

JavaFX Studies/ Aula08: Nodes do JavaFX

ScenBuilder: a função nos nodes de FitToParent faz com que o Node preencha o seu rootNode:

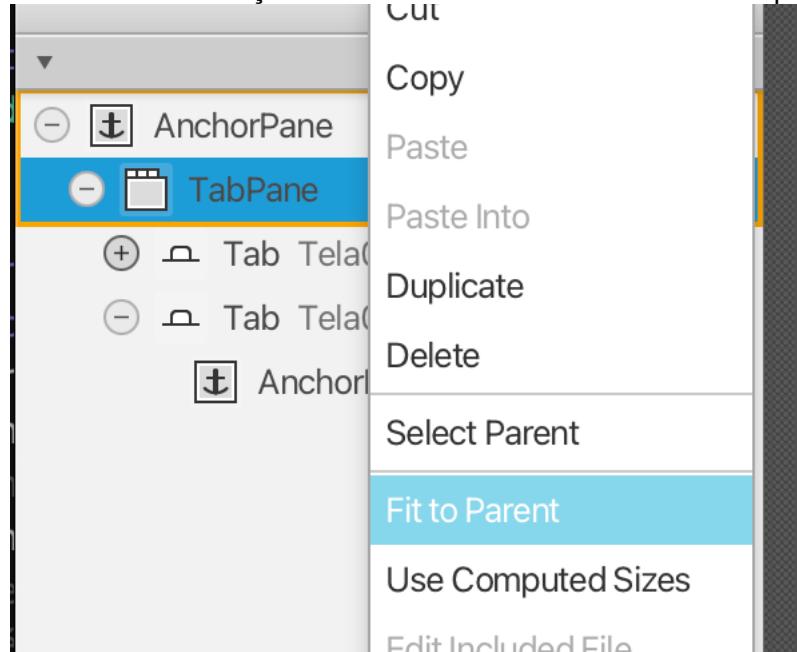
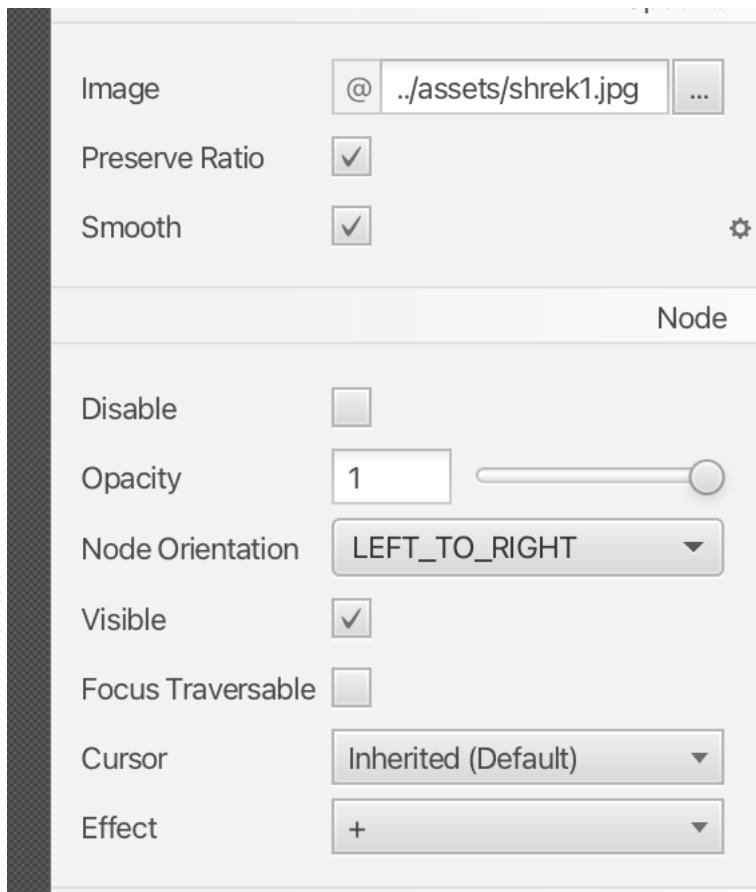
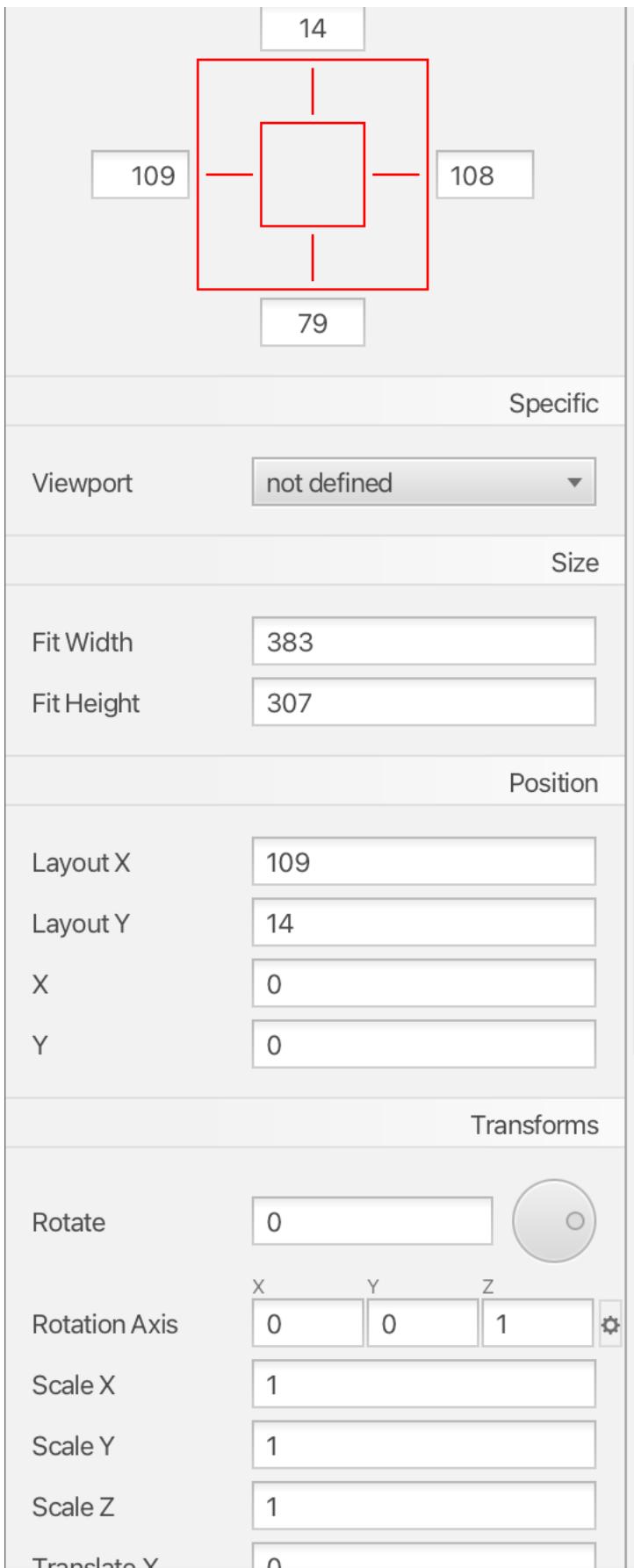


Image View: O image view é um node utilizado para “pintar” imagens carregadas pela classe Image.

Primeiramente vamos entender alguns básicos sobre o ImageView no SceneBuilder:



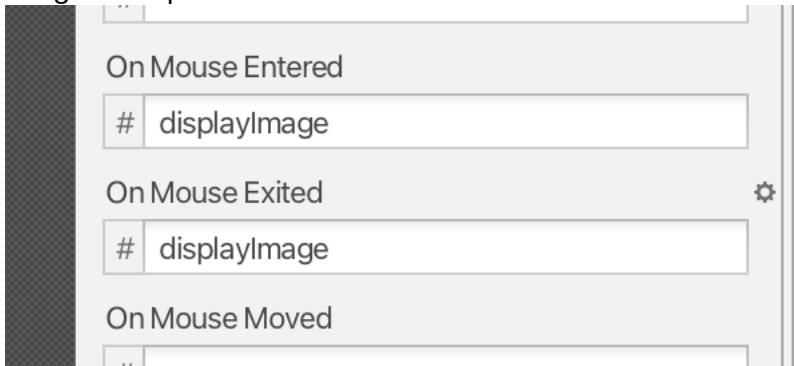
- Em Image declaramos a imagem inicial do nosso image View
- Preserve Ratio: se a imagem preserva ou não o seu tamanho padrão.
- Opacity: Muda a Opacidade da sua imagem.
- Cursor: o cursor que aparece ao passar o mouse em cima dela;
- Effect: Adiciona um efeito a imagem



- Widht e Height: são a altura e largura o ImageView
- Layout X e Layout Y: é a posição do ImageView em relação ao seu root
- Rotate: Rotaciona a imagem

- Scale: a escala da imagem em relação a largura e altura.

E por ultimo temos a parte de code, e o mais importante são as ações executadas pelo ImageView quando entramos e saímos com o mouse da imagem e etc:



Agora iremos ver um exemplo de código para troca de imagens em um ImageView:

```
public class Controller01 {

    @FXML
    Button button;
    @FXML
    ImageView imageView;

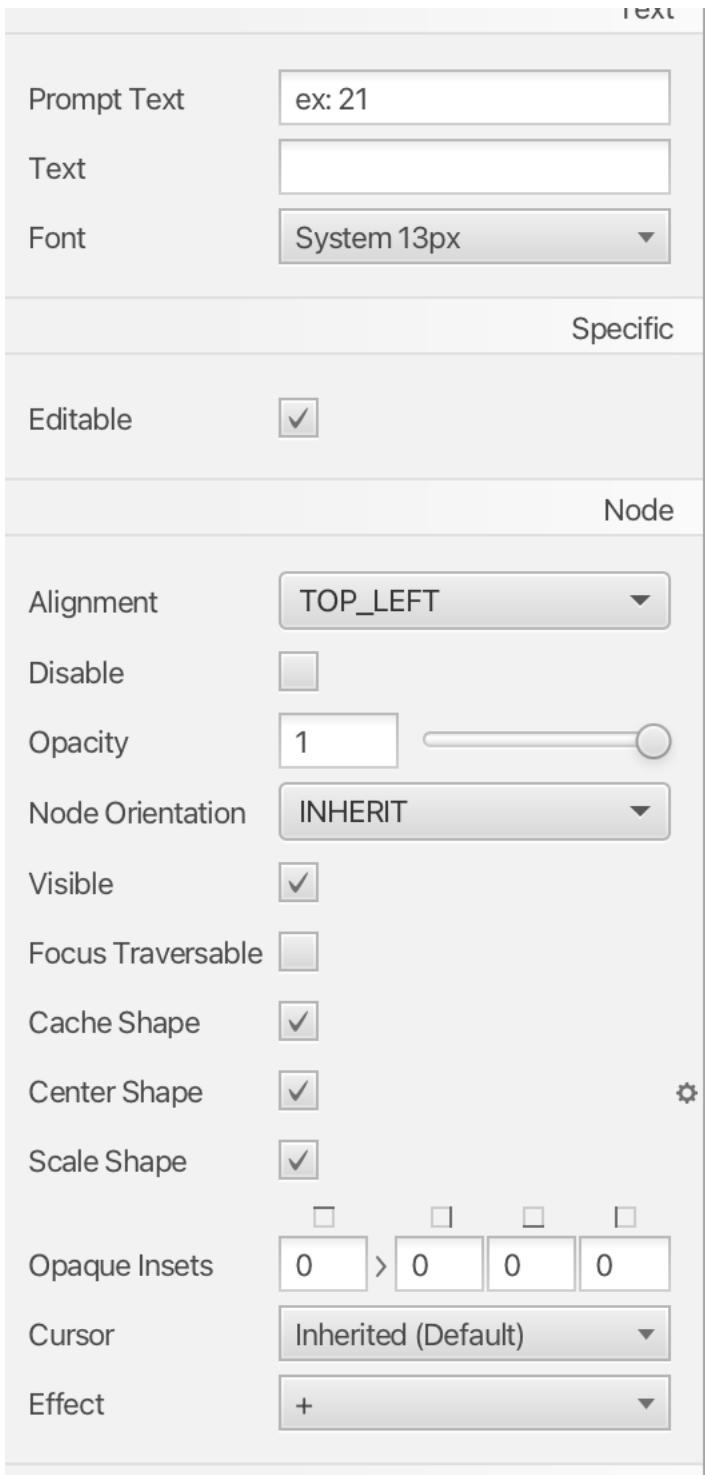
    Image image1 = new Image(getClass().getResourceAsStream(
        name:"/assets/shrek1.jpg"));
    Image image2 = new Image(getClass().getResourceAsStream(
        name:"/assets/shrek2.jpg"));

    public void displayImage(){
        if(imageView.getImage() == image1){
            imageView.setImage(image2);
        }else{
            imageView.setImage(image1);
        }
    }
}
```

Logo em um ImageView temos o .getImage() e o .setImage() que são de suma importância.

TextField:

O textfield é um campo que permite uma entrada digitada pelo usuário, é importante ressaltar que ele trabalha com Strings, então oq eu digitar dentro dele será considerado uma String. Agora vamos ver algumas opções dele no SceneBuilder e metodos dele no seu controller:



- Prompt Text: é literalmente um texto de dica para o usuário que quando eu clico no textfield ele “some”
- Alignment: Como será posicionado o texto dentro dele, como o texto será alinhado nele.
- Focus Transversable: Determina se quando eu iniciar o programa ele já vai vir “selecionado: ou não.

Agora vamos ver em um código alguns métodos importantes dele:

```

@FXML
TextField textfield;

//variável necessária para essa lógica de programação.
int idade;

public void checkField(ActionEvent event){
    //Pegando o texto do textField e convertendo para
    inteiro
    try {
        idade = Integer.parseInt(textfield.getText());
        //Verificando se a idade é maior ou igual a 18
        if(idade >= 18){
            label.setText(value:"Maior de idade, entrada
            permitida");
        }else{
            label.setText(value:"Menor de idade, entrada
            proibida");
        }
    } catch (NumberFormatException e) {
        textfield.setText(value:"");
        textfield.setPromptText(value:"Digite apenas
        números");
    }
}

```

- Primeiro é importante receber e tratar a entrada: nesse caso usando o Integer.parseInt
- .getText() retorna o texto que está atualmente no textField
- .setText() modifica o texto do textField
- .setPromptText modifica o “texto-dica” desse TextField

Check-Box:

As check-Box são simples, cada uma delas terão seu próprio ID e o importante é que elas tenham uma ação quando for acionado, logo elas precisam de um método no seu On-Action.

The screenshot shows the JavaFX Scene Builder interface. At the top, there's a tree view with nodes: fx:id (checkbox), Main, On Action (# changelimage), and DragDrop. Below the tree view is the controller code for Controller03:

```
public class Controller03 {  
  
    @FXML  
    ImageView imageview;  
    @FXML  
    CheckBox checkbox;  
    @FXML  
    Label label;  
  
    Image lampOn = new Image(getClass().getResourceAsStream  
        (name:"/assets/lightOn.jpg"));  
    Image lampOff = new Image(getClass().getResourceAsStream  
        (name:"/assets/lightOff.jpg"));  
  
    public void changeImage(ActionEvent event) {  
        if (checkbox.isSelected()) {  
            imageview.setImage(lampOn);  
            label.setText(value:"ON");  
        } else {  
            imageview.setImage(lampOff);  
            label.setText(value:"OFF");  
        }  
    }  
}
```

No caso o método em questão mudará uma imagem, mas basicamente o código de uma checkbox será sempre um if/else que faz a checagem se a checkbox está selecionado ou não pelo método .isSelected().

Para personalizar um checkBox pelo CSS é só usar .check-box{}.

Radio-Buttons:

Primeiramente vamos entender sobre a questão de grupos dos radio-buttons:



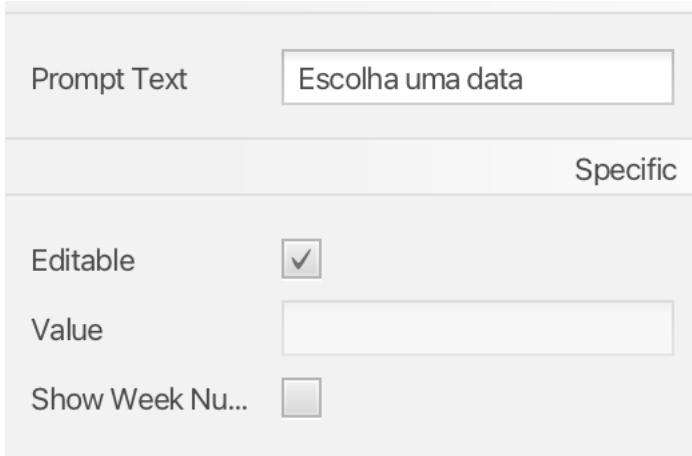
A função Toggle Group coloca ele em um grupo de radio-buttons, esses botões do mesmo grupo não podem ser selecionados de forma múltipla, sendo assim se eu tenho botões em um mesmo grupo, apenas um deles pode ser selecionado por vez.

```
public class Controller04 {  
    @FXML  
    Label label;  
    @FXML  
    RadioButton vbutton, lbutton, dbutton;  
  
    public void getGame(ActionEvent event){  
        if(vbutton.isSelected())  
            label.setText(vbutton.getText());  
        else if(lbutton.isSelected())  
            label.setText(lbutton.getText());  
        else if(dbutton.isSelected())  
            label.setText(dbutton.getText());  
        else  
            label.setText(value:"Nenhum jogo  
selecionado");  
    }  
}
```

Um exemplo de um código de um Radio-button, usamos a mesma função do .isSelected, porém é importante ressaltar que conseguimos acessar o texto do botão usando um.getText().

DatePicker:

Agora trabalhando com o date Picker a primeira coisa a citar é que no SceneBuilder o que mais importa é a função de Prompt Text que determina u texto de dica dentro do date-picker, e é importante declarar um ID para o date-picker, e colocar um método OnAction nele:



Já no código é importante termos consciência que o date-picker retorna um Objeto do tipo Genérico Date quando você usa seu método .get, então basicamente o que devemos fazer é receber esse objeto e trata-ló da forma como desejamos, principalmente formatando a data escolhida para ser usada de forma correta:

```
public class Controller05 {  
  
    @FXML  
    private Label label;  
    @FXML  
    private DatePicker datepicker;  
  
    public void getData(ActionEvent event){  
  
        LocalDate date = datepicker.getValue();  
        String formatedDate = date.format(DateTimeFormatter.  
            ofPattern(pattern:"dd/MM/yyyy")); //Posso usar ("dd/MM/  
            yyyy") ou ("dd-MM-yyyy"), caso eu use MMM ele me  
            retorna o nome do mês.  
        label.setText(formatedDate);  
    }  
  
}
```

ColorPicker:

O color-picker é bem simples, no SceneBuilder você pode declarar a cor inicial que ele vem escolhido e um texto de dica pelo PromptText, porém o importante no SceneBuilder é você declarar um ID para o seu colorPicker e declarar um método OnAction, para dessa forma acessar as cores selecionadas pelo usuário, também é de enorme importância você declarar um ID para o node ou root que você deseja modificar a cor, agora vamos demonstrar como modificar a cor de um node ou root a partir de uma cor selecionada pelo color-picker:

```

public class Controller06 {
    @FXML
    AnchorPane anchorPane;
    @FXML
    ColorPicker colorpicker;

    public void changeColor(ActionEvent event){
        Color color = colorpicker.getValue();
        anchorPane.setBackground(new Background(new
        BackgroundFill(color, CornerRadii.EMPTY, Insets.
        EMPTY)));
    }
}

```

Primeiro precisamos criar um Objeto da Classe Color para receber a cor selecionada pelo color-picker, logo após chamamos o ID do node ou do root e usamos o método .setBackground(), que precisa receber um objeto background que recebe um objeto de pintura chamado backgroundFill, e esse backgroundFill recebe uma cor e dois outros parâmetros que serão explicados abaixo:

- color: Este é o primeiro parâmetro e representa a cor que será usada para preencher o fundo. No seu código, você está usando o valor da cor selecionada de um ColorPicker, armazenado na variável color.
- CornerRadii.EMPTY: Este é o segundo parâmetro e representa os raios dos cantos do retângulo de preenchimento. No seu código, você está usando CornerRadii.EMPTY, o que significa que os cantos não têm nenhum raio especial; eles são quadrados.
- Insets.EMPTY: Este é o terceiro parâmetro e representa as margens internas do retângulo de preenchimento. Insets.EMPTY indica que não há margens internas, ou seja, o preenchimento se estende até as bordas do retângulo.

Então, o BackgroundFill está sendo usado para criar um objeto que especifica como o fundo (background) do anchorPane deve ser preenchido. Esse objeto, juntamente com outros BackgroundFills, pode ser usado para criar um Background que pode ter vários gradientes ou regiões coloridas. No seu caso, parece ser uma única cor sólida sem bordas arredondadas.

Choice-Box:

Em um choice-box, é importante citar que não podemos declarar um método OnAction para ele pelo SceneBuilder, afinal ele estará vazio e deveremos incrementar as escolhas nele depois, logo para trabalharmos com o choice-box a única coisa que deveremos fazer é declarar o seu ID pelo SceneBuilder.

Agora no Controlador da Cena precisamos fazer coisas diferentes, primeiro precisamos declarar nosso ChoiceBox e depois preenche-ló, isso será feito da seguinte forma:

```

11
12 public class Controller07 implements Initializable {
13     @FXML
14     private Label label;
15     @FXML
16     private ChoiceBox<String> choicebox;
17
18     private String[] games = {"Valorant", "CS:GO", "League of
19     Legends", "Minecraft", "GTA V", "Fortnite", "Among Us",
20     "FIFA 24", "PUBG", "Free Fire"};
21
22     @Override
23     public void initialize(URL location, ResourceBundle
24     resources) {
25         choicebox.getItems().addAll(games); //popularizando o
26         choicebox
27         choicebox.setOnAction(this::getGame); //quando eu
28         selecionar um item do choicebox ele vai chamar o
29         método getGame, por meio dessa linkagem do
30         setOnAction com o método getGame.
31     }
32
33
34     public void getGame(ActionEvent event) {
35         String game = choicebox.getValue();
36         label.setText(game);
37     }
38
39 }
```

Primeiramente declaramos o choice-box, como uma forma de List, afinal para ele ser preenchido ele irá receber diversos valores, após isso devemos criar um array com o que queremos preencher, porém deve ser do mesmo tipo do choice-box:

Agora para preenchermos o choice-box, precisamos entender sobre o método initialize da interface que o nosso Controlador precisa implementar.

O método initialize é um método da interface Initializable em JavaFX, e é chamado automaticamente quando o arquivo FXML associado à classe do controlador é carregado. O objetivo principal do método initialize é permitir que você realize inicializações necessárias para os elementos gráficos e lógica da interface do usuário antes que ela seja exibida. A assinatura do método initialize é a seguinte:

```

@Override
public void initialize(URL location, ResourceBundle
resources) {
```

- URL location: Representa a localização do arquivo FXML associado ao controlador. Pode ser usado para carregar recursos relacionados ao FXML.

- ResourceBundle resources: Representa um pacote de recursos que pode ser usado para internacionalização (i18n). Geralmente, não é muito usado em controladores FXML.

No exemplo fornecido, o método initialize é usado para popularizar um ChoiceBox com uma lista de jogos.

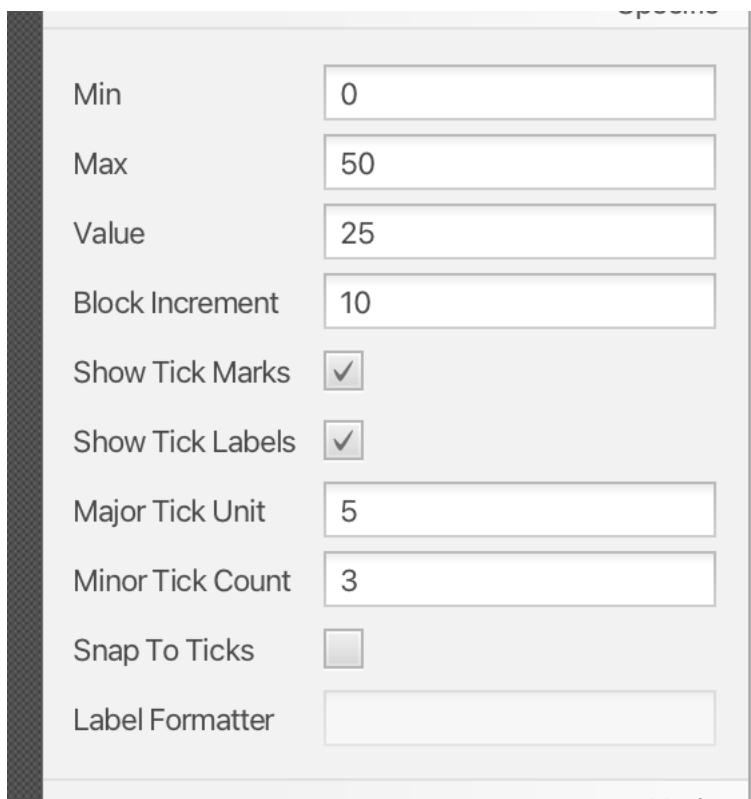
- choicebox.getItems().addAll(games): Adiciona os jogos ao ChoiceBox. Neste ponto, o ChoiceBox será preenchido com os jogos definidos no array games.
- choicebox.setOnAction(this::getGame): Associa o método getGame ao evento de seleção do ChoiceBox. Isso significa que sempre que o usuário selecionar um item no ChoiceBox, o método getGame será chamado.

Em resumo, o método initialize é usado para configurar a interface do usuário, realizar inicializações e associar eventos a elementos gráficos antes que a interface seja exibida ao usuário. Ele é parte integrante do ciclo de vida do controlador JavaFX.

Logo no caso do choiceBox necessitamos utilizar o initialize para preenche-ló, afinal o choicebox vem vazio, e preenchemos ele fora do SceneBuilder.

Sliders:

Os Sliders são controladores que podem modificar o seu padrão em tempo real, para isso são necessários listeners e métodos ChangeListeners, mas primeiro vamos ver e entender as funções dos sliders dentro do sceneBuilder:



- MIN e MAX : determinam o número máximo e mínimo do slider
- Value: o valor incial do slider
- block increment: Quando o slider é clicado ou controlado por teclas, ele aumenta e diminui esse valor caso a função snap to ticks não esteja ligada
- Snap to ticks: puxa o slider para o valor mais próximo

- show tick marks: mostra as marcações do Slider
- show tick labels: mostra os números do slider
- major tick unit: a numeração dos ticks principais baseado em saltos, como ai está 5 ele mostra 0,5,10,15,20,25,etc...
- minor tick unit: as marcações pequenas entre os major tick marks, caso eu coloque o major tick como 10 e o minor tick como 1 eu terei os ticks da seguinte forma: 0,5,10,15,20, sendo os números em 0 os major ticks e os números em 5 os minor ticks.

Agora entendendo o controlador:

```

@FXML
private Label label;
@FXML
private ImageView imageview;
@FXML
private Slider slider;

Image normal = new Image(getClass().getResourceAsStream
(name:"/assets/normal.jpeg"));
Image calor = new Image(getClass().getResourceAsStream
(name:"/assets/calor.jpg"));
Image frio = new Image(getClass().getResourceAsStream
(name:"/assets/frio.jpeg"));

int temperatura;
```

```

@Override
public void initialize(URL location, ResourceBundle
resources) {

    slider.valueProperty().addListener(new
    ChangeListener<Number>() {

        @Override
        public void changed(ObservableValue<? extends
Number> observable, Number oldValue, Number
newValue) {
            temperatura = (int) slider.getValue();
            label.setText(temperatura + "°C");
            if (temperatura <= 15) {
                imageview.setImage(frio);
            } else if (temperatura > 15 && temperatura <=
30) {
                imageview.setImage(normal);
            } else {
                imageview.setImage(calor);
            }
        }
    });
}

```

Primeiramente deve ser citado que no SceneBuilder a única coisa que deve ser declarada é o ID do seu slider, afinal ele não terá uma ação OnAction, pois ele utiliza do método Initialize implementado pelo Controller a partir da interface Initializable (Explicado no node: Choice-Box) e junto a isso ele utiliza um Listener para esse Slider, nesse caso um ChangeListener, que será explicado abaixo:

- **ChangeListener:** Este bloco de código adiciona um ChangeListener ao valueProperty do Slider. O método changed é chamado sempre que o valor do slider é alterado.
- **Obtenção do Novo Valor do Slider:** temperatura = (int) slider.getValue(); - Aqui, o novo valor do slider é obtido e armazenado na variável temperatura, deve ser usado um casting pois o método .getValue() em um slider retorna um double.
- **Atualização do Rótulo (Label):** label.setText(temperatura + "°C"); - O texto do rótulo é atualizado para exibir a temperatura atual.
- **Atualização da Imagem (ImageView):** Com base na temperatura, a imagem exibida no ImageView chamado imageview é atualizada. Se a temperatura for menor ou igual a 15, exibe a imagem frio, se estiver entre 15 e 30 (inclusive), exibe a imagem normal, e se for maior que 30, exibe a imagem calor.

Em resumo, este trecho de código está vinculando o comportamento do aplicativo à interação do usuário com o Slider. Sempre que o usuário move o slider, a temperatura é atualizada, o rótulo é

ajustado e a imagem exibida também é alterada com base nos intervalos de temperatura definidos.

Vamos a uma explicação mais detalhada sobre os listeners e o ChangeListener associado ao Slider no contexto de uma aplicação JavaFX:

Listeners em JavaFX:

Listeners são mecanismos em JavaFX que permitem que um objeto (o "ouvinte" ou "listener") seja notificado sobre mudanças em outro objeto (o "emissor" ou "source"). Isso é especialmente útil em interfaces gráficas de usuário, onde as interações do usuário podem acionar eventos que precisam ser tratados pelo código.

ChangeListener:

ChangeListener é uma interface em JavaFX projetada para receber notificações de alterações em propriedades observáveis. No caso do Slider, a propriedade observável é valueProperty().

Funcionamento do ChangeListener no Slider:

- **Associando o Listener ao Slider:**

```
slider.valueProperty().addListener(new ChangeListener<Number>() {...});
```

O método addListener é usado para adicionar um ouvinte à propriedade observável valueProperty()(source) do Slider. No exemplo, um novo ChangeListener é instanciado e seu método changed é implementado inline.

- **Método changed:**

```
@Override  
public void changed(ObservableValue<? extends Number> observable,  
        Number oldValue, Number newValue) {...}
```

Este método é chamado automaticamente sempre que o valor do slider é alterado. Ele recebe três parâmetros:

- **observable:** A propriedade observável que mudou (no caso, valueProperty() do Slider).
- **oldValue:** O valor antigo antes da alteração.
- **newValue:** O novo valor após a alteração.

Relação com a Tela:

- **Bidirecionalidade com a Interface Gráfica:**

O ChangeListener é parte fundamental da atualização bidirecional entre o modelo de dados (no caso, a temperatura) e a interface gráfica. Quando o usuário interage com o slider, o modelo (temperatura) é atualizado e, por sua vez, atualiza a interface gráfica. Essa relação bidirecional é crucial para manter a consistência entre a lógica de aplicação e a representação visual.

- **Reatividade:**

O uso de listeners, como o ChangeListener, torna a aplicação reativa. Isso significa que as atualizações na interface gráfica são acionadas automaticamente em resposta a mudanças no estado da aplicação. Sem um listener, a aplicação dependeria de verificações manuais e atualizações, o que tornaria o código mais complexo e menos eficiente.

Em resumo, o ChangeListener permite que a aplicação responda dinamicamente às mudanças no valor do Slider, atualizando a interface gráfica de acordo com as alterações na temperatura e proporcionando uma experiência de usuário interativa e reativa.

No contexto do código fornecido, o **ouvinte** (listener) é o objeto que implementa a interface ChangeListener<Number>. No código apresentado, esse ouvinte é anônimo, criado diretamente como uma instância de ChangeListener durante a adição do listener.

O **emissor** (source) é o objeto que dispara o evento quando há uma mudança. Neste caso, o emissor é a propriedade valueProperty() do Slider. A linha de código slider.valueProperty().addListener(...) indica que estamos adicionando um ouvinte à propriedade valueProperty() do Slider. Quando o valor dessa propriedade muda (quando o usuário interage com o slider), o ChangeListener é notificado, e o método changed é invocado.

- **Ouvinte (Listener):** new ChangeListener<Number>() { ... } - Esta instância anônima da interface ChangeListener<Number> age como o ouvinte/listener. Ela contém a lógica que deve ser executada quando ocorre uma mudança no valor do Slider.
- **Emissor (Source):** slider.valueProperty() - Essa expressão representa a propriedade valueProperty() do Slider, que é a fonte (emissor/source) dos eventos de mudança. Quando o valor desta propriedade muda (quando o usuário interage com o Slider), ela notifica todos os ouvintes associados.

Portanto, quando o usuário move o slider, o Slider (emissor) avisa aos ouvintes (Listeners) que estão interessados nas mudanças do valor (ChangeListener), e o método changed do ChangeListener é chamado, realizando as ações definidas nesse método.

Progress Bar:

A progress bar é simples, necessitando apenas de um ID para funcionar e alguns botões para aumenta-lá e diminui-lá, de resto no SceneBuilder podemos rotacionar uma Progress Bar para “modificar” a sua ideia de progressão.

Já no programa é importante declarar uma variável de controle do progresso da progressbar, sendo essa variável double, afinal a progress bar varia de 0 a 1 , sendo 1 o 100% e 0.5 o 50%.

Para mudar a cor da sua ProgressBar:

```
@Override  
public void initialize(URL  
location, ResourceBundle  
resources) {  
    progressbar.setStyle  
(value:"-fx-accent: red;");  
}
```

Logo para alterar o CSS de Nodes no JavaFX por meio de códigos inline é necessário usar o método initialize da interface Initializable(explicada no node Choice-Box)

Porém quando trabalhamos com Double, temos o problema de truncamento, então para incrementarmos o valor na nossa progressBar precisamos resolver esse problema, para resolver o problema de truncamento vamos utilizar um objeto do tipo BigDecimal, para controlar o progresso:

```

BigDecimal progress = new BigDecimal(val:0);

public void diminuir() {
    if(progress.doubleValue() > 0){
        progress = progress.subtract(new BigDecimal(val:0.
1));
        progressbar.setProgress(progress.doubleValue());
        label.setText(String.format(format:".0f%%",
            progress.doubleValue() * 100));
    }
}

public void aumentar() {
    if(progress.doubleValue() < 1){
        progress = progress.add(new BigDecimal(val:0.1));
        progressbar.setProgress(progress.doubleValue());
        label.setText(String.format(format:".0f%%",
            progress.doubleValue() * 100));
    }
}

@Override
public void initialize(URL location, ResourceBundle
resources) {

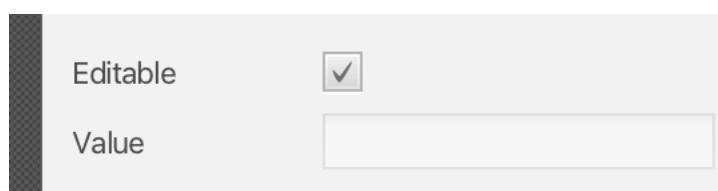
```

Assim conseguimos controlar a nossa progressBar sem gerar problemas de valores quebrados.

Spinner:

Os spinners são uma forma do usuário escolher um número ou um objeto de forma sequencial através de um node para ter controle sobre isso.

No scene builder apenas precisamos declarar um ID no nosso Spinner, e além disso tem uma opção que pode ser usada ou não:



A Opção Editable, permite o usuário digitar um dos objetos ou números que iremos incluir na nossa coleção, facilitando a interação do usuário que não precisa ficar clicando para chegar na opção que deseja.

Agora vamos entender o código, primeiro as declarações:

```
public class Controller10 implements Initializable{  
  
    @FXML  
    Spinner<Integer> spinner;  
    @FXML  
    Label label;  
  
    int currentValue;
```

Assim como a choice-box o Spinner é um Node de seleção, logo ele precisa receber uma coleção porém ele será preenchido de outra forma:

```

@Override
public void initialize(URL location, ResourceBundle
resources) {

    SpinnerValueFactory<Integer> valueFactory = new
    SpinnerValueFactory.IntegerSpinnerValueFactory(min:1,
max:10);

    valueFactory.setValue(newValue:1);

    spinner.setValueFactory(valueFactory);

    currentValue = spinner.getValue();|
```

label.setText("Current value: " + currentValue);

spinner.valueProperty().addListener(new
ChangeListener<Integer>() {

```

@Override
public void changed(ObservableValue<? extends
Integer> observable, Integer oldValue, Integer
newValue) {
    currentValue = spinner.getValue();
    label.setText("Current value: " +
currentValue);
}
```

});

Para utilizar o Spinner precisamos utilizar o initialize, e junto a isso diversas funções específicas dele, para começarmos devemos entender que ele pode possuir diversos objetos, sendo assim o tipo da coleção colocada a ele depende de qual tipo de Objeto você escolheu, nesse caso vamos utilizar o Integer para fazer um spinner que varia de 1 a 10, agora vamos explicar detalhadamente cada linha de código:

- SpinnerValueFactory<Integer> valueFactory: Aqui você está criando uma fábrica (SpinnerValueFactory) para o componente Spinner que irá gerenciar os valores que o Spinner pode exibir.
- new SpinnerValueFactory.IntegerSpinnerValueFactory(1, 10): Você está criando uma instância da classe IntegerSpinnerValueFactory, que é uma implementação específica de SpinnerValueFactory para valores inteiros. O intervalo permitido é de 1 a 10.

- `valueFactory.setValue(1)`: Aqui você está definindo o valor inicial da fábrica de valores do Spinner para 1.
- `spinner.setValueFactory(valueFactory)`: Agora você está associando a fábrica de valores que você criou ao componente Spinner.
- `currentValue = spinner.getValue()`: Você está obtendo o valor inicial do Spinner e atribuindo à variável `currentValue`.
- `label.setText("Current value: " + currentValue)`: Aqui você está atualizando o texto de um componente Label para exibir o valor inicial do Spinner.
- `spinner.valueProperty().addListener(new ChangeListener<Integer>() {`: Aqui você está adicionando um ouvinte de mudança ao Spinner, que será notificado sempre que o valor do Spinner for alterado.
- `public void changed(ObservableValue<? extends Integer> observable, Integer oldValue, Integer newValue) {}`: Você está implementando o método `changed` da interface `ChangeListener<Integer>`. Este método será chamado sempre que o valor do Spinner for alterado.
- `currentValue = spinner.getValue();`: Você está atualizando a variável `currentValue` com o novo valor do Spinner.
- `label.setText("Current value: " + currentValue);`: Finalmente, você está atualizando o texto do Label para refletir o novo valor do Spinner.

Em resumo, este trecho de código configura um Spinner para exibir valores inteiros de 1 a 10, inicializado com o valor 1. Ele também atualiza um Label para mostrar o valor atual do Spinner e adiciona um ouvinte para detectar mudanças no valor do Spinner e refletir essas mudanças no Label.

Mais detalhes sobre o Spinner Value Factory:

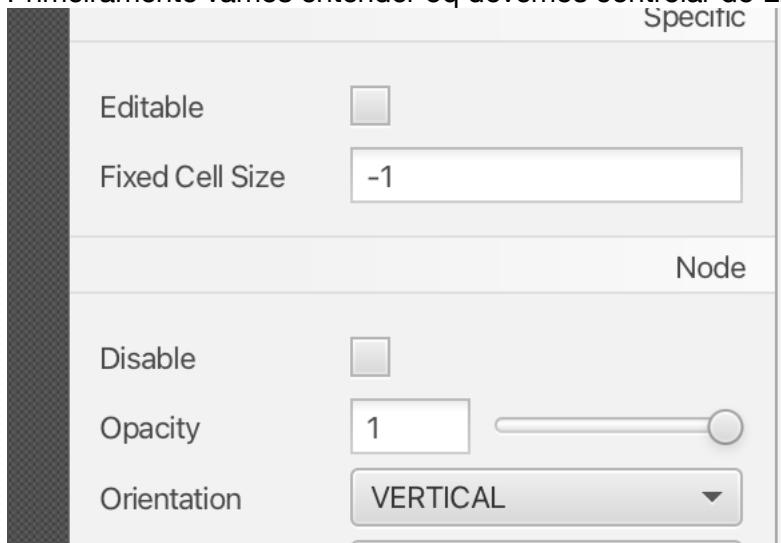
SpinnerValueFactory

Existem vários tipos de fábricas de valores (`SpinnerValueFactory`) em JavaFX, dependendo do tipo de dados que o Spinner deve manipular. Os principais tipos são:

- **IntegerSpinnerValueFactory**: Lida com valores inteiros. É configurado com um intervalo de valores inteiros.
Exemplo:
`SpinnerValueFactory<Integer> valueFactory = new SpinnerValueFactory.IntegerSpinnerValueFactory(1, 10);`
- **DoubleSpinnerValueFactory**: Lida com valores decimais. Configurado com um intervalo de valores decimais.
Exemplo:
`SpinnerValueFactory<Double> valueFactory = new SpinnerValueFactory.DoubleSpinnerValueFactory(0.0, 1.0, 0.1);`
- **ListSpinnerValueFactory**: Lida com uma lista predefinida de valores.
Exemplo:
`ObservableList<String> values = FXCollections.observableArrayList("Red", "Green", "Blue");
SpinnerValueFactory<String> valueFactory = new SpinnerValueFactory.ListSpinnerValueFactory(values);`

ListView:

Primeiramente vamos entender oq devemos controlar do ListView pelo SceneBuilder:



- Fixed Cell Size: Determina o tamanho do bloco na lista, caso vc deixe como -1, ele determina automaticamente, caso vc mude o valor, esse valor será o tamanho do bloco na lista de acordo o número de pixels que vc digitar.
- Orientation: Determina se a lista vai seguir de forma Vertical ou Horizontal.

No SceneBuilder Declaramos apenas o ID do ListView, já no controller faremos da seguinte forma:

```

public class Controller11 implements Initializable {

    @FXML
    Label label;
    @FXML
    ListView<String> listview;

    String[] food = {"Pizza", "Hamburger", "Hot Dog",
    "Pasta", "Salad", "Soup", "Sandwich", ""};

    String currentFood;

    @Override
    public void initialize(URL location, ResourceBundle resources) {

        listview.getItems().addAll(food);

        listview.getSelectionModel().selectedItemProperty().
        addListener(new ChangeListener<String>() {

            @Override
            public void changed(ObservableValue<? extends
            String> observable, String oldValue, String
            newValue) {
                currentFood = listview.getSelectionModel().
                getSelectedItem();
                label.setText("Current value: " +
                currentFood);
            }
        });
    }
}

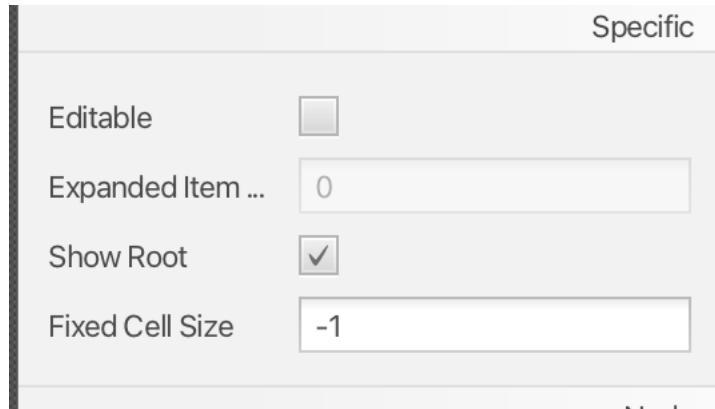
```

Vamos necessitar do método initialize da interface Initializable (Explicado no Node Choice-Box), para declaramos a nossa ListView, precisamos declarar também o tipo de coleção que ela irá receber, nesse caso uma coleção de Strings, junto a isso criamos a coleção que será adicionada e também uma variável do tipo String que vai me permitir receber e trabalhar com o objeto selecionado do ListView.

Já dentro do método initialize, primeiro fazemos o preenchimento da lista usando .getItems().addAll("a coleção") e logo após adicionamos um Listener a nossa ListView, nesse caso um ChangeListener com o método changed que a cada vez que eu clico na ListView, ou seja que ela sofre uma mudança, eu realizo esse método obtendo o valor selecionado e passando ele pra label, modificando ela.

TreeView:

Vamos entender agora o TreeView, que funciona como um organizador de arquivos, o seja são pastas que podem ser abertas e contém conteúdos, primeiramente vamos entender suas funcionalidades no SceneBuilder:



- Show Root: declara se eu vou ou não ver a primeira “pasta” do treeView, caso ele seja marcado como falso a primeira pasta já estará selecionada.
- Fixed Cell Size: delcara o tamanho do bloco em Pixels, caso esteja em -1, ele se ajusta automaticamente

Mas agora o importante é entender que para um TreeView no SceneBuilder nós vamos declarar apenas o seu ID e junto a isso declarar dois métodos:



Eles devem ter o mesmo método de resposta, nesse caso o método select.

Agora vamos ver o código do TreeView:

```

public class Controller12 implements Initializable{

    @FXML
    private TreeView<String> treeview;

    @Override
    public void initialize(URL location, ResourceBundle resources) {

        TreeItem<String> root = new TreeItem<>(value:"Files",
        new ImageView(new Image(url:"/assets/pasta.png")));

        TreeItem<String> branchItem1 = new TreeItem<>
        (value:"Pictures");
        TreeItem<String> branchItem2 = new TreeItem<>
        (value:"Video");
        TreeItem<String> branchItem3 = new TreeItem<>
        ([value:"Music"]);

        TreeItem<String> leafItem1_1 = new TreeItem<>(value:".png");
        TreeItem<String> leafItem2_1 = new TreeItem<>(value:".jpeg");
        TreeItem<String> leafItem1_2 = new TreeItem<>(value:".mp4");
        TreeItem<String> leafItem2_2 = new TreeItem<>(value:".wav");
        TreeItem<String> leafItem1_3 = new TreeItem<>(value:".mp3");
        TreeItem<String> leafItem2_3 = new TreeItem<>(value:".mp3");
    }
}

```

Primeiramente precisamos declarar o TreeView com alguma coleção específica, e junto a isso devemos utilizar o método inicializa da Interface Initializable, e a primeira coisa que fazemos é declarar o root do nosso TreeView, a sua raiz, a sua primeira pasta, utilizando o código: TreeItem<String> root = new TreeItem<>("valor que será passado", posso adicionar um ícone a essa raiz do TreeView utilizando a passagem de mais um parâmetro sendo ele um ImageView com uma Image dentro).

Logo após declaro as ramificações do TreeView nesse caso 3, que são os BranchItems e por ultimo delcaro as ramificações das ramificações desse TreeView, que nesse caso são os leafItems.

```

root.getChildren().addAll(branchItem1, branchItem2,
branchItem3);

branchItem1.getChildren().addAll(leafItem1_1,
leafItem2_1);
branchItem2.getChildren().addAll(leafItem1_2,
leafItem2_2);
branchItem3.getChildren().addAll(leafItem1_3,
leafItem2_3);

//treeview.setShowRoot(false);
treeview.setRoot(root);

```

E agora finalmente organizo e coloco cada parte do TreeView em seu devido lugar, primeiro adiciono no root as suas 3 ramificações e depois adiciono em cada uma das suas ramificações outras 2 ramificações, tudo isso utilizando do método: .getChildren().addAll("ramificações").

Logo após declaro através do método setRoot(), o root que o meu TreeView vai receber.

Agora vamos trabalhar com as ações que serão tomadas quando o usuário selecionar um item do TreeView:

```

public void select() {

    TreeItem<String> item = treeview.getSelectionModel().
        getSelectedItem();

    if (item != null) {
        System.out.println(item.getValue());
    }
}

```

Utilizamos do método select que foi declarado para ser usado quando 2 ações forem realizadas, essa ligação das ações com esse método foi feita pelo sceneBuilder. Nesse método o que fazemos é receber o item selecionado através do método .getSelectionModel().getSelectedItem(); e como ele vai nos retornar um TreeItem eu recebo ele por meio de um TreeItem, porém é importante que a ação a ser realizada a cada seleção do usuário seja testada checando se ela é ou não null, pois quando o usuário clica na seta para expandir o TreeView, esse clique retorna null a esse método, podendo assim gerar um erro, portanto é importante o uso desse if para checar se o item recebido é ou não null.

TableView:

Para entender uma tableView é mais complicado, primeiro precisamos entender o que fazer com uma no SceneBuilder, para isso precisamos entender que em uma TableView, nós iremos incluir dentro dela TableColumn e no SceneBuilder tanto a TableView quanto as TableColumn terão ID's, e ambas recebem uma coleção de algo, para entendermos melhor vamos para o código:

```
public class Controller13 implements Initializable {  
  
    //Id da tabela  
    @FXML  
    private TableView<Customer> tableview;  
  
    //Colunas da tabela  
    @FXML  
    private TableColumn<Customer, String> nameC;  
    @FXML  
    private TableColumn<Customer, Integer> ageC;  
    @FXML  
    private TableColumn<Customer, String> numberC;  
  
    @FXML  
    Button buttonAdd;  
    @FXML  
    Button buttonRemove;  
    @FXML  
    TextField tfname;  
    @FXML  
    TextField tfage;  
    @FXML  
    TextField tfnumber;
```

Primeiramente vamos notar que a TableView está com uma coleção de algo diferente nesse caso o tipo Customer, que foi uma Classe criada pelo usuário

- Classe Customer:

```
public class Customer {  
  
    private String name;  
    private int age;  
    private String number;  
  
    public Customer(String name, int  
                    age, String number) {  
        this.name = name;  
        this.age = age;  
        this.number = number;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

Já as TableColumn possuem duas Coleções, sendo a primeira sempre a mesma coleção da sua TableView, nesse caso Customer e a segunda o tipo de Coleção que essa coluna irá receber, sendo ela Integer, String ou diversos outros.

Vamos entender agora como relacionar as tabelas da TableView, com os respectivos aspectos e atributos da Classe Customer:

```
@Override  
public void initialize(URL location, ResourceBundle resources) {  
    nameC.setCellValueFactory(new  
        PropertyValueFactory<Customer, String>  
        (property:"name")); //Aqui é onde se define qual  
        atributo da classe Customer será exibido na coluna  
        nameColumn.  
    ageC.setCellValueFactory(new  
        PropertyValueFactory<Customer, Integer>  
        (property:"age")); //Aqui é onde se define qual  
        atributo da classe Customer será exibido na coluna  
        ageColumn.  
    numberC.setCellValueFactory(new  
        PropertyValueFactory<Customer, String>  
        (property:"number")); //Aqui é onde se define qual  
        atributo da classe Customer será exibido na coluna  
        numberColumn.  
}
```

Primeiramente vamos necessitar do método initialize da Interface Initializable, para iniciar a nossa tela já com as tabelas e colunas relacionadas corretamente.

Para fazer essa relação iremos utilizar o método: ID_da_coluna.setCellValueFactory(new PropertyValueFactory<Customer, String>) ("o nome do atributo igualmente está na Classe Customer que irá se relacionar com essa tabela"), Dessa forma podemos fazer a relação de um atributo na Classe Customer com a nossa tabela, sendo assim a nossa coluna recebe um atributo da Classe Customer e o transmite em forma de String.

Agora vamos ver os métodos para adiconar itens nessa nossa TableView:

```

public void add() {
    Customer customer = new Customer(tfname.getText(),
    Integer.parseInt(tfage.getText()), tfnumber.getText());
    //Cria um objeto do tipo Customer com os dados
    //digitados nos campos de texto.
    tableview.getItems().add(customer); //Adiciona o
    //objeto criado na tabela.
    tfname.setText(value:""); //Limpa o campo de texto.
    tfage.setText(value:""); //Limpa o campo de texto.
    tfnumber.setText(value:""); //Limpa o campo de texto.
}

```

Primeiros vamos criar um objeto da nossa Classe Customer, com os seus respectivos dados, nesse caso estamos passando o nome, idade e o número por meio de textFields, determinados por tfname,tfage, etc.

Então criamos o nosso Objeto Customer e agora usamos o método
Id_da_TableView.getItems().add(Objeto_customer_instanciado);

Agora vamos ver o método de remover itens da nossa TableView:

```

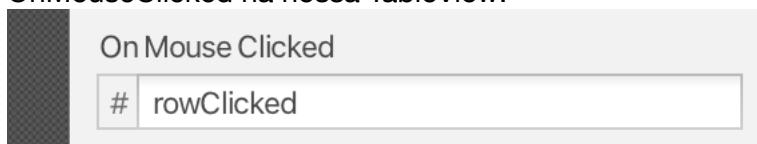
public void remove() {
    int index = tableview.getSelectionModel().
    getSelectedIndex(); //Pega o índice da linha
    selecionada.
    tableview.getItems().remove(index); //Remove a linha
    selecionada.
}

```

Quanto selecionamos um item e clicamos no botão que realiza o método remove, ele primeiro recebe o index do item selecionado por meio do método
nome_da_tableview.getSelectionModel().getSelectionIndex() e armazena esse index em um
inteiro. Logo após por meio do método tableview.getItems().remove(index); ele remove o item que
tem esse index selecionado.

Agora vamos ver o método de editar itens na nossa TableView:

Primeiramente precisamos declarar um método no SceneBuilder para a ação de
OnMouseClicked na nossa TableView:



On Mouse Clicked
rowClicked

No código esse método funciona da seguinte forma:

```
public void rowClicked(MouseEvent event){  
    Customer customer = tableview.getSelectionModel().  
    getSelectedItem(); //Pega o objeto Customer da linha  
    selecionada.  
    if(customer != null){ //Se o objeto Customer não for  
    nulo, preenche os campos de texto com os dados do  
    objeto Customer.  
        tfname.setText(customer.getName()); //Preenche o  
        campo de texto com o nome do objeto Customer.  
        tfage.setText(String.valueOf(customer.getAge()  
        )); //Preenche o campo de texto com a idade do  
        objeto Customer.  
        tfnumber.setText(Integer.toString(customer.  
        getNumber())); //Preenche o campo de texto com o  
        número do objeto Customer.  
    }  
}
```

Primeiro recebemos o Objeto da linha selecionada na nossa Tabela, como estamos trabalhando com uma tabela de Customers, a tabela retorna um customer no seu método `.getSelectionModel().getSelectedItem()`; Logo após verificamos se o objeto customer é ou não null, pois como o método é chamado sempre que clicarmos na tabela, caso cliquemos nela e ela não ter nenhum dado ainda dentro dela, ela nós retornará um `NullPointerException`, por isso essa verificação é importante.

Logo após essa verificação, preenchemos todos os `TextFields` com os dados do nosso objeto selecionado para assim podermos editá-los com o nosso método `Update`:

```

public void update(ActionEvent event){
    ObservableList<Customer> currentTableData = tableview.getItems(); //Pega a lista de
    objetos Customer da tabela.
    int currentCustomerID = Integer.parseInt(tfnumber.getText()); //Pega o número do objeto
    Customer que está sendo editado.

    for(Customer customer : currentTableData){ //Percorre a lista de objetos Customer da
    tabela.

        if(customer.getNumber() == currentCustomerID){ //Se o número do objeto Customer for
        igual ao número do objeto Customer que está sendo editado, atualiza os dados do
        objeto Customer.
            customer.setName(tfname.getText()); //Atualiza o nome do objeto Customer.
            customer.setAge(Integer.parseInt(tfage.getText())); //Atualiza a idade do
            objeto Customer.
            customer.setNumber(Integer.parseInt(tfnumber.getText())); //Atualiza o número
            do objeto Customer.

            tableview.setItems(currentTableData);
            tableview.refresh(); //Atualiza a lista de objetos Customer da tabela.

            tfname.setText(value:""); //Limpa o campo de texto.
            tfage.setText(value:""); //Limpa o campo de texto.
            tfnumber.setText(value:""); //Limpa o campo de texto.

            break; //Para o loop.
    }
}

```

Primeiro pegamos a lista toda da nossa TableView e armazenamos em uma ObservableList<tipo da nossa tabela> com o método .getItems();, afinal nossa tabela retorna uma ObservableList com o seu tipo determinado anteriormente, nesse caso uma ObservableList de Customers. Logo após recebo o ID da linha selecionado no momento, como o método de MouseClicked modifica o TextField, eu posso obter esse ID de lá

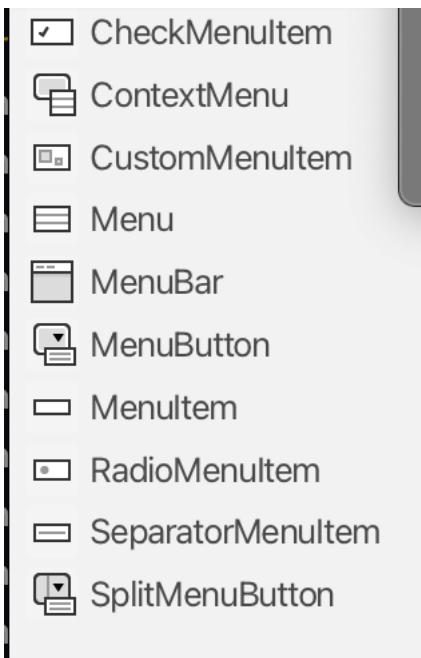
Logo após crio um forEach, que percorre a lista de objetos e utilizo um If pra ver onde o ID é igual ao ID na tabela, e nesse objeto em específico e atualizo com os valores passados pelos TextFields.

PAra que essa atualização funcione eu preciso setar novamente a tabela com a ObservableList que sofreu a modificação, nesse caso a currentTableData e por ultimo fazer o refresh da nossa TableView.

OBS: é importante voltar o texto dos textFields e dar um break no loop para não ocorrer erros.

MenuBar:

Basicamente uma MenuBar, é uma forma de criarmos um menu com várias opções de escolha, porém aqui iremos apenas descrever suas funcionalidades no SceneBuilder, que são muitas:

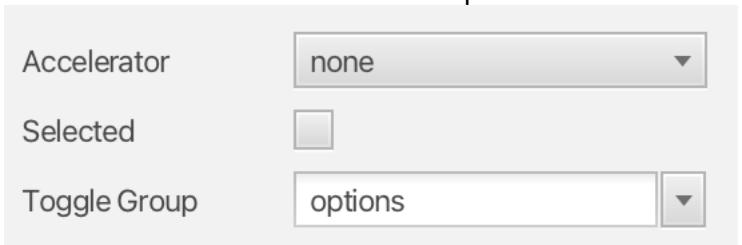


Esses são todos os tipos de nodes Menu que podemos adicionar, eles devem ser adicionados no nosso Node principal que é o MenuBar:

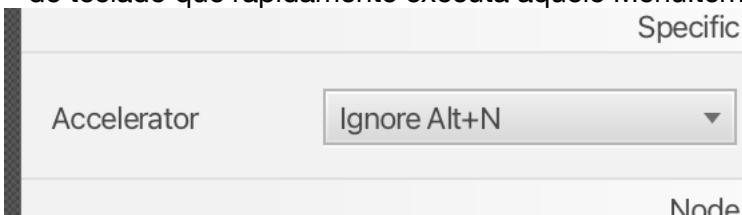
- Menu: adicionado na MenuBar ele vira um dos menus que podemos ter acesso clicando, um dos menus principais.
- MenuItem: adicionado no Menu, ele é um dos items do Menu que podemos escolher.
- checkMenuItem: adicionado no menu, ele é uma checkBox dentro de um Menu
- SeparatorMenuItem: Adicionado no Menu, ele serve apenas para estética, para dividir Menultens dentro de um Menu
- RadioMenuItem: Adicionado no Menu, é um radioButton dentro do menu.

Agora algumas observações:

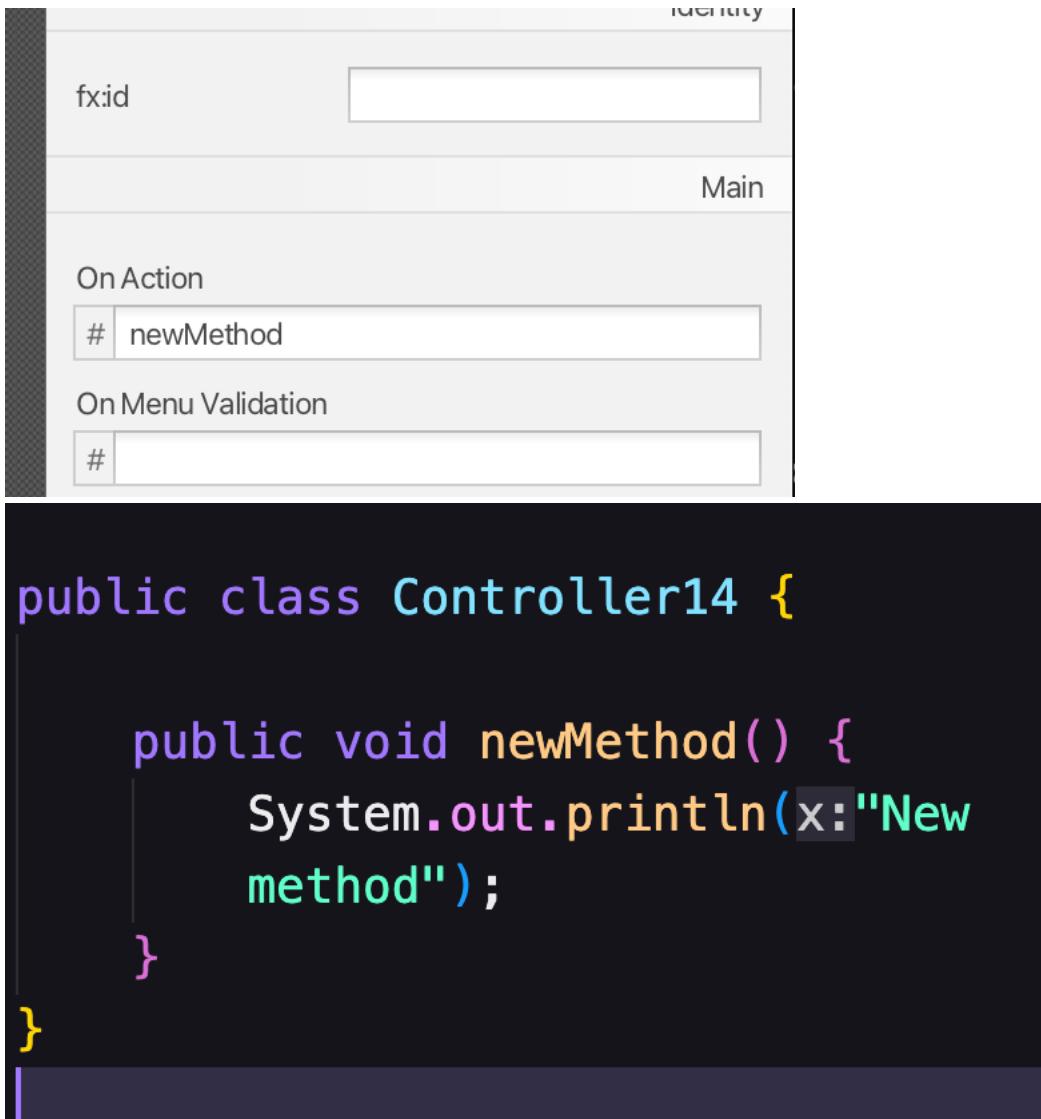
- O radioMenuItem pode ter um TogleGroup, permitindo que apenas um daquele grupo possa estar selecionado ao mesmo tempo:



- Todo MenuItem pode ter um Accelerator, um accelerator é uma forma de adicionar um atalho de teclado que rapidamente executa aquele MenuItem:



- E por ultimo é importante ressaltar que todo MenuItem tem o seu próprio ID e OnAction Event, porém geralmente só declaramos seu OnActionEvent no SceneBuilder para ser executado ao clicar nele, a não ser que a ação dele ser mudar de tela, ai vamos necessitar do seu ID declarado.



```
public class Controller14 {  
  
    public void newMethod() {  
        System.out.println("New  
method");  
    }  
}
```

FlowPane:

O FlowPane é apenas um tipo diferente de Pane, vou ressaltar que a maioria dos panes não terão um código explícito, logo vou ressaltar apenas na parte do SceneBuilder.

O flowpane parece com um anchorpane normal, porém ao colocarmos algum node dentro dele, notamos a diferença, ele organiza tudo em fileiras, todos os nodes em ordem, sendo ela vertical ou horizontal isso será definido por opções que veremos abaixo, porém o mais importante de se entender em um flowPane é que ele é Resizable, logo ele é reativo e muda a organização dos itens de acordo mudamos o tamanho dele em execução, agora vamos ver os detalhes do SceneBuilder do Flow Pane:

Row Valignment	CENTER
Column Halign...	LEFT
Pref Wrap Length	400
Node	
Alignment	TOP_CENTER
Disable	<input type="checkbox"/>
Opacity	1 <input type="range"/>
Orientation	HORIZONTAL
Node Orientation	INHERIT
Visible	<input checked="" type="checkbox"/>
Focus Traversable	<input type="checkbox"/>
Cache Shape	<input checked="" type="checkbox"/>
Center Shape	<input checked="" type="checkbox"/>
Scale Shape	<input checked="" type="checkbox"/>
Opaque Insets	<input type="text"/> 0 > <input type="text"/> 0 <input type="text"/> 0 <input type="text"/> 0
Cursor	Inherited (Default)

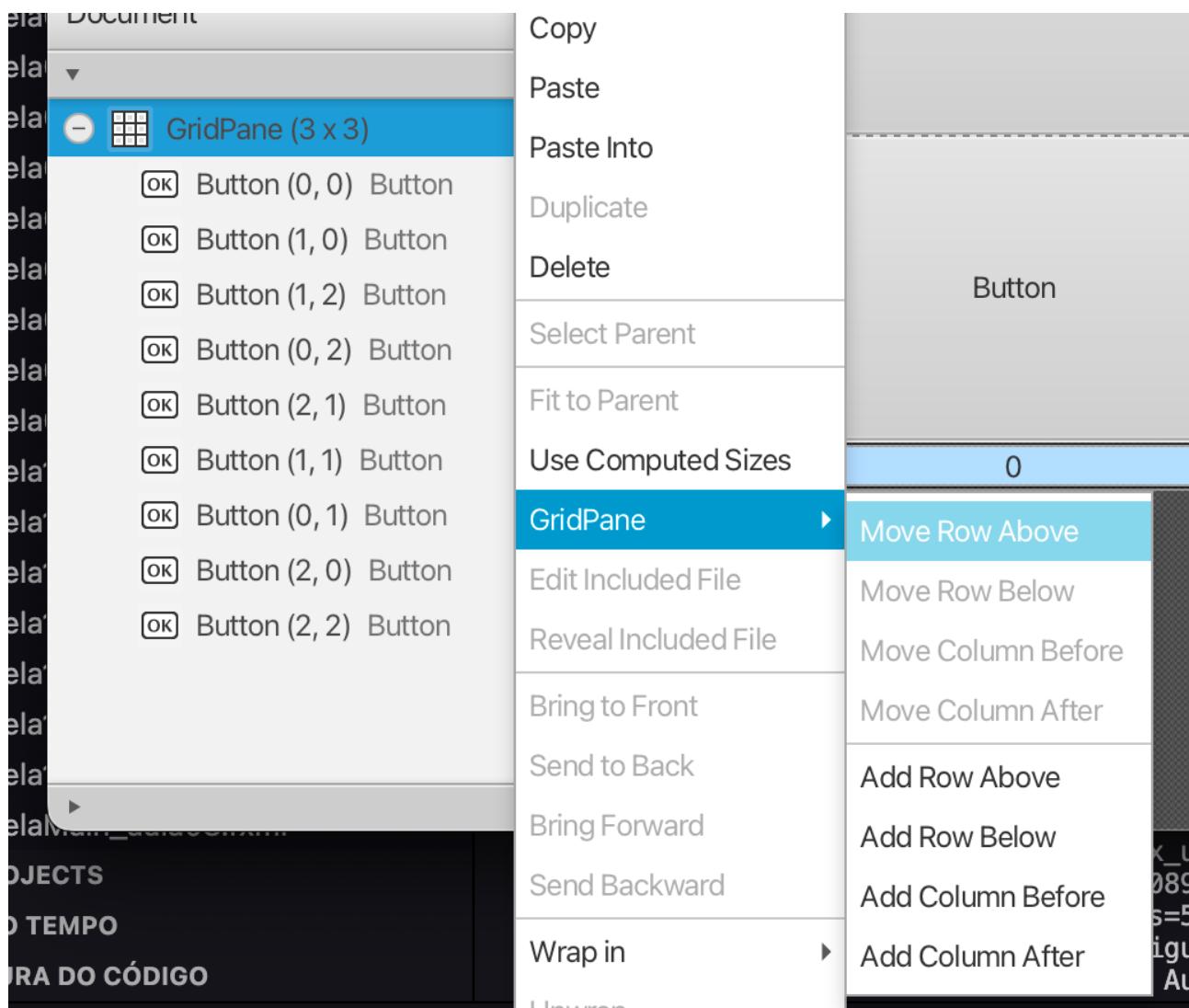
- Row Valignmet: o Alinhamento das linhas no FlowPane
- Column Halignment: O Alinhamento das colunas no FlowPane
- Pref Wrap Length: O tamanho de preferência que ele começa a ser reativo
- Alignment: O alinhamento dos Nodes em si, colocados dentro dele, determina onde os nodes começam
- Orientation: A Orientação das linhas podendo ser vertical ou horizontal.

Hgap	10
Vgap	10
Padding	<input type="text"/> 10 > <input type="text"/> 10 <input type="text"/> 10 <input type="text"/> 10

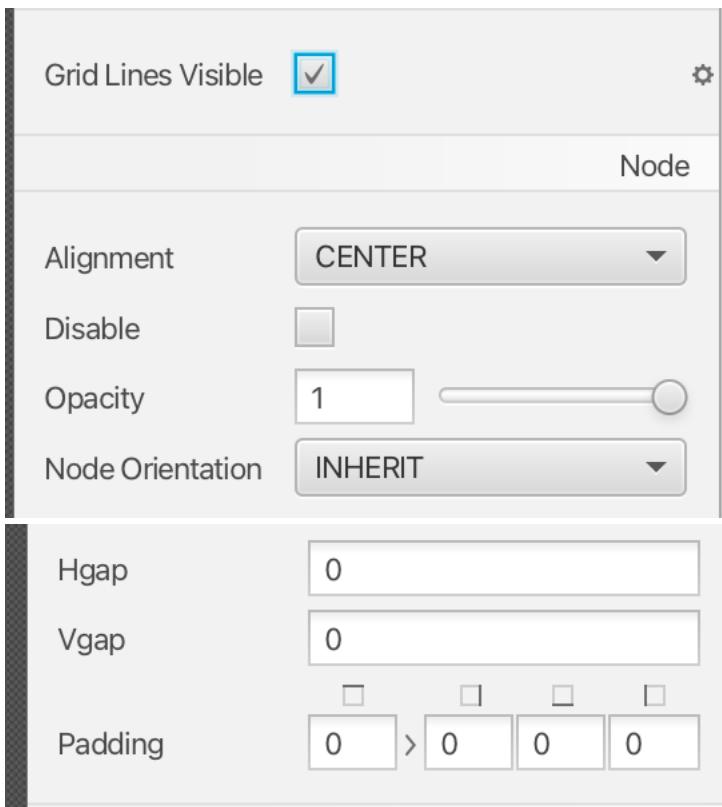
- Hgap: o gap hprizontal entre os nodes
- Vgap: o gap vertical entre os nodes
- Padding: o espaçamento referente entre as bordas do Pane e os nodes dentro dele.

GridPane:

O gridPane é uma forma de organização de um pane em uma matriz, podendo se adicionar colunas e linhas e fazendo com que cada coluna e cada linha tenha a mesma proporção, dessa forma mantendo um padrão e trabalhando apenas com o alinhamento de controler do espaço dentro de uma linha e coluna especifica, vamos ver mais sobre um gridPane no SceneBuilder.

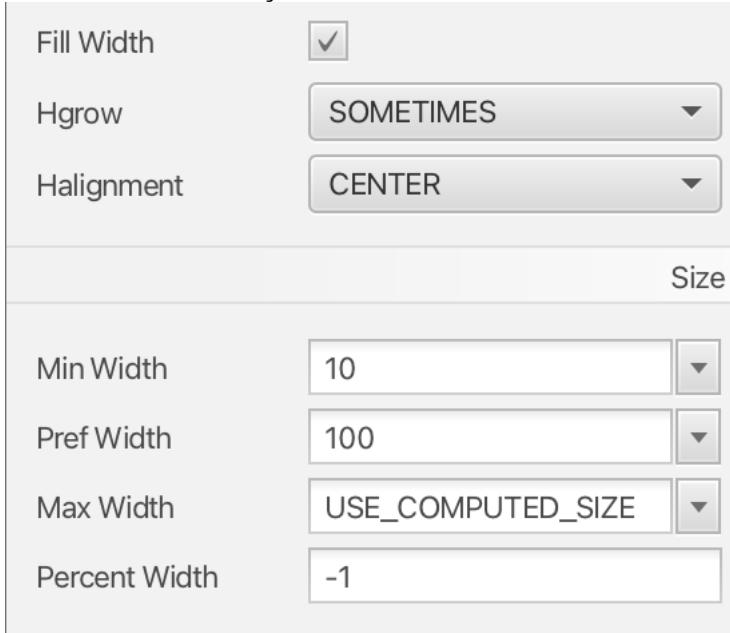


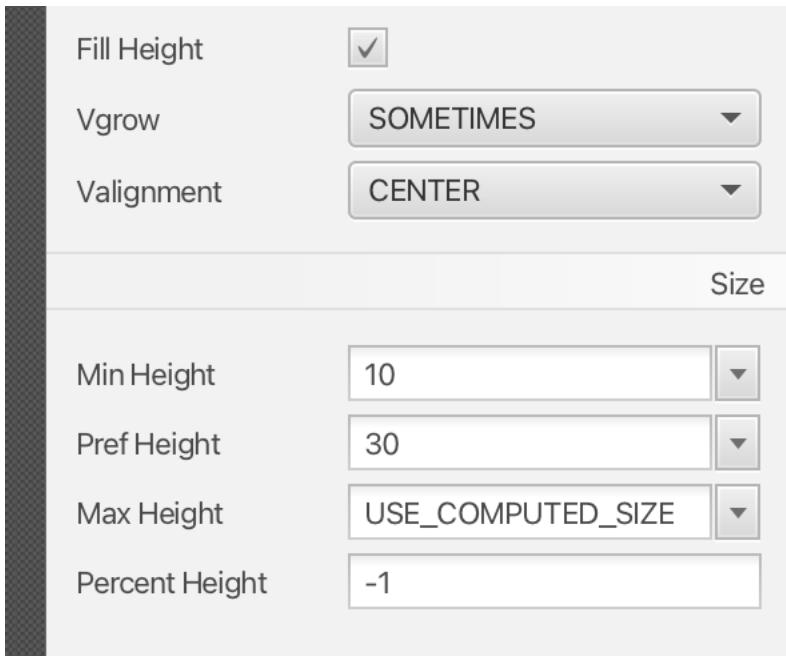
Aqui podemos adicionar linhas e colunas, adicionando elas antes ou depois das linhas e colunas já existentes, é importante ressaltar que quando modificamos o tamanho do gridPane, ele se adapta modificando o tamanho das linhas e colunas da matriz proporcionalmente.



- **GridLinesVisible:** determina se as linhas de divisão tanto das colunas como das linhas ficarão visíveis ou não.
- **Alignment:** Esse alignment é do gridPane em relação a outras panes
- **Hgap e Vgap:** gap horizontal e vertical dos nodes dentro do gridPane
- **Padding:** margem dos nodes em relação ao gridPane

Devemos ressaltar que podemos modificar as opções exclusivamente das linhas e das colunas separadamente, o mais importante é modificar o alinhamento dos nodes em relação a linha horizontal e em relação a coluna vertical:

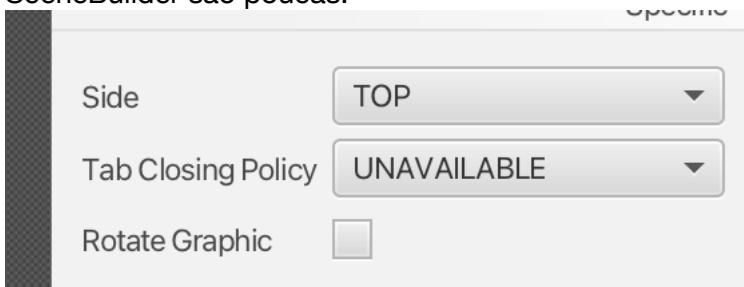




- Halignment e Valignment: Alinhamento do node em relação ao grid pane, horizontal e vertical respectivamente.

TabPane:

O tabPane é uma janela com escolhas que podem ser feitas, nesse caso as opções do SceneBuilder são poucas:



- Side: Local onde a barra de Tabs vai aparecer
- Tab Closing Policy: Onde e como vão aparecer as opções de fechar permanentemente aquele tab.

No TabPane colocamos tabs e nesses tabs podemos por um Pane e trabalhar com diversos nodes dentro dele, criando assim uma ideia de janela interativa, essa interação já é padrão do TabPane, portanto não precisamos declarar nenhum ID para ele, porém se queremos adicionar uma imagem para cada um dos tab's podemos utilizar ids individuais para cada um, e por meio desses IDs podemos no Controller definir sua imagem, é importante ressaltar que para isso teremos que utilizar o método initialize da interface implementada Initializable:

```

public class Controller17 implements Initializable{

    @FXML
    private Tab tab1;
    @FXML
    private Tab tab2;
    @FXML
    private Tab tab3;

    private ImageView addGraphics(String url) {

        ImageView imageView = new ImageView(new Image(getClass()
        .getResourceAsStream(url)));
        imageView.setFitHeight(value:25);
        imageView.setFitWidth(value:25);

        return imageView;
    }

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        tab1.setGraphic(addGraphics(url:"/assets/icon1.png"));
        tab2.setGraphic(addGraphics(url:"/assets/shrek1.
        jpg"));
        tab3.setGraphic(addGraphics(url:"/assets/shrek2.
        jpg"));
    }
}

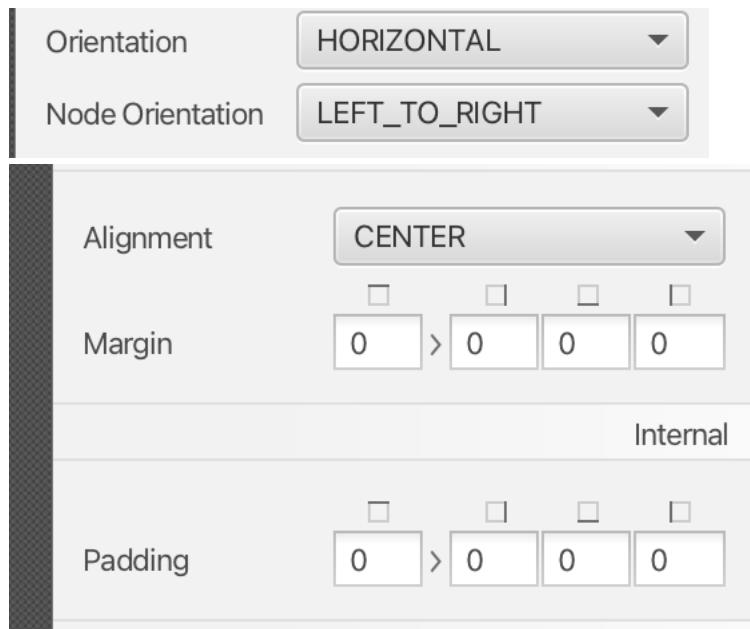
```

ToolBar e BorderPane:

Geralmente utilizamos o toolbar juntamente com o border pane para melhor organização da toolbar, primeiro vamos explicar melhor o BorderPane.

O BorderPane é simples, é apenas um pane de organização dividido em 4 partes, sendo elas o Top, Bottom, Left e Right, cada uma dessas partes pode receber um node ou até mesmo outro Pane, ele possui assim como tudo no javafx padding e alignment, porém isso só é feito quando utilizamos e colocamos algo dentro dele.

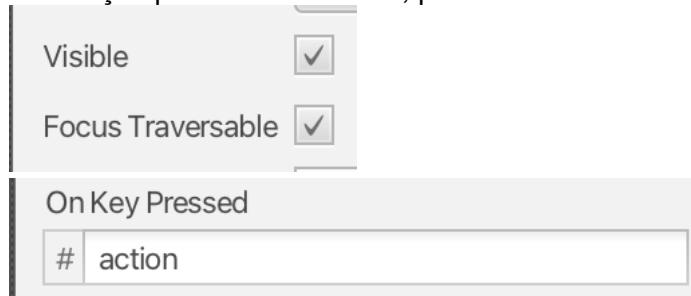
Já a toolbar é uma barra de ferramentas geralmente colocada no Top do BorderPane, e ela pode receber diversos nodes, nela temos algumas configurações que podemos fazer pelo SceneBuilder:



- Orientation: Se a toolbar vai ser horizontal ou vertical.
- Node Orientation: a orientação dos nodes dentro dela, sendo direita para esquerda ou esquerda para direita.
- Alignment: alinhamento dela em relação ao seu node root
- Margin e Padding: espaçamento entre ela e o seu node root, e espaçamento dos nodes dentro dela.

KeyEvents:

Primeiro para utilizar KeyEvents algumas coisas precisam de atenção e dependendo algumas mudanças precisam ser feitas, primeiramente no SceneBuilder:



É importante que o Node que vai receber o método do On Key Pressed esteja com Visible ligado e com o Focus Traversable ligado também, caso mesmo assim estaja ocorrendo erros, a forma de usar o KeyEvent deverá ser da maneira desse vídeo a seguir: https://youtu.be/tq_0im9qc6E?si=mA1oTisZmDTaEwqT

Caso esteja funcionando o código segue abaixo:

```

public class Controller19{

    @FXML
    ImageView imageview;
    @FXML
    Label label;

    Image img = new Image(getClass().getResourceAsStream
    (name:"/assets/seta.png"));

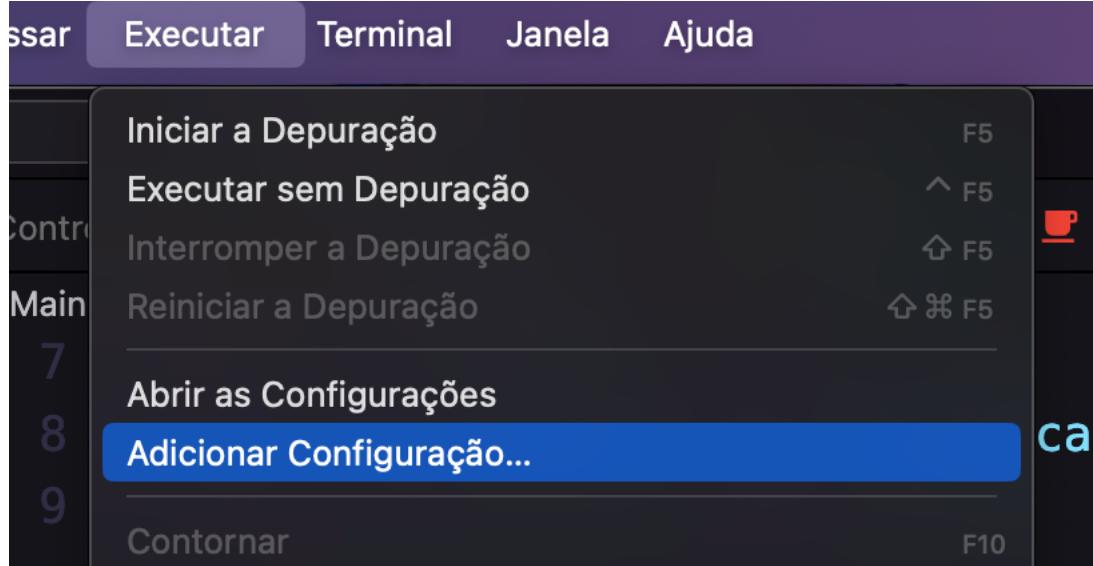
    @FXML
    void action(KeyEvent event) {
        label.setVisible(value:false);
        imageview.setImage(img);

        if(event.getCode().equals(KeyCode.RIGHT)){
            imageview.setRotate(value:0);
        }
        if(event.getCode().equals(KeyCode.LEFT)){
            imageview.setRotate(value:180);
        }
        if(event.getCode().equals(KeyCode.UP)){
            imageview.setRotate(value:270);
        }
        if(event.getCode().equals(KeyCode.DOWN)){
            imageview.setRotate(value:90);
        }
    }
}

```

MediaView:

Primeiramente a partir de agora devemos declarar vmArgs para prosseguir com os Nodes, no caso do mediaView não é diferente, primeiramente vamos aprender a colocar os vmArgs antes de tudo / Obs: o vmAgrs só é necessário caso não esteja utilizando java8.



Com a sua mainClass selecionada vá em executar e adicione Configurações:

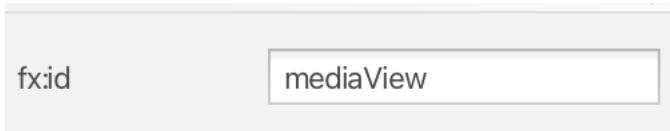
```
{  
    // Use o IntelliSense para saber mais sobre os atributos  
    // possíveis.  
    // Focalizar para exibir as descrições dos atributos  
    // existentes.  
    // Para obter mais informações, acesse: https://go.microsoft.com/fwlink/?linkid=830387  
    "version": "0.2.0",  
    "configurations": [  
        {  
            "type": "java",  
            "name": "Current File",  
            "request": "launch",  
            "mainClass": "${file}"  
        },  
        {  
            "type": "java",  
            "name": "Main",  
            "request": "launch",  
            "mainClass": "Main",  
            "projectName": "Aula08_249f3f27",  
            "vmArgs": "--add-modules javafx.controls,javafx.media,javafx.fxml,javafx.web,javafx.swing,javafx.graphics"  
        }  
    ]  
}
```

você irá para essa tela agora é só adicionar embaixo o vmArgs assim como está acima:

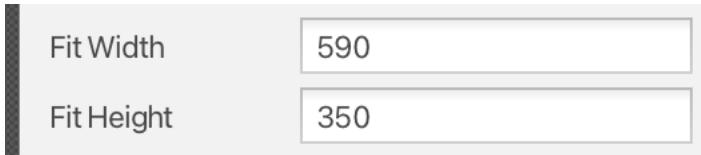
```
,"vmArgs": "--add-modules  
javafx.controls,javafx.media,javafx.fxml,javafx.web,javafx.swing,javafx.graphics"
```

Dessa forma será possível usar os Nodes de media, de webView e etc...

Seguindo agora vamos trabalhar com o MediaView no SceneBuilder, primeiramente a única coisa que declaramos no ScenBuilder do mediaView é o seu ID, porém é importante criar buttons com as funções de play, pause e etc...



A unica forma de modificar o seu tamanho é indo na aba de layout e modificando sua larhura e sua altura:



Agora no código existem muitas coisa pequenas que modificam tudo no mediaView:

```
public class Controller20 implements Initializable {  
  
    @FXML  
    private MediaView mediaView;  
    @FXML  
    private Button playBtt;  
    @FXML  
    private Button pauseBtt;  
    @FXML  
    private Button resetBtt;  
  
    private File file;  
    private Media media;  
    private MediaPlayer mediaPlayer;
```

primeiramente injetamos os buttons e o mediaView do FXML, logo após precisaremos de 3 outros objetos, sendo eles o File, o Media e o MediaPlayer, todos de classes javaFX. Além disso é necessário a criação do método initialize da interface Initializable para configurar nosso vídeo.

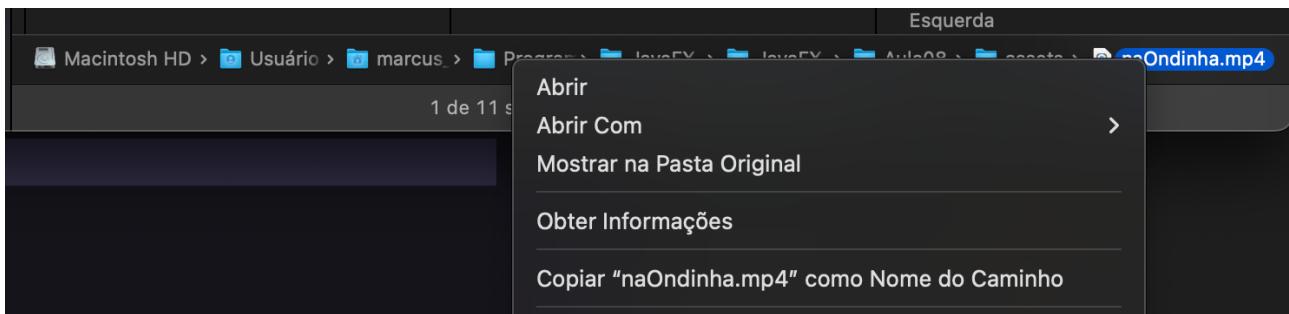
```
@Override  
public void initialize(URL location, ResourceBundle  
resources) {  
    file=new File(pathname:"/Users/marcus_cs_pereira/  
    Programação/JavaFX_Projects/JavaFX-Studies/Aula08/  
    assets/naOndinha.mp4");  
    media=new Media(file.toURI().toString());  
    mediaPlayer=new MediaPlayer(media);  
    mediaView.setMediaPlayer(mediaPlayer);  
}
```

Para iniciar e colocar um vídeo no nosso mediaPlayer nós primeiro criamos um file com um Path, esse Path é diferente dos outros, vc necessita que ele saia da raiz do programa por isso você deve utilizar:

./assets/naOndinha.mp4

Porém caso não funcione o path precisa ser extremamente específico e explicarei sobre isso logo abaixo, depois instanciamos a media passando o nosso file para URI e depois para String, depois definimos a media dentro no nosso mediaPlayer, e para podermos ver o vídeo nós definimos o MediaPlayer do nosso mediaView.

Explicando sobre o Path: as vezes o path do nosso vídeo gera problemas par ser encontrado, para isso precisamos pegar o vídeo direto da raiz da nossa máquina, sendo assim necessitamos copiar o endereço desde o nosso SistemaOperacional até o vídeo, no Windows é muito simples, já no MacBook é da seguinte forma:



Chegue no arquivo pelo finder, clique com o botão direito nele e escolha a opção de copiar como nome do caminho, dessa forma temos o caminho do vídeo.

Agora vamos definir as funções dos nossos botões para executar o vídeo e etc:

```

public void play(){
    mediaPlayer.play();
}

public void pause(){
    mediaPlayer.pause();
}

public void reset(){
    if (mediaPlayer.getStatus() != MediaPlayer.Status.
READY) {
        mediaPlayer.seek(Duration.seconds(s:0.0));
        mediaPlayer.play();
    }
}

```

- Deve ser ressaltado que esse If no método reset é de suma importância para que o vídeo não travie e gere um erro.

WebView:

Para começáros é importante ressaltar que o webView também necessita de um vmArgs assim como o MediaView acima, caso use o mesmo vmArgs do mediaView acima, o WebView funcionará.

O WebView apenas necessita no SceneBuilder de um ID, porém para melhor funcionamento é recomendado que seja usado buttons com handleEvents OnAction definidos e também um textField com ID e um HandleEvent OnAction também.

Agora vamos ver o código do nosso webView:

```

public class Controller21 implements Initializable{

    @FXML
    private WebView webView;
    @FXML
    private TextField textField;

    private WebEngine engine;
    private WebHistory history;

    private String homePage = "www.google.com.br";

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        engine = webView.getEngine();
        textField.setText(homePage);
        loadPage();
    }

    public void loadPage(){
        engine.load("https://" + textField.getText());
    }
}

```

Primeiramente para o nosso WebView nós vamos precisar injetar ele no nosso código com o @FXML e junto a ele vamos necessitar de um objeto instanciado WebEngine, os objetos WebHistory e a String home não são estritamente necessários, porém ajudam e facilitam a usabilidade do nosso webview. Além disso é necessário o uso do método initialize da interface Initializable para funcionamento do nosso webView.

Primeiro no nosso método initialize nós pegamos e declaramos a nossa engine por meio do nosso webView.getEngine();, logo após setamos o nosso texto do textFiel pois ele que executa a troca de webSite, e logo após executamos o método loadPage(); que us o método da nossa engine chmdo engine.load("url do site") que carrega o nosso site.

Porém temos outra forma de carregar nosso site, sendo essa uma forma que evita erros e trás uma melhor experiência para o usuário, para isso vamos usar import dessa forma:

```
import java.net.HttpURLConnection
```

```

public void loadPage() {
    String urlToLoad = "https://" + textField.getText();

    if (isValidURL(urlToLoad)) {
        engine.load(urlToLoad);
    } else {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle(title:"URL Inválida");
        alert.setHeaderText(headerText:"ERRO");
        alert.setContentText(contentText:"A URL fornecida
não é válida.");
        alert.showAndWait();
    }
}

private boolean isValidURL(String url) {
    try {
        URL u = new URL(url);
        HttpURLConnection connection =
        (HttpURLConnection) u.openConnection();
        connection.setRequestMethod(method:"HEAD");
        int code = connection.getResponseCode();
        return (code == HttpURLConnection.HTTP_OK);
    } catch (IOException e) {
        return false;
    }
}

```

Assim podemos testar antes se a URL digitada pelo usuário é válida ou não, caso seja válida ele avança, caso não seja, ele informa que a URL não é válida e faz o teste.

Agora o que veremos a seguir são apenas métodos que podemos atribuir a botões e etc para executar ações no nosso WebView:

```
public void loadPage(){
    engine.load("https://" + textField.getText());
}

public void refreshPage(){
    engine.reload();
}

public void zoomIn(){
    webView.setZoom(webView.getZoom() + 0.25);
}

public void zoomOut(){
    webView.setZoom(webView.getZoom() - 0.25);
}
```

Aqui temos o método de carregar nossa página, dar zoom e tirar zoom.

```
public void history(){

    history = engine.getHistory();
    ObservableList<WebHistory.Entry> entries = history.
    getEntries();

    String historico="";
    for(WebHistory.Entry entry : entries){
        String url = entry.getUrl() + " " + entry.
        getLastVisitedDate();
        historico += url + "\n";
    }

    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setTitle(title:"Histórico");
    alert.setHeaderText(headerText:"Histórico de
    navegação");
    alert.setContentText(historico);
    alert.showAndWait();

}
```

Aqui temos um método com a lógica de acessar e informar ao usuário o histórico de acesso de páginas do nosso webView e mostrar ao usuário como forma de alert informando a url do site e a sua data de acesso.

```

public void back(){
    history = engine.getHistory();
    ObservableList<WebHistory.Entry> entries = history.
    getEntries();

    int currentIndex = history.getCurrentIndex();

    if(currentIndex > 0){
        history.go(-1);
        textField.setText(entries.get(currentIndex - 1).
        getUrl());
    }
}

public void foward(){
    history = engine.getHistory();
    ObservableList<WebHistory.Entry> entries = history.
    getEntries();

    int currentIndex = history.getCurrentIndex();

    if(currentIndex + 1 < entries.size()){
        history.go(offset:1);
        textField.setText(entries.get(currentIndex + 1).
        getUrl());
    }
}

```

Aqui temos a lógica de avançar e retornar em sites que acessamos por meio de acesso a ObservableList do histórico e assim obtendo os index de cada um dos sites acessados e retornando a eles, temos também um controle para erros com esse IF.

```

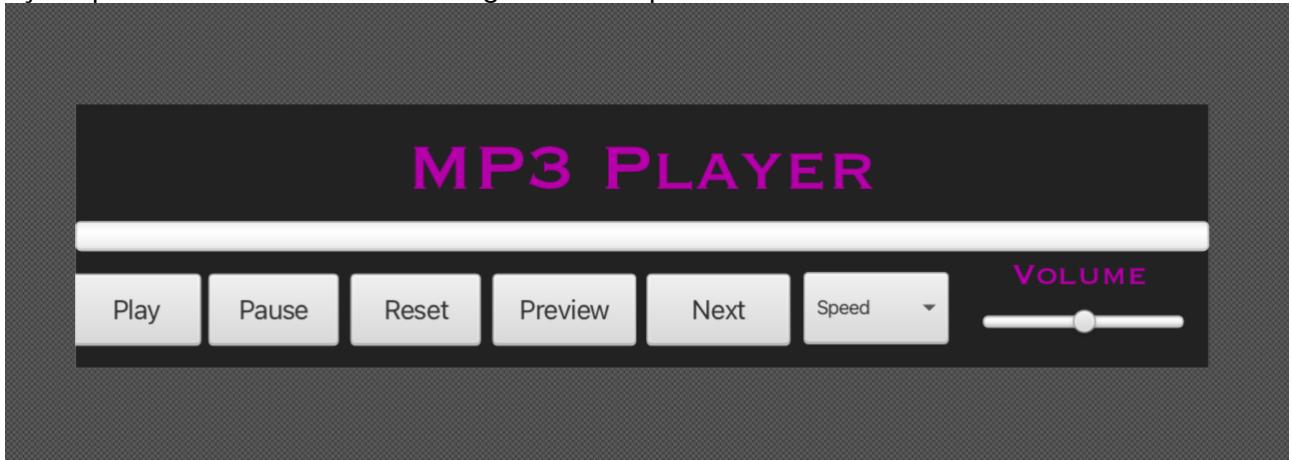
public void executeJS(){
    engine.executeScript(script:"window.location =
    \"https://www.youtube.com\";");
    textField.setText(value:"https://www.youtube.com");
}

```

E por fim temos um método de executar comandos JavaScript no nosso WebView.

Music Player:

Primeiramente vamos entender a nossa construção para entender o MusicPlayer, para utilizar um musicPlayer necessitamos apenas de um pane com ID, porém para controlar a musica e todas as ações possíveis é necessário usar alguns Nodes para facilitar esse controler:



Nesse caso estamos usando uma label, uma progressBar, 5 buttons e um comboBox.

Agora vamos para o código do nosso Controller:

Primeiro vamos ver as declarações necessária para esse código

```

public class Controller22 implements Initializable {

    @FXML
    AnchorPane pane;
    @FXML
    Label label;
    @FXML
    Button playBtt, pauseBtt, resetBtt, previewBtt, nextBtt;
    @FXML
    ProgressBar progressBar;
    @FXML
    ComboBox<String> comboBox;
    @FXML
    Slider slider;
    //Declarando o media player.
    private Media media;
    private MediaPlayer mediaPlayer;

    //Basicamente a organização das músicas.
    private File directory;
    private File[] files;
    private ArrayList<File> songs;
    private int songNumber=0;
    //Controle de tempo da música.
    private Timer timer;
    private TimerTask task;
    private boolean isPlaying;
    //Controle de velocidade da música.
    private int[] speeds = {25, 50, 75, 100, 125, 150, 175,
    200};
}

```

Primeiro injetamos todos os Nodes com Id's declarados no SceneBuilder, depois declaramos o nosso Media e o nosso MediaPlayer, basicamente isso seria o necessário para tocar uma música porém para fazer algo melhor organizado usaremos:

Uma organização de músicas criando uma File que terá o diretório, um array de File que terá todos os arquivos dentro de uma certa pasta, um ArrayList de Files que terá todos esses Files do array e por ultimo teremos o número da música para seguirmos uma ordem nesse ArrayList.

também usaremos um temporizador para calcular a progressão da música e passar para nossa progressBar e junto a isso passar de música quando ela acabar, faremos isso por meio de um Timer, um TimerTask e uma variável boolean para controlar se a música está ou não tocando.

Junto a tudo isso necessitaremos do método initialize da interface Initializable para iniciar diversos Nodes, preencher o ArrayList com as músicas, iniciar o Timer e declarar listeners.

Primeiramente vamos adicionar as músicas da pasta music no ArrayList:

```
@Override  
public void initialize(URL location, ResourceBundle resources) {  
  
    songs = new ArrayList<File>();  
    directory = new File(pathname:"./assets/music");  
    files = directory.listFiles();  
    if (files!= null) {  
        for (File file : files) {  
            songs.add(file);  
            //System.out.println(file);  
        }  
    }  
}
```

dessa forma o diretório recebe uma pasta com música, e com o método list files fazemos o array files receber os arquivos.mp3, e para cada arquivo .mp3 adicionamos ele no ArrayList para ter controle sobre os índices.

```
//Inicializando o media player com a música 1.  
media= new Media(songs.get(songNumber).toURI()  
.toString());  
mediaPlayer = new MediaPlayer(media);  
label.setText(songs.get(songNumber).getName());
```

Aqui inicializamos o mediaPlayer com a música 1, modificando o label para o nome da música.

```
//Inicializando o choice box.  
for (int i = 0; i < speeds.length; i++) {  
    comboBox.getItems().add(speeds[i]+"%");  
}  
comboBox.setOnAction(this::speed);
```

Aqui preenchemos o comboBox com os elementos do array Speed e além disso declaramos o setOnAction do comboBox com o método Speed que será demonstrado abaixo.

```
//Inicializando o slider.  
slider.valueProperty().addListener(new  
ChangeListener<Number>() {  
  
    @Override  
    public void changed(ObservableValue<? extends  
Number> observable, Number oldValue, Number  
newValue) {  
        mediaPlayer.setVolume(slider.getValue() * 0.  
01);  
    }  
  
});  
  
//Mudando cor da ProgressBar.  
progressBar.setStyle(value: "-fx-accent: #00ff00;");
```

Aqui adicionamos o change listener do slider de controle de volume do mediaPlayer e junto a isso mudamos a cor da nossa progressBar

Agora vamos demonstrar todos os métodos e explica-lós detalhadamente:

Primeiramente os métodos do Timer que são importantes para toda execução do programa principalmente para a animação da ProgressBar de acordo o andar da música:

```
public void beginTimer() {
    timer = new Timer();
    task = new TimerTask() {
        @Override
        public void run() {
            isPlaying = true;
            double current = mediaPlayer.getCurrentTime().
                toSeconds();
            double end = mediaPlayer.getCycleDuration().
                toSeconds();

            // Execute o código relacionado à interface do
            // usuário na Thread da Aplicação FX
            Platform.runLater(() -> {
                progressBar.setProgress(current / end);

                if (current / end == 1) {
                    endTimer();
                    next();
                }
            });
        }
    };
    timer.scheduleAtFixedRate(task, delay:1000, period:1000);
}
```

```
public void endTimer() {
    isPlaying = false;
    timer.cancel();
}
```

Aqui, uma instância de Timer é criada e atribuída à variável de instância timer. A classe Timer em Java é usada para agendar tarefas para serem executadas em momentos específicos no futuro.

Após isso uma nova instância anônima de TimerTask é criada e atribuída à variável de instância task. Uma TimerTask representa uma tarefa que pode ser agendada para execução.

Depois temos o método run(). Este método será chamado quando a tarefa for executada pelo temporizador, junto a isso a variável isPlaying é definida como true, isso indica que alguma reprodução de mídia está em andamento.

As variáveis double obtêm o tempo atual de reprodução (current) e a duração total da reprodução (end) em segundos.

`Platform.runLater()` é usado para garantir que o código dentro do bloco seja executado na Thread da Aplicação FX. Isso é importante quando se lida com interfaces gráficas em JavaFX, pois a manipulação da interface do usuário deve ocorrer na thread da aplicação, isso é necessário para correções de erro nessa execução.

`progressBar.setProgress(current / end);` Aqui, a barra de progresso é atualizada com o progresso atual da reprodução, calculado como a divisão do tempo atual pelo tempo total.

Se o progresso atingir 100% (ou seja, `current / end` é igual a 1), então o método `endTimer()` é chamado e, em seguida, a função `next()` é chamada. Isso indica que a reprodução atingiu o final e ai realiza transição para a próxima faixa.

No método `endTimer()`, declaramos como false a variável `isPlaying` para sinalizar que não tem música tocando no momento e após isso usamos o `timer.cancel()`, que finaliza o timer.

Finalmente, a tarefa (`task`) é agendada para execução periódica usando `timer.scheduleAtFixedRate()`. A tarefa será executada a cada 1000 milissegundos (1 segundo), repetindo-se a cada segundo.

Resumidamente, este código cria um temporizador que atualiza uma barra de progresso com base no tempo de reprodução de um objeto `mediaPlayer`. Quando a reprodução atinge o final, algumas ações adicionais são realizadas, como chamar métodos `endTimer()` e `next()`. O código utiliza `Platform.runLater()` para garantir que as atualizações da interface do usuário ocorram na Thread da Aplicação FX.

O Timer é usado para agendar a execução de tarefas em intervalos regulares, enquanto o `TimerTask` representa a tarefa que será executada. No caso desse código, a tarefa atualiza a interface do usuário relacionada à reprodução de mídia e executa ações específicas quando a reprodução atinge o final.

Vamos agora explicar melhor sobre a `Platform.runLater()`:

JavaFX é uma biblioteca gráfica para criação de interfaces de usuário em Java. No JavaFX, a maioria das operações que manipulam a interface gráfica deve ser realizada na Thread da Aplicação FX, também conhecida como JavaFX Application Thread. Isso ocorre porque a maioria dos componentes gráficos não é thread-safe, o que significa que não podem ser manipulados a partir de várias threads simultaneamente sem o risco de problemas de concorrência.

A `Platform.runLater()` é uma maneira de garantir que um bloco de código seja executado na Thread da Aplicação FX. Isso é fundamental quando você está manipulando a interface do usuário, como atualizações de componentes gráficos, alterações de cena, entre outras operações que envolvem a GUI.

Vamos analisar a parte específica do código onde `Platform.runLater()` é utilizada:

O bloco de código dentro do `runLater()` está atualizando a barra de progresso (`progressBar.setProgress(current / end)`) e realizando verificações relacionadas ao término da reprodução de mídia. Isso é feito dentro do `runLater()` para garantir que essas operações de interface do usuário ocorram na Thread da Aplicação FX.

Se essas operações fossem realizadas diretamente fora do `runLater()`, poderiam ocorrer problemas de concorrência e inconsistências na interface do usuário, já que o código pode estar sendo executado em uma thread diferente da Thread da Aplicação FX.

Em resumo, `Platform.runLater()` é uma ferramenta crucial para garantir a integridade e consistência da interface do usuário em aplicações JavaFX, pois permite que você move tarefas que afetam a GUI para a Thread da Aplicação FX, evitando assim problemas de concorrência e garantindo que as mudanças na GUI sejam feitas de formas corretas, garantindo a maior experiência pro usuário.

Agora vamos para os outros métodos da aplicação:

```
public void play() {
    beginTimer();
    mediaPlayer.play();
}

public void pause() {
    endTimer();
    mediaPlayer.pause();
}

public void reset() {
    progressBar.setProgress(value:0);
    mediaPlayer.seek(Duration.seconds(s:0.0));
}
```

Aqui temos o método play que inicia a música e o timer, junto a isso temos o pause que pausa a música e finaliza o timer, porém não zera a progressBar, logo quando voltamos a executar a lógica ainda funciona, já no reset há a necessidade de zerar a progressBar para dar funcionalidade ao método reset.

```
public void speed(ActionEvent event) {
    //mediaPlayer.setRate(Integer.parseInt(comboBox.
    getValue()) * 0.01);
    //Como uso a porcentagem preciso retira-lá aqui, e
    testar para ver se o valor é nulo, para quando passar
    de música manter a velocidade.
    if(comboBox.getValue() == null){
        mediaPlayer.setRate(value:1);
    }else{
        mediaPlayer.setRate(Integer.parseInt(comboBox.
        getValue().substring(beginIndex:0, comboBox.
        getValue().length()-1)) * 0.01);
    }
}
```

Aqui temos o método Speed, que primeiro tem um controle de segurança caso eu passe de tela sem modificar a velocidade ele iria retornar null, então daria um NullPointerException, por isso

usamos esse If, afinal estamos usando esse método Speed para que quando trocarmos de música a velocidade ainda continue alterada, para alterar a velocidade modificamos o Rate do mediaPlayer passando um inteiro de 0 a 1 que define a velocidade da música, além disso utilizamos uma substring para retirar a % presente no combobox.

```
public void next() {
    if(songNumber < songs.size()-1) {
        songNumber++;
        mediaPlayer.stop();

        if(isPlaying){
            endTimer();
        }

        media= new Media(songs.get(songNumber).toURI().
        toString());
        mediaPlayer = new MediaPlayer(media);
        label.setText(songs.get(songNumber).getName());
        play();
        speed(event:null);
    }else{
        songNumber = 0;
        mediaPlayer.stop();

        if(isPlaying){
            endTimer();
        }

        media= new Media(songs.get(songNumber).toURI().
        toString());
        mediaPlayer = new MediaPlayer(media);
        label.setText(songs.get(songNumber).getName());
        play();
        speed(event:null);
    }
}
```

Aqui temos o método para passar de música e nele temos vários pequenos detalhes de suma importância, primeiro temos um controle de segurança por meio de um if que checa o número da música pelo índice dela no ArrayList songs que nesse caso é usado a variável songNumber para controle desse index, após essa verificação aumentamos o número simbolizando que alteramos

de música, paramos o mediaPlayer da música anterior, fazemos uma verificação que caso essa música seja passada enquanto ainda está sendo executada nós paramos o timer.

Após tudo isso declaramos a nova música no MediaPlaye, mudamos o nome da música no label, tocamos a música por meio do método play() que liga novamente o timer, e como utilizamos a lógica para calcular a progressBar baseado no tempo atual da música com o tempo total dela, a progressBar é resetada com o timer, e como também escolhemos passar a velocidade da choicebox da música anterior pra essa precisamos chamar o método speed passando null como argumento pois ele recebe um ActionEvent.

Esse else é necessário apenas para que a “playlist” segue em loop, então quando eu chegar na última música eu volte para a primeira, por isso a única mudança do if é a declaração do songNumber que é o index do ArrayList.

```

public void preview() {
    if (songNumber > 0) {
        songNumber--;
        mediaPlayer.stop();

        if(isPlaying){
            endTimer();
        }
        media= new Media(songs.get(songNumber).toURI().
        toString());
        mediaPlayer = new MediaPlayer(media);
        label.setText(songs.get(songNumber).getName());
        play();
        speed(event:null);
    }else{
        songNumber = songs.size()-1;
        mediaPlayer.stop();

        if(isPlaying){
            endTimer();
        }
        media= new Media(songs.get(songNumber).toURI().
        toString());
        mediaPlayer = new MediaPlayer(media);
        label.setText(songs.get(songNumber).getName());
        play();
        speed(event:null);
    }
}

```

Para o método preview a única modificação é na condição do if e no controler da variável songNumber, afinal estou indo para trás, então o index do ArrayList deve ser decrementado e deve ser checado se o index está no 0, para assim ir para a última música do ArrayList.

Logo para o MusicPlayer, utilizamos diversos métodos ligados a Nodes e suas diferentes lógicas, além de trabalhar com Timer e até mesmo com Concorrência e Threads para a lógica da ProgressBar ser referente ao tempo da música.

Detalhamento e resumo sobre principais Panes:

Alguns Panes podem ser usados para melhor organização de nodes na sua tela, aqui vão alguns exemplos e resumos sobre eles:

Vbox e Hbox:

Podem ser usados como panes para organizar com espaçamento igual vertical ou horizontalmente dos seus nodes dentro da aplicação.

StackPanes:

Pode ser usado para colocar Nodes “dentro” de outros nodes, um exemplo seria colocar um checkBox dentro de um TextField, dessa forma simulando a ideia de clique para revelar uma senha por exemplo.

BorderPane:

O BorderPane divide a cena em cinco áreas distintas: superior, inferior, esquerda, direita e centro. É útil para layouts mais complexos, onde diferentes partes da interface ocupam diferentes regiões da tela.

GridPane:

O GridPane organiza os elementos em uma grade, permitindo que você especifique a posição dos elementos usando linhas e colunas. É adequado para layouts mais flexíveis e controlados.

AnchorPane:

O AnchorPane é usado para ancorar elementos em relação às bordas da cena. Ele é útil quando se deseja fixar um elemento em uma posição específica ou ancorá-lo em relação a outro elemento.

DialogPane:

O DialogPane é comumente usado para criar caixas de diálogo em JavaFX. Ele fornece uma estrutura para adicionar conteúdo à caixa de diálogo, como botões, rótulos e campos de texto.

FlowPane:

FlowPane organiza nodes em uma sequência flexível, tentando ajustá-los automaticamente ao tamanho disponível. É útil quando você deseja que os nodes sejam automaticamente redimensionados para se encaixarem no espaço disponível.

TilePane:

TilePane é semelhante ao FlowPane, mas tenta ajustar os nodes para formar uma grade regular, mantendo o tamanho dos nodes constante.

ScrollPane:

ScrollPane fornece uma área de rolagem para nodes que não cabem completamente na tela. É útil quando você tem uma grande quantidade de conteúdo que precisa ser exibida em uma área limitada.

Accordion e TitledPane:

Accordion e TitledPane são úteis quando você deseja criar interfaces de usuário com seções expansíveis e recolhíveis. Accordion contém vários TitledPane, cada um com um título e um conteúdo associado.

TabPane:

TabPane é usado quando você deseja organizar seu conteúdo em abas, permitindo que os usuários alternem entre diferentes conjuntos de nodes.

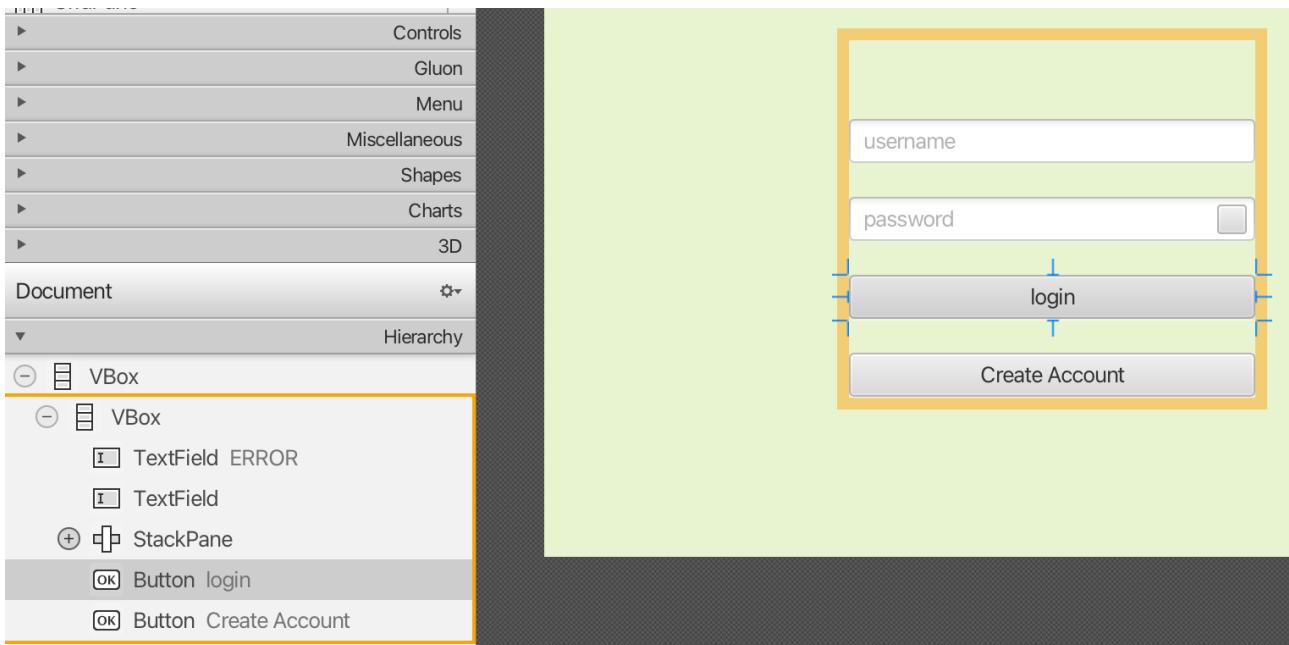
SplitPane:

SplitPane permite dividir a tela em duas ou mais áreas redimensionáveis, facilitando a interação do usuário para ajustar o tamanho relativo de diferentes partes da interface.

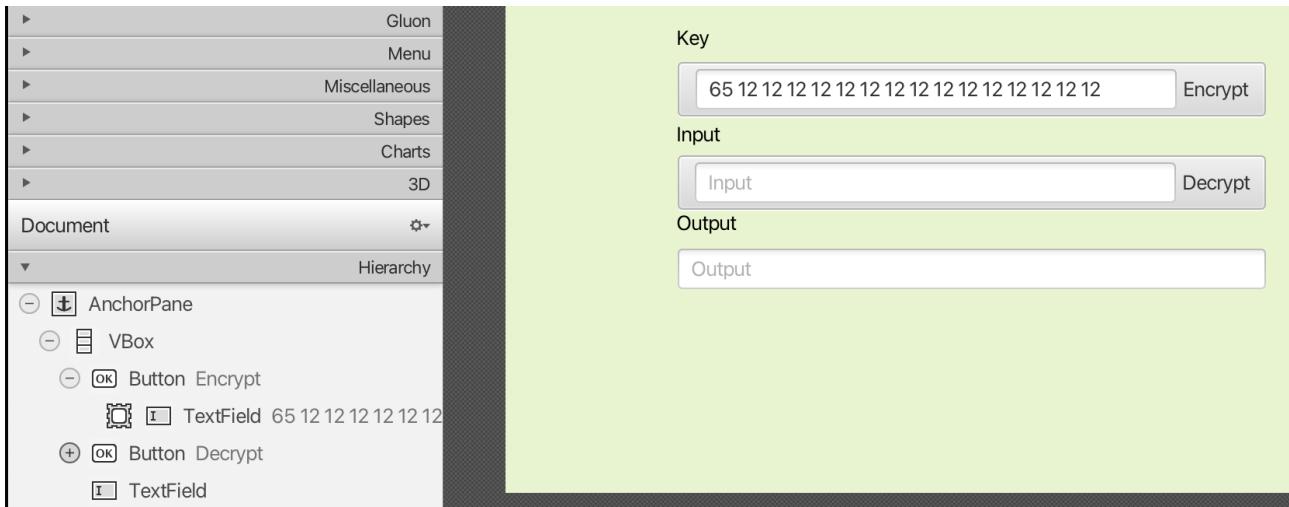
Extra(Buttons):

Outro fator importante é que dentro de buttons podemos colocar um Node, para isso no SceneBuilder, é só arrastar o node para o button pela barra de Hierarchy no SceneBuilder, dessa forma você consegue criar elementos de botão com um textfield dentro por exemplo.

Aqui vão alguns exemplos:



Utilização de VBox para alinhamento perfeito dos Nodes e do StackPane para por o checkBox dentro do textField.



Utilização de Vbox para alinhamento dos itens e Utilização de um textField dentro de um button.

#ID

