis`: primeiro, o menor número; segundo, o maior; e terceiro, de quanto em quanto é criado um *Tick*, ou seja, uma parada de números. Como a variação é de apenas 5 anos, colocamos o tick de 5, ou seja, só haverá parada em 2008 e 2013;
- Precisamos criar diversas `XYChart.Series`, pois haverá 5 linhas;
- Cada `XYChart.Data` receberá o ano (coordenada X) e a posição deste ano (coordenada Y).

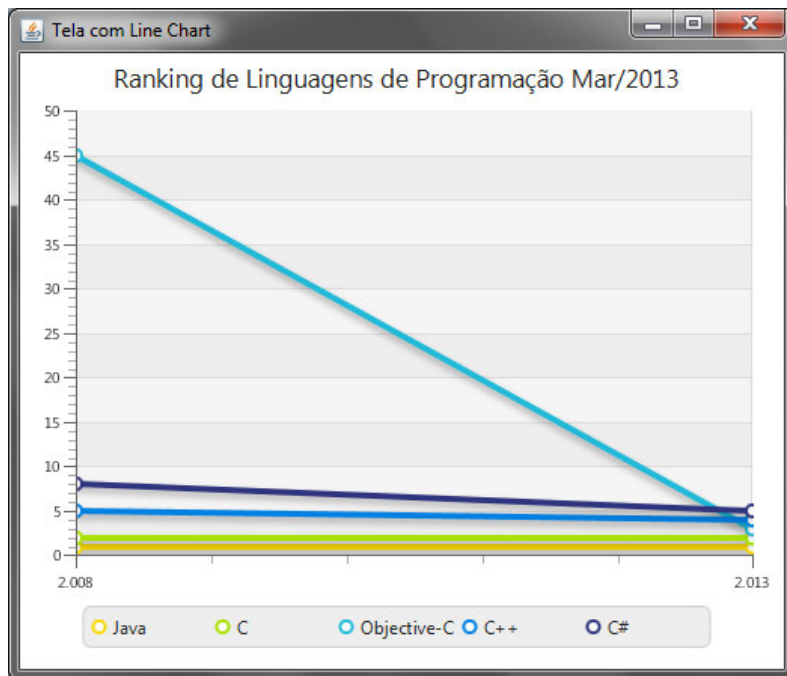


Figura 9.12: Line Chart

Por último, mostraremos o `PieChart`, o gráfico de pizza. Novamente, utilizaremos o exemplo do ranking, agora determinando a porcentagem de usuários em um dado de 100%. Para isto, precisamos criar o dado “Outros”, onde constará o restante das linguagens e sua porcentagem total. O uso do `PieChart` é mais simples do que os anteriores. Basta criarmos um novo `PieChart` e uma lista de `PieChart.Data`, que corresponde aos dados do gráfico.

```
private void initPieChart() {
    PieChart chartLinguagens = new PieChart();
    chartLinguagens.
        setTitle("Ranking de Linguagens de Programação Mar/2013");
    ObservableList<PieChart.Data> datas =
        FXCollections.observableArrayList(
            new PieChart.Data("Java", 18.156),
            new PieChart.Data("C", 17.141),
            new PieChart.Data("Objective-C", 10.230),
            new PieChart.Data("C++", 9.115),
```



```
        new PieChart.Data("C#", 6.597),  
        new PieChart.Data("Outros", 38.761));  
chartLinguagens.setData(datas);  
/* Indicar o PieChart em um painel principal... */  
}
```

Utilizamos uma `ObservableList` parametrizada com `PieChart.Data`, onde passamos diversos dados para compor o gráfico. O resultado final é este:

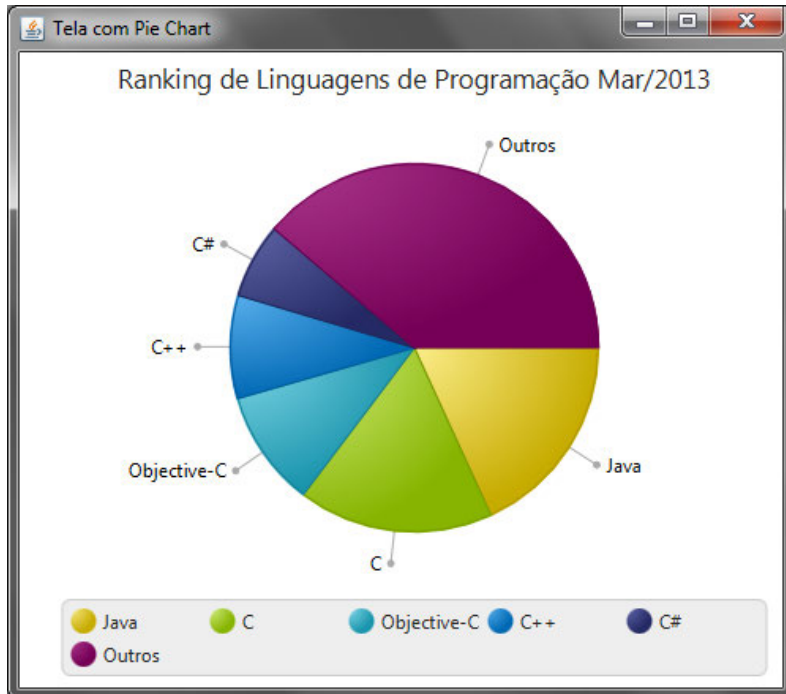


Figura 9.13: Pie Chart

Para aprender mais sobre o uso de *Charts*, entre no link <http://docs.oracle.com/javafx/2/charts/jfxpub-charts.htm>, no qual constam explicações e códigos relacionados ao seu uso, desenvolvido pela Oracle.

9.11 AUDIO E VIDEO

O JavaFX possui componentes para multimídia, com eles você pode controlar toda a estrutura de som, volume e reprodução.

Primeiro, mostraremos como executar um áudio MP3. Para isto, usaremos as classes `Media`, para identificar o nosso objeto multimídia, no caso a música, e `MediaPlayer`, que seria o tocador de áudio.

```
private void initAudioPlayer() {
    Label lbTocando = new Label("Tocando música...");
    Media media = new Media(TelaComSwing.class.getResource("audio.mp3")
        .toExternalForm());
    MediaPlayer mediaPlayer = new MediaPlayer(media);
    mediaPlayer.setAutoplay(true);
    /* Indicar Label em um painel principal... */
}
```

Na construção da `Media`, indicamos o caminho do arquivo a ser executado, um áudio MP3 chamado “audio.mp3”. Então, criamos a `MediaPlayer`, passando como argumento a `Media`. O método `setAutoplay(boolean value)` diz ao `Player` para executar o áudio na inicialização da `MediaPlayer`.

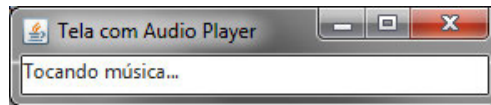


Figura 9.14: Audio Player

Para a execução de um vídeo, precisamos, além das classes `Media` e `MediaPlayer`, de uma `MediaView`, que representa a visualização do vídeo. Ela é um componente, portanto, precisa estar inserido em um painel principal.

```
private void initVideoPlayer() {
    Media media = new Media(TelaComSwing.class.getResource("video.mp4")
        .toExternalForm());
    MediaPlayer mediaPlayer = new MediaPlayer(media);
    mediaPlayer.setAutoplay(true);
    MediaView mediaView = new MediaView(mediaPlayer);
    mediaView.setFitWidth(300);
    mediaView.setFitHeight(200);
    /* Indicar MediaView em um painel principal... */
}
```

Basta criarmos uma nova `MediaView`, passando como parâmetro a nossa `MediaPlayer`, e indicarmos o tamanho do componente de reprodução do vídeo. No exemplo, está 300x200. Então, veremos o resultado:

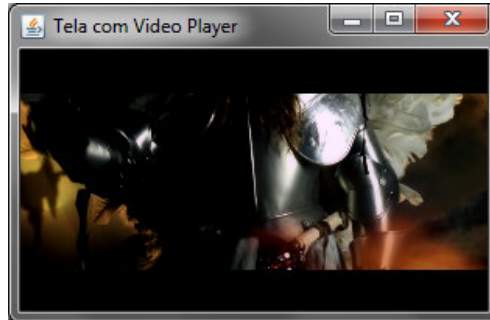


Figura 9.15: Video Player

9.12 SHAPES PARA O CONTROLE FINO

Shapes são objetos que representam figuras geométricas, como círculo, retângulo, linha etc. Podem ser utilizados para aplicações matemáticas ou representativas, como algo científico, por exemplo.

O primeiro *Shape* que mostraremos é o `Circle` que é o círculo. Podemos indicar várias propriedades, mas as que utilizaremos são: cor, centro e raio da circunferência.

```
private void initCircle() {  
    Circle circle = new Circle();  
    circle.setFill(Color.AQUA);  
    circle.setCenterX(100);  
    circle.setCenterY(100);  
    circle.setRadius(50);  
    FadeTransition transition = new FadeTransition(  
        Duration.millis(1000), circle);  
    transition.setFromValue(0.0);  
    transition.setToValue(1.0);  
    transition.setAutoReverse(true);  
    transition.setCycleCount(Transition.INDEFINITE);  
    transition.play();  
}
```

```

    /* Indicar Circle em um painel principal... */
}

```

O método `setFill(Paint paint)` indica a cor do interior do círculo, no nosso caso, `AQUA`. Os métodos `setCenterX(double value)` e `setCenterY(double value)` indicam o centro do círculo em coordenadas. E o método `setRadius(double value)` indica o raio da circunferência. Por fim, fizemos uma pequena brincadeira de efeito `Fade In/Fade Out` a cada 1 segundo, para sumir e surgir a bola infinitas vezes, com o `FadeTransition`.

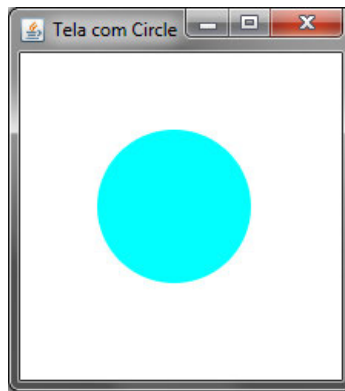


Figura 9.16: Circle

O segundo Shape é o `Line` que é a linha. Para definirmos os pontos da linha, precisamos indicar sua posição inicial e final. A linha pode ter posição negativa, o que significa que ela crescerá no sentido oposto de seu início.

```

private void initLine() {
    Line line = new Line();
    line.setLayoutX(10);
    line.setLayoutY(10);
    line.setStartX(0);
    line.setStartY(0);
    line.setEndX(100);
    line.setEndY(100);
    Line line2 = new Line();
    line2.setLayoutX(10);
    line2.setLayoutY(10);
}

```

```
line2.setStartX(100);  
line2.setStartY(0);  
line2.setEndX(0);  
line2.setEndY(100);  
/* Indicar Line's em um painel principal... */  
}
```

Os métodos para indicação de posição inicial são `setStartX(double value)` e `setStartY(double value)`, e os métodos para indicação de posição final são `setEndX(double value)` e `setEndY(double value)`. No nosso exemplo, criamos duas linhas que se ligam e formam um “X”, utilizando os métodos citados.

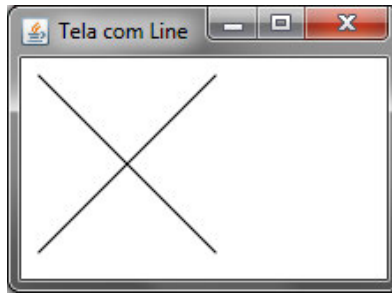


Figura 9.17: Line

Por último, mostraremos o `Rectangle`, o nosso retângulo. Para utilizá-lo, precisamos apenas indicar seu tamanho. No exemplo, citaremos uma cor com efeito gradiente, e também usaremos uma `Thread` para dar um efeito de barra de progresso de uma tela de Splash, por exemplo.

```
private void initRectangle() {  
    final Rectangle rect = new Rectangle();  
    rect.setWidth(10);  
    rect.setHeight(20);  
    rect.setLayoutY(30);  
    rect.setStyle("-fx-fill: linear-gradient(from 0% 0% to 100% 100%, " +  
        "blue 0%, silver 100%);");  
    Thread thread = new Thread() {  
        public void run() {  
            while (rect.getWidth() != 300) {
```

```
Platform.runLater(new Runnable() {
    @Override
    public void run() {
        rect.setWidth(rect.getWidth() + 30);
    }
});
try {
    sleep(1000);
} catch (Exception e) {
    e.printStackTrace();
}
}
};
thread.start();
/* Indicar o Label e Rectangle em um painel principal... */
}
```

Em nossa Thread, a cada segundo que passa (`sleep(1000)`), o retângulo aumenta 30 da sua largura, até chegar a 300 (lembrando-se de utilizar o método `Platform.runLater(Runnable runnable)`, quando for alterar layout em uma Thread). Veremos o resultado do código, quando o tempo chegar aos 5 segundos:

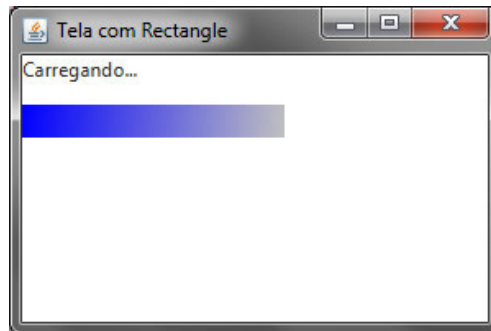


Figura 9.18: Rectangle

CAPÍTULO 10

JavaFX Scene Builder

10.1 CONHECENDO A FERRAMENTA

Os códigos realizados no JavaFX são simples e ajustáveis, tornando a vida do programador mais organizada. Vimos que podemos utilizar arquivos para auxiliar a construção de estilo do layout, como o CSS, e criamos diversos efeitos visuais com códigos simples. Porém, ainda percebemos um problema: os componentes continuam sendo criados à mão, assim como a antiga JFrame. Isto pode tornar o código cansativo e extenso, apenas pela criação de cada componente. Para resolver esta questão, o JavaFX possui um arquivo de criação de layout, chamado FXML. Ele utiliza uma linguagem de marcação semelhante ao XML. Para a geração deste arquivo, utilizamos uma ferramenta da Oracle chamada JavaFX Scene Builder. Esta ferramenta pode ser baixada pelo próprio site da Oracle, no link: <http://www.oracle.com/technetwork/java/javafx/downloads/index.html>. Sua instalação é bem simples, não necessita de configuração. Após a instalação, execute-a e veja a tela principal:

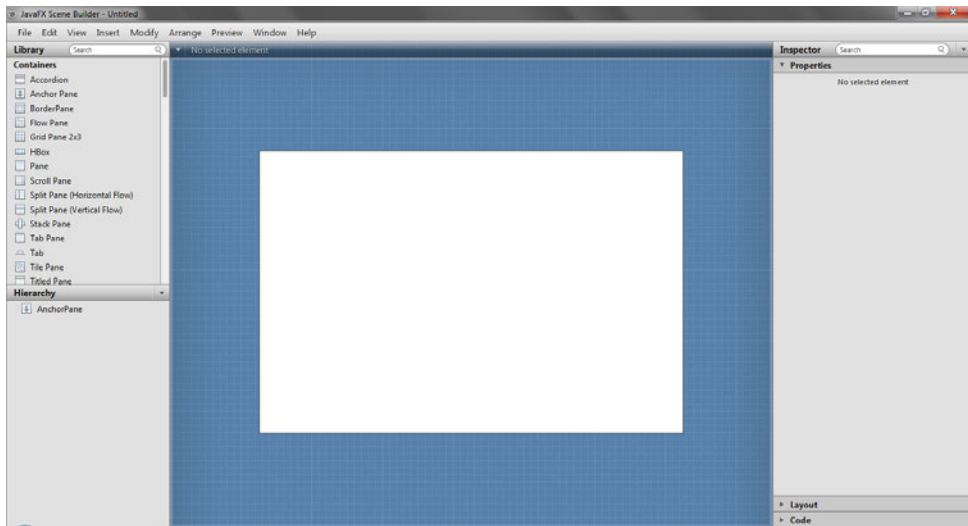


Figura 10.1: Tela Principal do JavaFX Scene Builder

Vamos, agora, conhecer cada área da IDE, para facilitar no entendimento e uso prático da ferramenta.

10.2 LIBRARY PANEL

Aqui, veremos todos os componentes possíveis do JavaFX. Para adicionarmos novos itens à tela, basta arrastarmos o componente para dentro do painel central, onde localiza-se a tela. Há uma divisão dos Nodes por tipo:

- Containers;
- Controls;
- Popup Controls;
- Menu Content;
- Miscellaneous;
- Shapes e
- Charts.

Sua identificação é fácil, pois existe um ícone ilustrativo sobre cada componente, facilitando a sua compreensão. Na Library Panel, há também um campo de texto para filtragem dos componentes. Basta digitar o nome de um componente específico para agilizar a procura.

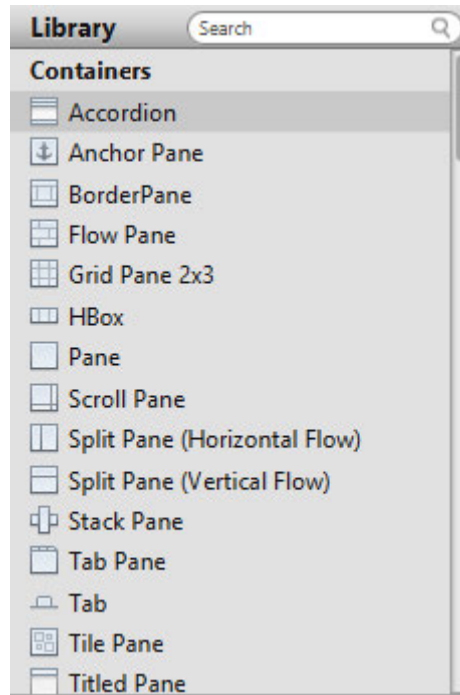


Figura 10.2: Library Panel

10.3 HIERARCHY PANEL

Este painel contém todos os componentes já adicionados à tela. Lembrando que, por padrão, é gerado um `AnchorPane` para representar o painel principal. No painel, é mostrado também um sistema de hierarquia dos Nodes, descrevendo a relação exata entre cada componente. Na figura abaixo, veremos uma tela com um `VBox` com três componentes. Este `VBox` está relacionado ao painel principal, um `AnchorPane`.

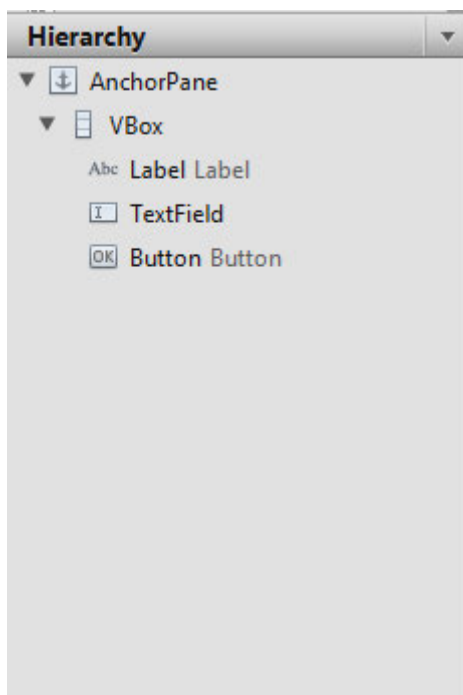


Figura 10.3: Hierarchy Panel

10.4 CONTENT PANEL

Todo o conteúdo da tela se localiza aqui. O desenvolvedor consegue visualizar a tela em *runtime*, ao mesmo tempo da inserção de novos componentes à tela. Este sistema é drag-and-drop, semelhante ao WindowBuilder, do Eclipse para aplicações Swing. Após inserir os componentes, pode-se modificar a posição de cada item livremente, ajustando seu layout da melhor forma possível. Este painel possui algumas facilidades ao desenvolvedor com o menu de contexto. Por exemplo, é possível agrupar componentes em um painel ou box. Basta selecionarmos os componentes que serão agrupados, clicar com o botão direito do mouse, irmos na opção “Wrap in” e escolhermos a opção de agrupamento desejada. Podemos também maximizar o tamanho de um componente específico para o tamanho completo da tela. Para isto, basta selecionarmos o componente desejado, clicar com o botão direito do mouse e clicarmos na opção “Fit to Parent”.

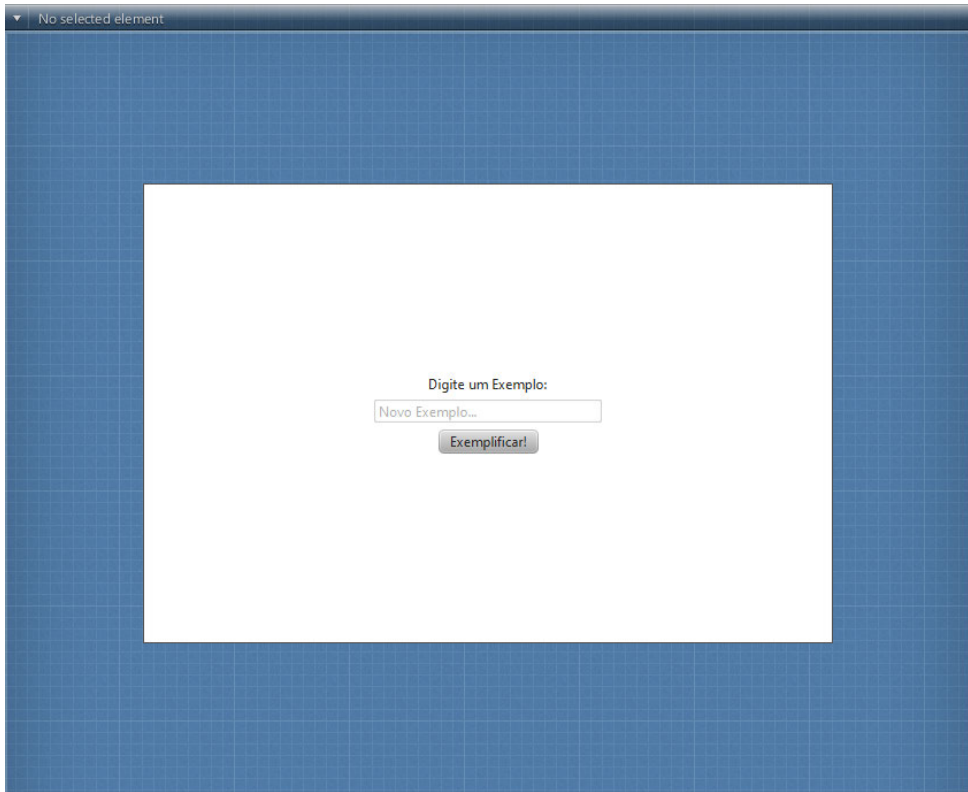


Figura 10.4: Content Panel

10.5 INSPECTOR PANEL

Aqui estão contidas todas as propriedades do componente selecionado. Vemos também uma caixa de texto para filtrar o nome da propriedade procurada. Nesta aba, veremos três

subabas: *Properties*, *Layout* e *Code*.

A subaba *Properties* possui todas as propriedades gerais do componente, como valor de texto, ID, fontes etc. Possui também a parte de implementação de CSS para o determinado componente, além de opções como rotação, translação e escala.

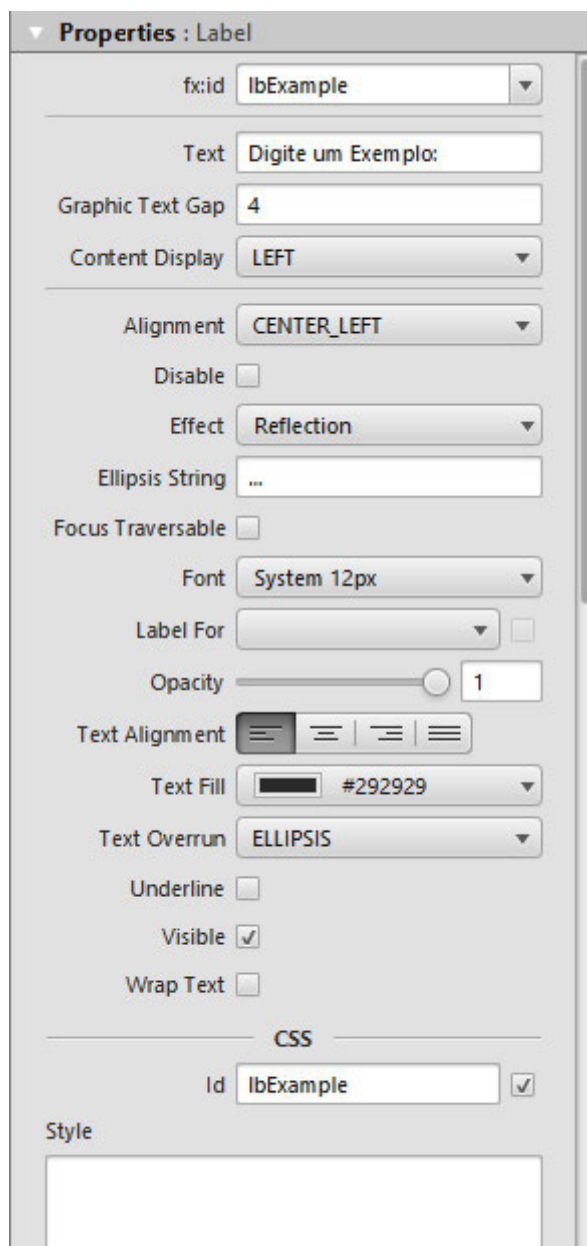


Figura 10.5: Properties

Na subaba *Layout*, veremos opções para modificar a posição, largura e altura do

componente, em sua especificação mínima, preferida e máxima, além de opções de margem e *constraints*, caso o componente esteja inserido em Group's ou Boxes.

▼ **Layout : Label**

VBox Constraints

Vgrow

Margin

TOP	RIGHT	BOTTOM	LEFT
0	0	0	0

Min Width

Min Height

Pref Width

Pref Height

Max Width

Max Height

Resizable ☒

Layout X

Layout Y

Width

Height

Layout Bounds

Bounds In Local

Baseline Offset

Figura 10.6: Layout

Por fim, na subaba *Code* localizam-se todos os *listeners* do componente, como

ação de clique, mouse ou tecla. Esta funcionalidade trabalhará com injeção de dependências, que será vista em prática em breve.

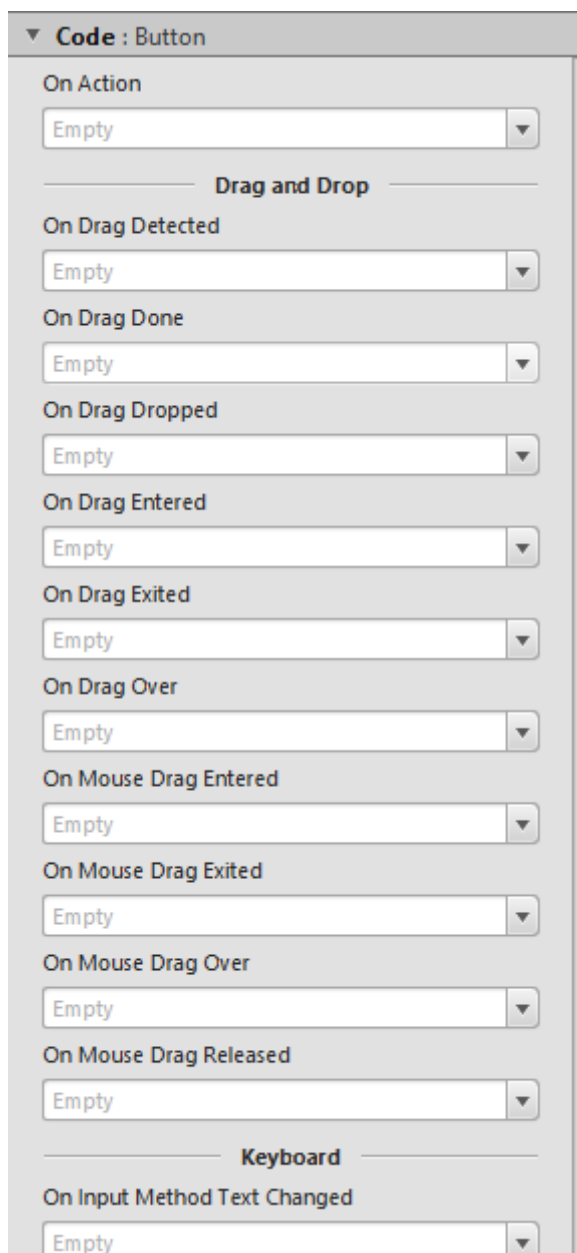


Figura 10.7: Code

10.6 INSERINDO CSS

Vimos que para criar um arquivo .CSS, podemos utilizar o próprio Eclipse, ou até mesmo um simples bloco de notas — o importante é que ele esteja identificado no projeto. Podemos visualizar este CSS também no JavaFX Scene Builder, podendo, assim, ver seu uso antes mesmo de rodar sua aplicação.

Clique na aba “*Preview*”, depois clique em “*Preview a Style Sheet...*”, então procure em seu computador um arquivo CSS criado.

Após isto, vá à subaba *Properties*, da aba *Inspector*, e vá no item *Style Class*. Clique no botão “+” e selecione o efeito correspondente pelo componente selecionado. Você pode também gerar seu código CSS no item *Style*, e digitar o código do efeito do componente referido. Veja na figura abaixo:

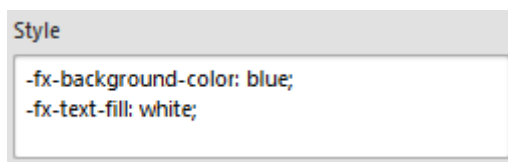


Figura 10.8: Uso do item Style

Para visualizar o resultado obtido até o momento, vá na aba “*Preview*” e clique em “*Preview in Window*”. Você verá sua aplicação sendo executada sem funcionalidades, apenas focando no layout e suas implementações.

Além disso, você pode gerar o esqueleto da classe *Initializable* (seu controller) pela IDE, apenas indo na aba “*View*” e clicando em “*Show Sample Controller Skeleton*”. Abrirá um pequeno bloco de notas interno, contendo o código inicial da sua tela.

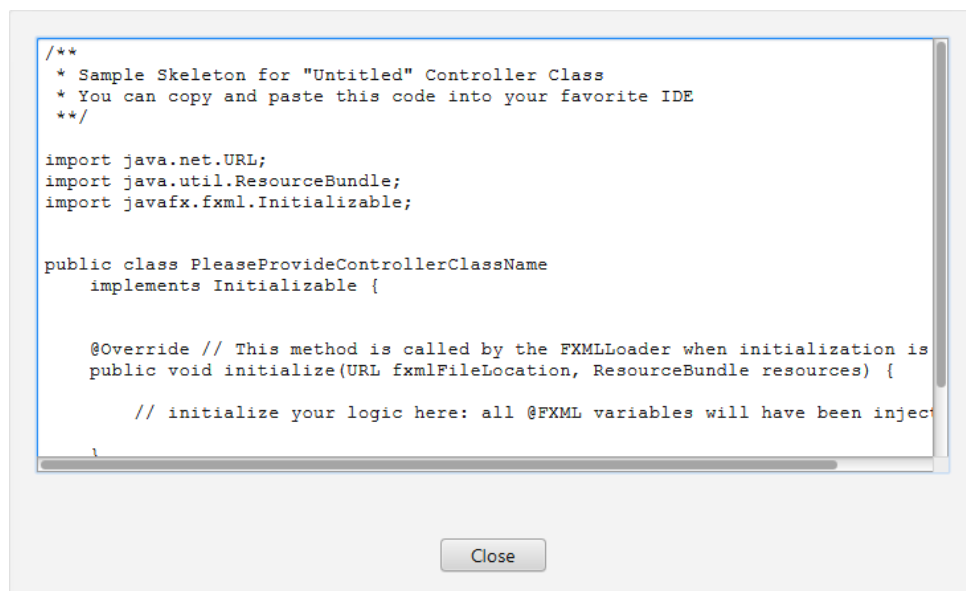


Figura 10.9: Esqueleto de classe Controller

Ao salvar o arquivo, seu formato é FXML semelhante a um XML tradicional. Este arquivo conterá todas as propriedades e gerenciamento da tela, assim como uma classe `Application` faria para criação de cada componente e sua configuração. Ou seja, com a criação do arquivo FXML, a identificação de componentes via código se torna muito mais fácil. Veremos que a estrutura muda: agora faremos uso de duas classes, a `Application`, apenas para criação e gerenciamento da tela (`Stage`) e a `Initializable`, que será responsável pelo gerenciamento dos componentes e seus *listeners*.

10.7 CLASSE APPLICATION

Primeiro, criaremos uma tela pelo JavaFX Scene Builder. No nosso exemplo, faremos novamente uma tela de login simples. Ao terminarmos sua implementação e salvarmos com o nome `login.fxml`, o arquivo ficará assim:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import java.lang.*?>
```

```
<?import java.util.*?>
```

```

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.paint.*?>

<AnchorPane id="AnchorPane" maxHeight="-Infinity" maxWidth="-Infinity"
    minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0"
    prefWidth="600.0" xmlns:fx="http://javafx.com/fxml">
    <children>
        <Label layoutX="200.0" layoutY="145.0" text="Login:" />
        <TextField fx:id="txLogin" layoutX="200.0" layoutY="161.0"
            prefWidth="200.0" />
        <Label layoutX="200.0" layoutY="183.0" text="Senha:" />
        <PasswordField fx:id="txSenha" layoutX="200.0" layoutY="200.0"
            prefWidth="200.0" />
        <Button fx:id="btEntrar" layoutX="272.0" layoutY="262.0"
            mnemonicParsing="false" text="Entrar" />
        <Button fx:id="btSair" layoutX="272.0" layoutY="301.0"
            mnemonicParsing="false" prefWidth="52.0" text="Sair" />
    </children>
</AnchorPane>

```

Então, a classe Application se reduzirá, pois não haverá necessidade de criarmos componentes e/ou listeners aqui. Veremos o seu conteúdo:

```

public class LoginApplication extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Parent parent = FXMLLoader.
            load(getClass().getResource("login.fxml"));
        Scene scene = new Scene(parent);
        stage.setScene(scene);
        stage.setTitle("Tela de Login");
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

A grande diferença é a criação de uma `Parent`, baseada no layout FXML. Para isto, usamos o `FXMLLoader.load(URL url)`, onde passamos como argumento o diretório do arquivo `login.fxml`. Dica: coloque-o na mesma pasta da classe criada. Então, indicamos o `Parent` para uma `Scene`, como se fosse o nosso painel principal. Voltamos a ressaltar: perceba a redução no código, com apenas 15 linhas (fora os *imports*), criamos a inicialização da tela. Porém, ainda falta a segunda parte, sem ela não podemos executar ações nos componentes, porém já é possível executar esta classe e veremos o layout.

10.8 CLASSE INITIALIZABLE

`Initializable` é uma interface do JavaFX para gerenciamento de componentes e suas propriedades e ações. Para identificar a criação de cada componente da tela, apenas devemos criá-las como variáveis globais e “anotá-las” com a `Annotation @FXML`. Esta anotação buscará no arquivo FXML um componente com o ID idêntico ao nome do componente criado na classe. Tal processo é conhecido como “injeção de dependência”, recurso bastante avançado, utilizado em frameworks conceituadas — como o RoboGuice, para instanciar componentes em `Activities`, em aplicações Android, por exemplo.

Então, podemos criar as ações dos componentes dentro do método `initialize(URL url, ResourceBundle bundle)`, implementado da interface `Initializable`. Vejamos o código:

```
public class LoginController implements Initializable {

    @FXML
    private TextField txLogin;
    @FXML
    private PasswordField txSenha;
    @FXML
    private Button btEntrar, btSair;

    @Override
    public void initialize(URL url, ResourceBundle bundle) {
        btEntrar.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                // TODO logar usuário
            }
        })
    }
}
```

```

    });

    btSair.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            // TODO sair da aplicação
        }
    });
}
}
}

```

Ótimo! Agora temos funcionalidades para os botões. Porém, como a aplicação saberá que esta classe Initializable será responsável pelo arquivo login.fxml? Para fazer esta identificação, precisamos abrir o arquivo FXML e identificar na tag do componente principal (painel) o nome do controller, utilizando o `fx:controller`. Veja a implementação:

```

<!-- Códigos acima da AnchorPane... -->

<AnchorPane id="AnchorPane" maxHeight="-Infinity" maxWidth="-Infinity"
    minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0"
    prefWidth="600.0" xmlns:fx="http://javafx.com/fxml"
    fx:controller="LoginController">

<!-- Códigos abaixo da AnchorPane... -->

```

Se o Controller estivesse em outra *package*, precisaríamos passar o caminho completo dela. Exemplo:

```

<AnchorPane id="AnchorPane" maxHeight="-Infinity" maxWidth="-Infinity"
    minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0"
    prefWidth="600.0" xmlns:fx="http://javafx.com/fxml"
    fx:controller="com.casadocodigo.controller.LoginController">

```

E assim vimos a estrutura de criação de layout's com o auxílio da ferramenta JavaFX Scene Builder e do arquivo FXML. Com esta dupla, o código se torna mais legível, simples e organizado, reduzindo número de linhas drasticamente, dependendo da quantidade de componentes de sua tela.

CAPÍTULO 11

Executando tarefas concorrentemente

Trabalhar com Concorrência (*Concurrency*, em inglês) é extremamente importante em aplicações que necessitam de uma resposta com dados em um tempo determinado. Porém, sua complexidade, dependendo da implementação, pode se tornar uma tarefa árdua para o desenvolvedor. Utilizar recursos avançados para este problema é bastante indicado para uma boa performance. Neste capítulo, veremos duas formas de implementação de concorrência: as `Threads` e `Tasks`, e utilizaremos uma `Task` para resposta da confirmação da compra de produtos do nosso sistema.

11.1 INICIANDO POR THREADS

O uso de `Threads` já foi demonstrado no nosso sistema GolFX, na tela de confirmação de compra. Ao clicarmos no botão “Confirmar Compra”, o sistema executa uma `Runnable` através de uma `Thread` por 5 segundos e, então, mostra uma men-

sagem na tela para o usuário: “Compra realizada com sucesso”. O problema do uso da `Thread` é que o próprio JavaFX já roda uma `Thread` especial para execução dos componentes na tela. Então, se rodar uma segunda `Thread`, teremos que nos atentar quando houver uma atualização dos componentes (uma troca de texto de um `TextField`, por exemplo), para utilizarmos a `Runnable` dos componentes, através do código `Platform.runLater(Runnable runnable)`. Veja o exemplo do próprio sistema a seguir:

```
Thread thread = new Thread() {
    public void run() {
        try {
            sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        JOptionPane.showMessageDialog(null,
            "Compra realizada com sucesso!");
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                CarrinhoApp.getStage().close();
                ItemApp.getStage().close();
            }
        });
    }
};
thread.start();
```

A sintaxe da `Thread` é simples: apenas inicializamos uma variável do tipo `Thread`, utilizando `{}` para sobrescrever o método `run()`, onde acontecerá a lógica. O método `sleep(long millis)` identifica por quantos milissegundos o sistema ficará inativo, para passar para a próxima etapa, que é a caixa de mensagem `JOptionPane`. Caso haja uma interrupção durante a inatividade, o sistema executa uma `InterruptedException`. Logo em seguida, há uma abertura de tela e fechamento de outra tela. Como isto pode afetar a `Thread` principal do JavaFX, precisamos fazê-lo utilizando o `Platform.runLater`.

```
Platform.runLater(new Runnable() {
    @Override
    public void run() {
```



```
        CarrinhoApp.getStage().close();
        ItemApp.getStage().close();
    }
});
```

`Threads` são bastante utilizadas em cálculos curtos, nos quais a lógica implementada seja pequena. Caso queira utilizar algo mais robusto, utilizamos as `Tasks`, que veremos a seguir.

11.2 CONHECENDO TASKS

`Task` é uma classe do JavaFX cujo objetivo é implementar tarefas (normalmente com longos cálculos) em um determinado tempo. `Task` é estendida da classe `FutureTask`, do antigo `java.util.concurrent`. Esta classe permite ao desenvolvedor cancelar a `Thread`, através do método `cancel(boolean value)`. A `Task` também é implementada da interface `Worker`, também do JavaFX. Esta interface realiza alguma lógica em uma ou mais `Threads`, e é possível observar seu estado na aplicação, podendo ser visualizada e usada na `Thread` principal do JavaFX. O uso de `Tasks` também é simples: a primeira coisa a se fazer é identificar a parametrização da `Task`, na qual indicamos que tipo de dado será retornado ao fim do processo de *loop*. Caso não queira retornar nenhum dado, utilize `Void`, com retorno nulo no método `call()`. A propósito, este método deve ser sobrescrito no uso da `Task`, é lá que constará a lógica (lembrando do método `run()` da `Thread`). Veremos um exemplo de um acumulador para calcular o valor de 2 elevado a 50.

```
Task<Long> potentiationTask = new Task<Long>() {
    @Override
    protected Long call() throws Exception {
        long result = 1;
        for (int i = 1 ; i <= 50 ; i++) {
            result *= 2;
        }
        return result;
    }
}
```

Ou podemos simplesmente executar uma lógica sem retorno, como será o nosso caso do sistema. Como exemplo, mostraremos apenas uma `JOptionPane`, após 10 segundos de execução da `Thread`.

```
Task<Void> resultTask = new Task<Void>() {
    @Override
    protected Void call() throws Exception {
        try {
            Thread.sleep(10000); // 10 segundos
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        JOptionPane.showMessageDialog(null, "Lógica realizada com sucesso!");
        return null;
    }
}
new Thread(resultTask).start();
```

Percebemos, aqui, que podemos utilizar o método `sleep(long millis)`, que é estático da classe `Thread`, para a parada de 10 segundos.

Agora, conhecendo mais sobre as `Tasks`, vamos implementá-las no projeto.

11.3 IMPLEMENTANDO TASK NO SISTEMA

Faremos, agora, o mesmo efeito realizado anteriormente pela `Thread`, mas utilizando a classe `Task`, do JavaFX. A lógica é tão simples quanto a da `Thread`. Como não há necessidade de retorno de dados, utilizaremos `Void` e `return null`.

```
Task<Void> task = new Task<Void>() {
    @Override
    protected Void call() throws Exception {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        JOptionPane.showMessageDialog(null,
            "Compra realizada com sucesso!");
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                CarrinhoApp.getStage().close();
                ItemApp.getStage().close();
            }
        });
    }
};
```

```

        return null;
    }
}
new Thread(task).start();

```

Vendo o exemplo da `Task<Void>` da seção anterior, fica fácil implementar no nosso sistema. Lembrando que o uso de `Platform.runLater(Runnable runnable)` é fundamental. Porém, não haveria sentido mudar a estrutura de código e não obter melhorias ou outras formas de lógica. Então, vamos adicionar uma coisa interessante: que tal mostrar para o usuário, de alguma forma, durante o *sleep*, que o programa “ainda está vivo”? Podemos pegar o próprio texto do botão “Confirmar Compra” para mostrar ao usuário algo como “aguarde...”. Para isto, usaremos o método `updateMessage(String msg)` para atualizar o texto. Vejamos a implementação:

```

Task<Void> task = new Task<Void>() {
    @Override
    protected Void call() throws Exception {
        updateMessage("Aguarde...");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        updateMessage("Confirmar Compra");
        JOptionPane.showMessageDialog(null,
            "Compra realizada com sucesso!");
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                CarrinhoApp.getStage().close();
                ItemApp.getStage().close();
            }
        });
        return null;
    }
}
new Thread(task).start();

```

Usamos o primeiro `updateMessage` para indicar que o programa está aguardando o *sleep* ser concluído para continuar o processo. O segundo `updateMessage`

volta o texto antigo do botão “Confirmar Compra”, executando a lógica subsequente. Agora, precisamos indicar esta mudança de texto ao botão referido. Utilizaremos o `Observer` do texto do botão, indicando a mudança com o `Observer` da `Task`, através do método `bind`.

```
btConfirmarCompra.textProperty().bind(task.messageProperty());
```

Adicione esta linha acima antes da execução da `Thread` com a `Task`. Dessa forma, a implementação completa da `Task` fica assim:

```
Task<Void> task = new Task<Void>() {
    @Override
    protected Void call() throws Exception {
        updateMessage("Um momento...");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        updateMessage("Confirmar Compra");
        JOptionPane.showMessageDialog(null,
            "Compra realizada com sucesso!");
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                CarrinhoApp.getStage().close();
                ItemApp.getStage().close();
            }
        });
        return null;
    }
};
btConfirmarCompra.textProperty().bind(task.messageProperty());
new Thread(task).start();
```

Execute o sistema, adicione alguns produtos ao carrinho de compras e confirme a compra para ver a `Task` em execução. Perceba a mudança do texto do botão “Confirmar Compra”.

MAIS MÉTODOS DA `Task`:

Além do método `updateMessage` para atualização de mensagens, a `Task` contém os seguintes métodos:

- `updateTitle(String title)`: atualiza o título de uma tela, por exemplo;
- `updateProgress(double min, double max)`: atualiza o progresso da `Thread`. Muito utilizado em `ProgressBar` e `ProgressIndicator`.

CAPÍTULO 12

Utilizando WebServices de forma simples

Diversas aplicações, tanto empresariais quanto pessoais, necessitam de dados e recursos disponibilizados em um ponto específico, seja em um disco rígido, um banco de dados, ou talvez em recursos em *cloud*. Para que estes dados sejam retornados ao usuário, precisamos trabalhar com serviços web, conhecidos como *WebServices*, para isto o trabalho com recursos físicos e lógicos devem ser coerentes e sincronizados. Neste capítulo, utilizaremos este conceito para retornar dados via web para a nossa aplicação GolFX, mostrando a capacidade do JavaFX em relação à integração de aplicações corporativas. Através dessas comunicações, podemos criar um canal de dados entre diversas aplicações de forma simples. Mais precisamente, iremos criar uma área onde serão disponibilizados todos os *Tweets* de pessoas que utilizarem a *Hashtag* #GolFX, para identificarmos quem e quantos usuários estão falando sobre nosso projeto, em tempo real. Vamos começar entendendo alguns conceitos de *WebServices*.

12.1 CHAMANDO SERVIÇOS REMOTOS VIA WEB

Como citado anteriormente, as aplicações Desktop podem receber informações vindas da web, através de *WebServices* (ou serviços remotos via web). Estes serviços podem ser lidos e utilizados independente da linguagem (seja Java ou não) e independente da plataforma utilizada (Desktop, Web ou Mobile). Existem diversas especificações que tratam exclusivamente sobre estes serviços, no nosso caso, citaremos os dois principais conceitos: o SOAP e o REST.

O SOAP — *Simple Object Access Protocol* (em português, Protocolo Simples de Acesso a Objetos) — foi inventado pela Microsoft em 1998, e passou a ser utilizado como o formato de intercâmbio entre aplicações Java e aplicações .Net. Este protocolo é baseado em XML, porém com um nível de detalhamento muito alto, tornando seu uso complexo. O Java possui várias ferramentas para a interpretação do SOAP.

E o REST — *Representational State Transfer* (em português, Transferência de Estado Representativo) — foi criado para suprir a necessidade do uso de serviços simples e específicos. Este protocolo trabalha com XML, JSON, Atom e diversos outros formatos, de forma flexível.

12.2 TWITTER API E TWITTER4J

Para o nosso projeto utilizaremos uma combinação de ferramentas para criarmos um canal de informações do Twitter: o Twitter API e a framework Twitter4j.

O Twitter API foi criado para que os desenvolvedores de diversas linguagens possam receber dados do Twitter para suas aplicações, além de criar post's. Com esta API, é possível criar botões de Tweets, ver a *Timeline* e os *Tweets* dos usuários, entre outros recursos. Pode-se ser utilizado tanto em Desktop, Web e Mobile, e seus dados são retornados através de JSON, um formato de fácil entendimento e interpretação. Para utilizarmos este serviço web, utilizando Java, precisamos de uma framework, e para isto, usaremos o Twitter4j.

O Twitter4j trabalha com a integração de aplicações Java com o serviço Twitter. É possível utilizá-la para Android, suporta autenticação (OAuth) e é compatível com a nova versão do Twitter API: Versão 1.1. Para baixar, vá até o site do Twitter4j: <http://twitter4j.org/en/index.html>, na parte de Download, e baixe a última versão (*Latest stable version*). Porém, é possível pegá-la juntamente com o projeto completo, no Github (pasta *libs*).

Caso faça o download, é necessário colocá-la no projeto. Apenas crie uma pasta chamada *libs* dentro do projeto e cole o arquivo *core* do Twitter4j (*lib/twitter4j-core*

3.0.3.jar, caso baixe a versão 3.0.3, a mais atual até o momento) para esta pasta. Então, volte para o Eclipse, clique com o botão direito do mouse sobre este jar, escolha a opção “Build Path” e clique em “Add to Build Path”.

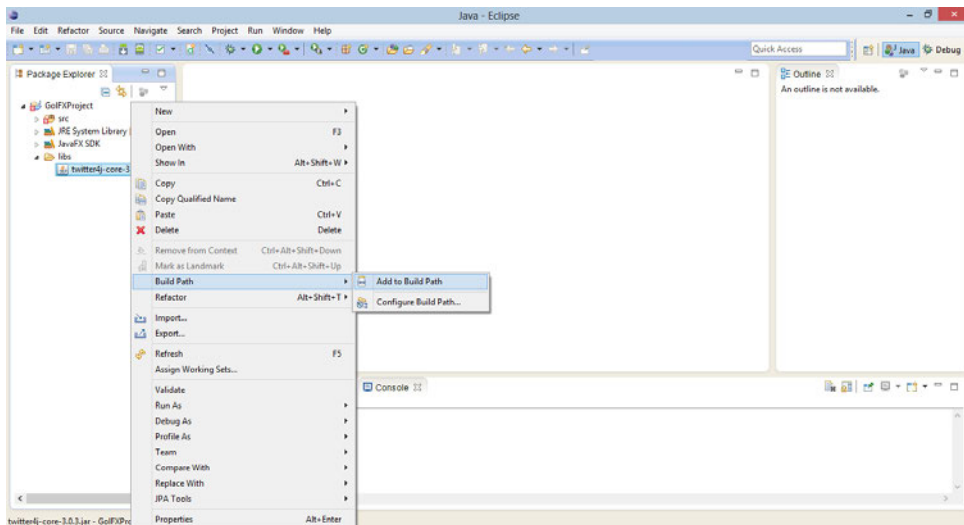


Figura 12.1: Adicionando Twitter4j ao Projeto

Agora, já podemos utilizar os códigos da framework em nosso projeto. Porém, precisamos, ainda, autenticar nossa conta no Twitter e a aplicação para podermos utilizar os dados do Twitter no nosso projeto.

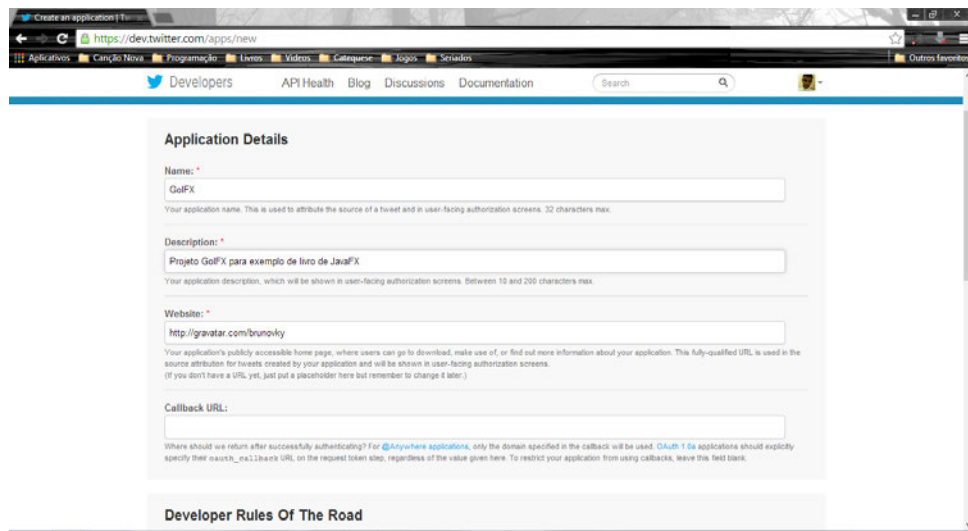
12.3 AUTENTICANDO USUÁRIO NO TWITTER DEVELOPERS

A primeira coisa a se fazer é criar uma conta no Twitter, caso ainda não tenha. Para isto, acesse o site <https://twitter.com/> e cadastre-se gratuitamente. Caso já tenha uma conta no Twitter, pule esta etapa.

Após se inscrever, iremos autenticar nossa aplicação no site do Twitter Developers. Acesse <https://dev.twitter.com/>, logue sua conta, clicando em *Sign in*.

Ao fazer o login, clique em seu avatar no canto direito da tela, e clique em “My applications”. Agora, você deve criar uma nova aplicação para identificação de sua autenticação. Clique em “Create a new application”. Preencha os campos corretamente, pode ser apenas os com um asterisco vermelho, significando que este campo

é obrigatório. No caso, preencha o nome do projeto (*Name*), a descrição dele (*Description*) e um site de seu projeto (*Website*). Caso não tenha um website para o projeto, coloque algum link pessoal para identificação (pode ser seu perfil do Facebook, ou seu avatar no Gravatar, por exemplo). O campo *Callback URL* é desnecessário para nós, no momento. Confirme as regras de uso e digite o *captcha* antes de confirmar a criação da aplicação, clicando em “*Create your Twitter application*”. Veja um exemplo de preenchimento dos campos abaixo:



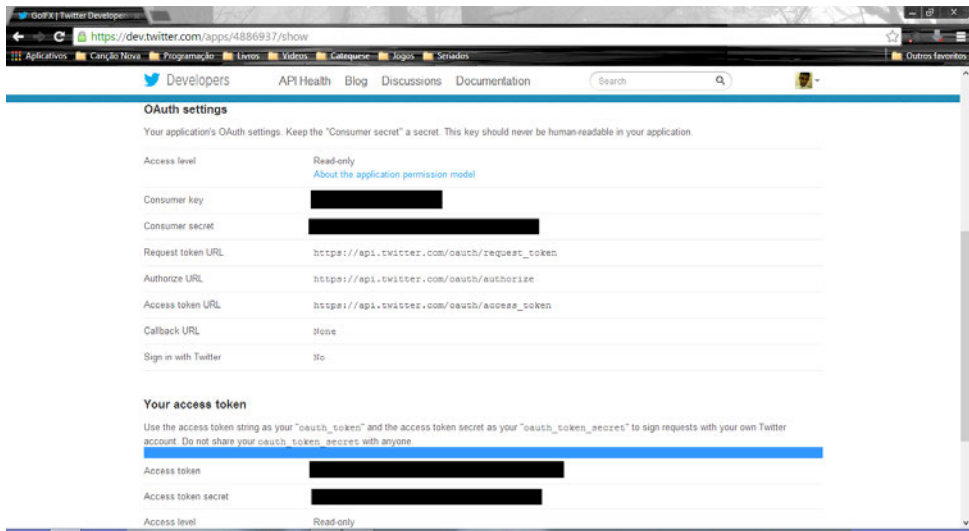
The screenshot shows a web browser window with the URL <https://dev.twitter.com/apps/new>. The page is titled "Create an application | Twitter" and has a navigation bar with links: Developers, API Health, Blog, Discussions, Documentation, and a search bar. The main content area is titled "Application Details" and contains the following form fields:

- Name:** * (Required) Input field containing "GoFX". Below it, a note says: "Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max."
- Description:** * (Required) Input field containing "Projeto GoFX para exemplo de livro de JavaFX". Below it, a note says: "Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max."
- Website:** * (Required) Input field containing "http://gravatar.com/brunoviky". Below it, a note says: "Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens. (If you don't have a URL, yes, just put a placeholder here but remember to change it later.)"
- Callback URL:** Input field (empty). Below it, a note says: "Where should we return after successfully authenticating? For @Anywhere applications, only the domain specified in the callback will be used. OAuth 1.0a applications should explicitly specify their oauth_redirect URL, on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank."

Below the form fields is a section titled "Developer Rules Of The Road".

Figura 12.2: Autenticando a aplicação no Twitter Developers

Após a criação, você verá uma página com todos os dados pertinentes à autenticação do projeto. Atente-se para os campos “*Consumer key*” e “*Consumer secret*”, serão necessários para a identificação pelo Twitter4j. Então, clique em “*Create my access token*”, mais abaixo. Após este passo, atualize a página para abrir seu *access token*, também necessário para o uso do serviço Twitter. Sua página deverá ficar mais ou menos assim:

Figura 12.3: Visualizando seu *access token*

Agora estamos prontos para utilizar o serviço Twitter com o Twitter4j para nossa aplicação!

12.4 CRIANDO UMA LISTA DE TWEETS

Implementaremos uma lista com os últimos Tweets de usuários, que contenham as palavras “GoFX” ou “JavaFX”. Primeiramente, esta lista iniciará invisível, no meio da tela. Ela surgirá no momento do clique de um `Hyperlink` no canto esquerdo da tela, no lado oposto do campo de texto de pesquisa de produtos. Então, criaremos mais dois componentes na classe `VitrineApp`: um `Hyperlink` e um `ListView`, recebendo como parametrização uma `String`, que será o conteúdo do Tweet (`Data + Usuário + Texto`).

```
private Hyperlink linkTwitter;  
private ListView<String> lvTweets;
```

Agora, iremos no método `initComponents()` para inicializarmos os componentes, aproveitando para colocar a nossa lista de Tweets invisível.

```
private void initComponents() {  
    // Demais códigos...
```

```

linkTwitter = new Hyperlink("Clique aqui para ver nossos tweets...");
lvTweets = new ListView<String>();
lvTweets.setVisible(false); // Tornando a lista invisível
pane.getChildren().addAll(txPesquisa, tbVitrine, linkTwitter, lvTweets);
}

```

No método `initLayout()`, indicamos a posição dos dois componentes: o `Hyperlink` ficará no canto superior esquerdo da tela e o `ListView` ficará no centro da tela, com um tamanho fixo de 500.

```

private void initLayout() {
    // Demais códigos...
    linkTwitter.setLayoutX(10);
    linkTwitter.setLayoutY(10);
    lvTweets.setLayoutX(150);
    lvTweets.setLayoutY(100);
    lvTweets.setMinWidth(500);
}

```

Agora, usaremos efetivamente a `Twitter4j`. Não explicaremos profundamente seu uso, para isto, acesse o site da framework, que possui uma boa documentação e diversos exemplos de código: <http://twitter4j.org/en/code-examples.html>. O método retornará uma `ObservableList<String>`, que será a nossa lista de Tweets, juntando os três dados indicados anteriormente (Data + Usuário + Texto). Comece o método inicializando uma `ObservableList<String>` vazia, e retornando ela mesma.

```

private ObservableList<String> getTweets() {
    ObservableList<String> tweets = FXCollections.observableArrayList();
    // Demais códigos...
    return tweets;
}

```

O método utiliza, primeiramente, a classe `Twitter`, inicializando através da *factory* `TwitterFactory`, e indicando os valores responsáveis pela autenticação (OAuth).

```

Twitter twitter = TwitterFactory.getSingleton();
twitter.setOAuthConsumer("meu_consumer_key",
    "meu_consumer_secret");
twitter.setOAuthAccessToken(new AccessToken(

```

```
"meu_access_token",
"meu_access_token_secret"));
```

Então, utilizaremos a classe `Query` para indicar qual a busca de Tweet será realizada. No nosso caso, serão duas buscas, para isto utilize a palavra “OR” entre as buscas.

```
Query query = new Query("source: golfx OR javafx");
```

IMPORTANTE: A palavra “source:” é necessária para a busca.

Por fim, faremos um loop para a busca, já que utilizamos duas *queries*. Utilizaremos a classe `QueryResult` para o retorno da busca, e a classe `Status` para indicar cada Tweet, através de um comando `for`, iterando na lista de Tweets. Utilizamos o comando `getTweets()`, da `QueryResult`, adicionando alguns dados do status para a nossa `ObservableList tweets`.

```
QueryResult result;
do {
    result = twitter.search(query);
    for (Status status : result.getTweets()) {
        tweets.add(status.getCreatedAt() + ":"
            + status.getUser().getScreenName() + ":"
            + status.getText());
    }
} while ((query = result.nextQuery()) != null);
```

O método completo é visualizado a seguir:

```
private ObservableList<String> getTweets() throws TwitterException {
    ObservableList<String> tweets = FXCollections.observableArrayList();
    Twitter twitter = TwitterFactory.getSingleton();
    twitter.setOAuthConsumer("WXjCt3vJi67ytqaVx9AaEQ",
        "AL3KgZaRy2fyWxQxV7IQUG0oYkVVZv2JgAdmRQQ");
    twitter.setOAuthAccessToken(new AccessToken(
        "574580115-DRIX3iCxu24FsQrsa8JiyEHKK3hCrwIt1gbb65HK",
        "ZGrcG1v2uVDqmlZSF4Hy8pmMEJrVyls7zYX5xcuBIM"));
    Query query = new Query("source: golfx OR javafx");
    QueryResult result;
    do {
        result = twitter.search(query);
        for (Status status : result.getTweets()) {
```

```

        tweets.add(status.getCreatedAt() + ":"
            + status.getUser().getScreenName() + ":"
            + status.getText());
    }
    } while ((query = result.nextQuery()) != null);
    return tweets;
}

```

Para finalizarmos, no método `initListeners()`, criaremos a ação do clique do `Hyperlink`. A ação tornará visível a lista de Tweets, e desabilitará o campo de texto de pesquisa e a tabela de produtos, para que não haja cliques, atrapalhando a lista. Ao mesmo tempo, se a lista já estiver visível, ao clicar no link, voltará a tornar invisível e habilitará os dois outros componentes. Caso a lista esteja visível, trocaremos os itens da lista, para atualizarmos sempre os Tweets.

```

private void initListeners() {
    // Demais códigos...
    linkTwitter.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent evt) {
            lvTweets.setVisible(!lvTweets.isVisible());
            txPesquisa.setDisable(!txPesquisa.isDisabled());
            tbVitrine.setDisable(!tbVitrine.isDisabled());
            if (lvTweets.isVisible()) {
                try {
                    lvTweets.setItems(getTweets());
                } catch (TwitterException e) {
                    e.printStackTrace();
                }
            }
        }
    });
}

```

Execute o projeto e veja o uso do *WebServices* do Twitter.

Referências Bibliográficas

- [1] Java-buddy blogspot. <http://java-buddy.blogspot.com.br/>.
- [2] Carl Dea. Javafx 2.0: Introduction by example. 2011.