



# JavaFX Prático

*William Antônio Siqueira*

---

# Tabela de conteúdos

Capa	1.1
Prefácio	1.2
Apresentação	1.3
JavaFX Prático	1.4
Um "Olá Mundo" com JavaFX	1.4.1
Mostrando Imagens e Figuras Geométricas	1.4.2
Tocando Áudio	1.4.3
Tocando Vídeo	1.4.4
Controles de interface: Rótulos, campos de texto e outros	1.4.5
Controles de interface: Tratamento de eventos e o botão	1.4.6
Controles de Interface: Radio Button, CheckBox e ToggleButton	1.4.7
Controles de Interface: ComboBox e ChoiceBox	1.4.8
Gerenciando Layout: HBox, VBox e StackPane	1.4.9
Gerenciando Layout: BorderPane, FlowPane e o GridPane	1.4.10
Transições	1.4.11
Desenhando com Canvas	1.4.12
Gráficos	1.4.13
Abrindo páginas Web	1.4.14
Tabelas com JavaFX	1.4.15
Mudando o estilo com CSS	1.4.16
Usando FXML	1.4.17
Projeto: Um simples CRUD	1.4.18
Conclusão	1.5

# JavaFX Prático

Leitura prática da API básica do JavaFX

*William Antônio Siqueira* <[william.a.siqueira@gmail.com](mailto:william.a.siqueira@gmail.com)>

# Prefácio

Ninguém é uma ilha durante o aprendizado no mundo da programação e, talvez, em qualquer processo de aprendizado, pois nosso maior diferencial sobre as outras espécies é aprender e passar pra frente o aprendizado. O mundo conectado é maravilhoso: posso parar o que estou fazendo agora e ter algumas aulas de neurociência no youtube, ou biologia e até aprender a fazer as unhas, por que não?

Nesse processo incrível nos esquecemos do principal, que são os que criam conteúdo. Tenho isso em meu coração desde o tempo do ensino médio ou quando aprendia eletrônica com revistinhas *\_Divirta-se com a Eletrônica\_*, do mestre Bêda Marques. Claro que o caos no mundo liberal que vivemos garante que, por motivações diversas, as pessoas poderão criar conteúdos e serem beneficiadas por isso, como ganhar dinheiro através de *\_adsense\_* do Google ou com *\_marketing\_*. *\_Mas há algo mais que move os criadores de conteúdo.*

Esse algo mais é uma força incrível, que nos remete a ideia de um mundo melhor através da disseminação do conhecimento. Sim, ainda existem coisas como essas no mundo líquido que vivemos. Eu acredito ter isso comigo, pois joguei minha carreira promissora como **ABAP** para trabalhar com *\_open source\_*. *\_Mas não há nada em especial nisso, todos temos isso e só precisamos nos manter firmes e exercitar essa ideia em tempos complicados para o homem romântico com suas ideias destruídas pelo século do ego.* Muitos que criaram conteúdos que permitiram a **sua** existência tinham isso forte com eles e hoje podemos contribuir de forma pequena, como esse livro, mas juntos é que crescemos. Quando se um dia houver o alinhamento desse pensamento, então inicia a história do homem.

Mas voltando a revistinha *Divirta-se com a Eletrônica*. Nessa revista não tínhamos horas e horas de cálculo para então ver algo concreto, nós, os hobbystas, como éramos chamados pelos escritores da revista, criávamos circuitos divertidos e rápidos e com o tempo íamos ganhando conhecimento teórico. Essa é a ideia desse livro. Com certeza, eu não acertei o ponto, mas pensei em escrever com editoras, com regras e tudo mais, mas, pra ser sincero, assim funcionou pra mim e deve funcionar para um punhado de pessoas, por isso escolhi essa plataforma gitbooks e organizei tudo aqui.

Ao ler esse livro eu espero que: 1) Você se divirta; 2) Aprenda algo que vai lhe ser útil; 3) Possa compartilhar as coisas que aprendeu aqui com outras pessoas .

# JavaFX Prático

Um livro rápido, direto, sem muita enrolação. Objetivo é introduzir as classes mais básicas da plataforma JavaFX com evolução natural dos conceitos sem que haja buracos no aprendizado dos componentes básicos da plataforma. É necessário somente o conhecimento básico de Java e o uso de uma IDE que suporte Maven.

Notem que esse livro é baseado no blog [Aprendendo JavaFX](#), mas revisado, editado e formatado para ficar no formato de um livro!

## Software Necessário

Utilizando JavaFX 8 e Eclipse. O download pode ser realizado nos seguintes links:

- [Java JDK 8](#)
- [Eclipse IDE for Java Developers\(para esse livro foi usada a versão "Mars"\)](#)

## Como ler esse livro

O livro não está organizando em capítulos. Pelo contrário, vamos em uma ordem linear, começando com conceitos básicos, até aplicações um pouco mais complexas.

Inicialmente teremos explicações mais extensas e detalhadas, no decorrer do livro, sai o texto, entram explicações breves e mais código.

Espero que o leitor seja curioso e pró-ativo: Abra os links e leia a documentação; Edite o código; crie coisas legais para você com os exemplos que encontrar.

## Código

Há exemplos de código em todos os capítulos e estão em um projeto Maven que poderá ser baixado em <http://aprendendo-javafx.blogspot.com>.

Em um anexo você encontra **javafx-pratico.zip** e nesse arquivo há todo o código organizado e testado. Cabe a você explorar.

Você pode abrir o eclipse e importar o projeto usando os seguintes passos:

- *Deszipe* o arquivo **javafx-pratico.zip** em algum lugar no seu computador;

- No Eclipse você deve usar o menu **File -> Import -> Existing Maven Projects**
- Então você aponta para o diretório onde está o arquivo pom.xml, que você *deszipou*;
- É só um projeto com todo o código mencionado nesse livro. Você pode rodar as classes que tiverem um método *main\_usando* **Run As -> Java Application\_**

# O que é JavaFX?

Calma, em alguns minutos você vai estar se divertindo com JavaFX, passeando por esse fascinante *toolkit* gráfico!

## ***Mas o que é JavaFX?***

JavaFX é a nova biblioteca gráfica da plataforma Java.

Mas o que você pode fazer com JavaFX?

- Criar interfaces gráficas
- Animações
- Desenhar na tela
- Efeitos
- Gráficos
- Programar arrastando e soltando
- Tocar vídeo e áudio
- Jogos
- Muitas, muitas outras coisas

As aplicações em JavaFX rodam no computador localmente, não em um navegador WEB. No entanto, há projetos maduros da comunidade que permitem portar o seu código para celulares ou para rodar dentro de navegadores.

# Um "Olá Mundo" com JavaFX

Nesse primeiro artigo vamos mostrar um passo a passo de como executar sua primeira aplicação usando JavaFX! A versão usada é a que vem no Java 8.

*Antes de prosseguir, confira se você instalou o JDK 8 da Oracle corretamente na sua máquina e também o Eclipse!*

Ótimo! Já temos tudo o que precisamos. O próximo passo agora é abrir o Eclipse e criar um projeto do tipo **Maven Project**. Nesse projeto iremos criar um pacote e nele uma classe Java. Veja abaixo uma explicação mais detalhada (nomes em negrito e itálico entre parêntesis foram os que eu usei):

1. Acesse o menu **File -> New -> Maven Project**, marque a opção "Create a simple project (skip archetype selection)", preencha os valores para **groupId(org.javafxpratico)**, **artifact-id(javafx-pratico)**, **version(1.0)** e clique em **Finish**;
2. Clique com o botão direito sobre o seu projeto e acesse o menu **New -> Package**, dê um nome (***javafxpratico***) e em seguida clique em **Finish**;
3. Clique com o botão direito no pacote main e acesse o menu **New -> Class**. Dê um nome para a classe (eu usei ***OlaMundoJavaFX***) e clique em **Finish**.

Pronto! É nesse mesmo projeto que iremos explorar o JavaFX. O código da classe `OlaMundoJavaFX` deve se parecer com o seguinte:



```
package javafxpratico;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class OlaMundoJavaFX extends Application { // 1

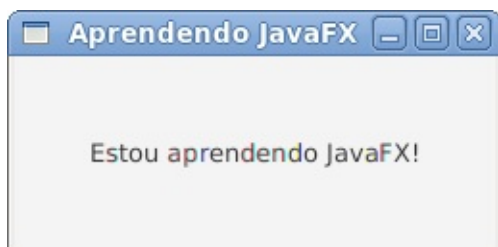
    public static void main(String[] args) {
        launch(); // 2
    }

    @Override
    public void start(Stage palco) throws Exception { // 3
        StackPane raiz = new StackPane(); // 4
        Label lblMensagem = new Label(); // 5

        lblMensagem.setText("Estou aprendendo JavaFX!"); // 6
        raiz.getChildren().add(lblMensagem); // 7

        Scene cena = new Scene(raiz, 250, 100); // 8
        palco.setTitle("Aprendendo JavaFX"); // 9
        palco.setScene(cena); // 10
        palco.show(); // 11
    }
}
```

Essa é uma das aplicações mais básicas que você pode fazer com JavaFX! Você pode executá-la clicando em **Run** ou clicando com o botão direito em cima dela e escolher no menu o seguinte: **Run As -> Java Application**. Se fizer tudo certinho a seguinte janela irá aparecer:



Ótimo! Sua primeira aplicação. Você deve ter notado que o código acima está com comentários numéricos. Isso foi proposital, pois aqui vai a explicação desse código aí! Preste bastante atenção pois esse mesmo código é usado em todas as aplicações JavaFX!

1. Perceba que a classe principal herda de **javafx.application.Application**. **TODA** classe principal de JavaFX deve herdar de Application e implementar o método start;
2. No método *main* chamamos o método *launch* para começar a nossa aplicação. Aqui não

vai código JavaFX, o código vai no método `start` (notem que no JavaFX 8 isso é mais necessário, abaixo mais detalhes);

3. A implementação do método `start`, herdado da classe `Application`. O atributo recebido é do tipo **`javafx.stage.Stage`**. Sendo direto, podemos ver o `Stage` (palco) como o frame, a janela da nossa aplicação, mas na verdade ele não pode ser representado sem se pensarmos nos diversos dispositivos que podem rodar(em um futuro próximo) JavaFX: Celulares, televisores, "tablets", etc;
4. Nesse ponto nós criamos um elemento chamado "pai", pois permite adicionarmos outras coisas dentro dele. No nosso caso, o **`javafx.scene.layout.StackPane`** permite adicionar vários elementos os quais tem seu leiaute de pilha, ou seja, eles serão empilhados um sobre o outro. No futuro falaremos mais sobre isso, mas lembre-se que tudo no JavaFX é um nó, ou seja, herda da classe **`Node`**;
5. Não há nada de mais aqui, simplesmente criamos um objeto do tipo **`javafx.scene.control.Label`**, que é um controle de interface para mostrar texto. Ponto;
6. Aqui informamos o texto que o `Label` irá mostrar. Note que isso poderia ter sido feito pelo construtor, na criação do `Label`;
7. Como o `StackPane` é um elemento pai, ele também tem elementos filhos. Nessa linha de código, recuperamos os filhos dele(`getChildren()`) e adicionamos nosso `Label`(`add(Node)`), fazendo que o `Label` seja um filho dele;
8. É hora de aprender outro conceito do JavaFX. Nessa linha criamos uma **`javafx.scene.Scene`**(cena). Uma cena é o contêiner principal de todos os elementos do JavaFX e na criação dela aproveitamos para informar a raiz (como o nome diz, a raiz de todos os componentes), largura e altura da cena;
9. Agora vamos voltar a mexer com nosso palco. Nessa linha informamos o título dele, no nosso caso atual, o título da janela que será mostrada;
10. O palco precisa de uma cena, simplesmente é isso que é feito nessa linha.
11. Simplesmente mostrando o palco! Se esse método não for chamado, nada irá acontecer quando executar esse código.

Pronto! Muita coisa nova, né? Mas não desista, no decorrer dos artigos você vai ficando cada vez mais confortável com JavaFX e seus componentes!

# Mostrando Imagens e Figuras Geométricas

## Mostrando Imagens

Um dos pontos mais interessantes do JavaFX é como ele facilitou a criação de aplicações, assim podemos fazer algumas coisas antes difíceis da forma mais fácil possível.

Um exemplo é carregar imagens na aplicação. Temos somente dois passos a serem seguidos nesse caso:

- **Carregar a imagem:** Para isso usamos a classe `javafx.scene.image.Image`. Perceba que essa classe é uma representação alto nível de uma imagem, sendo que a mesma pode vir de um `java.io.InputStream`, um arquivo do seu computador e até mesmo um endereço da internet, sem você se preocupar com qualquer outra coisa;
- **Mostrar a imagem:** Visualizamos a imagem através da classe `javafx.scene.image.ImageView`. Repetimos o mesmo discurso para dizer que `ImageView` é um `Node` e é ele que adicionamos à nossa aplicação, aplicamos efeitos, animações etc (calma, iremos mostrar tudo isso em breve). Após saber disso, você entenderá perfeitamente que com três linhas de código podemos ter uma imagem em uma aplicação JavaFX. Mas código só daqui a pouco :)

## Formas Geométricas

Todas as formas geométricas do JavaFX, adivinhe, também são `Nodes` e podem sofrer efeitos, transformações e animações, além de ter diversas propriedades em comum. Você pode ver as figuras geométricas oferecidas pelo JavaFX no pacote `javafx.scene.shape`. Todas as figuras geométricas herdam de `javafx.scene.shape.Shape` (que por sua vez herda de... `Node`), portanto temos propriedades comuns só para as figuras geométricas, entre as quais destacamos:

- **fill:** O preenchimento da figura geométrica. Esse é um parâmetro do tipo `Paint`, que representa ou uma cor simples(exato: azul, verde, rosa...) ou um gradiente complexo;
- **stroke:** Também do tipo `javafx.scene.paint.Paint`, mas se aplica a linha que envolve a forma geométrica;
- **strokeWidth:** A largura da "stroke".

## Hora do Código

Ufa, hora de se divertir. No código atual iremos mostrar a carga de uma imagem e algumas figuras geométricas, ficando a seu cargo baixar o projeto anexado e fazer algo mais elaborado e divertido. Se formos explorar todas as propriedades de cada forma geométrica, estaríamos sendo redundantes, verbosos e chatos e programação se aprende com ação e não exposição :) Após o código, temos uma explicação linha a linha de tudo que é feito.

*Usando o projeto anterior, crie uma classe chamada ImagemFigurasGeometricas no pacote javafxpratico*

```
package javafxpratico;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class ImagemFigurasGeometricas extends Application {

    private final String IMG_URL =
        "http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/402460.gif";

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

    @Override
    public void start(Stage palco) throws Exception {
        Image imagem = new Image(IMG_URL); // 1
        ImageView visualizadorImagem = new ImageView(imagem); // 2
        visualizadorImagem.setTranslateX(80); // 3
        visualizadorImagem.setTranslateY(5); // 4

        Text textoInformativo = new Text("Algumas figuras Geométricas desenhadas com J
avaFX");
        textoInformativo.setTranslateX(30);
        textoInformativo.setTranslateY(70);

        Rectangle retangulo = new Rectangle(100, 40); // 5
        retangulo.setTranslateX(5);
        retangulo.setTranslateY(90);
        retangulo.setFill(Color.GREEN); // 6

        Circle circulo = new Circle(30);
        circulo.setTranslateX(160);
```

```

        circulo.setTranslateY(110);
        circulo.setFill(Color.YELLOW);

        Circle circuloBranco = new Circle(30);
        circuloBranco.setTranslateX(240);
        circuloBranco.setTranslateY(110);
        circuloBranco.setStroke(Color.BLACK); // 7
        circuloBranco.setStrokeWidth(3.0); // 8
        circuloBranco.setFill(Color.WHITE);

        Ellipse elipse = new Ellipse(30, 10);
        elipse.setTranslateX(320);
        elipse.setTranslateY(110);
        elipse.setFill(Color.BLUE);

        Group componentes = new Group(); // 9
        componentes.getChildren().addAll(visualizadorImagem,
            textoInformativo, retangulo, circulo,
            circuloBranco, elipse); // 10
        Scene cena = new Scene(componentes, 400, 150);
        palco.setTitle("Gráficos e Imagens em JavaFX");
        palco.setScene(cena);
        palco.show();
    }
}

```

1. Nessa linha carregamos nossa imagem usando uma URL da internet. Lembrando que essa não é a única maneira, podemos também usar um arquivo ou qualquer coisas que forneça um `InputStream`;
2. Agora criamos o nó da imagem e informamos que imagem será mostrada no construtor;
3. Usamos uma propriedade nova nessa linha e ela nem é só do `ImageView`, é da classe `Node`! Com o *translateX* informamos a posição X (imagine o posicionamento como um plano cartesiano) de um componente no pai. O método `set` serve para configurar um valor, você já viu o uso dele em classes como o `Stage`;
4. Fazemos o mesmo descrito em 3, mas agora para a posição Y. ATENÇÃO: Posicionar componentes assim não é legal de ser feito em aplicações grandes, vamos mostrar em breve os gerenciadores de layout do JavaFX;
5. Aqui já estamos criando uma forma geométrica, um Retângulo (**Rectangle**). De acordo com a forma criada, teremos parâmetros que devem ser informados. No caso do retângulo, informamos a largura e a altura;
6. Mudamos a cor padrão da forma geométrica (preta) para verde. Note que aqui usamos uma constante da classe `javafx.scene.paint.Color`;
7. É informado a cor da linha que envolve essa forma geométrica, nesse caso: preta;
8. Também aumentamos a largura (ou grossura) da linha;
9. Outra novidade nesse código: a classe `javafx.scene.Group`. Como o próprio nome

disse, usamos ela para agrupar outros nós(classes que herdam de **javafx.scene.Node**). Ela é um nó de nós, ou seja, podemos aplicar efeito, animar, transformar que isso será aplicado a todos os nós;

10. Finalmente usamos um novo método dos filhos do grupo que é adicionar todos os nossos nós de uma vez. Podemos fazer isso, ou adicionar um por um.

Por fim, ao executar o código acima, isso é o que deveria aparecer:



# Tocando Áudio

Incrível como as coisas são simples no JavaFX: tocar áudio exige 2 linhas de código, em outras palavras, simplesmente o áudio deve ser carregado e então tocado. O áudio não dispõe de uma classe para visualização, já que essa não é necessária e temos somente uma classe necessária para a carga e execução do áudio, a classe **javafx.scene.media.AudioClip**. Claro que você pode criar um player se achar necessário. Vejam abaixo o código de uma pequena aplicação de exemplo e em seguida a explicação (se você criar uma classe TocandoAudio no projeto que criamos, você pode testar esse código):

```
package javafxpratico;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.media.AudioClip;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TocandoAudio extends Application {

    // Áudio pego do seguinte site: http://sampleswap.org/mp3/song.php?id=105
    private String AUDIO_URL =
        "http://sampleswap.org/mp3/artist/5101/" +
        "Peppy--The-Firing-Squad_YMXB-160.mp3";

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

    @Override
    public void start(Stage palco) throws Exception {
        AudioClip clip = new AudioClip(AUDIO_URL); // 1
        clip.play(); // 2
        StackPane raiz = new StackPane();
        raiz.getChildren().add(new Text("Tocando Música ")); // 3
        Scene cena = new Scene(raiz, 600, 100);
        palco.setTitle("Tocando Audio em JavaFX");
        palco.setScene(cena);
        palco.show();

    }
}
```

1. Aqui carregamos o clipe de áudio e informamos a URL do audio a ser tocado; 2/ O

áudio irá começar a tocar nessa linha. Semelhante ao vídeo, temos vários outros métodos que poderíamos utilizar

2. Não é relacionado ao áudio, mas essa linha adiciona um simples texto informativo, pois não temos nada visual quanto tocamos áudio

A aplicação pode demorar um pouquinho para rodar, pois o áudio está na internet. Você pode incrementar essa aplicação quando aprender sobre botões e outros controles de interface que falaremos em breve.



# Tocando Vídeo

Tocar vídeo em JavaFX é tão simples que chega a assustar quem já tentou tocar um vídeo usando Java. Nós só temos que seguir três pequenos passos:

- **Carregar:** Você primeiramente deve carregar a mídia que vai tocar. A mídia pode estar no seu disco rígido como em um um servidor remoto(carregando assim através da sua URL);
- **Tocar.** A partir da mídia, você deverá controlar a mesma: tocar, parar, pausar...
- **Mostrar:** Nesse ponto iremos colocar o vídeo dentro de sua aplicação JavaFX.

Para realizar esses três passos temos três classes JavaFX que ficam no pacote **javafx.scene.media**. Para 1 nós temos a classe **Media**, 2 podemos fazer com a classe **MediaPlayer** e finalmente conseguimos o passo três com o uso da **MediaView**. O uso dessas classes é muito simples, abaixo temos o código de um aplicação de exemplo e em seguida um breve explicação de cada linha nova. Se você quiser testar, crie uma classe chamada **TocandoVideo** no mesmo projeto que já criamos e cole o código abaixo - note que a execução pode levar uns segundos já que ele baixa um arquivo da internet:

```
package javafxpratico;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;

public class TocandoVideo extends Application {

    // Vídeo de exemplo pego desse site:
    // http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html
    private String VIDEO_URL =
        "http://www.mediacollege.com/video-gallery/testclips/20051210-w50s.flv";

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

    @Override
    public void start(Stage palco) throws Exception {

        Media media = new Media(VIDEO_URL); // 1
        MediaPlayer mediaPlayer = new MediaPlayer(media); // 2
        MediaView mediaView = new MediaView(mediaPlayer); // 3

        StackPane raiz = new StackPane();
        raiz.getChildren().add(mediaView); // 4
        Scene cena = new Scene(raiz, 600, 400);
        palco.setTitle("Tocando Video em JavaFX");
        palco.setScene(cena);
        palco.show();

        mediaPlayer.play(); // 4
    }
}
```

1. Nessa linha estamos carregando o nosso vídeo. No construtor é recebido a URL do vídeo que pode ser ou um arquivo local ou um link para um vídeo publicado em algum servidor;
2. Após informar a mídia, nós temos que criar um player ("tocador") que será responsável por interagir com a mídia em si e assim poder tocar, parar, pausar (entre outras ações) o referido vídeo;
3. Nós já temos o vídeo e já sabemos como controlar ele, no entanto, temos que adicionar ele a nossa aplicação JavaFX para que o vídeo seja mostrado.
4. O **MediaView** é uma classe que herda de Node, ou seja, podemos adicionar ele a

qualquer Node que herda de Parent (não se preocupe, futuramente teremos posts específicos sobre essas classes). Nessa linha nós adicionamos o vídeo ao pai;

5. Por fim tocamos o vídeo através do método `play()`. Há também outros métodos para controle do vídeo, sendo possível criar um player dinâmico...

Conforme mostrado acima, são quatro linhas para podermos tocar um vídeo em uma aplicação JavaFX. Veja abaixo o resultado:



No futuro, quando você aprender mais sobre JavaFX, você pode voltar para essa mesma aplicação e fazer melhorias: adicionar botões de controle, permitir usuário escolher a URL do vídeo e por aí vai.

# Controles de interface: Rótulos, campos de texto e outros

Fazendo jus ao seu principal objetivo, JavaFX oferece muitos controles de interface entre simples rótulos para mostrar texto até complexas tabelas com milhares de registros. Nesse artigo vamos mostrar os controles mais básicos de interface que JavaFX oferece.

## Controles de interface

Todos os controles que o JavaFX oferece herdam da classe **Control**. Como alguns já devem ter deduzido, **Control** vai herdar em algum momento de **Node**, mas nesse caso ele antes herda de **Parent**.

```
java.lang.Object  
  
javafx.scene.Node  
  
javafx.scene.Parent  
  
javafx.scene.control.Control
```

Logos todos os controles usufruem com atributos como *height*, *maxHeight*, *width*, *minWidth*, entre outros.

Os controles são muito fáceis de lidar, no entanto, muitos tem suas características específicas, o que leva a uma complicação maior somente para um capítulo, então iremos separar a abordagem dos controles em vários, nesse iremos abordar "Rótulos, campos de texto, separadores e controles deslizantes".

## Rótulos, campos de texto, separadores e controles deslizantes

Rótulos, campos de texto, separadores e controles deslizantes são os controles mais simples possíveis, deixando para capítulos futuros outros controles que exigem tratamento de evento ou maior conhecimento prévio. Não vamos perder mais tempo e vamos para o código em seguida uma explicação!

```
package main;
```

```
import javafx.application.Application;
import javafx.geometry.Orientation;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Separator;
import javafx.scene.control.Slider;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ControlesSimples extends Application {

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

    @Override
    public void start(Stage palco) throws Exception {
        VBox raiz = new VBox(10); // 1
        raiz.setAlignment(Pos.CENTER); // 2

        Label rotuloDemo = new Label("Sou um rótulo de texto!"); // 3
        rotuloDemo.setTooltip(new Tooltip(
            "Esse é um rótulo para mostrar textos de forma simples")); // 4

        TextField campoTexto = new TextField("Digite algo"); // 5
        campoTexto.setTooltip(new Tooltip(
            "Campo de texto para entrada de uma só linha "));

        TextArea areaTexto = new TextArea("Digite algo com várias linhas"); // 6
        areaTexto.setTooltip(new Tooltip(
            "Campo de texto para entrada de múltiplas linhas"));

        Separator separadorHorizontal = new Separator(); // 7
        Separator separadorVertical = new Separator(Orientation.VERTICAL); // 8
        Slider deslizando = new Slider(); // 9
        deslizando.setShowTickLabels(true); // 10
        deslizando.setShowTickMarks(true); // 11
        deslizando
            .setTooltip(new Tooltip(
                "O controle deslizante tem um valor numérico de acordo com sua posição"));

        raiz.getChildren().addAll(rotuloDemo, campoTexto, areaTexto,
            separadorVertical, separadorHorizontal, deslizando);

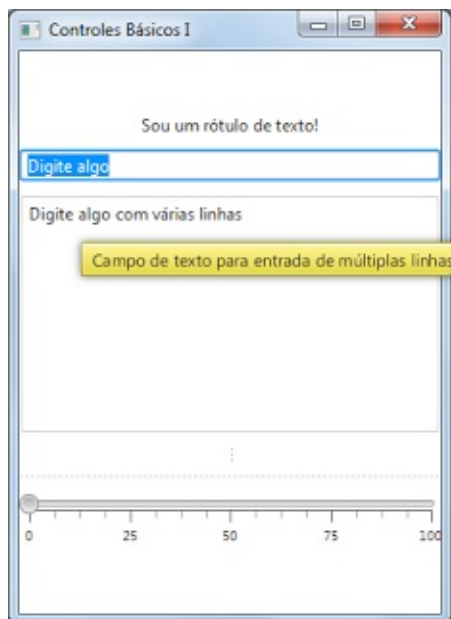
        Scene cena = new Scene(raiz, 300, 400);
        palco.setTitle("Controles Básicos I");
        palco.setScene(cena);
        palco.show();
    }
}
```

```
}  
}
```

1. Nessa linha introduzimos um novo componente pai, ou seja, um componente que podemos colocar outros dentro dele. Nesse caso a **VBox**, que irá organizar os componentes verticalmente. O construtor informa o espaçamento entre os componentes, no nosso caso 10;
2. Você também pode determinar qual o posicionamento dos componentes dentro da VBox. Você usa o enum **Pos** para informar qual posicionamento quer. Nessa linha, informamos o posicionamento **CENTER**, logo todos os componentes serão posicionados no centro da nossa **VBox**;
3. Esse é nosso primeiro controle de interface apresentado um **Label**, ou o rótulo. Esse componente nada mais é do que um texto estático, semelhante ao **Text**, mas herda de **Control**, o que significa que ele é um controle de interface, no entanto, o usuário não muda o texto do Label diretamente. Como você já deve ter deduzido, o Label é um **Node**, assim como todos os controles e assim como e qualquer componente dentro da Cena do JavaFX. Pense quão poderoso esse detalhe arquitetural é!
4. Os controles tem algumas características em comum e uma delas é a possibilidade de informar um **Tooltip**, que nada mais é um texto demonstrativo informado quando mantemos o mouse sobre o componente;
5. Nessa linha criamos um campo de texto, o **TextField**. Esse campo, ao contrário do Label, permite a modificação do usuário diretamente. Futuramente você pode recuperar esse texto através do método *getText()*;
6. Semelhante ao campo de texto, temos o uma área de texto. Não há muita diferença entre essas duas classes, na verdade a maior diferença é que o **TextArea** permite a entrada de várias linhas e também contém propriedades para trabalhar com essa característica. Detalhe que a **TextArea** e o **TextField** herdam de **TextInputControl**;
7. Nessa linha mostramos um separador horizontal. Ele é sem graça assim mesmo, só serve para separar as coisas. No entanto, lembre-se que ele é um... **Node** e também um controle;
8. Aqui também criamos um separador, mas esse fica na vertical conforme informado pelo Enum **Orientation**;
9. O controle deslizando, ou **Slider**, permite você escolher um valor numérico dentro de uma faixa determinada. De acordo com a posição do controle, você terá um valor numérico que pode ser recuperado depois através do método *getValue()*;
10. Se você quiser que o controle deslizante mostre marcas para indicação relativa de qual valor o Slider tem no momento, você deve chamar o método *setShowTickLabels* que vai configurar se devem aparecer essas marcas, ou não (true, false);
11. Na linha anterior, mostramos como colocar marcas, nessa linha chamamos o método *setShowTickMarks* e informamos que queremos ver números que indicam o valor do

controle deslizante.

Finalmente, olhe como ficou nossa aplicação:



# Controles de interface: Tratamento de eventos e o botão

Vamos continuar falando de controles de interface e esse capítulo será dedicado a somente um componente: o Botão. O motivo disso é que para falar de botão, devemos falar de tratamento de evento, logo, precisaremos focar nisso, pois esse conhecimento é muito utilizado em qualquer aplicação JavaFX.

## Tratamento de evento

Tratar evento é determinar qual código será executado de acordo com uma ação do usuário. O usuário interage com sua aplicação através da interface gráfica, e dela ele faz diversas coisas: passa o cursor do mouse sobre um componente, arrasta coisas, clica, move o cursor para dentro ou para fora de um componente, entre outros. Qualquer componente é passível de sobre ações desse tipo por parte do usuário, logo, na classe **Node** podemos tratar qualquer um desses eventos. Nesse capítulo, no entanto, não iremos falar de todos eles, mas só de um que é o evento de ação. Quando o usuário clica em um botão, ele dispara esse evento. Iremos apresentar o botão, fazer um programa simples com ele e explicar como tratar o clique em um botão. Quando entrarmos em detalhes na classe **Node** iremos mostrar como tratar cada um desses outros eventos

## Botões e Ouvintes

Botões estão em todos os lugares, com certeza você deve ter clicado em algum botão hoje! No JavaFX, um botão pode ser usado através da classe **Button**. No entanto, simplesmente adicionar um botão na cena não ajuda em muita coisa, você deve informar para ele o que deve ser feito assim que um usuário clicar no mesmo. Isso é feito através de ouvintes de evento, que são classes que implementam uma determinada interface.

No caso de botão, temos o atributo *onAction* que é do tipo da interface **EventHandler**. Essa interface exige que você implemente o método *handle*, que é será chamado quando você clicar no botão. Em resumo, você deve criar uma classe que implementa essa interface e informar ao botão uma instância dessa classe para que o botão chame o método quando o usuário do seu programa clicar nele! O tipo do evento pode ser parametrizado usando Genéricos - assim o tipo especificado pelo genérico será o tipo do parâmetro recebido no método *handle*.



Felizmente com o Java 8 a gente não precisa escrever muito código, podemos usar expressões **Lambda**! No final desse livro você deverá encontrar um apêndice que fala sobre Java 8 e as expressões lambdas.

Nossa, muita informação! Mas isso ficará melhor nas próximas linhas, pois mostremos exemplos de código e tudo ficará mais claro.

## Clique em mim!

Vamos começar mostrando um exemplo do clássico clique em mim para assim demonstrar o tratamento mais simples possível de evento. A nossa classe ouvinte, ou tratadora de evento se encontra abaixo e em seguida uma breve explicação do código.

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public static class TratadorEvento implements EventHandler<ActionEvent> { // 1

    public void handle(ActionEvent evento) { // 2
        System.out.println("Evento tratado por uma classe externa");
    }
}
```

1. Conforme já falamos, a classe destinada a tratar eventos deve implementar a Interface `EventHandler`. Nesse caso também fica claro o uso de Genéricos, onde devemos informar um tipo de classe que herda de **Event**, no caso informamos o **ActionEvent**, pois é o que o botão requer;
2. Esse é o método que será chamado quando clicarmos no botão. O parâmetro evento será enviado pelo gerador do evento, no caso o botão, e deles poderíamos saber tudo de quem gerou o evento.

Mas de quem esse tratador de evento trata eventos? O próximo passo é informar para um botão qual será o seu tratador de evento. Veja abaixo que uso duas linhas para isso e basicamente criamos um botão e informamos quem será o tratador de evento através do método `setOnAction`.

Ótimo, já sabemos tratar eventos, mas e se você não quiser criar uma classe para toda vez que quiser tratar um clique? Você pode utilizar um classe anônima ou as modernas expressões lambdas.

Para ilustrar tudo isso, criamos uma simples aplicação com dois botões e três formas de se registrar um tratador de evento. O código abaixo mostra essa aplicação e também explica diretamente no comentário!

```
package javafxpratico;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

//Se quisermos que essa classe trate evento, ela deve herdar de EventHandler
public class TratamentoEventoComBotao extends Application
    implements EventHandler<ActionEvent> {

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

    @Override
    public void start(Stage palco) throws Exception {
        VBox raiz = new VBox(20);
        raiz.setAlignment(Pos.CENTER);
        raiz.setTranslateY(5);

        Button botao1 = new Button("Clique em mim! (Tratador externo)");
        Button botao2 = new Button("Clique em mim! (Expressão Lambda)");
        Button botao3 = new Button("Clique em mim! (Própria classe)");

        // usamos a classe TratadorEvento para cuidar dos eventos
        botao1.setOnAction(new TratadorEvento());
        // Criando uma instância de uma classe anônima para tratar evento
        botao2.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent evento) {
                System.out.println("Evento tratado por uma classe anônima!");
            }
        });
        // o botão 3 usa essa própria classe para tratar seus eventos
        botao3.setOnAction(this);

        raiz.getChildren().addAll(botao1, botao2, botao3);

        Scene cena = new Scene(raiz, 300, 200);
        palco.setTitle("Tratando eventos");
        palco.setScene(cena);
        palco.show();
    }

    /*
     * Como a própria classe TratamentoEventoComBotao implementa
     * a interface EventHandler, ela mesma pode responder a eventos do botão
     */
}
```

```

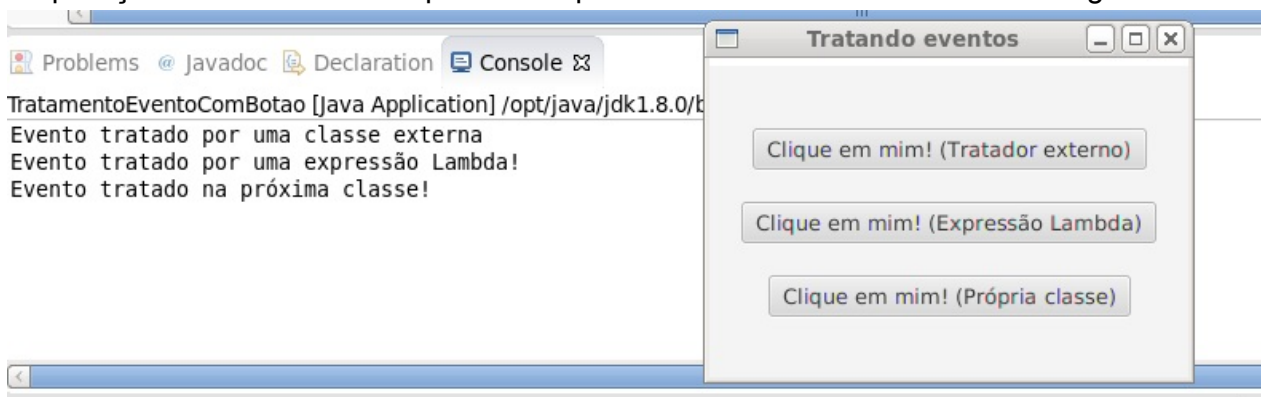
    *
    */
    public void handle(ActionEvent evento) {
        System.out.println("Evento tratado na próxima classe!");
    }

    /**
     * Uma classe para tratar os eventos
     * poderia ser uma classe externa
     *
     */
    public static class TratadorEvento implements EventHandler<ActionEvent> { // 1

        public void handle(ActionEvent evento) { // 2
            System.out.println("Evento tratado por uma classe externa");
        }
    }
}

```

A aplicação final bem como o que será impresso no console é mostrado na imagem abaixo:



# Radio Button, CheckBox e ToggleButton

Vamos para mais um capítulo e nesse vamos falar sobre mais alguns controles de interface, especificamente sobre botões agrupados, mas que permitem a seleção individual, e também sobre a caixa de checagem, que representam um valor booleano (sim ou não, verdadeiro ou falso).

## Várias possibilidades, uma escolha

Os botões de rádio, **RadioButton**, ou os botões alternados, **ToggleButton**, permitem apresentar um grupo de botão no qual é permitido selecionar somente um. Esse controle de não deixar dois botões serem selecionados é feito através de um grupo de botões, **ToggleGroup**, que recebe os mesmos e então controla o comportamento deles.

O uso desse tipo de botão é feito em diversos casos onde você tem escolher bem pré-definidas, por exemplo: campos para escolha de sexo, faixa de salário, alternativas de questões multivaloradas com uma resposta válida, entre outros.

## Três estados, um controle

O segundo controle de interface que vamos mostrar hoje é simples e bastante utilizado em formulários para pedido de informações de usuários quando temos uma questão direta (exemplo: "Você Fuma?"). Estamos falando da caixa de checagem, **CheckBox**, que na verdade tem três estados: selecionado, não selecionado e indeterminado(você pode escolher se quer habilitar esse estado ou não).

Pode parecer estranho, mas um problema que podemos ter é pedir a entrada de um usuário e o mesmo sequer ligar para esse campo, daí ele larga o campo em branco, logo, como você vai fazer para saber se ele não respondeu ou não marcou com o objetivo de responder de forma negativa à pergunta?

## A hora mais feliz

Vamos agora colocar esses botões em uma tela JavaFX e mostrar algumas classes novas para posicionar os componentes. Vamos avançar um pouco o nível: já foram alguns capítulos só com aplicações simples, temos que colocar um pouco mais de emoção e daqui a pouco já estaremos trabalhando com aplicações grandes com JavaFX.

O programa abaixo representa um programa de uma pesquisa simples sobre programação. Com ele vamos mostrar o que discutimos nesse capítulo e algumas coisas novas:

```
package javafxpratico;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class RadioCheckToggle extends Application {
    public static void main(String[] args) {
        launch();
    }

    @Override
    public void start(Stage palco) throws Exception {
        VBox raiz = new VBox(10);
        raiz.setTranslateX(10);
        raiz.setTranslateY(20);

        Label lblTitulo = new Label("Pesquisa sobre Programação");
        lblTitulo.setUnderline(true); // 1

        final TextField txtNome = new TextField();
        HBox hbNome = new HBox(10); // 2
        hbNome.getChildren().addAll(new Label("Nome"), txtNome);

        HBox hbSo = new HBox(20);
        ToggleButton tbLinux = new ToggleButton("Linux"); // 3
        tbLinux.setSelected(true);
        ToggleButton tbWindows = new ToggleButton("Windows");
        ToggleButton tbMac = new ToggleButton("Mac");
        final ToggleGroup tgSo = new ToggleGroup(); // 4
        tgSo.getToggles().addAll(tbLinux, tbWindows, tbMac); // 5
        hbSo.getChildren().addAll(new Label("Sistema Operacional utilizado"),
            tbLinux, tbWindows, tbMac);

        final ToggleGroup tgLinguagem = new ToggleGroup();
        HBox hbLinguagens = new HBox(20);
        RadioButton rbJava = new RadioButton("Java"); // 6
        rbJava.setSelected(true);
        RadioButton rbC = new RadioButton("C");
        RadioButton rbPython = new RadioButton("Python");
        tgLinguagem.getToggles().addAll(rbJava, rbC, rbPython);
        hbLinguagens.getChildren().addAll(new Label(
            "Linguagem de programação Predileta:"), rbJava, rbC, rbPython);
    }
}
```

```

final CheckBox chkFrequencia = new CheckBox("Programa todo dia?"); // 7
final CheckBox chkGosto = new CheckBox("Gosta de programação?");
chkGosto.setAllowIndeterminate(true); // 8
chkGosto.setIndeterminate(true);

Button btnSubmeter = new Button("Submeter pesquisa");
btnSubmeter.setOnAction(e -> {
    System.out.println("\t\tResultado da pesquisa para \""
        + txtNome.getText() + "\"\n");
    // Podemos não ter um SO selecionado
    ToggleButton tbSo = (ToggleButton) tgSo.getSelectedToggle(); // 9
    System.out.print("Sistema Operacional predileto: ");
    System.out.println(tbSo == null ? "Não selecionado." : tbSo.getText());

    // Deve ter uma linguagem selecionada
    RadioButton rbLinguagem = (RadioButton) tgLinguagem.getSelectedToggle();
    System.out.println("Linguagem de programação: " + rbLinguagem.getText());
    // 10
    System.out.println(
        (chkFrequencia.isSelected() == true ? "P" : "Não p") +
        "rograma todo dia.");

    System.out.print("Gosta de programação: ");
    if (chkGosto.isSelected()) {
        System.out.println("Sim.");
        // 11
    } else if (chkGosto.isIndeterminate()) {
        System.out.println("Não respondido.");
    } else {
        System.out.println("Não.");
    }
    System.out.println("\n\n");
});
raiz.getChildren().addAll(lblTitulo, hbNome,
    hbSo, hbLinguagens, chkFrequencia,
    chkGosto, btnSubmeter);

Scene cena = new Scene(raiz, 500, 250);
palco.setTitle("Pesquisa sobre programação");
palco.setScene(cena);
palco.show();
}
}

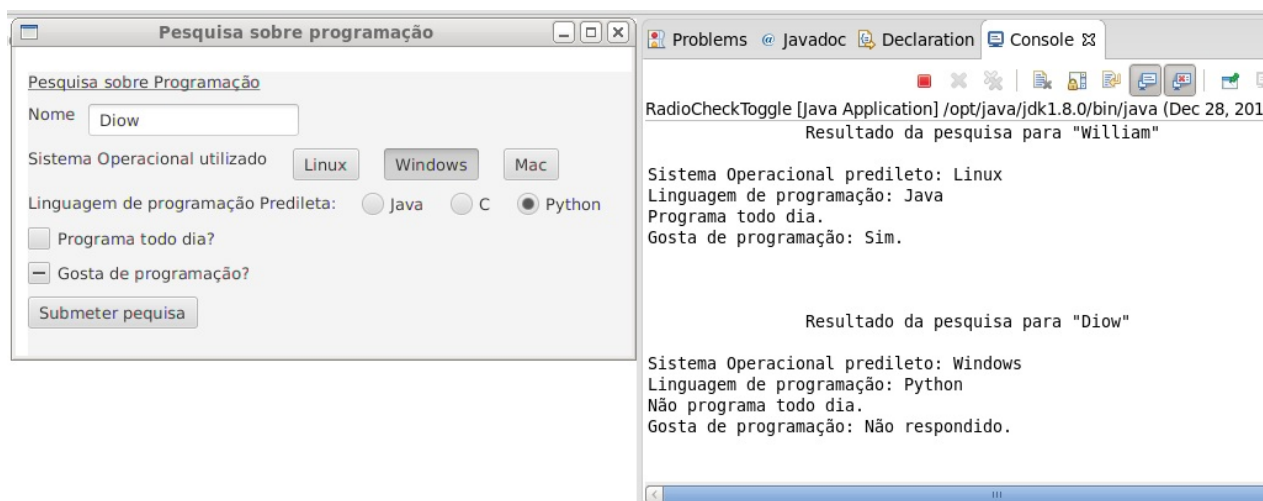
```

### Explicação do Código

1. Já começamos mostrando uma propriedade do **Label**. Se você chamar esse método e enviar **true**, ele será sublinhado;
2. Esse componente é novo, mas o primo dele, **VBox**, funciona de forma muito semelhante. As propriedades são exatamente as mesmas, a diferença é que a **HBox**

- os componentes são organizados horizontalmente;
3. Nessa linha estamos criando o tão falado botão alternado **ToggleButton**. Vamos usar três. Daqui a pouco teremos que adicionar ele a um grupo junto com outros botões alternados, assim somente um poderá ser escolhido nesse grupo, mas há a possibilidade de não escolher nenhum. Perceba que estamos informando no construtor o texto que ele irá mostrar;
  4. O grupo que falamos no passo anterior é representado pela classe **ToggleGroup**. Ele agrupa objetos do tipo **Toggle**, que é a classe pai do **ToggleButton** e do **RadioButton**;
  5. Nessa linha simplesmente adicionamos os botões já declarados para o grupo, dizendo que eles não podem ser selecionados simultaneamente;
  6. Semelhante ao **ToggleBox**, temos o **RadioButton**. A maior diferença está na aparência e que um grupo de **RadioButton** não permite um botão não selecionado;
  7. Aqui usamos uma caixa de checagem, **CheckBox**. A String enviada no construtor também é o texto associado à ela;
  8. Dizemos que essas caixas de checagem têm três estados. Nessa linha habilitamos o terceiro estado, o estado indeterminado (quântico?);
  9. Dentro do listener do botão usando Lambda, estamos lendo as propriedades dos controles de interface apresentados. Essa linha especificamente retorna o **Toggle** selecionado. Como sabemos que esse grupo contém toggles do tipo **ToggleButton**, já fazemos um *cast* para esse tipo. Se nada estiver selecionado, null é retornado;
  10. O *isSelected* retorna um valor booleano para informar se essa caixa está selecionada;
  11. Por fim(ufa), usamos o método *isIndeterminate* para sabermos se o estado dessa **CheckBox** é indeterminado.

Nossa aplicação final se parece com o seguinte:



# ComboBox e ChoiceBox

Nesse capítulo também sobre controles, vamos falar das caixas de combinação, **ComboBox**, e caixas de escolhas, **ChoiceBox**. Ambas permitem você escolher um valor entre um conjunto de opções.

## Muitas opções, uma possível escolha

Os dois controles que vamos apresentar hoje funcionam de maneira muito semelhante, a maior diferença está na forma visual que elas são apresentadas dentro da sua aplicação. Ambas permitem que você faça a escolha de uma só opção entre diversas outras. no entanto, há uma pequena diferença visual entre elas e o **ComboBox** permite que você fabrique as células que serão mostradas. Criar fábrica de células é um assunto que facilmente ocuparia um capítulo inteiro, vamos deixar isso para outra hora. Para tratar dos dados individualmente da parte visual, usamos em ambas classes o conceito de **SelectionModel**, onde abstraímos a parte de seleção de dados. Temos duas abstrações: **MultipleSelectionModel**(permite selecionar vários itens ao mesmo tempo) e **SingleSelectionModel**(permite selecionar um único item ao mesmo tempo. Hoje também vamos falar de um conceito novo e específico do JavaFX, então vamos nos limitar a mostrar o ComboBox e uma aplicação super simples, no entanto, o leitor deve poder acessar a documentação *JavaDoc* do ChoiceBox e criar a mesma aplicação usando ele. Enfim, vamos ao código!

## Selecione a cor

Para demonstrar o **ComboBox**, criamos uma aplicação bem simples que permite que você selecione a cor da cena e de um retângulo através do uso desse controle de interface. Ao selecionar a cor, a mudança já é imediata. Como fazemos isso? Confira o código e em seguida tenha acesso a uma explicação rápida:

```
package javafxpratico;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
```



```

public class TestandoComboBox extends Application {

    String cores[] = { "Blue", "Black", "Red", "White",
        "Green", "Yellow", "Gray", "Pink", "Salmon" };

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

    @Override
    public void start(Stage palco) throws Exception {
        StackPane raiz = new StackPane();

        HBox opcoes = new HBox(10);
        opcoes.setAlignment(Pos.CENTER);

        final Rectangle retangulo = new Rectangle(300, 100);
        final Rectangle fundo = new Rectangle(450, 250);

        ComboBox<String> cbCorCena = new ComboBox<>(); // 1
        ComboBox<String> cbCorRetangulo = new ComboBox<>();

        cbCorCena.getItems().addAll(cores); // 2
        cbCorRetangulo.getItems().addAll(cores);

        opcoes.getChildren().addAll(cbCorCena, cbCorRetangulo);

        raiz.getChildren().addAll(fundo, retangulo, opcoes);

        final Scene cena = new Scene(raiz, 450, 250);
        palco.setTitle("Uso de ComboBox");
        palco.setScene(cena);
        palco.show();
        // 3
        cbCorCena.getSelectionModel().selectedItemProperty().addListener(
            (valorObservado, valorAntigo, valorNovo) -> {
                // 4
                fundo.setFill(Color.valueOf(valorNovo));
            });
        cbCorRetangulo.getSelectionModel().selectedItemProperty()
            .addListener((valorObservado, valorAntigo, valorNovo) -> {
                retangulo.setFill(Color.valueOf(valorNovo));
            });
        cbCorCena.getSelectionModel().select(0); // 5
        cbCorRetangulo.getSelectionModel().select(1);
    }
}

```

1. Começamos instanciando uma **ComboBox**. Perceba que ela faz uso de genéricos, onde informamos que tipo de dado serão os itens contidos nela, no nosso caso, String;

2. Nessa linha adicionamos os itens. Isso é feito de forma semelhante com a qual adicionamos itens a um componente pai como o **Group**, **VBox**, **HBox**, entre outros. Nesse caso estamos adicionando uma lista de Strings, pois esse é o tipo aceitado por essa combobox;
3. Para observar quando o usuário escolhe algum valor no Combobox, adicionamos um ouvinte ao valor observável a propriedade do item selecionado. Isso pode parecer bastante estranho, mas em breve você vai se acostumar. JavaFX traz uma API com atributos diferentes, onde nós podemos observar as modificações feitas nesses atributos e tomar alguma ação. Novamente isso é feito através de Listeners, no nosso caso usamos um listener de mudança, o **ChangeListener**. Note que o uso dele é bastante semelhante ao uso dos listeners de botões. Em resumo, nós estamos observando o valor selecionado do combobox através de um listener, quando ele muda, nós trocamos a cor de fundo da cena e usamos a sintaxe das expressões lambda para simplificar;
4. A classe Color do JavaFX é bastante interessante e contém muito mais do que um aglomerado de constantes para definir cores. É possível também converter o valor de texto de uma cor para um objeto **Color**. É isso o que fazemos aqui;
5. Por fim nós usamos o método select do modelo de seleção para escolher uma cor. Esse método equivale a ação de um usuário ao escolher uma cor na combobox usando a sua aplicação, ou seja, isso também vai disparar o ouvinte de mudança que falamos na explicação 3.

Por fim, essa é a aplicação resultante:



# Gerenciando Layout: HBox, VBox e StackPane

Organizar os componentes que estão em uma aplicação gráfica é sempre um desafio, pois você tem que posicionar e dimensionar cada elemento que adicionar a sua aplicação. Os gerenciadores de leiaute(ou layout, você decide) servem para auxiliar você nessa missão.

O JavaFX traz alguns gerenciadores pronto para auxiliar você. Nesse capítulo vamos começar a falar dessa API, no entanto, você deve ter percebido o uso de alguns em capítulos anteriores.

## O que é um gerenciador de leiaute?

O conceito é bem simples. Quando você vai criar sua aplicação, a "telinha" dela, você tem necessidades comuns de posicionamento e dimensionamento dos componentes. Exemplo: as vezes você quer colocar eles alinhados verticalmente ou horizontalmente, quer empilhar, colocar em uma "grade" e por aí vai. Fazer isso na mão já é difícil e ainda teremos que recriar o código para cada nova aplicação que escrevermos... Um gerenciador de leiaute é uma classe que faz esse gerenciamento para você de acordo com uma forma de posicionamento já estabelecida. Em termos práticos, a classe **VBox**, por exemplo, organiza os componentes verticalmente e podemos adicionar nela os componentes de nossa aplicação como os botões, rótulos, tabelas, ou até outros gerenciadores de leiaute. As principais classes do JavaFX que exercem essa função são:

- **VBox**: Alinha os componentes verticalmente;
- **HBox**: Alinha os componentes horizontalmente;
- **StackPane**: Empilha um componente sobre o outro;
- **BorderPane**: Divide os elementos em regiões e coloca um componente em cada uma desses regiões
- **FlowPane**: Ajeita os componentes de acordo com uma orientação e com o fluxo da aplicação;
- **GridPane**: Cria uma grade com os componentes. É possível informar qual a posição do componente na grade;
- **AnchorPane**: Os componentes filhos são "ancorados" em uma parte do painel;
- **TilePane**: Um painel com pedaços (Tile vem do inglês e significa azulejo) para os componentes semelhante ao GridPane, mas permite tratamento dos "Tiles";

Todas essas classes estão no pacote `javafx.scene.layout`.

## Arquitetura

Como você já deve ter percebido, essas os gerenciadores aceitam classes que herdam de `Node` e os próprios são `Nodes` também. Eles estendem diretamente de **Pane** cujo método mais comum é *getChildren*.

O método *getChildren()* retorna os elementos filhos em uma **ObservableList**. Essas listas simplesmente atualizam automático a visualização quando adicionamos componentes nela. Com ela podemos adicionar os componentes um a um, usando um array ou simplesmente adicionar quantos elementos quiser na chamada do método. Métodos mais comuns da `ObservableList` são:

- **addAll(E... elements)**: adiciona 1 ou mais elementos em uma só chamada de método. Você pode passar vários elementos também ou até um array. É possível adicionar um único componente usando o método `add`;
- **removeAll**: remove todos os elementos da lista; Com o método `remove`, é possível remover um componente de um determinado índice ou um objeto informado.
- **setAll (E... elements)**: remove os elementos anteriores e adiciona os novos informados; Temos também o método `set` onde é possível "setar" o nó de um determinado índice.

Há outros métodos. As listas observáveis são poderosas classes do mundo JavaFX, no entanto, esses são os mais usados frequentemente. Lembre-se que essa lista é tipada para `Node`, o que significa que você só pode adicionar elementos desse tipo ou subclasses de `Node` e também reforçando que mexer na lista já garante que a sua aplicação seja atualizada.

## Conhecendo na prática

Depois de tanto conceito e uma pequena exploração da API, vamos criar aplicações simples que usam essas classes e dessa forma vamos ver em ação tudo que falamos, tornando mais prazeroso o aprendizado e a exploração da API. Hoje vamos mostrar só três classes: **VBox**, **HBox** e **StackPane**.

### VBox

Um dos gerenciadores mais simples, a classe `VBox` simplesmente alinha os componentes verticalmente e permite determinar o espaçamento entre eles e ainda o alinhamento. Abaixo um código que simplesmente adiciona umas forma geométricas a uma `VBox` e em seguida a explicação do mesmo.

```
package javafxpratico;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class PraticaVBox extends Application {

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

    @Override
    public void start(Stage stage) throws Exception {
        VBox caixaVertical = new VBox(); // 1
        caixaVertical.setSpacing(5); // 2
        caixaVertical.setAlignment(Pos.CENTER); // 3

        // 4
        caixaVertical.setTranslateX(10);
        caixaVertical.setTranslateY(20);

        // 5
        Rectangle retanguloAzul = new Rectangle(40, 20);
        Rectangle retanguloVerde = new Rectangle(40, 20);
        Rectangle retanguloVermelho = new Rectangle(40, 20);

        retanguloAzul.setFill(Color.BLUE);
        retanguloVerde.setFill(Color.GREEN);
        retanguloVermelho.setFill(Color.RED);

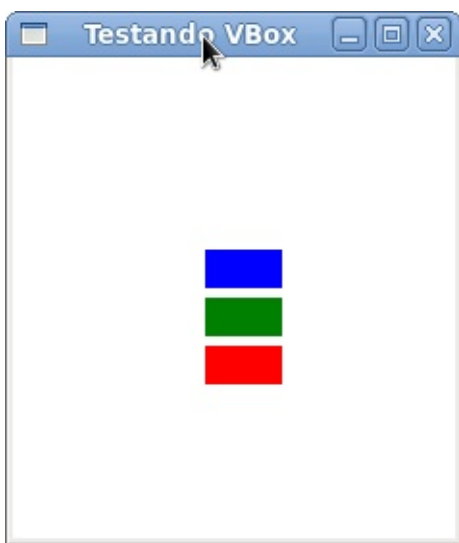
        // 6
        caixaVertical.getChildren().addAll(retanguloAzul, retanguloVerde, retanguloVermelho);
        stage.setScene(new Scene(caixaVertical, 230, 250));
        stage.show();
        stage.setTitle("Testando VBox");
    }
}
```

1. Criando nossa VBox. Não estamos enviando nada no construtor, mas é possível informar o espaçamento entre os componentes diretamente no construtor;
2. Informamos o espaçamento. Esse atributo é do tipo Double e ele informa qual distância os componentes devem entre eles;
3. Aqui determinamos que os componentes fiquem centralizados no nosso VBox.

Podemos também escolher colocar eles para esquerda, direita, entre outros. Veja o enum Pos.

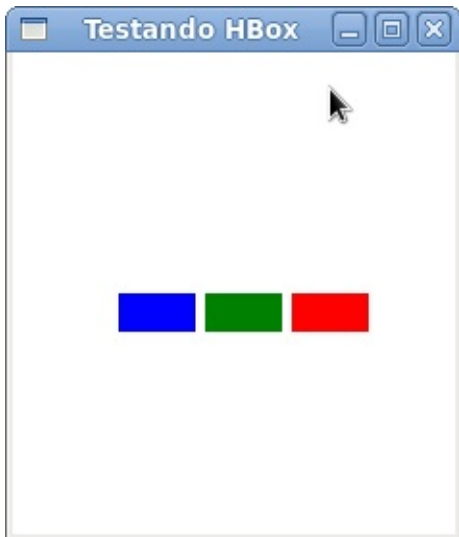
4. Nessa linha mudamos a posição X e Y do nosso VBox. O interessante é que os componentes filhos terão que obedecer a posição do pai, para isso eles também são deslocados automaticamente;
5. Criamos nossos componentes que estarão no VBox. Criamos retângulos e lembrando que eles herdam de Node. Isso é um requisito para nosso gerenciador.
6. Agora adicionamos todos de uma vez. A ordem de adição é a ordem que eles serão mostrados na aplicação.

O resultado está abaixo. Lembre-se de sempre fuçar bastante as classes, essa é a melhor forma de aprender e se divertir :)



## HBox

Não iremos entrar em detalhes dessa classe por que ela trabalha exatamente igual a VBox, com exceção que ela alinha os componentes horizontalmente. Abaixo temos uma imagem de como o código acima se comportaria simplesmente trocar a "instanciação" de VBox para HBox.



## StackPane

Um comportamento não muito comum, mas que pode ser útil em diversas aplicações, é o trazido pela classe **StackPane**. Nela os componentes são empilhados uns nos outros, ou seja, quem entrou primeiro fica embaixo do segundo e assim por diante. O exemplo de código abaixo mostra isso muito claramente. Perceba que não iremos explicar o código pois ele é semelhante ao mostrado anteriormente, a exceção está no resultado visual. Estamos desenhando uma bandeira do Brasil (ou tentando), para isso criamos um retângulo, um losango um círculo e, o mais difícil, um arco. Usamos as formas geométricas do JavaFX para fazer isso e no fim colocamos em um gerenciador do tipo StackPane, veja que ele empilha as imagens de forma que por final temos algo que é quase uma bandeira do Brasil... Quase...

```
package javafxpratico;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.ArcTo;
import javafx.scene.shape.Circle;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class PraticaStackPane extends Application {

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

```
@Override
public void start(Stage palco) throws Exception {
    StackPane painelEmpilhador = new StackPane();
    Rectangle retangulo = new Rectangle(220, 120);
    retangulo.setFill(Color.GREEN);

    Polygon losango = new Polygon();
    losango.getPoints().addAll(new Double[]{
        50.0, 50.0,
        150.0, 0.0,
        250.0, 50.0,
        150.0, 100.0,
        50.0, 50.0 });
    losango.setFill(Color.YELLOW);
    Path arco = new Path();
    MoveTo moveTo = new MoveTo();
    moveTo.setX(0.0);
    moveTo.setY(0.0);

    ArcTo arcTo = new ArcTo();
    arcTo.setX(55.0);
    arcTo.setY(0.0);
    arcTo.setRadiusX(50.0);
    arcTo.setRadiusY(50.0);

    arco.getElements().add(moveTo);
    arco.getElements().add(arcTo);
    arco.setStroke(Color.WHITE);

    arco.setRotate(180);
    arco.setStrokeWidth(5);

    Circle circulo = new Circle(30, Color.BLUE);

    painelEmpilhador.getChildren().addAll(retangulo, losango, circulo, arco);

    Scene cena = new Scene(painelEmpilhador, 250, 100);
    palco.setTitle("Testando StackPane");
    palco.setScene(cena);
    palco.show();
}
}
```

Mais uma vez note que a ordem que adicionamos os componentes é muito importante. Veja o resultado:





# Gerenciando Layout: BorderLayout, FlowPane e o GridPane

Nesse capítulo vamos apresentar o BorderLayout, FlowPane e o GridPane. A arquitetura deles é a mesma, se você quiser saber mais sugiro que releia o capítulo anterior.

## BorderPane

Uma das heranças do bom e velho Swing é o **BorderPane**. A idéia é bem simples: você coloca os componentes em regiões do painel. Essas regiões são referenciadas como: norte, sul, leste, oeste e centro.

Não há nenhum segredo no uso dele, mas há um pequena diferença que é termos que informar na hora de adicionar os componentes a qual região o mesmo pertence. Vamos ao exemplo de código e em seguida a explicação.

```
BorderPane borderPane = new BorderLayout();
Label lblTop, lblEsquerda, lblBaixo, lblDireita, lblCentro;
// 1
borderPane.setTop(lblTop = new Label("Topo"));
borderPane.setLeft(lblEsquerda = new Label("Esquerda"));
borderPane.setBottom(lblBaixo = new Label("Baixo"));
borderPane.setRight(lblDireita = new Label("Direita"));
borderPane.setCenter(lblCentro = new Label("Centro"));
// 2
BorderPane.setAlignment(lblTop, Pos.CENTER);
```

1. Ao contrário dos layouts apresentados no último capítulo, o BorderLayout não utiliza os métodos add, set e remove que falamos no nosso último capítulo, mas sim métodos correspondentes à região da tela. Os métodos *setTop*, *setLeft*, *setBottom*, *setRight* e *setCenter*, são respectivamente para informar os nós do topo, esquerda, baixo, direita e centro. Esses nós são agrupados dentro do espaço do BorderLayout
2. Nessa parte mostramos como alinhamos um nó. Veja que usamos um método estático para isso, o que já gerou muitas críticas. Cabe a você tentar outros valores de **Pos** na chamada desse método!

Abaixo o resultado desse código:

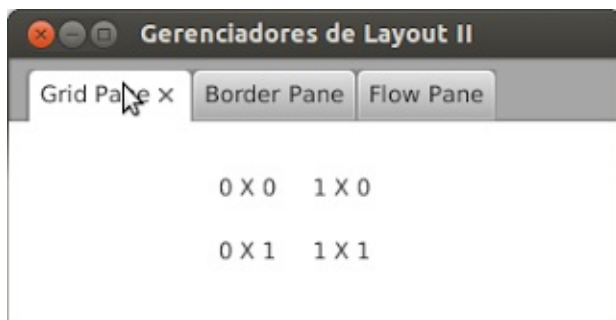


## GridPane

O GridPane, ou painel de grade, permite adicionar componentes em posições específicas semelhantes a uma grade de nós. Para entender melhor imagine a área do **GridPane** como um tabuleiro de batalha naval. Nesse tipo de jogo, temos os campos definidos por divisões, como por exemplo A1, B5, etc. O GridPane atua de forma semelhante, posicionando componentes nessa grade. No entanto, especificamos a posição com dois números invés de uma letra e um número, exemplo: 1-1, 5-2, etc. Esse gerenciador de leiaute também não utiliza os métodos add, set e remove. Na verdade, no momento de adicionar o componente já especificamos qual a "gradezinha" ele vai ocupar. Vamos a um exemplo de código para esclarecer.

```
GridPane gridPane = new GridPane();
// 1
gridPane.add(new Label("0 X 0"), 0, 0);
gridPane.add(new Label("0 X 1"), 0, 1);
gridPane.add(new Label("1 X 0"), 1, 0);
gridPane.add(new Label("1 X 1"), 1, 1);
// 2
gridPane.setVgap(20);
gridPane.setHgap(20);
// 3
gridPane.setTranslateX(120);
gridPane.setTranslateY(30);
```

1. Estamos adicionando um componente ao **GridPane** e informando a posição X e Y que ele vai ocupar na grade. Um label está sendo adicionado para mostrar para você onde cada elemento vai ficar
2. Esses métodos server para informar o espaçamento entre os componentes adicionados. *setVgap* é para o espaço na vertical e *setHgap* para o espaço na horizontal. "Gap" pode ser traduzido como "lacuna".
3. Por fim configuramos a posição dos componentes na tela. Assim como qualquer nó, podemos informação a posição X e Y do componente no "pai" dele. E isso resulta em:



## FlowPane

"Painel de Fluxo" é um significado ao pé da letra para **FlowPane**. Como o nome diz, é um painel que segue o fluxo da coisa, ou seja, você vai adicionando componentes e ele vai colocando de acordo com o fluxo. O fluxo pode ser na vertical ou horizontal e os componentes podem ser posicionados de forma centralizada, centralizada na vertical, tudo ao topo, etc. Parece difícil, mas é um dos painéis mais simples. Por exemplo, digamos que você utilize um desses do tipo horizontal e adicione 4 componentes. O 4 não caberia na mesma linha, então o FlowPane coloca ele na "linha abaixo", mesmo sobrando espaço na terceira linha. O mesmo acontece para o FlowPane na vertical, mas aí ele colocaria os componentes na próxima coluna, seguindo o fluxo. Lembrando que isso é dinâmico, se você redimensiona o pai, ele realoca os componentes sempre seguindo o fluxo. Mas chega de papo e vamos ao código.

```
// 1
FlowPane flowPane = new FlowPane();
// 2
flowPane.setAlignment(Pos.CENTER_RIGHT);
for (int i = 0; i < 10; i++) {
    // 3
    flowPane.getChildren().add(new Label("Label " + i));
}
```

1. Criamos o nosso **FlowPane**. Falei que podemos a direção do fluxo. Isso pode ser feito direto no construtor usando a classe **Orientation**;
2. Aqui setamos o alinhamento: centro, centro para a direita, centro // para esquerda, topo para esquerda, etc. Brinque com outros valores; :)
3. Adicionando um componente ao nosso Painel. Isso é feito da forma simples já mostrada em seções anteriores anterior.

Bem, aqui está o resultado do código acima:

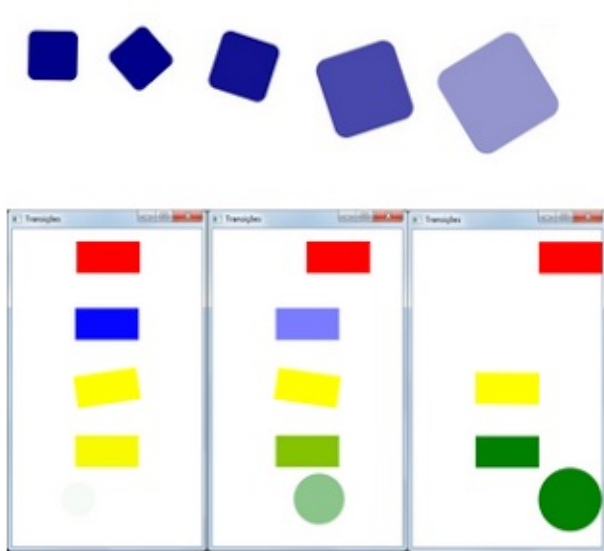


O código geral pode ser encontrado nesse mesmo repositório, lá vocês poderão ver os valores de *import* para as classes usadas.

## O que são transições?

Transições são um conjunto de classes da API do JavaFX que permitem que se crie animações. Com ela nós podemos modificar o valor de uma propriedade em um dado tempo. Por exemplo, você poderia fazer um objeto mover da posição X 0 até a posição X 150 em 2 segundos e isso iria criar uma animação.

Após configurar os parâmetros corretamente, você pode decidir quando a transição começa e até parar antes da mesma terminar. Ou seja, você pode, por exemplo, utilizar um botão para disparar o início da transição e outro para parar:



## Transições disponíveis na API do JavaFX

As classes de transição oferecidas no JavaFX ficam no pacote [javafx.animation](#) e as seguintes transições estão disponíveis para uso:

- **FadeTransition**: Muda a opacidade(tranparência) de um nó. O FadeTransition permite variar essa opacidade em um dado tempo. Por exemplo, você pode configurar uma transição para em 1 segundo variar a opacidade de um nó(um botão, uma imagem, uma figura geométrica, etc) de 0 até 1 e isso fará com que o mesmo passe de invisível até ser completamente visível no tempo de 1 segundo;
- **FillTransition**: Com a FillTransition nós podemos mudar a cor de preenchimento de um objeto. Se um objeto tem o preenchimento com a cor azul e você usa a transição para mudar a cor de azul para rosa em um segundo, isso significa que a cor de preenchimento do mesmo irá iniciar totalmente azul e no espaço de tempo de 1 segundo irá se tornar completamente rosa, dando um efeito bastante interessante;
- **RotateTransition**: Uma das propriedades mais comuns em um nó é a **rotate** (rotação). A RotateTransition mudará a rotação de um objeto de acordo com o tempo configurado.

Por exemplo, você pode rotacionar um objeto de 0° até 90° em 2 segundos e após iniciar essa transição, ele irá ter a rotação modificada gradualmente até que o tempo termine;

- **ScaleTransition**: A ScaleTransition serve para você modificar a escala de um objeto, seja a altura ou a largura. Em outras palavras, você pode fazer a escala de um objeto ser modificada de X para X<sup>2</sup> em um dado intervalo de tempo;
- **TranslateTransition**: Similarmente ao que foi falado na introdução desse artigo, a TranslateTransition irá modificar a posição X ou Y de um objeto em um dado intervalo de tempo;
- **StrokeTransition**: A StrokeTransition é muito similar à FillTransition, no entanto, nessa transição a modificação não é do preenchimento, mas sim da linha que contorna o mesmo;
- **PathTransition**: Embora complexa, a PathTransition é muito útil, pois permite que muitas ações sejam tomadas de acordo com o objeto **Path** passado;

A API do JavaFX também disponibiliza transições que permitem que várias transições sejam "tocadas" ao mesmo tempo, são elas:

- **SequentialTransition**: A SequentialTransition serve para adicionar diversas transições para serem "tocadas" de forma sequencial, ou seja, uma após a outra. Por exemplo, se adicionamos a transição t1, t2, t3 em uma SequentialTransition e tocar ela, t1 será ativada, em sequência será ativada t2 e por fim iremos fechar com t3; Se quisermos adicionar um tempo de espera entre as transições, podemos usar a **PauseTransition**, que é um tipo especial de transição cuja função é simplesmente "dar um tempo" antes da próxima transição da sequência;
- **ParallelTransition**: A ParallelTransition atua de forma semelhante à SequentialTransition, mas toca todas as transições em paralelo, sendo que o tempo de duração será a da transição mais longa.

**Lembre-se que todos os componentes em uma aplicação JavaFX herda de (é um) nó e todos os nós tem propriedades em comum como rotação, posição X e Y, opacidade, etc;**

## Transições na prática

Como muitos já devem ter imaginado, temos uma classe comum para todas as transições, a **classe Transition**. Abaixo temos uma explicação de seus principais métodos:

**play()**: Começa a "tocar" a transição. Se a transição estava pausada antes, chamar esse método fará com que a transição volte a tocar do ponto onde tinha parado. Similar ao play, temos o **playFromStart**, que começa uma transição desde o começo e **playFrom(Duration)**, onde podemos informar um ponto para a transição começar a tocar;

**pause():** Para a transição em um dado momento, sendo que ao chamar **play**, a transição irá começar no momento parado por **pause**;

**stop():** Para a transição por completo. A próxima vez que chamarmos **play** a transição irá iniciar do ponto inicial;

**setOnFinished(EventHandler<ActionEvent>):** Esse método é interessante! Com ele podemos informar uma ação que você deseja que seja feita assim que a transição termina de "tocar";

**setAutoReverse(boolean value):** Se chamarmos esse método e informar o valor **true** para **value**, iremos fazer com que a transição "volte", ou seja, se a transição for fazer um nó ir da posição X 0 para a posição 10, ao chamar **setAutoReverse(true)**, o mesmo irá até a posição 10 e irá voltar para a posição 0 invés de parar na posição X 10;

**setNode(Node):** Aqui é onde falamos qual será nosso alvo ou o nó que será manipulado por essa transição;

**Outros métodos:** Há outros métodos e atributos interessantes(acessados através de métodos) que você queria dar uma olhada. Para isso veja a documentação da classe que Transition veio, a [Animation](#).

## Exemplo de uso

Vamos agora explorar uma pequena aplicação que irá mostrar as funcionalidades das principais transições. Você pode encontrar o código da classe no arquivo `AprendendoTransicoes.java`. Veja nossa aplicação de exemplo:





Para criar as transições, usamos uma classe chamada **FabricaTransicao**.. Nessa classe há um método chamado *fazerTransicao* que, dado um [enum](#), duração e um nó, sempre cria uma nova transição:

```
public static class FabricaTransicao {

    public static enum Transicoes {
        FADE, TRANSLATE, SCALE, FILL, ROTATE
    }

    public static Transition fazerTransicao(Transicoes transicao, double duracaoSegundos, Node alvo) {
        Duration duracao = new Duration(duracaoSegundos * 1000);
        Transition t = null;

        switch (transicao) {
            case FADE:
                FadeTransition fadeTransition = new FadeTransition();
                fadeTransition.setFromValue(1);
                fadeTransition.setToValue(0);
                fadeTransition.setDuration(duracao);
                fadeTransition.setNode(alvo);
                t = fadeTransition;
                break;
            case FILL:
                FillTransition fillTransition = new FillTransition();
```

```
        fillTransition.setFromValue(Color.RED);
        fillTransition.setToValue(Color.DARKGREEN);
        fillTransition.setDuration(duracao);
        fillTransition.setShape((Shape) alvo);
        t = fillTransition;
        break;
    case ROTATE:
        RotateTransition rotateTransition = new RotateTransition();
        rotateTransition.setByAngle(360);
        rotateTransition.setDuration(duracao);
        rotateTransition.setNode(alvo);
        t = rotateTransition;
        break;
    case SCALE:
        ScaleTransition scaleTransition = new ScaleTransition();
        scaleTransition.setFromX(1);
        scaleTransition.setFromY(1);
        scaleTransition.setToX(4);
        scaleTransition.setToY(4);
        scaleTransition.setDuration(duracao);
        scaleTransition.setNode(alvo);
        t = scaleTransition;
        break;
    case TRANSLATE:
        TranslateTransition translateTransition = new TranslateTransition();
        translateTransition.setToX(600);
        translateTransition.setToY(250);
        translateTransition.setDuration(duracao);
        translateTransition.setNode(alvo);
        t = translateTransition;
        break;
    }
    t.setAutoReverse(true);
    t.setCycleCount(2);
    return t;
}
```

Isso obviamente não é "bonito", mas esse código não utiliza as melhores práticas de codificação, ele é somente para demonstrar as transições. As 5 transições aí criadas mostram como usamos os métodos mais básicos da transição. Primeiramente temos um atributo que é do tipo **Transition**, ou seja, uma classe abstrata, então de acordo com o valor do enum criamos uma transição concreta e atribuímos à [transição abstrata](#).

Note a repetição de uso dos métodos *setNode* e *setDuration*. Esses métodos estão em quase todas as transições, mas não em todos, por isso eles não puderam ser definidos na [classe abstrata](#). já nas seguintes duas linhas em destaque, temos dois métodos que são da classe abstrata(*setAutoReverse* e *setCycleCount*), logo não precisamos repetir a chamada

do mesmo para cada transição criada. Esses dois métodos simplesmente irão fazer com que a transição mude o atributo do nó e volte para o valor original. Para isso precisamos ter dois ciclos (ciclos são quantas vezes a transição será tocada).

Os botões na parte acima da aplicação são do tipo "ToggleButton" e são parte de um grupo. Esse grupo é gerado baseado no seguinte Enum que você pode ver declarado na classe **FabricaTransicao** mostrado acima. O Enum contém as transições que a Fábrica suporta e de acordo com os valores dele, a gente gera os botões. Veja o código abaixo:

```
private HBox criaPainelSuperior() {
    HBox hbTopo = new HBox(10);
    hbTopo.setSpacing(10);
    hbTopo.setAlignment(Pos.CENTER);
    Transicoes[] transicoes = Transicoes.values();
    // grupo para todas as transições
    botoesTransicao = new ToggleGroup();
    Stream.of(transicoes).map(t -> {
        ToggleButton tb = new ToggleButton(t.name());
        tb.setUserData(t);
        tb.setToggleGroup(botoesTransicao);
        return tb;
    }).forEach(hbTopo.getChildren()::add);
    botoesTransicao.getToggles().get(0).setSelected(true);
    return hbTopo;
}
```

Perceba que através do método *setUserData* nós adicionamos a cada botão o valor do enum que esse botão representa e usamos esse valor para fabricar nossa transição. Quando você clica no botão "Tocar", a "action" do mesmo será usar a fábrica para produzir uma transição de acordo com o botão selecionado, configurar o comportamento dos botões para que os mesmos não fiquem ativos quando a transição estiver tocando(ou que fiquem ativos só quando a transição estiver tocando). Veja a ação do botão Tocar:

```

btnTocar.setAction(e -> acaoTocar(btnParar, btnTocar, btnPausar, btnAjusta));
// mais
private void acaoTocar(final Button btnParar, final Button btnTocar, final Button btnPausar,
    final Button btnAjusta) {
    // antes de tocar, pegamos a mais nova transição selecionada
    Transicoes t = (Transicoes) botoesTransicao.getSelectedToggle().getUserData();
    transicaoAtual = FabricaTransicao.fazerTransicao(t, sldTempo.getValue(), alvo);
    // lógicas de habilitação dos botões, temos que setar todas as
    // vezes pq trocamos as transições
    btnParar.disableProperty().bind(transicaoAtual.statusProperty().isNotEqualTo(Status.RUNNING));
    btnTocar.disableProperty().bind(transicaoAtual.statusProperty().isEqualTo(Status.RUNNING));
    btnPausar.disableProperty().bind(transicaoAtual.statusProperty().isNotEqualTo(Status.RUNNING));
    btnAjusta.disableProperty().bind(transicaoAtual.statusProperty().isEqualTo(Status.RUNNING));
    sldTempo.disableProperty().bind(transicaoAtual.statusProperty().isEqualTo(Status.RUNNING));
    System.out.println("Tocando transição " + t);
    transicaoAtual.play();
}

```

Os outros botões irão controlar a transição (parar e pausar) e um serve para "ajustar" o texto quando a transição é parada no meio. Veja o método que é chamado quando clicamos nesse botão:

```

private void criaNoAlvo() {
    // configurar coisas do texto alvo...
    alvo = new Text("** Transições **");
    alvo.setFont(new Font(60));
    // efeitinsss
    Reflection efeito = new Reflection();
    efeito.setFraction(0.7);
    alvo.setEffect(efeito);
    alvo.setFill(Color.RED);
    raiz.setCenter(alvo);
}

```

O "slider" no canto direito será usado para que possamos selecionar o tempo total da transição. Ao criar a transição, pegamos o valor dele e enviamos para o método de criação. Veja a criação e o uso do slider:

```
// criando o slider  
sldTempo = new Slider(1, 10, 5);  
//... criando a transição  
transicaoAtual = FabricaTransicao.fazerTransicao(t, sldTempo.getValue(), alvo);
```

Terminamos aqui nossa seção sobre transições. Lembrando que a aplicação discutida acima foi demonstrada em um [vídeo no youtube](#).

# Desenhando com Canvas

Mostramos como desenhar [formas geométricas](#) com as classes que herdam de Shape no capítulo **Mostrando Imagens e Figuras Geométricas**. Essas formas, no entanto, ficam como nós separados no JavaFX. Como faremos se quisermos somente desenhar formas geométricas, textos e outras coisas livremente? Para isso temos o [Canvas](#), uma classe muito interessante para desenharmos livremente!

## Canvas

O canvas é simplesmente um nó que representa uma área para desenhar. Isso mesmo, nada mais do que isso. A mágica acontece com a class [GraphicsContext](#), que pode ser adquirida assim que criamos um Canvas. Com o *GraphicsContext* você pode desenhar formas geométricas, textos, imagens, mudar cor, aplicar efeito, etc. Veja um exemplo:



Para criar essa aplicação usamos o seguinte código:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.effect.BoxBlur;
import javafx.scene.effect.Reflection;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.stage.Stage;
```

```

public class DesenhandoComCanvas extends Application {

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

    @Override
    public void start(Stage palco) throws Exception {
        // O construtor do Canvas recebe a largura e a altura
        Canvas canvas = new Canvas(300, 300);
        // O objeto principal do Canvas é o GraphicsContext, que usamos para desenhar
        GraphicsContext ctx = canvas.getGraphicsContext2D();
        // estamos prontos para desenhar coisas! Vamos começar mudando a cor
        ctx.setFill(Color.RED);
        // podemos configurar uma fonte para os textos
        ctx.setFont(Font.font("Serif", FontWeight.BOLD, 25));
        // desenhando um texto, o primeiro param é o texto, os seguintes são a pos X e
        Y
        ctx.fillText("Olá Mundo Canvas", 15, 30);
        // podemos configurar efeitos e novamente trocar a cor
        ctx.setFill(Color.BLUE);
        ctx.setEffect(new BoxBlur());
        ctx.fillText("Olá Mundo Canvas", 15, 60);
        // agora vamos trocar o efeito, cor e desenhar um retângulo(x,y, largura, altu
        ra)
        ctx.setEffect(new Reflection());
        ctx.setFill(Color.AQUA);
        ctx.fillRect(15, 90, 240, 20);
        // ou um retângulo sem preenchimento
        ctx.setStroke(Color.GREEN);
        ctx.strokeRect(15, 135, 240, 30);
        // ou círculos (forma oval)
        ctx.setEffect(null);
        ctx.setFill(Color.BROWN);
        ctx.fillOval(15, 175, 90, 25);
        ctx.setStroke(Color.YELLOWGREEN);
        // ou formas ovais sem preenchimento
        ctx.strokeOval(160, 175, 90, 25);
        // ou até desenhar uns poligonos locos, usando diversos pontos X e Y
        double xs[] = {15, 30, 45, 60, 75, 90, 105, 120, 135, 150, 165,
            180, 195, 210, 225, 240, 255, 270};
        double ys[] = {205, 235, 250, 265, 205, 235, 205, 205, 235, 250,
            265, 205, 235, 205, 205, 235, 250, 205};
        ctx.setFill(Color.MAGENTA);
        ctx.setEffect(new Reflection());
        ctx.fillPolygon(xs, ys, 18);
        Scene cena = new Scene(new StackPane(canvas), 800, 600);
        palco.setTitle("Canvas");
        palco.setScene(cena);
        palco.show();
    }
}

```

```
}
```

Nesse momento, você deve estar se perguntando qual é a diferença do Canvas para uma aplicação JavaFX comum, já que podemos fazer tudo isso no JavaFX. Bem, o Canvas é uma tela de desenho, é ótimo para animações, jogos ou visualizações fantásticas. Já uma aplicação JavaFX, temos como maior objetivo objetos que se aninham hierarquicamente e são, no geral, estáticos, como essa página WEB. Desenhar essa página com o Canvas seria um buta dor de cabeça, já usar controles e outras características do JavaFX torna o serviço muito mais fácil.



## Introdução a gráficos com JavaFX

Os desenvolvedores de aplicações desktop e applets que vieram antes do JavaFX tinham muito trabalho criando gráficos.

JavaFX felizmente tem uma API pronta para criação de gráficos de diversos tipos. Nesse breve artigo vamos, através de exemplos, mostrar os gráficos mais simples do JavaFX.

### A API de gráficos

As classes da API de gráficos do JavaFX ficam no pacote [javafx.scene.chart](#). Para cada tipo de gráfico há uma classe (por exemplo, **PieChart**) e um tipo de objeto a ser populado com dados para serem exibidos nesse gráfico (como **PieChart.Data**). Lembrando que, assim como todo componente em uma aplicação JavaFX, gráfico [herda de nó](#).

Os principais gráficos são mostrados na imagem abaixo retirada do [tutorial da Oracle](#):



## O gráfico de Pizza

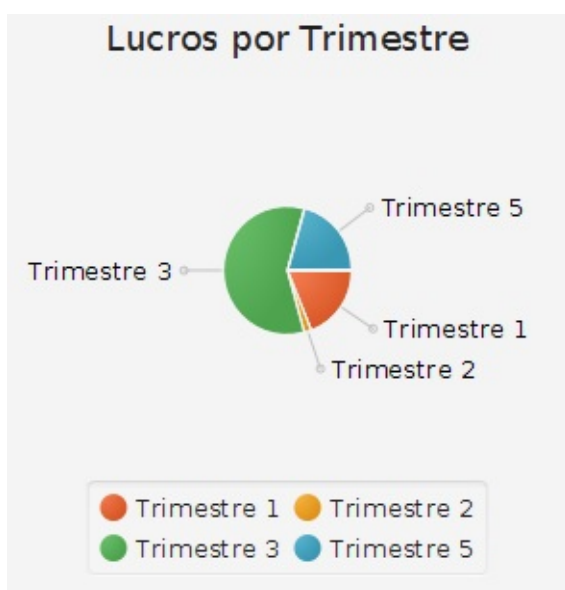
Esse é com certeza o gráfico mais fácil de usar :) Como dito, cada gráfico tem o seu tipo de dados. Os tipos de dados do gráfico de pizza é o **PieChart.Data** e esse tipo de dado contém uma String, que é o nome do dado, e um valor double, que é o valor desse dados. Os valores são somados e divididos pela quantidade de dados, assim descobrimos qual a fatia de cada um. Veja como é fácil criar um gráfico com dados fictícios de uma empresa com os ganhos de cada trimestre do ano:

```
PieChart graficoPizza = new PieChart();
graficoPizza.getData().addAll(new PieChart.Data("Trimestre 1", 11),
    new PieChart.Data("Trimestre 2", 1),
    new PieChart.Data("Trimestre 3", 34),
    new PieChart.Data("Trimestre 5", 12));
graficoPizza.setTitle("Lucros por Trimestre");
graficoPizza.setPrefSize(GRAFICO_LARGURA, GRAFICO_ALTURA);
```

Viu! O que temos acima é:

- **PieChart:** Um nó que representa o gráfico a ser desenhado;
- **PieChart.Data:** os dados do gráfico;

Por fim podemos adicionar o gráfico a uma cena e teremos o seguinte resultado:



## O gráfico de linha

O [gráfico de linha](#) é representado pela classe `LineChart` e não é difícil de ser usado também, no entanto, os dados já não são tão simples de criar. Ele é um subtipo de gráfico com plano cartesiano X e Y, ou seja, [XYChart](#) e o tipo de dados para esse gráfico são o [XYChart.Series](#). Mas o que vem a ser esse tipo de dados?

Esse tipo de dado consiste principalmente de valores para X, valores para Y de um dado grupo de dados. Um exemplo seria comparar o lucro de cada produto de uma empresa ao longo do ano.

O gráfico de linha, no entanto, também exige a definição dos eixos X e Y através da classe [Axis](#).

Bem, vejamos um exemplo de comparação de lucros por trimestre da empresa separado por produtos:

```
LineChart graficoLinha = new LineChart<>(
    new CategoryAxis(), new NumberAxis());
final String T1 = "T1";
final String T2 = "T2";
final String T3 = "T3";
final String T4 = "T4";

XYChart.Series prod1 = new XYChart.Series();
prod1.setName("Produto 1");

prod1.getData().add(new XYChart.Data(T1, 5));
prod1.getData().add(new XYChart.Data(T2, -2));
prod1.getData().add(new XYChart.Data(T3, 3));
prod1.getData().add(new XYChart.Data(T4, 15));

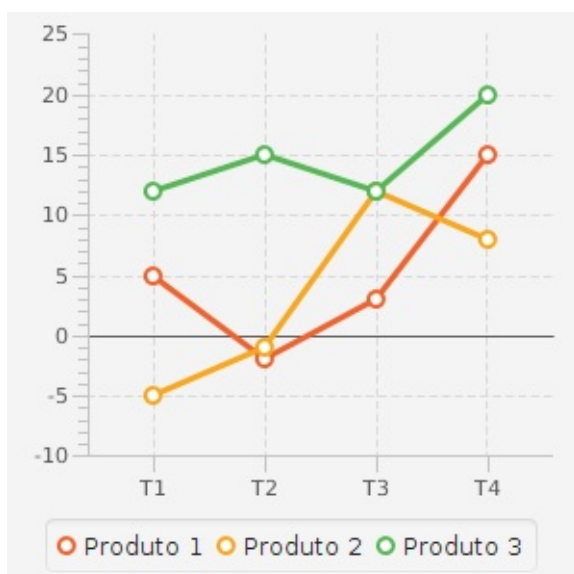
XYChart.Series prod2 = new XYChart.Series();
prod2.setName("Produto 2");

prod2.getData().add(new XYChart.Data(T1, -5));
prod2.getData().add(new XYChart.Data(T2, -1));
prod2.getData().add(new XYChart.Data(T3, 12));
prod2.getData().add(new XYChart.Data(T4, 8));

XYChart.Series prod3 = new XYChart.Series();
prod3.setName("Produto 3");

prod3.getData().add(new XYChart.Data(T1, 12));
prod3.getData().add(new XYChart.Data(T2, 15));
prod3.getData().add(new XYChart.Data(T3, 12));
prod3.getData().add(new XYChart.Data(T4, 20));
graficoLinha.getData().addAll(prod1, prod2, prod3);
graficoLinha.setPrefSize(GRAFICO_LARGURA, GRAFICO_ALTURA);
```

O resultado desse código é o seguinte:

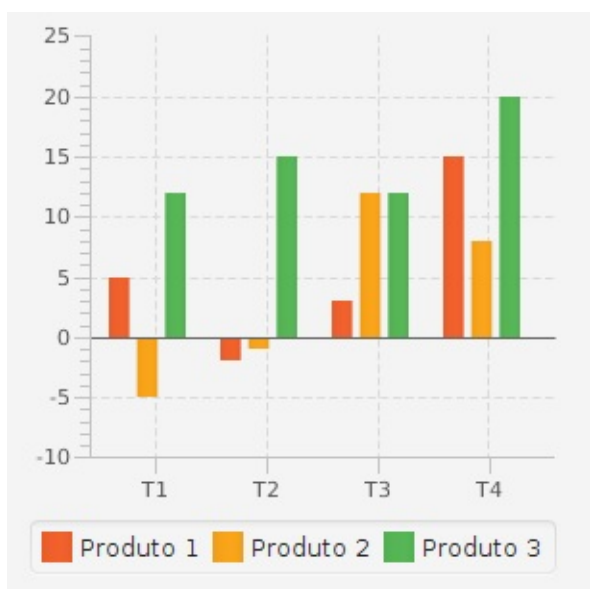


Notem que o código consiste de agrupamento de dados em tipo de categoria. No nosso caso agrupamos os dados do Produto 1 e a categoria foi determinada pelo trimestre.

## O gráfico de Barras

Muito similar ao gráfico de linha, temos o gráfico de barras. Esse gráfico usa o mesmo tipo de dados, mas representa de forma diferente: através de barras. Veja como o exemplo anterior, de linhas, usando o mesmo conjunto de dados, mas em um gráfico de barras:

```
BarChart graficoBarra = new BarChart<>(
    new CategoryAxis(), new NumberAxis());
// igualzinho ao feito para o gráfico de linha...
graficoBarra.getData().addAll(prod1, prod2, prod3);
graficoBarra.setPrefSize(GRAFICO_LARGURA, GRAFICO_ALTURA);
```



Claro que a API é muito avançada. Nesse pequeno livro vamos abordar o básico e cabê ao leitor explorar as APIs mencionadas.

## Abrindo e renderizando páginas WEB em uma aplicação JavaFX

Nesse capítulo vamos falar sobre como carregar páginas WEB dentro de sua aplicação JavaFX. Mais uma vez: essa é uma tarefa muito fácil em JavaFX!

### Um conto de duas classes: **WebView** e **WebEngine**

Vamos à parte teórica primeiramente e relembrar que tudo que é visual em uma aplicação JavaFX herda da classe `Node`, que [abordamos anteriormente](#). O `WebView` é a classe que você vai usar para mostrar o conteúdo de uma página e ele herda de `Node`, ou seja, é passível de [efeitos](#), [animações](#), [CSS](#), transformações, entre outros.

Já a classe `WebEngine` é onde manipulamos a [DOM](#) da página, executamos javascript, entre outros. É simplesmente o browser em si, baseado no [WebKit](#), o mesmo motor de navegadores famosos, como o Chrome.

A relação entre essas classes é simples: O `WebView` usa uma instância de `WebEngine` e cria a representação visual de um conteúdo HTML na sua aplicação JavaFX. Mas como usar os mesmo?

### Usando **WebView**

O uso de `WebView` consiste em três passos:

- Instanciar a classe `javafx.scene.web.WebView`;
- Acessar a `WebEngine` usando o método `getEngine_` e usar o método `_load` para carregar a página WEB;
- Adicionar o `WebView` ao `Scenograph` da sua aplicação.

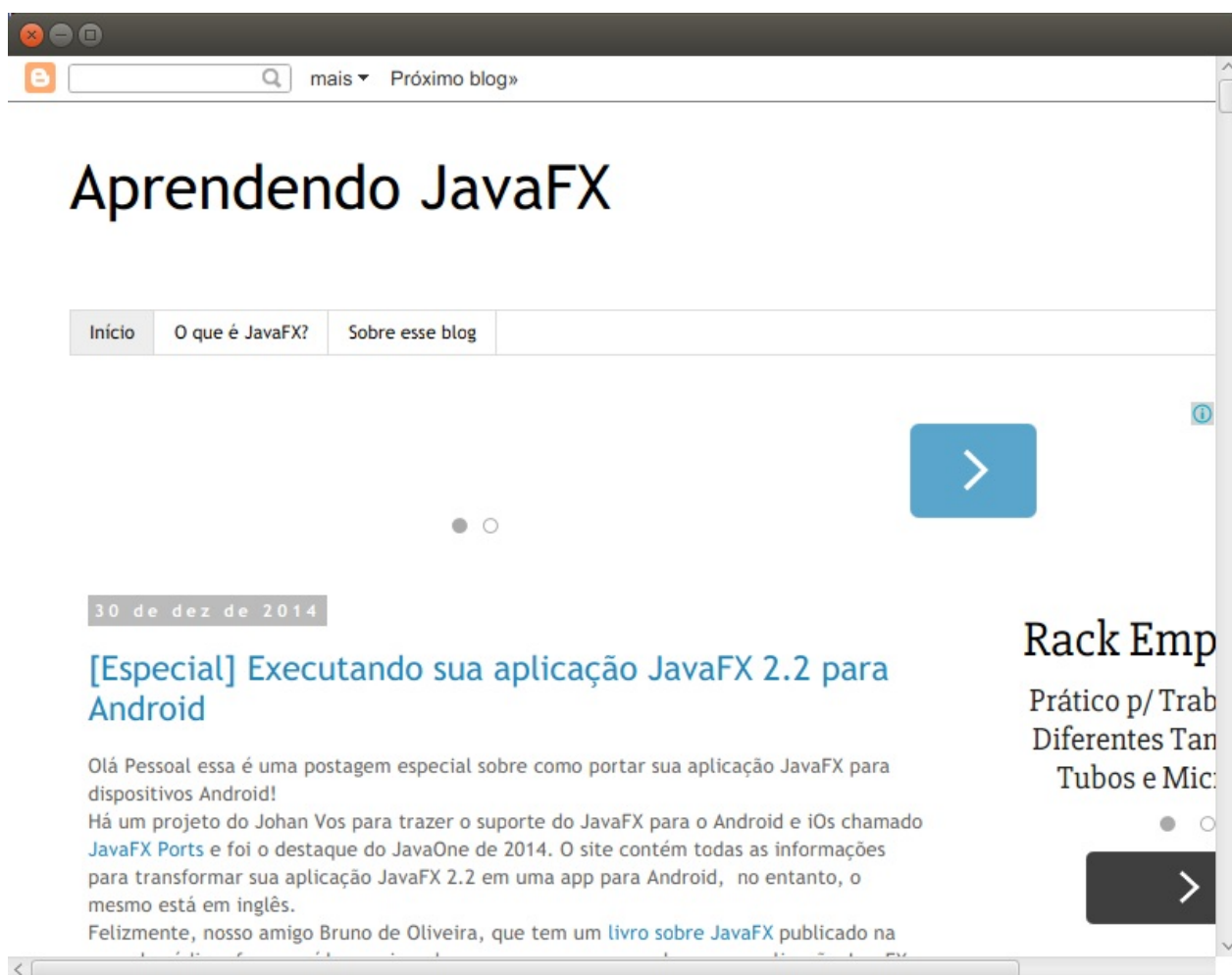
Em código, isso fica assim:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.web.WebView;
import javafx.scene.Scene;

public class SimplesWebView extends Application {

    @Override
    public void start(Stage s) {
        // O nó em sí, que será mostrado
        WebView webView = new WebView();
        // A engine é do tipo WebEngine
        webView.getEngine().load("http://aprendendo-javafx.blogspot.com");
        // criamos a cena e adicionamos no nosso stage
        s.setScene(new Scene(webView));
        // mostramos
        s.show();
    }
}
```

O resultado disso é uma aplicação simples que mostra o nosso blog:

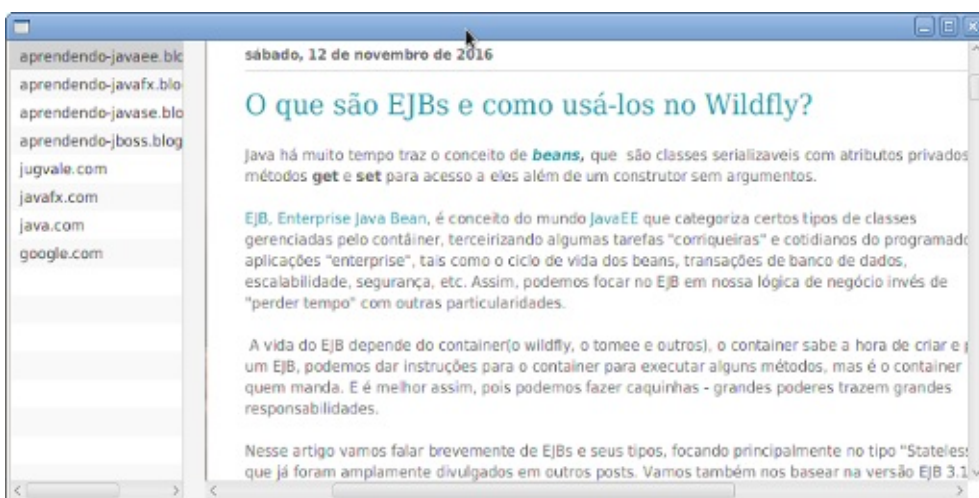


Exatamente, meus caros amigos. Foram 3 linhas de código e uma página WEB na sua app JavaFX.

## Uma aplicação um pouco mais elaborada

Agora vamos fazer uma aplicação que mostra alguns componentes JavaFX interagindo com a WEB Engine. Vamos fazer o básico, pois, como quase todos os componentes JavaFX, isso tomaria muitas páginas desse livro que deveria ser o arroz com feijão, não a feijoada, o filé, lasanha...

Nessa aplicação temos uma lista de URLs no lado esquerdo e quando o usuário seleciona uma, temos a mesma carregada no lado direito no nosso WEB View , vejam:



O funcionamento é simples: a ListView e o WebView ficam em um gerenciador de layout do tipo HBox. Quando o usuário seleciona um elemento na lista, pegamos a URL selecionada e carregamos na WebEngine usando o WebView. Quando a WebEngine está carregando a página, nós desabilitamos a lista. Fácil, não? Vejam o código inteiro:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.web.WebView;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.HBox;
import java.util.stream.Stream;

/**
 *
 * @author william
 */
public class NavegaSites extends Application {

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



```
}

final String[] sites = {
    "aprendendo-javaee.blogspot.com",
    "aprendendo-javafx.blogspot.com",
    "aprendendo-javase.blogspot.com",
    "aprendendo-jboss.blogspot.com",
    "jugvale.com",
    "javafx.com",
    "java.com",
    "google.com"
};

@Override
public void start(Stage s) {
    // criando uma lista visual de Strings, um webView e uma caixinha pra colocar
as coisas
    ListView<String> listaSites = new ListView<>();
    WebView webView = new WebView();
    HBox raiz = new HBox(20);

    // Para cada site no array adicionamos na lista visual
    Stream.of(sites).forEach(listaSites.getItems()::add);

    // Quando o usuário seleciona um item, carregamos a página
    listaSites.getSelectionModel().selectedItemProperty().addListener(
        (obs, o, n) -> {
            if(n != null) webView.getEngine().load("http://www." + n);
        });

    // a lista não permite seleção de página quando uma página está sendo carregada
    listaSites.disableProperty().bind(webView.getEngine().getLoadWorker().runningPrope
rty());

    // formalidades...
    raiz.getChildren().addAll(listaSites, webView);
    s.setScene(new Scene(raiz));
    s.show();
}
}
```

## Usando Tabelas com JavaFX: TableView


Um dos controles mais utilizados em aplicações cotidianas é a Tabela. Com controles que permitam visualizar dados em forma de tabelas podemos mostrar: lista de funcionários, de produtos, dados já ordenados em planilhas, informações diversas que vêm de algum servidor.

Nesse aspecto, estamos bem "servidos" com o [TableView](#) do JavaFX. A `TableView`, juntamente com a classe **TableColumn**, facilita visualizações de dados em uma aplicação JavaFX.

Nesse capítulo iremos mostrar o básico sobre criação de tabelas usando JavaFX.

### Visão geral da classe TableView

`TableView` é mais um controle de interface. Para começarmos falando dela, veja a seguinte imagem:



Nome	Idade	E-mail	
William	17	william@email.com	
Luana	32	luana@email.com	
Maria	12	maria@email.com	
João	15	joao@email.com	
Antônio	28	antonio@email.com	
Teles	17	teles@email.com	
Eduan	30	eduan@email.com	
Gabu	22	gabu@email.com	

Uma tabela tem colunas e linhas. Quando configuramos as colunas de uma [TableView](#), temos que também dizer como cada coluna vai representar os dados da tabela.

### Itens

Os dados devem ser do tipo da declaração da tabela. Por exemplo, se declaramos uma tabela usando:

```
TableView<Pessoa> tabela = new TableView<>();
```

A tabela só irá aceitar dados do tipo **Pessoa**:

```
List<Pessoa> pessoas = ....
```

```
tabela.setItems(FXCollections.observableArrayList(pessoas));
```

Caso você tente inserir dados de outro tipo, simplesmente você terá um **erro de compilação**.

## Colunas

As colunas são definidas pela classe `TableColumn` e devem saber lidar com cada linha da tabela. Isso é feito através de fábricas de células, que são classes que recebem cada item da tabela e retornam uma [célula da tabela](#)

. A fábrica você deve fornecer ou usar algumas colunas que já sabem fabricar as células de um tipo já conhecido. Veja um exemplo onde criamos colunas para as propriedades nome, idade e email de uma pessoa:

```
colunaNome.setCellValueFactory(new PropertyValueFactory<>("nome"));
colunaIdade.setCellValueFactory(new PropertyValueFactory<>("idade"));
colunaEmail.setCellValueFactory(new PropertyValueFactory<>("email"));
```

## Um exemplo completo de uso da TableView

Abaixo temos um código simples que mostra como mostrar uma lista de pessoas em uma TableView. Veja que não é nada sofisticado, mas com esse código, você já estará apto a construir suas próprias aplicações que usam tabelas.

```
package javafxpratico;

import java.util.Arrays;
import java.util.List;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.stage.Stage;

public class Tabela extends Application {

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

    @Override
    public void start(Stage primaryStage) {
```

```
List<Pessoa> pessoas = Arrays.asList(
    new Pessoa("William", 32, "william@email.com"),
    new Pessoa("Luana", 17, "luana@email.com"),
    new Pessoa("Maria", 12, "maria@email.com"),
    new Pessoa("João", 15, "joao@email.com"),
    new Pessoa("Antônio", 28, "antonio@email.com"),
    new Pessoa("Teles", 17, "teles@email.com"),
    new Pessoa("Eduan", 30, "eduan@email.com"),
    new Pessoa("Gabu", 22, "gabu@email.com"));

TableView<Pessoa> tabela = new TableView<>();
TableColumn<Pessoa, String> colunaNome = new TableColumn<>("Nome");
TableColumn<Pessoa, String> colunaIdade = new TableColumn<>("Idade");
TableColumn<Pessoa, String> colunaEmail = new TableColumn<>("E-mail");

colunaNome.setCellValueFactory(new PropertyValueFactory<>("nome"));
colunaIdade.setCellValueFactory(new PropertyValueFactory<>("idade"));
colunaEmail.setCellValueFactory(new PropertyValueFactory<>("email"));

tabela.setItems(FXCollections.observableArrayList(pessoas));
tabela.getColumns().addAll(colunaNome, colunaIdade, colunaEmail);
primaryStage.setScene(new Scene(tabela));
primaryStage.setWidth(250);
primaryStage.setHeight(300);
primaryStage.setTitle("Tabelas no JavaFX");
primaryStage.show();
}

public static class Pessoa {

    private String nome;
    private int idade;
    private String email;

    public Pessoa(String nome, int idade, String email) {
        this.nome = nome;
        this.idade = idade;
        this.email = email;
    }

    public String getNome() {
        return nome;
    }

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

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
```

```
        this.idade = idade;
    }

    public String getEmail() {
        return email;
    }

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

## Adicionando estilo à sua aplicação com CSS

Até o momento nesse livro fizemos aplicações e deixamos a "cara" delas a padrão que vem com o JavaFX, ou seja, não nos preocupamos em mudar a aparência das da nossa aplicação. Para fazer isso no JavaFX, não precisamos escrever código Java, mas sim conhecer [CSS \(Cascade Style Sheet\)](#), onde podemos declarativamente configurar a aparência de nossa aplicação.

## O que é e o que pode ser feito com CSS?

Com o CSS do javaFX é possível adicionar efeitos, mudar cores, dimensionar e completamente trocar a aparência da aplicação. É possível também definir a aparência quando o mouse fica sobre o elemento.

CSS é uma linguagem declarativa onde identificamos os elementos da nossa aplicação e em seguida definimos valores para as propriedades suportadas para aquele componente.

Essa tecnologia já é utilizada em páginas WEB e para o JavaFX todas as possibilidades de uso do CSS estão descritas em um

[completo guia de referência](#).

## Como referenciar os componentes do JavaFX no CSS?

Os componentes javafx podem ter várias classes CSS que referenciamos para usarmos na declaração do CSS. Assim, no código Java falamos qual a classe daquele componente e no CSS referenciamos a classe com **ponto**("."). Também podemos dar um id para nosso componente e referenciar a classe dele usando **cerquilha**("#"). Note que o ID deve ser único, já a classe pode ser aplicada a diversos nós.

## Como carregar CSS no JavaFX?

O CSS pode ficar em um arquivo separado e ser carregado na [Scene](#) da aplicação ou podemos adicionar estilo a qualquer classe que estenda de [nó](#) (todas as classes de uma cena no JavaFX) usando o método **setStyle**.

## Exemplo

Criamos a seguinte aplicação de exemplo para que você possa ver o que abordamos anteriormente na prática.



Note que a aplicação está diferente. Isso é por que eu carreguei o arquivo **app.css** na cena:

```
cena.getStylesheets().add("app.css");
```

```
.root{
    -fx-base: darkblue;
    -fx-background: lightblue;
}
.button {
    -fx-font-weight: bold;
    -fx-font-size: 15px;
}

.label {
    -fx-font-style: italic;
    -fx-font-size: 9px;
    -fx-text-fill: darkgreen;
}

.titulo {
    -fx-font-style: normal;
    -fx-font-weight: bolder;
    -fx-font-size: 30px;
    -fx-alignment: center;
}
```

Também temos conteúdos para o label **IbITitulo** que têm a classe **titulo**, que referenciamos para modificar o estilo:

No código Java:

```
IbITitulo.getStyleClass().add("titulo");
```

No CSS você pode ver acima a classe *.titulo*.

Notem que alguns componentes já tem uma classe usado pelo próprio JavaFX, mas que você pode modificar como quiser. No lado esquerdo podemos adicionar o CSS que queremos e ao apertar o botão, esse CSS é aplicação no Text do lado direito.

```
btnAplicar.setOnAction( event -> {  
    txtAlvo.setStyle(txtCSS.getText());  
});
```

Por fim, a aplicação acima foi construída com o seguinte código:

```
package javafxpratico;  
import javafx.application.Application;  
import javafx.geometry.Pos;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.control.TextArea;  
import javafx.scene.layout.BorderPane;  
import javafx.scene.layout.HBox;  
import javafx.scene.layout.VBox;  
import javafx.scene.text.Text;  
import javafx.stage.Stage;  
  
public class CssComJavaFX extends Application {  
  
    String txt = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. "  
        + "Nunc porta erat id lectus interdum, a pharetra est luctus. "  
        + "Nulla interdum convallis molestie. In hac habitasse platea "  
        + "dictumst. Ut ullamcorper ultricies viverra. Quisque blandit "  
        + "libero in ante sagittis sagittis. Ut gravida nibh quis justo "  
        + "sodales rutrum. Fusce euismod diam diam, vitae vulputate urna "  
        + "placerat vel. ";  
  
    public void start(Stage s) {  
        // declarações  
        BorderPane raiz = new BorderPane();  
        HBox pnlCentro = new HBox(50);  
        VBox vbEsquerda = new VBox(10);  
        VBox vbDireita = new VBox(10);  
        Button btnAplicar = new Button("Aplicar CSS");  
        TextArea txtCSS = new TextArea();  
        Text txtAlvo = new Text(txt);  
        Label lblTitulo = new Label("Testando CSS");  
        Scene cena = new Scene(raiz, 800, 250);  
  
        // configurando Layout e adicionando componentes  
        vbEsquerda.getChildren().addAll(new Label("Entre o CSS aqui"), txtCSS);  
        vbDireita.getChildren().addAll(new Label("O texto Alvo"), txtAlvo);  
  
        pnlCentro.getChildren().addAll(vbEsquerda, btnAplicar, vbDireita);  
        pnlCentro.setAlignment(Pos.CENTER);  
  
        raiz.setCenter(pnlCentro);  
        raiz.setTop(lblTitulo);  
        BorderPane.setAlignment(lblTitulo, Pos.CENTER);
```



```

txtCSS.setMinWidth(350);
txtAlvo.setWrappingWidth(220);
btnAplicar.setMinWidth(120);
btnAplicar.setOnAction( event -> {
    txtAlvo.setStyle(txtCSS.getText());
});
// configuramos classes para os nossos labels especiais
lblTitulo.getStyleClass().add("titulo");
// informamos o arquivo principal de CSS
cena.getStylesheets().add("app.css");
s.setScene(cena);
s.setTitle("Teste de CSS do JavaFX");
s.show();
}
}

```

Veja como ficaria se aplicarmos um CSS específico:



## Introdução ao FXML: Criando interfaces em JavaFX usando XML e o Scene Builder

Até agora utilizamos código Java na criação das [interfaces gráficas\(GUI\)](#). O uso de Java pode parecer atrativo a primeira vista, no entanto, o código pode se tornar complexo e de difícil manutenção, mas não é só em Java que podemos criar as telas.

O uso de [XML](#) é possível no JavaFX para que possamos criar a GUI de uma aplicação e então referenciar o mesmo no código Java. Os XMLs que contém componentes JavaFX são chamados de FXML.

### O modo Java de se fazer interfaces

Digamos que temos uma aplicação que contém um campo de texto, um botão e um campo de texto de leitura com uma saudação ao nome entrado. O código de interface para esse simples programa ficaria como segue:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.effect.Effect;
import javafx.scene.effect.Reflection;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class DigaOlaComJavaFX extends Application {

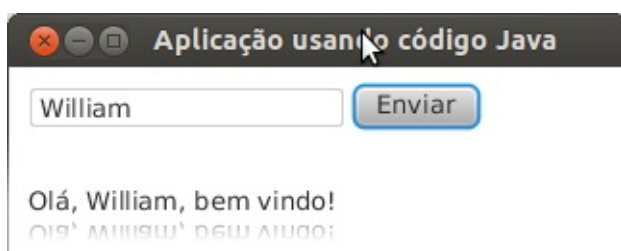
    public static void main(String[] args) {
        launch();
    }

    @Override
    public void start(Stage palco) throws Exception {
        final Effect r = new Reflection();
        final VBox raiz = new VBox(30);
        final HBox hbTopo = new HBox(5);
        final TextField txtNome = new TextField();
        final Button btnAcao = new Button("Enviar");
        final Label lblMensagem = new Label();
        raiz.setTranslateX(10);
        raiz.setTranslateY(10);
        lblMensagem.setText("Digite seu nome e clique no botão");
        hbTopo.getChildren().addAll(txtNome, btnAcao);
        raiz.getChildren().addAll(hbTopo, lblMensagem);
        lblMensagem.setEffect(r);
        Scene cena = new Scene(raiz, 250, 100);
        palco.setTitle("Aplicação usando código Java");
        palco.setScene(cena);
        palco.show();

        btnAcao.setOnAction(e -> lblMensagem.setText("Olá, " + txtNome.getText() + ",
bem vindo!"));
    }
}

```

Essa é uma aplicação simples, temos poucos componentes e um leiaute muito fácil de se montar. Mas e quando temos aplicações cheia de controles e leiaute complexo?



## Usando FXML

Utilizar XML em interface gráfica não é uma novidade. O [Pivot](#), um framework Java para criação de aplicações desktop, e o [Adobe Flex](#) também utilizam esse conceito. A grande vantagem do uso dessa linguagem declarativa está na possibilidade de usar uma ferramenta para a geração da interface e a possibilidade de modificar o XML sem ter que recompilar a aplicação inteira.

O JavaFX traz suporte ao FXML, uma forma de declarar todos os elementos de interface sem escrever uma linha de código de Java. Veja como o programa que demonstramos anteriormente ficaria com FXML:

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

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.effect.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.paint.*?>

<AnchorPane id="AnchorPane" prefHeight="88.0" prefWidth="306.0" xmlns:fx="http://javafx.com/fxml"
    fx:controller="main.ControleAplicacao">
    <children>
        <TextField layoutX="14.0" layoutY="14.0" prefWidth="200.0" fx:id="txtNome" />
        <Button layoutX="226.0" layoutY="15.0" mnemonicParsing="false"
            onAction="#atualizaMensagem" text="Clique!" />
        <Label fx:id="lblMensagem" layoutX="14.0" layoutY="44.0" prefHeight="21.0"
            prefWidth="264.0" text="Digite seu nome e clique no botão">
            <effect>
                <Reflection fraction="0.9" />
            </effect>
        </Label>
    </children>
</AnchorPane>
```

Claro que o FXML também pode ser complexo e gigante, já falamos mais sobre isso. No código Java acima é fácil adicionar ações ao botão e manipular os elementos da interface. Mas como fazer com FXML?

## O controller de um FXML

A lógica e o tratamento de eventos em uma aplicação JavaFX que usa FXML fica em uma classe que referenciamos no próprio FXML, essa classe é chamada de controller. Nessa classe podemos referenciar os elementos declarados no FXML para manipulação deles, lá

também declaramos o método que irá tratar os eventos. Veja como fica o FXML completo e pronto para uso com o nosso controller.

O código do controller [ControleAplicacao](#) está mais abaixo e ele tem declarado campos correspondentes ao nosso [TextField](#) e [Label](#). Notem que usamos a anotação [@FXML](#) no campo, é ela que informa o nome do mesmo e o JavaFX injeta o campo declarado lá no XML para nosso uso aqui! Veja o método *atualizaMensagem* também, é ele que é invocado quando clicamos no botão.

```
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;

public class ControleAplicacao {

    @FXML
    Label lblMensagem;

    @FXML
    TextField txtNome;

    public void atualizaMensagem() {
        lblMensagem.setText("Olá, " + txtNome.getText() + ", bem vindo!");
    }
}
```

## Executando sua aplicação com FXML

Nesse momento, nossa aplicação está funcionando. Só precisamos carregar o FXML e isso é feito através da classe [FXMLLoader](#), que lê o nosso arquivo .fxml e retorna um objeto do tipo [Parent](#). Com ele é possível configurar a raiz da cena da aplicação, veja:

```

public class DigaOlaComFXML extends Application {

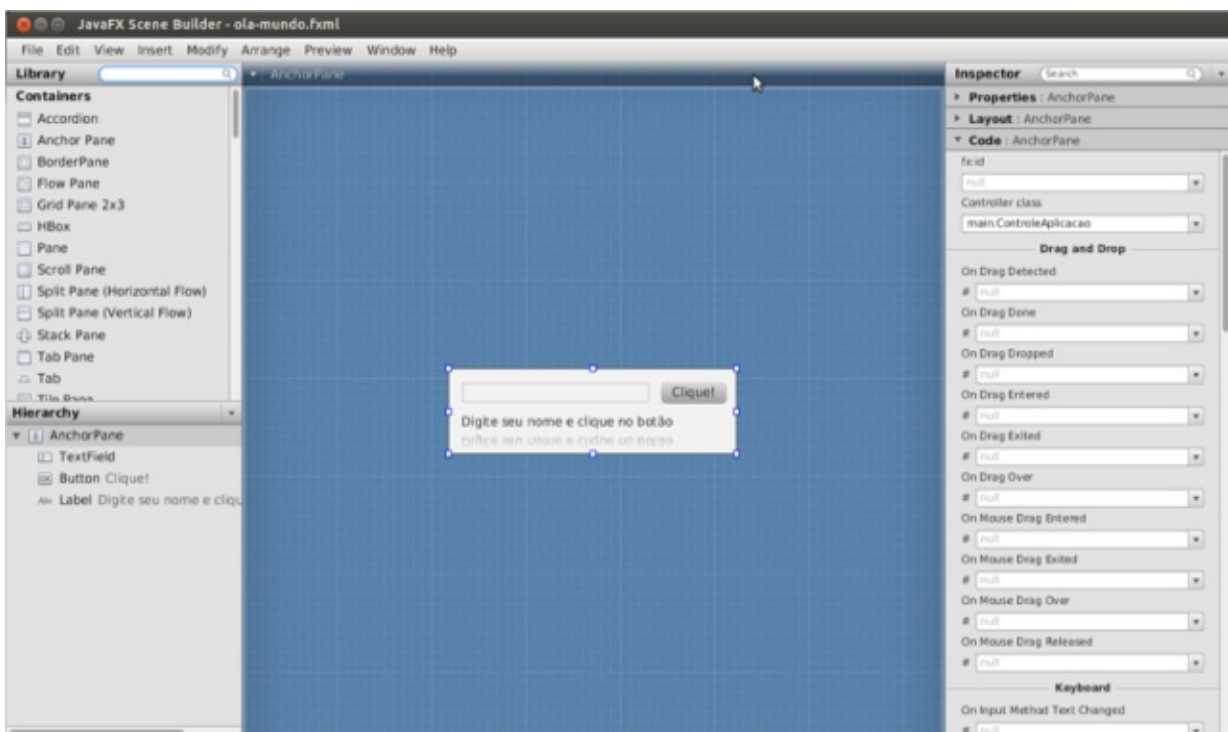
    public static void main(String[] args) {
        launch();
    }

    @Override
    public void start(Stage palco) throws Exception {
        URL arquivoFXML = getClass().getResource(
            "./ola-mundo.fxml");
        Parent fxmlParent = (Parent) FXMLLoader.load(arquivoFXML);
        palco.setScene(new Scene(fxmlParent, 300, 100));
        palco.setTitle("Olá mundo com FXML");
        palco.show();
    }
}

```

## Criando interfaces visualmente com o Scene Builder

Falamos que com Java o projeto poderia ficar muito complexo, mas você pode perceber que o XML não é lá aquelas facilidades... Felizmente temos uma ferramenta onde podemos arrastar e soltar componentes para desenhar nossa tela. Em seguida, exportamos um arquivo .fxml para uso em nossa aplicação JavaFX! Veja uma imagem da ferramenta.



Essa ótima ferramenta é mantida pela empresa Gluon e pode ser baixada no seguinte endereço: <http://gluonhq.com/labs/scene-builder>



## Um CRUD com JavaFX

Todo programador já fez(ou vai fazer) um **CRUD** na vida e é, no geral, a forma que aprendemos conceitos básicos de uma tecnologia. Nesse capítulo vamos mostrar um simples CRUD em JavaFX para explorar algumas características da tecnologia. O objetivo é exercitar o que aprendemos para podemos fechar o livro. Sim, senhores e senhoras, estamos chegando n final.

### Definição de CRUD

CRUD é uma sigla que vem das operações básicas que podemos fazer um algo armazenado em uma fonte de dados:

**Create:** Criar uma nova instância de algo na fonte dos dados;

**Retrieve:** Trazer os dados já armazenados;

**Update:** Atualizar dados já armazenados;

**Delete:** Apagar.

### Onde armazenar os dados?

A fonte de dados pode ser um banco de dados(MySQL, MariaDB, Postgres, etc), um arquivo(arquivos em diversos formatos, como CSV), um Web Service (que após chamado, salva os dados em algum lugar) ou até mesmo a própria memória dinâmica do computador(sendo que os dados se perdem quando a aplicação é fechada). Aqui vamos usar um arquivo CSV, uma forma simples de armazenar objetos em um arquivo de texto.

O motivo de não usarmos um banco de dados tradicional é que o foco do artigo é mostrar JavaFX. Um banco de dados comum implica em termos que falar de SQL, conexão com o banco, select, etc, isso vai fugir do foco.

### Nosso CRUD

Nossa aplicação simples vai ser um CRUD de Contas. Sim, bills, contas! Pagamos todo mês contas de luz, água, gás, faturas, etc, etc, argh. Para representar a conta, criamos um objeto Java chamado Conta com três atributos: id (gerado automaticamente para controle interno), concessionária, descrição e data de vencimento. É isso, simples, não? Em breve mostramos o código dessa classe, mas agora que conhecemos os campos, veja a telinha que modelamos usando o SceneBuilder que gerou um FXML (se não sabe o que é isso, veja esse [artigo sobre FXML](#), ou veja o capítulo Usando FXML).



Concessionaria	Descrição	Data vencimento
Bandeirantes	Conta de luz	Thu Mar 03 00:00:00 BRT 2016
Sabesp	água	Fri Mar 04 00:00:00 BRT 2016
bostacard	minhas correntes	Sat Mar 12 00:00:00 BRT 2016

Começamos com uma [tabela](#) com três colunas representando os campos, após a tabela temos os campos de texto para entrada do nome, descrição e um campo para entrada de data do tipo [DatePicker](#), que não foi abordado nesse livro, e por fim os botões de ações.

A lógica da aplicação é a seguinte:

- A tabela tem ID **tblContas** e três colunas: **clConc**, **clDesc** e **clVenc**. Elas são populadas com os dados de um objeto do tipo Conta;
- Os campos de texto e o campo de data tem um (**txtConc**, **txtDesc**, **dpVenc**) serão injetados no controller para que possamos saber o valor que o usuário entrou;
- Cada um dos botões ação:
  - **salvar**: salva o objeto de acordo com a informação entrada pelo usuário. Não está habilitado quando um campo está selecionado na tabela;
  - **atualizar**: Só está habilitado quando selecionamos uma linha da tabela e permite atualizar os dados dessa linha (os campos de entrada de dados vão ser atualizados com o valor selecionado para serem modificados pelo usuário);
  - **apagar**: apaga a linha selecionada;
  - **limpar**: limpa o campo selecionado atualmente.

As operações e os elementos da tela ficam na classe **ContasController**. Código que veremos já já.

## Fazendo as operações com o banco de dados

As operações em si com o banco ficam na interface `ContasService`. Ela contém os métodos **salvar**, que recebe uma instância de conta a ser salva, **atualizar**, que recebe a conta já salva para ser atualizada, **apagar**, que apaga uma conta e **buscarTodos**,

que retorna todas as contas selecionadas. É nessa classe que fazemos as operações.

Todo o código poderia ficar dentro do controller, MAS ISSO É COISA FEIA, temos que definir os métodos em uma interface e, vejam que interessante, usando a capacidade do Java 8 de definir métodos padrões, criamos um método **getInstance** para retornar a implementação que queremos dessa interface. Assim, criamos a classe `ContasCSVService`, que é uma implementação da interface, e retornamos uma nova instância nesse método! Podemos, obviamente, criar uma interface, por exemplo **ContasBDService** que faria a mesma coisa, mas que invés de usar um arquivo CSV, se comunica com um banco de dados, a mesma ideia poderia ser aplicada para um arquivo XLS, como **ContasXLSService**, e por aí vai. O código do controller, que os métodos da interface, não iria sofrer nenhuma modificação, pois só precisaríamos trocar a instância de **ContasService** retornada no método **getNewInstance**! (claro que com [CDI](#) e outras tecnologias, sequer criar manualmente precisaríamos). Veja a nossa interface:

```
public interface ContasService {  
  
    // CREATE  
    public void salvar(Conta conta);  
  
    // RETRIEVE  
    public List<Conta> buscarTodas();  
  
    public Conta buscaPorId(int id);  
  
    // DELETE  
    public void apagar(int id);  
  
    // UPDATE  
    public void atualizar(Conta conta);  
  
    // retorna a implementação que escolhemos - no nosso caso o ContasCSVService,  
    // mas poderia ser outro, como ContasDBService...  
    public static ContasService getNewInstance() {  
        return new ContasCSVService();  
    }  
}
```

## VAMOS AO CÓDIGO!

Agora, depois desse mooonnnntteee de papo, vamos ao código. Claro que se você leu acima com atenção, vai ser muito simples de entender tudo, mas mesmo assim o código está comentado na medida do possível.

Não vou colocar o todo código nesse capítulo, pois já está muito extenso, mas veja o código da classe `Conta`, da interface **`ContasService`** e da classe **`ContasController`**. Note que você pode encontrar o código no anexo ***`javafx-pratico.zip`*** \_no pacote **`javafxpratico.crud_`**.

```
import java.util.Date;

/**
 *
 * Nossa classe de modelo do objeto que "sofrerá" as operações de CRUD
 * @author wsiqueir
 *
 */
public class Conta {

    private int id;
    private String concessionaria;
    private String descricao;
    private Date dataVencimento;

    // gets e sets omitidos..

}
```

```
import java.net.URL;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;
import java.util.ResourceBundle;

import javafx.beans.binding.BooleanBinding;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.DatePicker;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;

/**
 *
 * O controller da aplicação, onde a mágica acontece
 * @author wsiqueir
 *
 */
```

```
public class ContasController implements Initializable {

    @FXML
    private TableView<Conta> tblContas;
    @FXML
    private TableColumn<Conta, String> clConsc;
    @FXML
    private TableColumn<Conta, String> clDesc;
    @FXML
    private TableColumn<Conta, Date> clVenc;
    @FXML
    private TextField txtConsc;
    @FXML
    private TextField txtDesc;
    @FXML
    private DatePicker dpVencimento;
    @FXML
    private Button btnSalvar;
    @FXML
    private Button btnAtualizar;
    @FXML
    private Button btnApagar;
    @FXML
    private Button btnLimpart;

    private ContasService service;

    //     Esse método é chamado ao inicializar a aplicação, igual um construtor.
    // Ele vem da interface Initializable
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        service = ContasService.getNewInstance();
        configuraColunas();
        configuraBindings();
        atualizaDadosTabela();
    }

    // métodos públicos chamados quando o botão é clicado

    public void salvar() {
        Conta c = new Conta();
        pegaValores(c);
        service.salvar(c);
        atualizaDadosTabela();
    }

    public void atualizar() {
        Conta c = tblContas.getSelectionModel().getSelectedItem();
        pegaValores(c);
        service.atualizar(c);
        atualizaDadosTabela();
    }
}
```

```
public void apagar() {
    Conta c = tblContas.getSelectionModel().getSelectedItem();
    service.apagar(c.getId());
    atualizaDadosTabela();
}

public void limpar() {
    tblContas.getSelectionModel().select(null);
    txtConsc.setText("");
    txtDesc.setText("");
    dpVencimento.setValue(null);
}

// métodos privados do controller

// pega os valores entrados pelo usuário e adiciona no objeto conta
private void pegaValores(Conta c) {
    c.setConcessionaria(txtConsc.getText());
    c.setDescricao(txtDesc.getText());
    c.setDataVencimento(dataSelecionada());
}

// método utilitário para pega a data que foi selecionada (que usa o tipo novo do
// java 8 LocalDateTime)
private Date dataSelecionada() {
    LocalDateTime time = dpVencimento.getValue().atStartOfDay();
    return Date.from(time.atZone(ZoneId.systemDefault()).toInstant());
}

// chamado quando acontece alguma operação de atualização dos dados
private void atualizaDadosTabela() {
    tblContas.getItems().setAll(service.buscarTodas());
    limpar();
}

// configura as colunas para mostrar as propriedades da classe Conta
private void configuraColunas() {
    clConsc.setCellValueFactory(new PropertyValueFactory<>("concessionaria"));
    clDesc.setCellValueFactory(new PropertyValueFactory<>("descricao"));
    clVenc.setCellValueFactory(new PropertyValueFactory<>("dataVencimento"));
}

// configura a lógica da tela
private void configuraBindings() {
    // esse binding só é false quando os campos da tela estão preenchidos
    BooleanBinding camposPreenchidos = txtConsc.textProperty().isEmpty()
        .or(txtDesc.textProperty().isEmpty())
        .or(dpVencimento.valueProperty().isNull());
    // indica se há algo selecionado na tabela
    BooleanBinding algoSelecionado = tblContas.getSelectionModel().selectedItemPro
    perty().isNull();
    // alguns botões só são habilitados se algo foi selecionado na tabela
    btnApagar.disableProperty().bind(algoSelecionado);
}
```

```

        btnAtualizar.disableProperty().bind(algoSelecionado);
        btnLimpart.disableProperty().bind(algoSelecionado);
        // o botão salvar só é habilitado se as informações foram preenchidas e não te
m nada na tela
        btnSalvar.disableProperty().bind(algoSelecionado.not().or(camposPreenchidos));
        // quando algo é selecionado na tabela, preenchemos os campos de entrada com o
s valores para o
        // usuário editar
        tblContas.getSelectionModel().selectedItemProperty().addListener((b, o, n) ->
{
    if (n != null) {
        LocalDate data = null;
        data = n.getDataVencimento().toInstant()
            .atZone(ZoneId.systemDefault()).toLocalDate();
        txtConsc.setText(n.getConcessionaria());
        txtDesc.setText(n.getDescricao());
        dpVencimento.setValue(data);
    }
});
}
}

```

```

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;
import java.util.stream.Collectors;

/**
 *
 * Uma implementação do ContasService para lidar com arquivo CSV
 * @author wsiqueir
 *
 */
public class ContasCSVService implements ContasService {

    // divisor de colunas no arquivo
    private static final String SEPARADOR = ";";

    // o caminho para o arquivo deve ser selecionado aqui
    private static final Path ARQUIVO_SAIDA = Paths.get("./dados.csv");

    // os dados do arquivo
    private List<Conta> contas;

    // formato de data usado no arquivo

```

```
final SimpleDateFormat formatoData = new SimpleDateFormat("dd/MM/yyyy");

public ContasCSVService() {
    carregaDados();
}

@Override
public void salvar(Conta conta) {
    conta.setId(ultimoId() + 1);
    contas.add(conta);
    salvaDados();
}

@Override
public void atualizar(Conta conta) {
    Conta contaAntiga = buscaPorId(conta.getId());
    contaAntiga.setConcessionaria(conta.getConcessionaria());
    contaAntiga.setDataVencimento(conta.getDataVencimento());
    contaAntiga.setDescricao(conta.getDescricao());
    salvaDados();
}

@Override
public List<Conta> buscarTodas() {
    return contas;
}

@Override
public void apagar(int id) {
    Conta conta = buscaPorId(id);
    contas.remove(conta);
    salvaDados();
}

public Conta buscaPorId(int id) {
    return contas.stream().filter(c -> c.getId() == id).findFirst()
        .orElseThrow(() -> new Error("Conta não encontrada"));
}

// salva a lista de dados no arquivo, gerando um novo CSV e escrevendo o arquivo c
// ompletamente
private void salvaDados() {
    StringBuffer sb = new StringBuffer();
    for (Conta c : contas) {
        String linha = criaLinha(c);
        sb.append(linha);
        sb.append(System.getProperty("line.separator"));
    }
    try {
        Files.delete(ARQUIVO_SAIDA);
        Files.write(ARQUIVO_SAIDA, sb.toString().getBytes());
    } catch (IOException e) {
```

```
        e.printStackTrace();
        System.exit(0);
    }
}

// o ID mais alto é retornado aqui para continuarmos contando os IDs
private int ultimoId() {
    return contas.stream().mapToInt(Conta::getId).max().orElse(0);
}

// carrega os dados do arquivo para a lista contas
private void carregaDados() {
    try {
        if(!Files.exists(ARQUIVO_SAIDA)) {
            Files.createFile(ARQUIVO_SAIDA);
        }
        contas = Files.lines(ARQUIVO_SAIDA).map(this::leLinha).collect(Collectors.toList());
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
    }
}

// transforma uma linha do CSV para o tipo Conta
private Conta leLinha(String linha) {
    String colunas[] = linha.split(SEPARADOR);
    int id = Integer.parseInt(colunas[0]);
    Date dataVencimento = null;
    try {
        dataVencimento = formatoData.parse(colunas[3]);
    } catch (ParseException e) {
        e.printStackTrace();
        System.exit(0);
    }
    Conta conta = new Conta();
    conta.setId(id);
    conta.setConcessionaria(colunas[1]);
    conta.setDescricao(colunas[2]);
    conta.setDataVencimento(dataVencimento);
    return conta;
}

// transforma um objeto conta em um arquivo CSV
private String criaLinha(Conta c) {
    String dataStr = formatoData.format(c.getDataVencimento());
    String idStr = String.valueOf(c.getId());
    String linha = String.join(SEPARADOR, idStr, c.getConcessionaria(), c.getDescricao(),
                                dataStr);
    return linha;
}
```



```
}
```

Esse foi nosso projeto de exemplo. Não há nada demais nele, não com o código.

## Conclusão

Chegamos ao fim do livro! Se você, nesse ponto, consegue criar aplicações simples com JavaFX, se divertiu e está apto a repassar conhecimento desse livro para frente, então meu objetivo foi concluído!

Há a ideia de novos livros com tópicos mais avançados. Mas só o tempo dirá se teremos ou não!

Até lá, fique de olho em ***[aprendendo-javafx.blogspot.com](http://aprendendo-javafx.blogspot.com)***