

JavaFX Studies/ Aula06:

Entendendo sobre Utilização de imports para compilação correta do programa , Loader, vmArgs, Trocando de telas e Comunicação entre controladores

Importante: Para os trabalhos de concorrente, temos que nós atentar a seguintes casos:

- Como ele irá compilar e executar o programa apenas usando o comando `javac Principal.java`, quando digitarmos esse comando ele deve gerar todos os `.class` de todos os nossos arquivos `.java` no programa, para que isso ocorra vamos utilizar imports, importante nossas classes, principalmente os nossos controllers, aqui vai um exemplo:

```
import controller.ControllerTela1;// Importa o controller da tela principal para poder compilar o
programa
import controller.ControllerTela2;// Importa o controller da tela secundaria para poder compilar o
programa

@SuppressWarnings("unused");// Suprime os avisos de variaveis nao utilizadas, nesse caso, os imports
dos controllers, mesmo esses imports nao estarem sendo utilizados nessa classe em especifico, eles
sao necessarios para a compilacao correta do programa

public class Principal extends Application {
```

- Importamos os controllers, e dentro deles eles possuem já as importações das outras classes que estão no model, util e etc, sendo assim para testarmos quando digitarmos `javac Principal.java`, esse comando deve agora gerar todos os `.class` de todos os nossos arquivos `.java`
- o `@SuppressWarnings("unused")` serve para suprimir os avisos de variáveis não utilizadas, nesse caso, os imports dos controllers, mesmo esses imports não estarem sendo utilizados nessa classe em específico, eles são necessários para a compilação correta do programa, portanto esse `@Suppress` é apenas opcional e para uso “estético”, basicamente para remover aqueles avisos amarelos que aparecem no código quando não estamos utilizando diretamente aquele código na classe em específico, porém ele é importante para funcionamento do programa em geral.

Loader:

O `FXMLLoader` é uma classe fornecida pelo JavaFX para carregar interfaces de usuário definidas em FXML (FXML é uma linguagem de marcação baseada em XML usada para definir interfaces

gráficas no JavaFX). O FXMLLoader permite carregar um arquivo FXML e criar uma hierarquia de objetos de interface do usuário correspondente.

A principal diferença entre usar o FXMLLoader e não usá-lo reside na forma como a interface do usuário é carregada e como a hierarquia de objetos é gerenciada.

Usando FXMLLoader com loader.load():

```
FXMLLoader loader = new FXMLLoader(getClass().getResource("sua_interface.fxml")); Parent root = loader.load();
```

- **Criação da Hierarquia de Objetos:**

- O FXMLLoader lida com a leitura do arquivo FXML e a criação da hierarquia de objetos correspondente.

- O método load() retorna o nó raiz (Parent) da hierarquia de objetos do FXML.

- **Controle de Eventos e Inicialização:**

- O FXMLLoader é capaz de associar controladores (classe Java) ao arquivo FXML, permitindo que você especifique a lógica de controle.

- Os métodos initialize() ou construtores definidos na classe do controlador são chamados automaticamente.

- **Injeção de Dependência:**

- O FXMLLoader é capaz de injetar automaticamente dependências no controlador, se estiver usando o recurso fx:controller no arquivo FXML.

Sem usar FXMLLoader:

```
Parent root = FXMLLoader.load(getClass().getResource("sua_interface.fxml"));
```

- **Criação Direta da Hierarquia de Objetos:**

- Neste caso, o método estático load() da classe FXMLLoader é chamado diretamente.

- A hierarquia de objetos é criada e retornada, sem a necessidade de uma instância específica do FXMLLoader.

- **Limitações no Controle de Eventos e Inicialização:**

- Sem um objeto FXMLLoader, você perde a capacidade de associar controladores diretamente e, portanto, perde alguns recursos, como a execução automática do método initialize().

- **Injeção de Dependência (Desvantagem):**

- Se você estiver usando injeção de dependência no controlador, perderá esse recurso, a menos que implemente manualmente.

Em resumo, o uso do FXMLLoader oferece mais flexibilidade, controle e recursos, especialmente quando se trata de gerenciamento de eventos, injeção de dependência e inicialização. É a abordagem preferida ao trabalhar com interfaces gráficas em JavaFX, proporcionando um código mais organizado e fácil de manter.

O uso do loader.load() com o FXMLLoader em aplicações JavaFX traz vários benefícios e é altamente recomendado para o desenvolvimento de interfaces gráficas. Vamos explorar esses benefícios e entender quando e por que você deve utilizá-lo:

Benefícios de usar loader.load():

1. Separação de Interface e Lógica:

- O FXML permite uma separação clara entre a interface do usuário e a lógica de controle. O loader.load() carrega a descrição da interface a partir do arquivo FXML, enquanto a lógica é implementada em uma classe de controlador separada.

1. Facilidade de Manutenção:

- A separação entre a interface e a lógica facilita a manutenção do código. As alterações na interface podem ser feitas no arquivo FXML sem afetar diretamente a lógica de controle.

1. Reusabilidade de Componentes:

- O FXML permite a definição de componentes reutilizáveis. Com o `loader.load()`, você pode facilmente reutilizar componentes definidos em outros arquivos FXML em diferentes partes da aplicação.

1. Controle de Eventos e Inicialização:

- O `FXMLLoader` permite associar um controlador (classe Java) ao arquivo FXML. Isso facilita o controle de eventos e a inicialização da interface, com métodos como `initialize()` sendo chamados automaticamente.

1. Injeção de Dependência:

- O `FXMLLoader` suporta injeção de dependência, permitindo que você injete automaticamente dependências no controlador usando anotações como `@FXML`.

1. Configuração de Estilos e Recursos:

- Com o `FXMLLoader`, é fácil configurar estilos, adicionar folhas de estilo CSS e vincular recursos externos, como imagens, ao arquivo FXML.

Quando Usar `loader.load()`:

1. Aplicações JavaFX Convencionais:

- Em aplicações JavaFX convencionais, onde você está construindo interfaces gráficas, é altamente recomendado usar `FXMLLoader` para aproveitar os benefícios listados acima.

1. Projetos Grandes ou de Equipe:

- Em projetos grandes ou em equipes de desenvolvimento, a separação entre interface e lógica é ainda mais crucial. O uso do `FXMLLoader` ajuda a manter um código mais modular e gerenciável.

1. Melhor Organização e Estruturação:

- Ao utilizar `loader.load()`, você organiza sua aplicação de maneira mais clara, facilitando a compreensão do código por você e por outros desenvolvedores.

Quando Não Usar `loader.load()`:

1. Aplicações Simples ou Pequenos Protótipos:

- Em aplicações muito simples ou protótipos rápidos, onde a complexidade da interface é baixa, pode ser aceitável criar a interface diretamente no código Java sem usar um arquivo FXML.

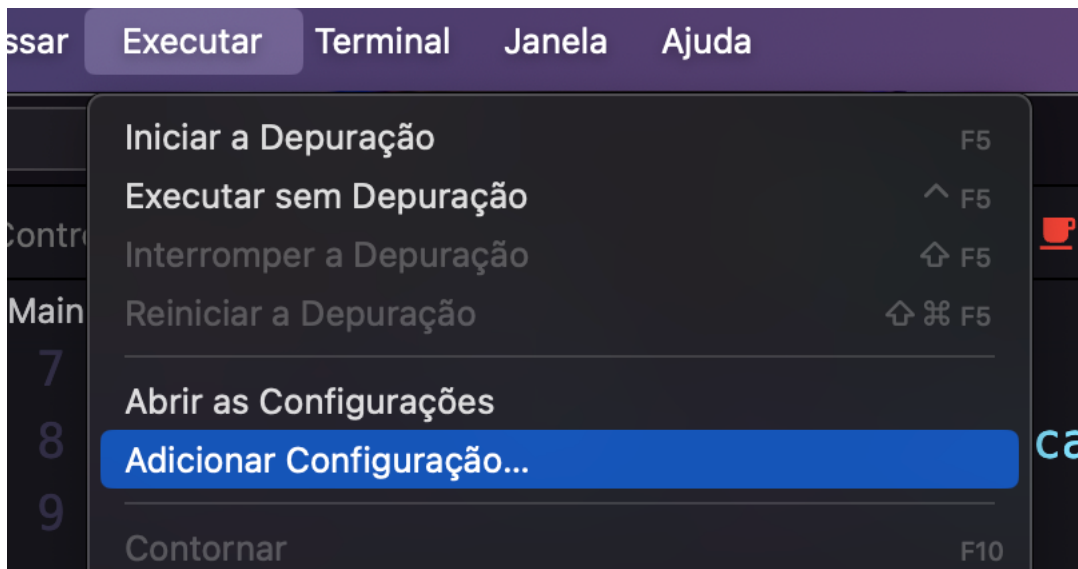
1. Casos Muito Específicos:

- Em alguns casos muito específicos, como quando a interface é gerada dinamicamente de uma maneira que não se beneficia do FXML, pode-se optar por construir a interface diretamente no código.

Em resumo, o uso do `loader.load()` com o `FXMLLoader` é geralmente recomendado para aplicações JavaFX convencionais, pois oferece uma abordagem mais modular, organizada e fácil de manter. A separação entre a interface e a lógica facilita a colaboração em equipe e a evolução da aplicação ao longo do tempo.

vmArgs: (Necessário caso não esteja utilizando java 8).

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:



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...

Trocando de Telas:

Algumas coisas são importantes ressaltar, como: esse método é usado quando as telas tem o mesmo controller, logo por esse controller eu faço os métodos que trocam de tela.

Primeiramente chamo a tela 1 padrão:

```
//Passando a tela para o root  
Parent root = FXMLLoader.load(getClass()  
.getResource(name:"./view/  
aula05_tela01.fxml"));  
  
Scene scene = new Scene(root);  
  
//Adicionando CSS  
String css = this.getClass().  
getResource(name:"./css/aula05.css").  
toExternalForm();  
scene.getStylesheets().add(css);  
  
primaryStage.setScene(scene);  
primaryStage.setTitle(value:"Aula 05  
- JavaFX");  
primaryStage.show();
```

```
}
```

Agora no controlador dela, que inclusive é o mesmo da tela 02 eu faço os seguintes códigos:

```
ntroller > SceneController.java > SceneController > switchScene2(ActionEvent)
12 public class SceneController {
13     private Stage stage;
14     private Scene scene;
15     private Parent root;
16
17     public void switchScene1(ActionEvent e) throws
18         IOException{
19         root = FXMLLoader.load(getClass().getResource
20             (name:"/view/aula05_tela01.fxml"));
21         stage = (Stage)((Node)e.getSource()).getScene().
22             getWindow();
23         scene = new Scene(root);
24         scene.getStylesheets().add(this.getClass().
25             getResource(name:"/css/aula05.css").
26             toExternalForm());
27         stage.setScene(scene);
28         stage.show();
29     }
30
31     public void switchScene2(ActionEvent e) throws
32         IOException{
33         root = FXMLLoader.load(getClass().getResource
34             (name:"/view/aula05_tela02.fxml"));
35         stage = (Stage)((Node)e.getSource()).getScene().
36             getWindow();
37         scene = new Scene(root);
38         scene.getStylesheets().add(this.getClass().
39             getResource(name:"/css/aula05.css").
40             toExternalForm());
41         stage.setScene(scene);
42         stage.show();
43     }
44 }
```

Algumas coisas que precisam ficar claras:

- Eu não uso os caminhos com o ./ pois não estou saindo da raiz do projeto.
- O método switchScene1 deve ser usado pelo botão da tela 2.

- É importante sempre carregar o CSS, se não ele não será passado e a tela irá abrir sem a estilização.
- É importante sempre carregar os icons das janelas se não, eles não serão carregados na troca de tela.

A parte desse código mais importante é o carregamento do Stage, logo vamos explicar a linha 19 e a linha 28 desse código:

Aqui estão as explicações para cada parte dessa linha de código:

- `e.getSource()`: `e` é um parâmetro do tipo `ActionEvent`, que é comumente usado para eventos de interface do usuário, como cliques em botões. `getSource()` retorna o objeto que acionou o evento, que no seu caso seria um nó da interface gráfica (por exemplo, um botão).
- `((Node)e.getSource())`: Este trecho converte o objeto obtido em um tipo `Node`, que é uma classe base para todos os elementos de interface gráfica no JavaFX.
- `.getScene()`: Obtém a cena à qual o nó pertence. A cena é basicamente um contêiner para o conteúdo gráfico em JavaFX.
- `.getWindow()`: Obtém a janela (window) à qual a cena está associada. Uma janela é uma área retangular na tela onde a interface gráfica é exibida.
- `(Stage)((Node)e.getSource()).getScene().getWindow()`: Finalmente, converte a janela obtida para o tipo `Stage`. Um `Stage` é uma janela JavaFX principal que geralmente contém uma ou mais cenas.

Eu armazeno o stage aberto, ou seja a janela já aberta e só troca a scene que ela está associada, pois como eu troco o root da scene eu troco a Scene, logo a janela continua a mesma, porém a scene e o root que é carregado e demonstrado por ela é modificado.

(OBS): Eu posso criar essa classe com esses métodos, porém caso eu queira construtores diferentes eu posso criar um construtor da mesma e em cada controller instanciar um objeto do tipo `SceneController` e usar os métodos criados nele.

Seria da seguinte forma, a Classe `SceneController`:

```
package controller;
```

```
import javafx.event.ActionEvent;  
import javafx.fxml.FXMLLoader;
```

```

import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.Node;
import java.io.IOException;

public class SceneController {

    private Stage stage;
    private Scene scene;
    private Parent root;

    public void switchScene(ActionEvent e, String fxmlPath, String cssPath) throws IOException {
        root = FXMLLoader.load(getClass().getResource(fxmlPath));
        stage = (Stage)((Node)e.getSource()).getScene().getWindow();
        scene = new Scene(root);

        if (cssPath != null && !cssPath.isEmpty()) {
            scene.getStylesheets().add(this.getClass().getResource(cssPath).toExternalForm());
        }

        stage.setScene(scene);
        stage.show();
    }
}

```

A Classe Controller01 sa tela 1:

```

package controller;

import javafx.event.ActionEvent;
import java.io.IOException;

public class Controller1 {

    private SceneController sceneController = new SceneController();

    public void goToScene2(ActionEvent event) throws IOException {
        sceneController.switchScene(event, "/view/aula05_tela02.fxml", "/css/aula05.css");
    }
}

```

A Classe Controller02 da tela 2:

```

package controller;

import javafx.event.ActionEvent;
import java.io.IOException;

public class Controller2 {

    private SceneController sceneController = new SceneController();

    public void goToScene1(ActionEvent event) throws IOException {
        sceneController.switchScene(event, "/view/aula05_tela01.fxml");
    }
}

```


Uma forma melhor seria usando da seguinte forma(obs: o nome do arquivo.css e arquivo.fxml tem que ser os mesmos para cada tela):

A classe SceneController:

```
package controller;

import javafx.event.ActionEvent;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.Node;
import java.io.IOException;

public class SceneController {

    private Stage stage;
    private Scene scene;
    private Parent root;

    public void switchScene(ActionEvent e, String fxmlPath) throws IOException {
        root = FXMLLoader.load(getClass().getResource(fxmlPath));
        stage = (Stage)((Node)e.getSource()).getScene().getWindow();
        scene = new Scene(root);

        String cssPath = getCSSPath(fxmlPath);

        if (getClass().getResource(cssPath) != null) {
            scene.getStylesheets().add(getClass().getResource(cssPath).toExternalForm());
        }

        stage.setScene(scene);
        stage.show();
    }

    private String getCSSPath(String fxmlPath) {
        // Assume que o CSS tem o mesmo nome que o FXML
        return "/css/" + fxmlPath.replace(".fxml", ".css");
    }

    public String getFXMLPath(String fileName) {
        return "/view/" + fileName + ".fxml";
    }

    public String getCSSPath(String fileName) {
        return "/css/" + fileName + ".css";
    }
}
```

A classe Controller01:

```
package controller;

import javafx.event.ActionEvent;
```

```
import java.io.IOException;

public class Controller1 {

    private SceneController sceneController = new SceneController();

    public void goToScene2(ActionEvent event) throws IOException {
        sceneController.switchScene(event, sceneController.getFXMLPath("aula05_tela02"));
    }
}
```

A classe Controller02:

```
package controller;

import javafx.event.ActionEvent;
import java.io.IOException;

public class Controller1 {

    private SceneController sceneController = new SceneController();

    public void goToScene2(ActionEvent event) throws IOException {
        sceneController.switchScene(event, sceneController.getFXMLPath("aula05_tela02"));
    }
}
```

Comunicação entre controllers:

Utilizaremos o mesmo programa, porém agora estamos usando a forma 2 de trocar de telas, cada tela agora tem seu controlador e quem troca a tela é o SceneController.

Primeiramente precisamos declarar em ambos os controladores os Nodes que vão se comunicar com os Id's declarados no SceneBuilder, ex:

Controlador 01:

```
1 public class Controller01 {
2
3     @FXML
4     TextField textField;
5
6     @FXML
7     Button loginButton;
```

Controlador 02:

```
public class Controller02 {  
  
    @FXML  
    Label label;
```

Para realizar essa comunicação declaramos primeiro como será recebido pelo Node da cena, e após isso utilizamos de algum método na outra cena para passar essa informação da forma solicitada, aqui estamos usando um exemplo de uma informação de um text-field e passando para uma label de outra cena, logo primeiros criamos o método de modificar a label no Controller02:

```
public class Controller02 {  
  
    @FXML  
    Label label;  
  
    public void displayName(String username){  
        label.setText("Hello " + username);  
    }  
}
```

Agora no Controller01, precisamos entender alguns aspectos:

Primeiro precisamos carregar uma instância da outra cena, pois só assim ela recebe os chamados dos métodos da tela dela, e modifica, fazemos esse pre-carregamento da seguinte forma:

```
public void login(ActionEvent event) throws IOException {  
  
    String username = text_field.getText();  
  
    FXMLLoader loader = new FXMLLoader(getClass().  
        getResource(name:"/view/aula05_tela02.fxml"));  
    root = loader.load();  
}
```

Aqui, a partir do momento que clicamos no botão que realiza esse método, ele faz o pré-carregamento da tela02.

A partir daí acessamos a tela 02 por meio de uma instância dela e chamando o seu método:

```
Controller02 controller02 = loader.getController();
controller02.displayName(username);
```

Aqui conseguimos mudar a label da tela02, antes mesmo dela aparecer na tela.

Agora, realizamos a chamada da tela02 e exibimos ela na tela:

```
stage = (Stage) ((Node) event.getSource()).getScene().
getWindow();
scene = new Scene(root);
stage.setScene(scene);
scene.getStylesheets().add(this.getClass().getResource
(name: "/css/aula05.css").toExternalForm());
stage.show();
```

E dessa forma conseguimos fazer essa comunicação entre as telas, é importante ressaltar que caso eu volte para a tela 1, e depois para a tela 2, essa mudança não fica salva.

Comunicação entre Controladores utilizando uma Classe externa DATA:

Primeiramente criamos duas telas e adicionamos seus controladores, adicionamos todos os seus Nodes injetáveis do FXML, e iniciamos em uma das telas qualquer, no início de toda tela é importante alguns aspectos, primeiro a criação de duas variáveis sendo elas o Stage e o root, e junto a isso, caso você queira que as informações sejam passadas para ambas as telas, é importante que ambas tenham o método Inicializa, dessa forma elas modificam os seus Nodes que recebem as informações da outra tela antes de serem carregadas:

```
public class Controller01 implements Initializable{

    Stage stage;
    Parent root;

    @FXML
    TextField textfield;
    @FXML
    ColorPicker color;
```

```
public class Controller02 implements Initializable {

    Stage stage;
    Parent root;

    @FXML
    TextArea text;
    @FXML
    AnchorPane pane;
}
```

Agora vamos explicar a Classe Data e a sua instância em ambos os controllers.

A classe Data é uma classe de modelo (model) que segue o padrão de design Singleton, que é utilizado para garantir que uma classe tenha apenas uma instância e forneça um ponto global de acesso a essa instância. Vamos analisar a classe e entender como ela funciona:

Padrão Singleton:

```
private static final Data instance = new Data();
```

```
private Data() {
}
```

```
public static Data getInstance() {
    return instance;
}
```

- A classe possui um membro privado e estático chamado instance, que é uma única instância estática da classe Data.
- O construtor da classe Data é privado (private), o que significa que não pode ser chamado de fora da própria classe. Isso impede que outras classes criem instâncias de Data diretamente.
- Em vez disso, a única instância disponível é criada e mantida dentro da própria classe (private static final Data instance = new Data();).
- O método público estático getInstance() é fornecido para obter a referência à única instância de Data. Se essa instância ainda não existir, o método cria uma nova instância.

A classe possui dois atributos privados: text (uma string) e color (um objeto Color do JavaFX). Esses atributos são usados para armazenar dados que precisam ser compartilhados globalmente na aplicação.

Métodos públicos são fornecidos para acessar e modificar os valores dos atributos color e text. Esses métodos permitem que outras classes alterem e obtenham os valores armazenados na instância única de Data.

A Classe Data:

```
package model;

import javafx.scene.paint.Color;

public class Data {

    private static final Data instance = new Data();

    private String text;
    private Color color;

    private Data() {
    }

    public static Data getInstance() {
        return instance;
    }

    public void setColor(Color color){this.color = color;}

    public Color getColor(){return this.color;}

    public void setText(String text) {this.text = text;}

    public String getText() {return this.text;}
}
```

Como essa classe é usada e acessada em cada uma das telas:

Primeiro na tela um: (Obs: A metodologia usada é de comunicação entre as duas telas, logo a tela 1 tem um initialize que faz certos testes com a data:

```

    ColorPicker color;

    Data data = Data.getInstance();

    @Override
    public void initialize(URL location, ResourceBundle
resources) {
        if (data.getText() != null) {
            textfield.setText(data.getText());
        }
        if (data.getColor() != null) {
            color.setValue(data.getColor());
        }
    }

    public void submit(ActionEvent event) throws IOException {

        data.setText(textfield.getText());
        data.setColor(color.getValue());

        stage = (Stage) textfield.getScene().getWindow();
        root = FXMLLoader.load(getClass().getResource
(name: "../view/tela02.fxml"));
        stage.setScene(new Scene(root));
        stage.setTitle(value: "Tela 02");
    }
}

```

O controlador01 acessa a instância única de Data usando Data.getInstance(). Isso permite que eles compartilhem e atualizem os dados armazenados na instância de Data.

O métodos initialize nos controladores são usados para configurar o estado inicial das interfaces gráficas com base nos dados armazenados na instância de Data, nesse caso a cor escolhida no ColorPicker e o texto salvo em Data.

Em submit a função é de um botão que quando apertado a Data seta os dados text e Color por meio dos métodos .set() e salva eles naquela instância, que é uma instância única de Data.

logo após ele recebe o Stage do Node da tela01 que pode ser qualquer node com ID declarado, armazena esse Stage, troca o root para o da tela02, e logo após troca a Scene desse Stage para

a Scene com o root da tela02, mudando assim para tela02, sem a necessidade de criar um novo Stage.(Obs: nesse caso ainda estamos no primaryStage da Main, trocando apenas as Scenes).

Agora vamos analisar como a Tela02 recebe essa Data e utiliza dela vindo o Controller02:

```
Data data = Data.getInstance();

@Override
public void initialize(URL location, ResourceBundle
resources) {
    text.setText(data.getText());
    pane.setBackground(new Background(new
    BackgroundFill(data.getColor(), CornerRadii.
    EMPTY, Insets.EMPTY)));
}

public void back(ActionEvent event) throws IOException {

    stage = (Stage) text.getScene().getWindow();
    root=FXMLLoader.load(getClass().getResource("../
view/tela01.fxml"));
    stage.setTitle(value:"Tela 01");
    stage.setScene(new Scene(root));
}
```

O controller02 recebe a mesma instância de data do controller01, que seria a Data data = Data.getInstance();

No seu método initialize ela modifica o Texto da TexArea com o texto presente no objeto data, além disso modifica a cor de fundo com a cor recebida pela data.

Por último temos o método back, que volta para a tela anterior fazendo a mesma lógica de receber o Stage e trocar apenas o Scene.

Podemos notar que nesse caso não modificamos a Data apenas a recebemos, porém caso eu queria posso adicionar outro botão que modifica a Data e quando clicar no botão de voltar para a tela01, ela irá receber essa Data por isso utilizamos o método Initialize em ambas as telas.

O propósito geral da classe Data é servir como um recipiente centralizado para dados compartilhados entre diferentes partes da aplicação. Isso é especialmente útil em aplicativos JavaFX, onde várias telas podem precisar compartilhar informações. O padrão Singleton garante que haja apenas uma instância global desses dados.

Mas caso eu queria comunicações entre várias telas, porém por meio de canais separados: Sendo assim você pode criar mais instâncias Singleton e liga-las com os controllers que você deseja comunicar entre eles:



```
private static final Data data1Instance = new Data();
private static final Data data2Instance = new Data();

private String text;
private Color color;

private Data() {
}

public static Data getData1Instance() {
    return data1Instance;
}

public static Data getData2Instance() {
    return data2Instance;
}
```



Mas caso eu queria atributos diferentes em cada classe Singleton:

Sendo assim você deve criar outra classe com diferentes atributos e cada uma terá uma ou mais instâncias Singleton e dessa maneira você terá a comunicação entre telas por meio de classes Datas com atributos e ligações específicas.

Comunicação de telas semelhantes com abertura de novos Stage e fechamento dos antigos:

Primeiramente devemos entender que as telas tem Nodes semelhantes, e vamos utilizar disso para passar dados entre as telas:

Vamos ver o padrão dos controladores dessas telas explicar após:

```
public class Controller03 implements Initializable {

    @FXML private TextField agetf1, emailtf1, nametf1;

    @FXML void oneSendThree(ActionEvent event) throws
    IOException{
        FXMLLoader loader = new FXMLLoader(getClass().
        getResource(name:"../view/tela05.fxml"));
        Parent root = loader.load();
        Controller05 controller05 = loader.getController();
        controller05.showInformation(nametf1.getText(),
        emailtf1.getText(), agetf1.getText());
        Stage stage = new Stage();
        stage.setScene(new Scene(root));
        stage.setTitle(value:"Tela 03");
        stage.show();
        closeStage();
    }
}
```

```

@FXML void oneSendTwo(ActionEvent event) throws
IOException {
    FXMLLoader loader = new FXMLLoader(getClass().
getResource(name:"../view/tela04.fxml"));
    Parent root = loader.load();
    Controller04 controller04 = loader.getController();
    controller04.showInformation(nametf1.getText(),
emailtf1.getText(), agetf1.getText());
    Stage stage = new Stage();
    stage.setScene(new Scene(root));
    stage.setTitle(value:"Tela 02");
    stage.show();
    closeStage();
}

public void showInformation(String name, String email,
String age) {
    nametf1.setText(name);
    emailtf1.setText(email);
    agetf1.setText(age);
}

public void closeStage() {
    Stage stage = (Stage) nametf1.getScene().getWindow();
    stage.close();
}

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

```

Algumas coisas precisam ser ressaltadas, primeiramente que todos os controladores precisam do método initialize da interface Initializable, porém dentro do método initialize nada é usado.

Nós métodos de enviar dados para as outras telas nós criamos um FXMLLoader e carregamos a tela com o root = loader.load();, junto a isso criamos uma instância do Controlador dessa tela que estamos indo, por meio do nosso loader usando o método loader.getController();, dessa forma conseguimos usar a instância do novo controlador ainda nessa tela, acessando o método showInformation(); que recebe como parâmetro os dados da tela atual e modifica os Nodes da tela que estamos indo, afinal essa tela que estamos indo já está carregada pelo método loader.load() anteriormente usado.

Após isso criamos um novo Stage, setamos ele com a Scene que possui o root da nova tela que já estava recém carregado, colocamos um título nesse Stage, demonstramos ele na tela e após isso fechamos o Stage atual por meio do método closeStage();, que por meio de um Node da tela "anterior" consegue acessar seu Stage e fecha-ló.

É importante ressaltar que essa forma de comunicação de Controllers necessita que a classe que receberá os dados tenha um método `showInformation` no seu controlador, além disso para dar a ideia de passagem informações de tela1 para tela2 e vice-versa, é necessário que ambas tenham o método `showInformation` formatado da sua forma e que ambas tenham um método para acessá-las

E por ultimo é importante ressaltar que caso queiramos as duas telas abertas ao mesmo tempo, podemos retirar o método `closeStage()` no final de cada método de transporte de informações porém devemos fechar a ultima instância da tela, criando assim um novo método para resolver essa problemática, e não ter diversas tela1 por exemplo.

#JavaFX