

Programação Avançada

Introdução ao *JavaFX*

Programação gráfica em Java

- A criação de ambientes gráficos em *Java* foi considerado desde o seu início
 - <https://www.oracle.com/java/technologies/painting.html>
- Inicialmente o desenvolvimento de ambientes gráficos foi baseado em *AWT – Abstract Windowing Toolkit*
 - Os seus componentes são considerados "*heavyweight*" porque são suportados por código nativo do sistema operativo
- Foi substituído pelo *Swing*
 - Os componentes são considerados "*lightweight*" porque são todos desenvolvidos em Java
 - A maior parte dos componentes foram desenvolvidos sobre os componentes AWT existentes
 - A utilização desta API não substitui a AWT, podem ser ambas usadas embora deva ser evitado

JavaFX

- Biblioteca para desenvolvimento de aplicações gráficas (*Rich Internet Applications*) em Java, apresentada como evolução e para substituição do *Swing*
- A primeira versão foi disponibilizada em Outubro de 2008
- Framework desenvolvida em Java
- Inicialmente pertencia à distribuição do JDK, mas passou a ser distribuído à parte, a partir do JDK 11
 - A Oracle passou o seu desenvolvimento para o OpenJDK, mais concretamente para um projeto específico designado OpenJFX

JavaFX

- Características principais:
 - Escrito em *Java*
 - FXML
 - *Scene Builder*
 - Compatibilidade *Swing*
 - *Built-in Controls*
 - Suporte de configurações similares a CSS
 - API's e conjunto de classes para suporte de gráficos 2D, 3D e conteúdos avançados
 - Tira partido de mecanismos de aceleração gráfica disponível no dispositivo onde a aplicação é executada

OpenJFX

- A documentação mais atualizada pode ser obtida a partir do *website*
 - <https://openjfx.io/>
- A integração do *JavaFX* num projeto *Java* pode ser realizada fazendo *download* da biblioteca e associando-a ao projeto
 - A biblioteca é constituída por diversos ficheiros *jar*

Instalação

- Fazer *download* a partir do *website*: <http://openjfx.io>
 - *Direct link*: <https://gluonhq.com/products/javafx/>
- Para novos projetos, fazer *download* de uma versão *LTS* ou versão mais recente
 - Sugestão: *JavaFX 17*
 - Fazer *download*
 - *SDK* para o sistema operativo pretendido
 - Documentação (*JavaDoc*)
- Descompactar para um diretório, preferencialmente junto ao *JDK* em uso (a documentação pode ser descompactada para um diretório *doc* dentro do diretório base criado)
 - *Windows*: C:\Program Files\Java
 - *MacOS*: /Library/Java/JavaVirtualMachines

Configuração do IntelliJ

- Criar um projeto Java como realizado para projetos Java anteriores
- Ir a opção File → Project Structure
 - Global Libraries
 - "+" → New Java Library...
 - Indicar o caminho completo para o diretório lib do JavaFX
 - Adicionar à biblioteca os *URL* para a documentação
 - Exemplo para ficheiros locais:
 - file:///C:\Program Files\Java\javafx-sdk-17\doc
 - file:///C:\Program Files\Java\javafx-sdk-17\doc\javafx.base
 - file:///C:\Program Files\Java\javafx-sdk-17\doc\javafx.controls
 - file:///C:\Program Files\Java\javafx-sdk-17\doc\javafx.fxml
 - file:///C:\Program Files\Java\javafx-sdk-17\doc\javafx.graphics
 - file:///C:\Program Files\Java\javafx-sdk-17\doc\javafx.media
 - Estes passos são realizados apenas uma vez

Configuração do IntelliJ

- Criar um projeto Java como realizado para projetos Java anteriores
- Nos projectos onde se pretende usar o *JavaFX*
 - File → Project Structure
 - Global Libraries
 - Dar um toque com o botão direito sobre a biblioteca e adiciona-se ao módulo (projeto criado)

Configuração do IntelliJ

- Nas configurações de execução
 - Adicionar as seguintes configurações às opções VM Options
 - module-path /path_to_javafx_sdk/lib
 - add-modules javafx.controls
 - Se for usado xml para a definição da interface:
--add-modules javafx.controls,javafx.fxml
 - Incluir todos os módulos do JavaFX
--add-modules ALL-MODULE-PATH

Aplicação *JavaFX*

- Uma aplicação *JavaFX* é encapsulada através de um objeto `javafx.application.Application`
- O desenvolvimento de uma aplicação *JavaFX* inicia-se normalmente pela criação de uma nova classe que estende a classe `Application`
 - Deve ser definido o método abstrato `void start(Stage);`
- O objeto `Application` é criado pelo sistema quando o método estático `Application.launch(...)` for executado
 - O método `launch` só pode ser chamado numa vez no contexto de uma aplicação *JavaFX*, ou seja, apenas deve existir uma objeto `Application`

Application

- Quando a classe que deriva de Application é a mesma onde está definida a função main

```
public class Main extends Application {  
  
    public static void main(String [] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        // TODO  
    }  
}
```

Application

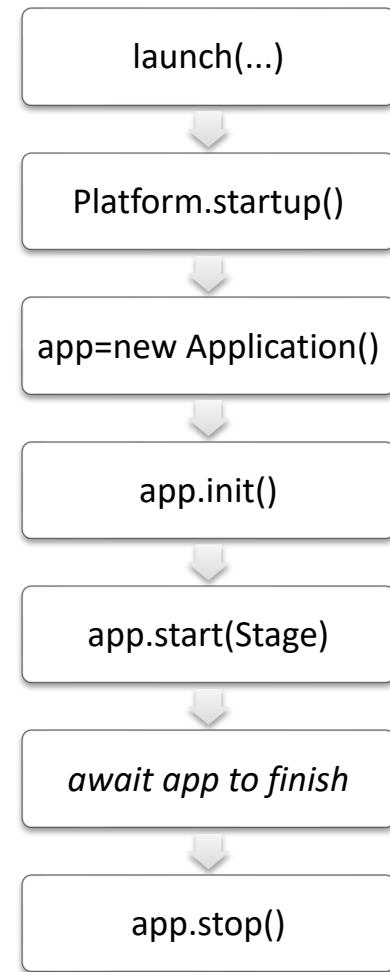
- Quando a classe que deriva de Application é diferente da classe onde está definida a função main

```
public class Main {  
    public static void main(String [] args) {  
        Application.launch(JavaFXMain.class,args);  
    }  
}
```

```
public class JavaFXMain extends Application {  
    @Override  
    public void start(Stage stage) throws Exception {  
        // TODO  
    }  
}
```

Ciclo de vida de uma aplicação

- Quando o método `launch` é chamado são realizadas as seguintes ações, por ordem:
 - Garantir que o "*JavaFX runtime*" foi devidamente iniciado
 - Caso não tenha ainda sido iniciado, é executado o método `Platform.startup(...)`
 - Cria uma instância do objeto `Application` especificado
 - Chama o método `init()` do objeto `Application`
 - Pode ser redefinido para tarefas de iniciação da aplicação e reserva de recursos
 - Chama o método `start()` do objeto `Application`
 - Criação de todo o ambiente gráfico e configuração dos comportamentos
 - Espera pelo fim da aplicação
 - Ocorre quando a última janela for fechada
 - ou, o método `Platform.exit()` for executado
 - Chama o método `stop()` do objeto `Application`
 - Permite libertar recursos alocados e outras tarefas de finalização da aplicação



Scene Graph: Stage, Scene e root node

- A interface gráfica que caracteriza uma aplicação JavaFX deve ser criada no método `start` do objeto `Application`
- Os objetos que constituem a interface são organizados através de uma árvore de objetos, designada *Scene Graph*, a qual é constituído por:
 - `Stage`
 - `Scene`
 - *Root node* e outros *nodes*

Scene Graph: Stage, Scene e root node

- A interface gráfica que caracteriza uma aplicação *JavaFX* deve ser criada no método `start` do objeto `Application`
- Os objetos que constituem a interface são organizados através de uma árvore de objetos, designada *Scene Graph*, a qual é constituído por:
 - **Stage**
 - Corresponde à janela que suporta a aplicação, adaptada ao sistema operativo
 - Podem ser criados vários Stage caso se pretendam ter várias janelas
 - É recebido por parâmetro no método `start` um objeto Stage já previamente criado
 - A um objeto Stage deverá ser associado um objeto Scene
 - **Scene**
 - **Root node** e outros *nodes*



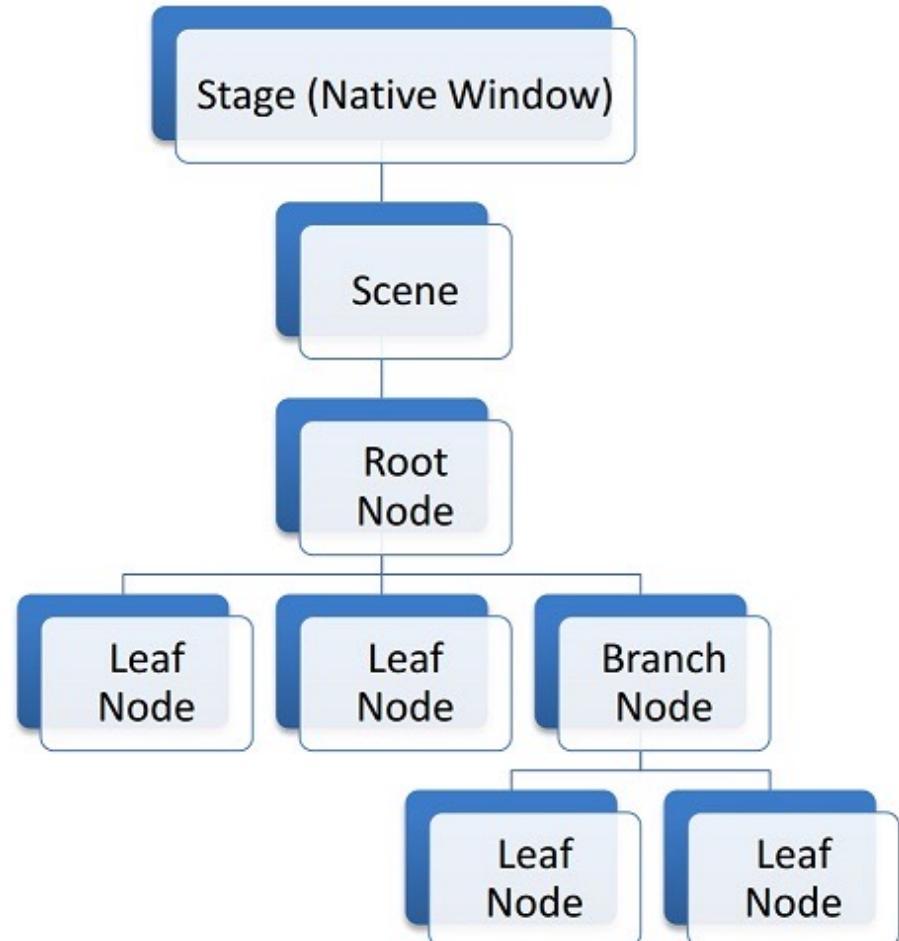
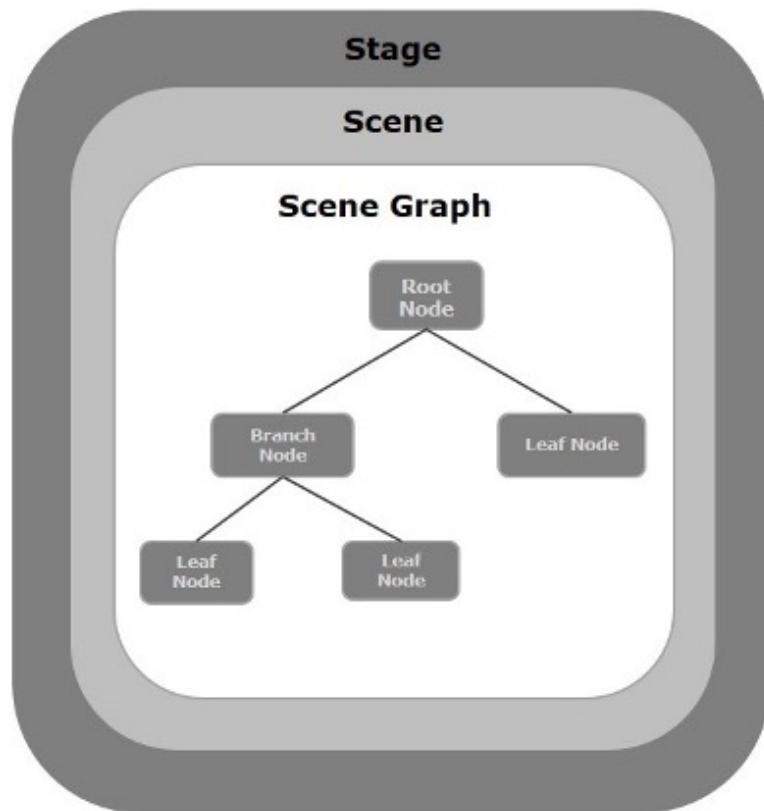
Scene Graph: Stage, Scene e root node

- A interface gráfica que caracteriza uma aplicação *JavaFX* deve ser criada no método `start` do objeto `Application`
- Os objetos que constituem a interface são organizados através de uma árvore de objetos, designada *Scene Graph*, a qual é constituído por:
 - `Stage`
 - `Scene`
 - Representa os componentes que constituem o *Scene Graph*
 - Pode ser definida uma largura e uma altura, definindo assim o tamanho da janela de suporte
 - *Root node* e outros *nodes*

Scene Graph: Stage, Scene e root node

- A interface gráfica que caracteriza uma aplicação *JavaFX* deve ser criada no método `start` do objeto `Application`
- Os objetos que constituem a interface são organizados através de uma árvore de objetos, designada *Scene Graph*, a qual é constituído por:
 - `Stage`
 - `Scene`
 - *Root node* e outros *nodes*
 - Todos os componentes que constituem a interface (botões, caixas de texto, elementos de organização de outros elementos, ...) derivam direta ou indiretamente da classe `Node`
 - Alguns elementos `Node` (elementos de *layout: pane*) podem incluir outros elementos `Node`, criando-se assim uma hierarquia de componentes
 - O primeiro deles – a base da hierarquia – é designado *Root Node*
 - Este é o elemento que deve ser indicado na criação do objeto `Scene`

Scene Graph: Stage, Scene e root node



https://www.tutorialspoint.com/javafx/javafx_application.htm

<https://fxdocs.github.io/docs/html5/>

Hierarquia de objetos JavaFX

- `javafx.stage.Window`
 - `PopupWindow`
 - `Popup`
 - `PopupControl`
 - `ContextMenu, Tooltip`
 - **Stage**
- `javafx.scene.Scene`
- `javafx.scene.Node`
 - `Parent`
 - `Group`
 - `Region`
 - *Next slide...*
 - `WebView`
 - **Shape**
 - `Arc, Circle, CubicCurve, Ellipse, Line, Path, Polygon, Polyline, QuadCurve, Rectangle, SVGPath, Text`
 - `Shape3D`
 - `Box, Cylinder, MeshView, Sphere`
 - **Canvas**
 - **ImageView**
 - `Camera, LightBase, MediaView, SubScene, SwingNode`

Hierarquia de objetos JavaFX

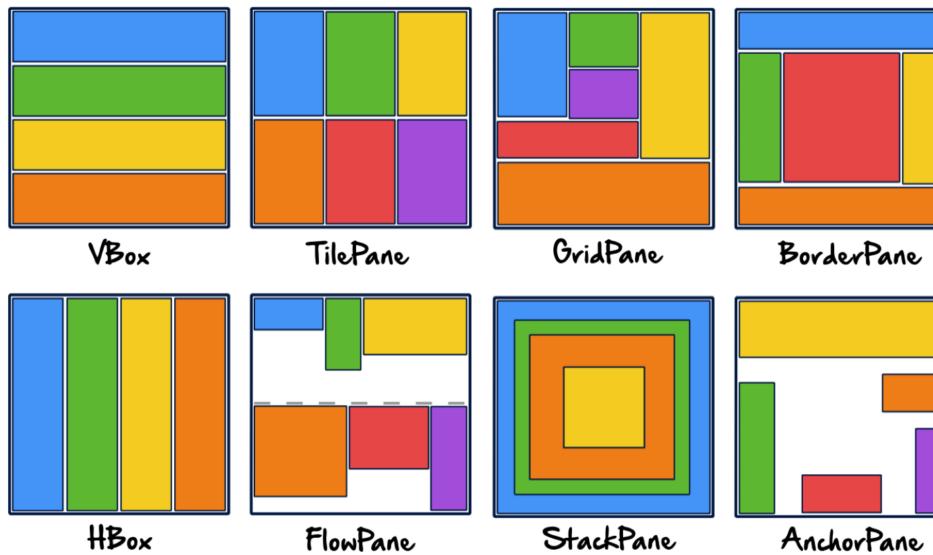
- Region
 - Control
 - TextInputControl
 - **TextArea**, **TextField**
 - ComboBoxBase
 - **ColorPicker**, **ComboBox**, **DatePicker**
 - Labeled
 - ButtonBase
 - **Button**
 - MenuButton
 - SplitMenuButton
 - ToggleButton
 - **RadioButton**
 - **CheckBox**
 - Hyperlink
 - Cell, **Label**, TitledPane
 - Accordion, ButtonBar, **ChoiceBox**, HTMLEditor, **ListView**,MenuBar, Pagination, ProgressIndicator, ScrollBar, **ScrollPane**, Separator, Slider, Spinner, SplitPane, TableView, TabPane, ToolBar, TreeTableView, **TreeView**
 - Pane
 - AnchorPane, BorderPane, DialogPane, FlowPane, GridPane, HBox, PopupControl.CSSBridge, StackPane, TextFlow, TilePane, VBox
 - Axis, Chart, TableColumnHeader, VirtualFlow

Hierarquia de objetos JavaFX

- Existem outras *sub-hierarquias* com muitos outros objetos que permitem disponibilizar muitas das funcionalidades e modos de interação usuais nas interfaces atuais

Objetos Pane (*layout*)

- AnchorPane
- BorderPane
- FlowPane
- GridPane
- HBox
- StackPane
- TilePane
- VBox



Formatação do *layout*

- Dependendo do tipo de objeto de *layout* usado, estão disponíveis diversos parâmetros de formatação e formas de adicionar os componentes que gera (*children*)
 - `getChildren().add`, `getChildren().addAll`
 - `setPadding`
 - `setAlignment`
 - `setSpacing`
 - `setTopAnchor`, `setBottomAnchor`, `setLeftAnchor`, `setRightAnchor`
 - `setTop`, `setBottom`, `setLeft`, `setRight`, `setCenter`
 - `setMargin`
 - ...

Formatação de Nodes

- As cores e outras configurações de cada componente podem ser configuradas através de métodos específicos
 - `setBorder`
 - `setMaxSize`, `setMinWidth`, `setMaxHeight`, `setPrefHeight`, ...
 - `setText`, `setTextAlignment`, `setTextFill`
 - `setStyle("CSS style string")`
 - Ex: `obj.setStyle("-fx-background-color: #ffffd0;");`
 - ...

Exemplo

```
public class MainJFX extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
    @Override  
    public void start(Stage stage) throws Exception {  
        BorderPane root = new BorderPane();  
        root.setStyle("-fx-background-color: #ffffe0;");  
        Label label = new Label("Advanced Programming");  
        label.setTextFill(Color.INDIGO);  
        label.setFont(new Font("Times New Roman", 24));  
        root.setCenter(label);  
        Scene scene = new Scene(root, 400, 300);  
        stage.setTitle("DEIS-ISEC");  
        stage.setResizable(false);  
        stage.setScene(scene);  
        stage.show();  
    }  
}
```



Programação orientada por eventos

- No contexto do *JavaFX*, assim como de outros ambientes de programação gráficos, a programação de aplicações...
 - deixa de ter uma sequência linear e bem definida de execução das instruções
 - passa a ser baseada nos eventos assíncronos, que podem ocorrer sobre os diversos componentes da interface, normalmente resultado da interação do utilizador com esses componentes (p. ex.: clicar um botão)

Programação orientada por eventos

- A programação dos eventos corresponde a programar previamente aquilo que deverá ser executado quando o evento ocorre
 - Os objetos relacionados com os eventos fornecem métodos adequados para definir o que fazer quando os eventos acontecerem
 - `setOn*`, `addEventHandler`, `addEventFilter`
 - Por exemplo
 - Se existir um botão na interface o programa não fica à espera num ciclo "infinito" que o botão seja pressionado (tanto que, podem existir vários botões na interface)
 - Indica-se o código que deverá ser executado quando algum dos botões é pressionado
- Cada evento é encapsulado através de um objeto adequado
 - Por exemplo, o evento relativo ao *click* num botão é representado através de um objeto o `ActionEvent`

ActionEvent

- Como referido, quando um botão (Button) é clicado é gerado um evento representado através de um objeto ActionEvent
- O processamento dos eventos é realizado através de objetos EventHandler<T>
 - No objeto EventHandler<T> deve ser redefinido o método `void handle(T event)`, no contexto do qual se deve fazer o processamento pretendido
- Formas de criar um objeto EventHandler<ActionEvent>
 - criar uma instância de uma classe que implementa a interface EventHandler<ActionEvent>
 - criar um objeto *inline* (classe anónima) que implementa a interface EventHandler<ActionEvent>
 - *Lambda expression*

ActionEvent

- Exemplo de uma classe que para processar um evento ActionEvent

```
class MyHandler implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent actionEvent) {  
        //TODO  
    }  
}
```

... a qual pode ser associada a um botão da seguinte forma:

```
Button button = new Button("Go");  
button.setOnAction(new MyHandler());
```

ActionEvent

- Exemplos com *Lambda Expressions*

- `setOnAction`

```
Button button = new Button("Increment");
button.setOnAction(actionEvent -> {
    //TODO
});
```

- `addEventFilter`

```
Button button = new Button("Decrement");
button.addEventFilter(
    ActionEvent.ACTION,
    actionEvent -> {
        //TODO
    }
);
```

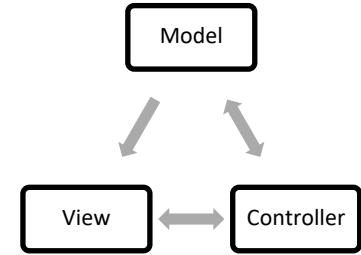
- Com os métodos `addEventFilter/addEventHandler` podem-se tratar diferentes tipos de eventos e podem ser configurados vários processamentos para o mesmo evento

Organização do projeto

- De modo a manter independência entre o modelo de dados e as formas de visualização e/ou interação com o utilizador, há que organizar as entidades que constituem os programas
- Essa separação/organização favorece outros aspetos do desenvolvimento, por exemplo:
 - Divisão da complexidade dos programas
 - Evolução das diferentes partes da aplicação de forma independente
 - Alocação de tarefas diferentes às equipas de desenvolvimento
 - Reaproveitamento de código
 - Teste das aplicações
 - Suporte facilitado
 - ...

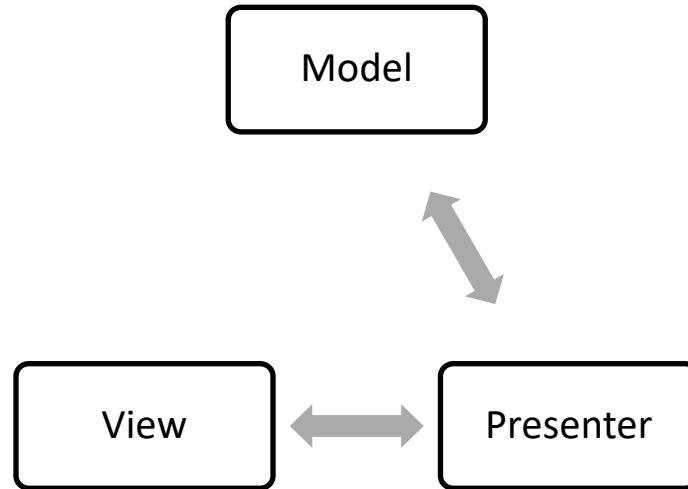
MVC

- No modelo MVC, *Model-View-Controller*, as responsabilidades são divididas entre:
 - Modelo
 - Conjunto de classes que permitem gerir os dados, a lógica/regras de negócio e respetivos algoritmos
 - Vista
 - Classes e componentes que permitem apresentar as informações relevantes a cada momento
 - Apresenta os elementos necessários para que o utilizador possa interagir com a aplicação
 - Controlador
 - Permite redirecionar a reação do utilizador para execução das tarefas adequadas do modelo
 - Garante que as vistas são atualizadas para representar a informação mais atual



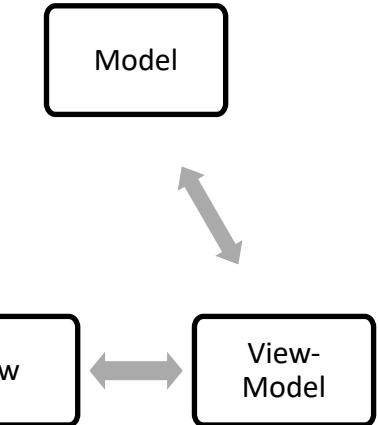
MVP

- O modelo MVP, *Model-View-Presenter*, tem como base o MVC, mas em que:
 - Todas trocas de informação entre o modelo e a vista passam obrigatoriamente pelo *Presenter*
 - Normalmente existe uma relação de 1 para 1 entre a *View* e o *Presenter*



MVVM

- O modelo MVVM, *Model View View-Model*, é um desenvolvimento dos anteriores em que:
 - É criada uma camada entre a vista e o modelo, *View-Model*, que permite disponibilizar a informação essencial para a vista
 - Esta camada intermédia serve como um modelo adaptado às necessidades da vista
 - As ações do utilizador sobre a vista vão atuar sobre o *View-Model*, o qual realiza sobre o modelo as operações necessárias
 - Neste modelo a comunicação do *View-Model* para a *View* é feita com base no padrão *Observer/Observable*, para permitir que as alterações nas vistas possam ser realizadas de forma assíncrona e mais transparente



- Por vezes é difícil fazer a distinção entre o "controlador" e as "vistas" uma vez que a programação das informações e configuração do aspetto gráfico são realizadas sobre os mesmos objetos nos quais se configuram os *listeners/handlers* dos eventos
- No contexto da Unidade Curricular de Programação Avançada opta-se pela seguinte organização para os projetos JavaFX:
 - **Modelo**
 - Conjunto de classes que permitem gerir os dados, a lógica/regras de negócio e respetivos algoritmos
 - **View-Controller**
 - Classe ou classes que implementam os vários ecrãs da interface gráfica da aplicação
 - Para dar suporte à aplicação JavaFX são definidas as classes *main*
 - *Main*
 - possui o método `main` do Java, o qual chamará o método `launch` do JavaFX
 - *MainJFX*
 - Deriva da classe `Application` do JavaFX
 - Responsável por configurar os objetos `Stage`
 - Responsável por criar o primeiro objeto *View-Controller*

MVC@PA – Modelo

- **Modelo**
 - Conjunto de classes que permitem gerir os dados, a lógica/regras de negócio e respetivos algoritmos
 - Não devem conter qualquer código que interaja com o utilizador ("nem *in*, nem *out*")
 - Idealmente é fornecida uma classe *Proxy*, *Facade* ou similar, que permita esconder todas as especificidades internas do modelo
 - Embora esta classe possa ser *Singleton*, em algumas aplicações esse tipo de implementação limita as possibilidades futuras para tratamento simultâneo de vários modelos (por exemplo: aplicações que possuem várias janelas para permitir edição simultânea de vários documentos)
 - Alternativa: *Multiton*

MVC@PA – View-Controller

- *View-Controller*
 - Classe ou classes que implementam os vários ecrãs da interface gráfica da aplicação
 - Deriva normalmente de um objeto *Pane* ou seus derivados, para possibilitar a inclusão dos vários elementos gráficos adequados
 - Métodos chamados no seu construtor
 - `createViews`
 - Responsável pela criação das vistas, configuração de propriedades de visualização e criação do *layout* geral da interface
 - `registerHandlers`
 - Associação de *listeners/handlers* aos eventos relevantes dos diversos elementos da vista criada
 - Atua sobre o modelo para invocar os métodos que irão acionar os comportamentos/métodos do modelo
 - `update`
 - Método responsável por atualizar as informações das vistas com base nos dados obtidos a partir do modelo (apenas são esperadas chamadas a métodos *get* do modelo)
 - Para o seu bom funcionamento é necessário que a *View-Controller* possua uma referência para o modelo

Nota: os nomes a atribuir às classes e métodos podem ser diferentes dos indicados, mas devem permitir perceber facilmente o seu objetivo nesta arquitetura

Exemplo de projeto

```
public class Main {  
    public static void main(String[] args) {  
        Application.launch(MainJFX.class, args);  
    }  
}  
  
public class MainJFX extends Application {  
    ModelData data;  
  
    public MainJFX() { data = new ModelData(); } // It can also be created in 'init'  
    @Override  
    public void start(Stage stage) throws Exception {  
        RootPane root = new RootPane(data);  
        Scene scene = new Scene(root,600,400);  
        stage.setScene(scene);  
        stage.setTitle("TeoIntroJFX");  
        stage.show();  
    }  
}
```

Exemplo de projeto

```
public class RootPane extends VBox { //View-Controller
    ModelData data;
    // variables, inc. views

    public RootPane(ModelData data) {
        this.data = data;

        createViews();
        registerHandlers();
        update();
    }

    private void createViews() { /* create and configure views */ }

    private void registerHandlers() { /* handlers/listeners */ }

    private void update() { /* update views */ }
}
```

Multiton

- O *Multiton* é um padrão de desenvolvimento que generaliza o padrão *Singleton*, permitindo centralizar a gestão de múltiplas instâncias identificadas com base num determinado elemento

```
public class ModelMultiton {  
    private static final HashMap<Object,ModelData> models = new HashMap<>();  
  
    public static ModelData getModel(Object scope) {  
        ModelData model = models.get(scope);  
        if (model == null) {  
            model = new ModelData();  
            models.put(scope,model);  
        }  
        return model;  
    }  
}
```

- Como parâmetro do `getModel` (o qual corresponde à chave do `HashMap`) pode-se usar, por exemplo, a referência para o `Stage`
 - Após a associação de um objeto `Node` a um objeto `Scene`, pode-se obter a referência ao `Stage` fazendo `obj.setScene().getWindow()`

Evolução do modelo

- Na base apresentada para a implementação do MVC, subentende-se que, de cada vez que houver uma alteração aos dados, deverá ser feita uma chamada ao método update para que a vista seja atualizada
- A incorporação de mecanismos que permitam a atualização automática das vistas facilitará este processo

MVVM e *Observer/Observable*

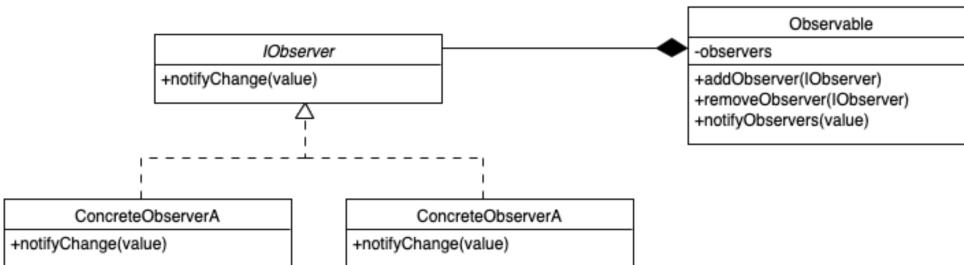
- Uma das possibilidades será a incorporação das funcionalidades do modelo MVVM
 - Neste caso iremos estudar uma forma de integrar as funcionalidades assíncronas do MVVM no modelo MVC@PA
- O padrão *Observer/Observable* permite implementar mecanismos em que existe um conjunto de entidades (*Observers*) que manifestam interesse nos dados ou anúncios de uma outra entidade (*Observable*)
 - Na aplicação aos projetos teremos
 - *Observable* – Modelo de dados (ou uma classe intermédia que sirva de interface para o modelo)
 - *Observer* – as classes *View-Controller* que têm interesse nos dados

Observer/Observable

```
interface IObserver {  
    void notifyChange(Object value);  
}  
  
class A implements IObserver {  
    @Override  
    public void notifyChange(Object value) {  
        System.out.println("A: "+value);  
    }  
}  
  
class B implements IObserver {  
    @Override  
    public void notifyChange(Object value) {  
        System.out.println("B: "+value);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Observable observable = new Observable();  
        A a = new A();  
        B b = new B();  
        observable.addObserver(a);  
        observable.addObserver(b);  
        observable.notifyObservers("DEIS-ISEC");  
    }  
}
```

```
class Observable {  
    HashSet<IObserver> observers = new HashSet<>();  
  
    public void addObserver(IObserver observer) {  
        observers.add(observer);  
    }  
  
    public void removeObserver(IObserver observer) {  
        observers.remove(observer);  
    }  
  
    public void notifyObservers(Object value) {  
        for(IObserver observer : observers)  
            observer.notifyChange(value);  
    }  
}
```

Output:
A: DEIS-ISEC
B: DEIS-ISEC



Observer/Observable

- No package `java.util` é fornecida uma implementação do padrão *Observer/Observable*, no entanto está marcada como *deprecated*
- Existe um outro padrão similar ao *Observer/Observable* designado por *Publisher/Subscriber* que em termos práticos oferece funcionalidades similares

Property Change

- No package `java.beans` é fornecido um conjunto de classes que permitem implementar as funcionalidades do padrão *Observer* de uma forma mais robusta e com mais funcionalidades, designado por *Property Change*
- Classes principais
 - **PropertyChangeSupport**
 - Gestor do alvo que se pretende gerir, onde vão existir alterações aos dados que devem ser sinalizadas
 - Gestor do conjunto de *listeners* (similares a *Observers*), que manifestam interesse nas alterações
 - Podem ser geridos interesses em propriedades específicas (definidas por um nome) ou de interesse geral (sem nome)
 - Permite sinalizar uma alteração que tenha existido:
 - `firePropertyChange(<property_or_null>, <old_value>, <new_value>)`
 - **PropertyChangeListener**
 - Corresponde ao objeto que manifesta interesse nas mudanças que venham a ocorrer
 - **PropertyChangeEvent**
 - Objeto que representa a notificação sobre uma alteração que tenha acontecido
 - Permite a obtenção de diferentes valores, como por exemplo, novo valor da propriedade e valor anterior

Exemplo com Property Change

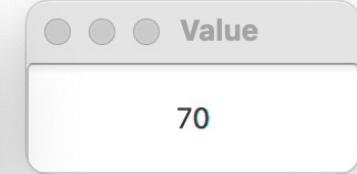
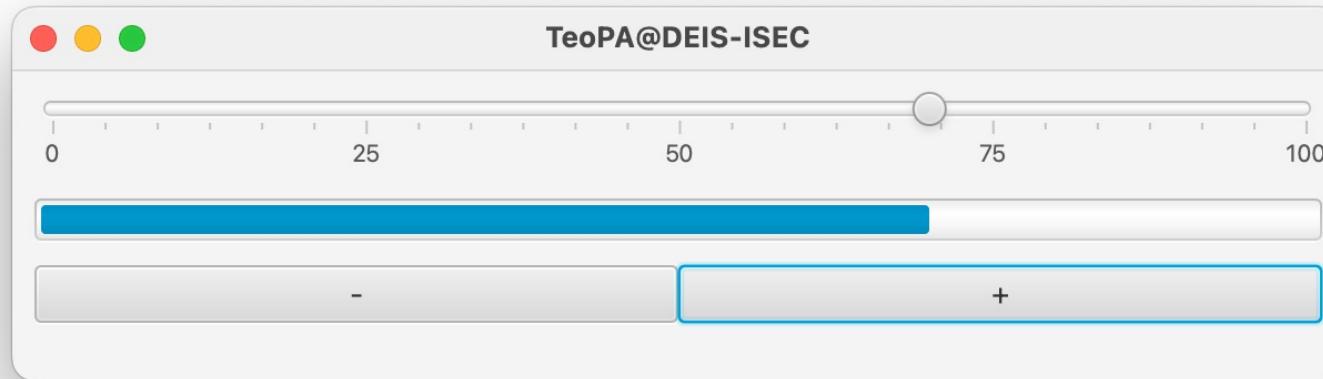
```
class MyModel {  
    String data;  
  
    public String getData() {  
        return data;  
    }  
  
    public void setData(String data) {  
        this.data = data;  
    }  
}  
  
  
public class Main {  
    public static void main(String[] args) {  
        ModelManager mm = new ModelManager();  
        mm.addPropertyChangeListener(ModelManager.TYPE1, evt -> {  
            System.out.println("L1: "+evt.getNewValue());  
        });  
        mm.addPropertyChangeListener(ModelManager.TYPE2, evt -> {  
            System.out.println("L2: "+evt.getNewValue());  
        });  
        mm.addPropertyChangeListener(evt -> {  
            System.out.println("L3: "+evt.getNewValue());  
        });  
  
        mm.setData1("DEIS-ISEC");  
        mm.setData2("ISEC-DEIS");  
    }  
}
```

Output:
L3: DEIS-ISEC
L1: DEIS-ISEC
L3: ISEC-DEIS
L2: ISEC-DEIS

```
class ModelManager {  
    public static final String TYPE1 = "type1";  
    public static final String TYPE2 = "type2";  
    MyModel myModel;  
    PropertyChangeSupport pcs;  
  
    public ModelManager() {  
        myModel = new MyModel();  
        pcs = new PropertyChangeSupport(this);  
    }  
    public void addPropertyChangeListener(  
        String property, PropertyChangeListener listener) {  
        pcs.addPropertyChangeListener(property,listener);  
    }  
    public void addPropertyChangeListener(  
        PropertyChangeListener listener) {  
        pcs.addPropertyChangeListener(listener);  
    }  
    public String getData() {return myModel.getData(); }  
    public void setData1(String value) {  
        myModel.setData(value);  
        pcs.firePropertyChange(TYPE1,null,value);  
    }  
    public void setData2(String value) {  
        myModel.setData(value);  
        pcs.firePropertyChange(TYPE2,null,value);  
    }  
}
```

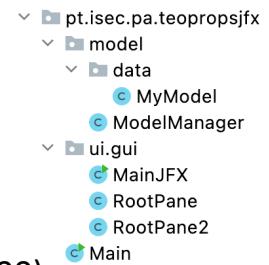
Exemplo de *Property Changes* com JavaFX

- Aplicação que permite atualizações automáticas de interface quando um determinado valor do modelo é alterado
 - O valor pode ser alterado através de um componente Slider ou por edição direta numa janela à parte
 - Simultaneamente é visualizado através de uma ProgressBar



Exemplo de *Property Changes* com JavaFX

- Projeto
 - model
 - data
 - MyModel
 - classe para gerir os dados (no caso concreto um inteiro que varia entre 0 e 100)
 - ModelManager
 - classe para ofuscar pormenores internos de implementação do modelo
 - Adiciona ao modelo a funcionalidade de sinalização de *property change*
 - ui.gui
 - MainJFX
 - classe JavaFX Application para criar os objetos Stage para suporte das duas janelas do exemplo (um dos objetos Stage é o recebido como parâmetro no start)
 - RootPane
 - Árvore de objetos do *Scene Graph* da janela com a Slider e ProgressBar
 - RootPane2
 - Árvore de objetos do *Scene Graph* da janela que permite a edição direta do valor
 - Main
 - Classe com o método main, no qual é carregada toda a plataforma JavaFX (launch)



MyModel

```
public class MyModel {  
    private int value;  
  
    public MyModel(int value) {  
        this.value = value;  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    public void setValue(int value) {  
        this.value = Math.min(Math.max(0,value),100);  
    }  
  
    public void inc() {  
        if (value < 100)  
            value++;  
    }  
  
    public void dec() {  
        if (value > 0)  
            value--;  
    }  
}
```

ModelManager

```
public class ModelManager {  
    public static final String PROP_VALUE = "prop_value";  
    MyModel model;  
    PropertyChangeSupport pcs;  
  
    public ModelManager() {  
        model = new MyModel(50);  
        pcs = new PropertyChangeSupport(this);  
    }  
  
    public void addPropertyChangeListener(String property, PropertyChangeListener listener) {  
        pcs.addPropertyChangeListener(property,listener);  
    }  
  
    public int getValue() { return model.getValue(); }  
  
    public void setValue(int value) {  
        int old = model.getValue();  
        model.setValue(value);  
        pcs.firePropertyChange(PROP_VALUE,old, model.getValue());  
    }  
  
    public void inc() {  
        model.inc();  
        pcs.firePropertyChange(PROP_VALUE,null,null);  
    }  
  
    public void dec() {  
        model.dec();  
        pcs.firePropertyChange(PROP_VALUE,null,null);  
    }  
}
```

MainJFX e Main

```
public class MainJFX extends Application {  
    ModelManager mm;  
    public MainJFX() {  
        mm = new ModelManager();  
    }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        RootPane root = new RootPane(mm);  
        Scene scene = new Scene(root,600,140);  
        stage.setScene(scene);  
        stage.setTitle("TeoPA@DEIS-ISEC");  
        stage.show();  
  
        Stage st2 = new Stage();  
        RootPane2 root2 = new RootPane2(mm);  
        Scene scene2 = new Scene(root2,150,50);  
        st2.setScene(scene2);  
        st2.setX(stage.getX()+stage.getWidth());  
        st2.setY(stage.getY());  
        st2.setTitle("Value");  
        st2.show();  
    }  
}
```

Main:

```
public class Main {  
    public static void main(String[] args) {  
        Application.launch(MainJFX.class,args);  
    }  
}
```

RootPane

```
public class RootPane extends VBox {  
    ModelManager mm;  
    Slider slider;  
    ProgressBar progressBar;  
    Button btnPlus,btnMinus;  
  
    public RootPane(ModelManager mm) {  
        this.mm = mm;  
        createViews();  
        registerHandlers();  
        update();  
    }  
    private void registerHandlers() {  
        mm.addPropertyChangeListener(  
            ModelManager.PROP_VALUE, evt -> { update(); }  
        );  
  
        btnMinus.setOnAction(actionEvent -> { mm.dec(); });  
        btnPlus.setOnAction(actionEvent -> { mm.inc(); });  
        slider.setOnMouseReleased(mouseEvent -> {  
            mm.setValue((int)slider.getValue());  
        });  
        slider.setOnMouseDragged(mouseEvent -> {  
            mm.setValue((int)slider.getValue());  
        });  
    }  
    (...)  
}  
(...)
```

```
(...)  
    private void createViews() {  
        setPadding(new Insets(10));  
        setSpacing(10);  
        slider = new Slider(0,100,50);  
        slider.setShowTickMarks(true);  
        slider.setShowTickLabels(true);  
        slider.setMajorTickUnit(25);  
        slider.setMinorTickCount(5);  
        slider.setPrefWidth(Integer.MAX_VALUE);  
        progressBar = new ProgressBar(0.5);  
        progressBar.setPrefWidth(Integer.MAX_VALUE);  
  
        HBox hbox = new HBox();  
        btnMinus = new Button("-");  
        btnMinus.setPrefWidth(Integer.MAX_VALUE);  
        btnPlus = new Button("+");  
        btnPlus.setPrefWidth(Integer.MAX_VALUE);  
        hbox.setAlignment(Pos.CENTER);  
        hbox.getChildren().addAll(btnMinus,btnPlus);  
        this.getChildren()  
            .addAll(slider,progressBar,hbox);  
    }  
  
    private void update() {  
        System.out.println(mm.getValue());  
        slider.setValue(mm.getValue());  
        progressBar.setProgress(mm.getValue()/100.0);  
    }  
}
```

RootPane2

```
public class RootPane2 extends TextField {  
    ModelManager mm;  
  
    public RootPane2(ModelManager mm) {  
        this.mm = mm;  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() { this.setAlignment(Pos.CENTER); }  
  
    private void registerHandlers() {  
        mm.addPropertyChangeListener(ModelManager.PROP_VALUE, evt -> { update(); });  
  
        this.setOnKeyPressed(keyEvent -> {  
            if (keyEvent.getCode() == KeyCode.ENTER) {  
                try {  
                    mm.setValue(Integer.parseInt(this.getText()));  
                } catch (Exception e) { }  
            }  
        });  
    }  
  
    private void update() { this.setText(""+mm.getValue()); }  
}
```

FSM com JavaFX + *Property Change*

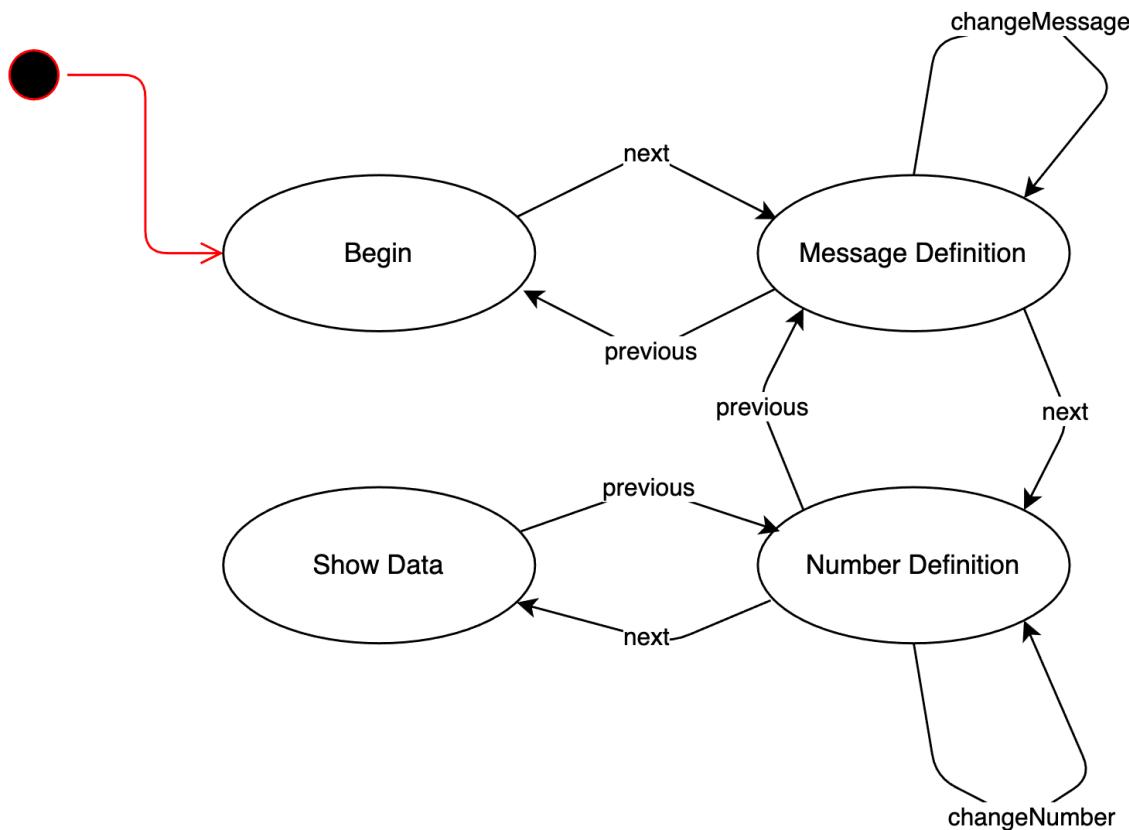
- A criação interfaces para máquinas de estados pode ser realizada de diversas formas.
- Uma possibilidade é ...
 - Criar uma árvore de elementos visuais para cada estado
 - Semelhante ao realizado nos exemplos anteriores para o RootPane, mas um novo para cada um dos estados, eventualmente diferentes entre si
 - Cada uma das árvores criadas irá alternadamente ocupar o espaço definido por um objeto Pane ou seus derivados, colocado na janela principal da aplicação para esse efeito
 - A utilização de StackPane para este efeito poderá ser uma boa opção, uma vez que todos os "root pane" irão sobrepor-se
 - A forma de garantir que apenas a árvore correspondente ao estado ativo é visualizada passa por
 - Definir uma *Property Change* que permita verificar quando o estado ativo se altera
 - Todos os "root pane" de cada estado manifestam o interesse nessa *Property Change*
 - Quando surgir o evento, todas os "root pane" serão avisados dessa mudança
 - Caso o estado ativo corresponda ao "root pane", ele configurar-se-á como visível
 - Caso o estado ativo não corresponda ao "root pane", ele configurar-se-á como invisível

Exemplo FSM + JavaFX

- Nos slides seguintes será apresentada uma aplicação com um exemplo académico, em que:
 - Existe um modelo de dados que gera um valor inteiro e uma *string*
 - As alterações da *string* e do valor inteiro devem ser geridas através de uma máquina de estados, com estados separados para alterar cada um dos valores
 - Os valores atuais deverão estar sempre visíveis no ecrã (numa *status bar*)

Exemplo FSM + JavaFX

- Diagrama de estados



- Estrutura do projeto

```
pt.isec.pa.teostatejfx
└── model
    └── data
        └── ModelData.java
    └── fsm
        └── states
            └── BeginState.java
            └── MessageDefinitionState.java
            └── NumberDefinitionState.java
            └── ShowDataState.java
            └── Context.java
            └── IState.java
            └── State.java
            └── StateAdapter.java
            └── ModelManager.java
└── ui.gui
    └── BeginUI.java
    └── MainJFX.java
    └── MessageDefinitionUI.java
    └── NumberDefinitionUI.java
    └── RootPane.java
    └── ShowDataUI.java
    └── StatusBar.java
    └── Main.java
```

ModelData

```
public class ModelData {  
    String message;  
    int number;  
  
    public ModelData() {  
        message = "none";  
        number = 0;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
  
    public int getNumber() {  
        return number;  
    }  
  
    public void setNumber(int number) {  
        this.number = number;  
    }  
}
```

State, IState

```
public enum State {  
    BEGIN, MESSAGE_DEFINITION, NUMBER_DEFINITION, SHOW_DATA;  
  
IState createState(Context context, ModelData data) {  
    return switch (this) {  
        case BEGIN          -> new BeginState(context,data);  
        case MESSAGE_DEFINITION -> new MessageDefinitionState(context,data);  
        case NUMBER_DEFINITION -> new NumberDefinitionState(context,data);  
        case SHOW_DATA       -> new ShowDataState(context,data);  
    };  
}  
}
```

```
public interface IState {  
    void changeMessage(String msg);  
    void changeNumber(int nr);  
    void next();  
    void previous();  
  
    State getState();  
}
```

StateAdapter

```
public abstract class StateAdapter implements IState {  
    protected Context context;  
    protected ModelData data;  
  
    protected StateAdapter(Context context, ModelData data) {  
        this.context = context;  
        this.data = data;  
    }  
  
    protected void changeState(State newState) {  
        context.changeState(newState.createState(context,data));  
    }  
  
    @Override  
    public void changeMessage(String msg) { }  
  
    @Override  
    public void changeNumber(int nr) { }  
  
    @Override  
    public void next() { }  
  
    @Override  
    public void previous() { }  
}
```

BeginState

```
public class BeginState extends StateAdapter {  
    public BeginState(Context context, ModelData data) {  
        super(context,data);  
    }  
  
    @Override  
    public void next() {  
        changeState(State.MESSAGE_DEFINITION);  
    }  
  
    @Override  
    public State getState() {  
        return State.BEGIN;  
    }  
}
```

MessageDefinitionState

```
public class MessageDefinitionState extends StateAdapter {  
    public MessageDefinitionState(Context context, ModelData data) {  
        super(context,data);  
    }  
  
    @Override  
    public void next() { changeState(State.NUMBER_DEFINITION); }  
  
    @Override  
    public void previous() { changeState(State.BEGIN); }  
  
    @Override  
    public void changeMessage(String msg) { data.setMessage(msg); }  
  
    @Override  
    public State getState() { return State.MESSAGE_DEFINITION; }  
}
```

NumberDefinitionState

```
public class NumberDefinitionState extends StateAdapter {  
    public NumberDefinitionState(Context context, ModelData data) {  
        super(context, data);  
    }  
  
    @Override  
    public void next() { changeState(State.SHOW_DATA); }  
  
    @Override  
    public void previous() { changeState(State.MESSAGE_DEFINITION); }  
  
    @Override  
    public void changeNumber(int nr) { data.setNumber(nr); }  
  
    @Override  
    public State getState() { return State.NUMBER_DEFINITION; }  
}
```

ShowDataState

```
public class ShowDataState extends StateAdapter {  
    public ShowDataState(Context context, ModelData data) {  
        super(context, data);  
    }  
  
    @Override  
    public void previous() {  
        changeState(State.NUMBER_DEFINITION);  
    }  
  
    @Override  
    public State getState() {  
        return State.SHOW_DATA;  
    }  
}
```

Context

```
public class Context {  
    ModelData data;  
    IState state;  
  
    public Context() {  
        data = new ModelData();  
        state = State.BEGIN.createState(data);  
    }  
  
    void changeState(IState state)          { this.state = state; }  
  
    public State getState()                { return state.getState(); }  
  
    public void next()                     { state.next(); }  
  
    public void previous()                 { state.previous(); }  
  
    public void changeMessage(String msg) { state.changeMessage(msg); }  
  
    public void changeNumber(int nr)      { state.changeNumber(nr); }  
  
    public int getNumber()                { return data.getNumber(); }  
  
    public String getMessage()           { return data.getMessage(); }  
}
```

ModelManager

```
public class ModelManager {  
    public static final String PROP_STATE = "state";      public static final String PROP_DATA  = "data";  
  
    Context context;  
    PropertyChangeSupport pcs;  
  
    public ModelManager() {  
        this.context = new Context();  
        pcs = new PropertyChangeSupport(this);  
    }  
  
    public void addPropertyChangeListener(String property, PropertyChangeListener listener) {  
        pcs.addPropertyChangeListener(property, listener);  
    }  
  
    public State getState() { return context.getState(); }  
  
    public void next() { context.next(); pcs.firePropertyChange(PROP_STATE,null,context.getState()); }  
  
    public void previous() { context.previous(); pcs.firePropertyChange(PROP_STATE,null,context.getState()); }  
  
    public void changeMessage(String msg) { context.changeMessage(msg); pcs.firePropertyChange(PROP_DATA,null,null); }  
  
    public void changeNumber(int nr) { context.changeNumber(nr); pcs.firePropertyChange(PROP_DATA,null,null); }  
  
    public int getNumber() { return context.getNumber(); }  
  
    public String getMessage() { return context.getMessage(); }  
}
```

Main, MainJFX

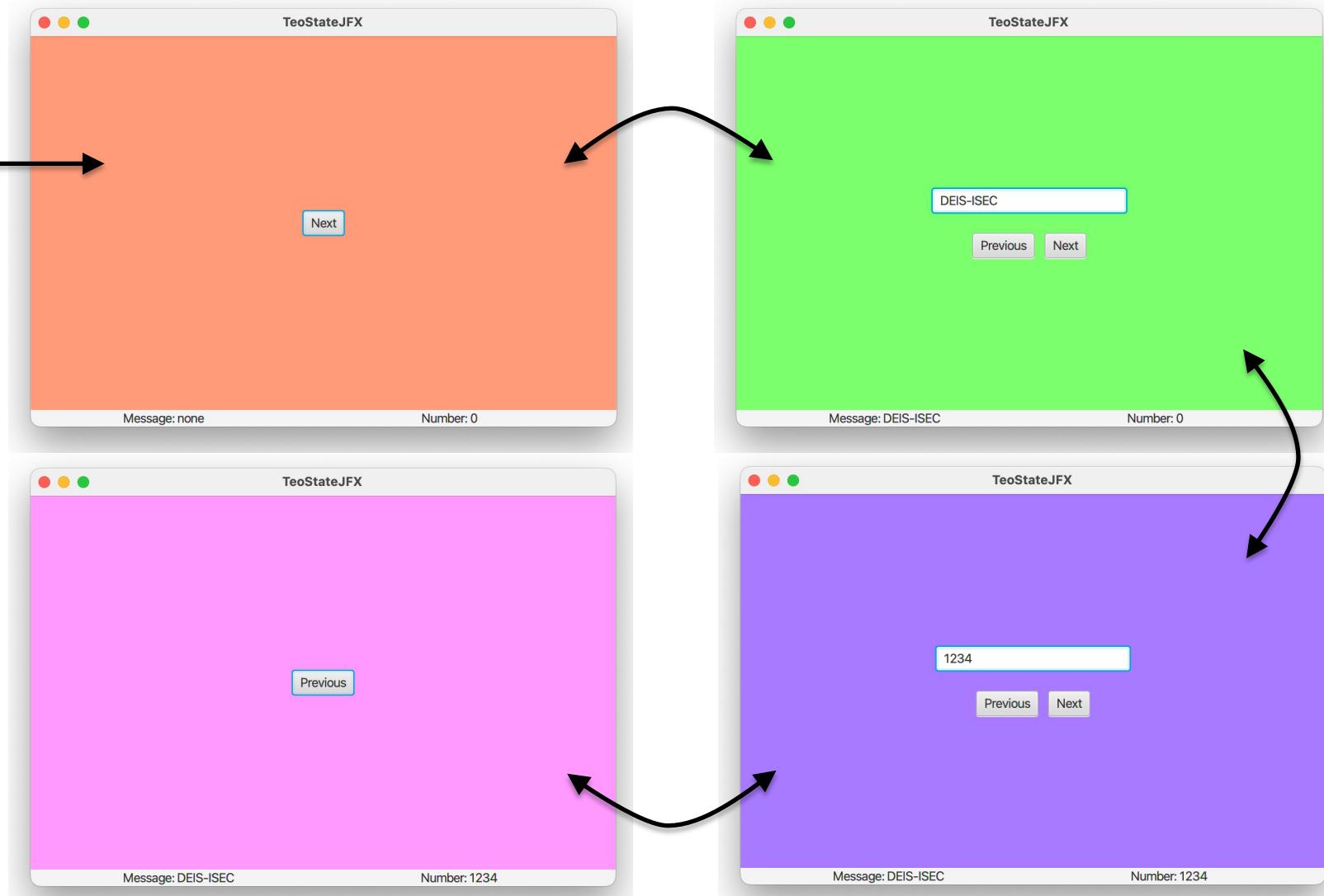
```
public class Main {  
    public static void main(String[] args) {  
        Application.launch(MainJFX.class, args);  
    }  
}
```

```
public class MainJFX extends Application {  
    ModelManager model;  
  
    public MainJFX() { model = new ModelManager(); }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        RootPane root = new RootPane(model);  
        Scene scene = new Scene(root, 600, 400);  
        stage.setScene(scene);  
        stage.setTitle("TeoStateJFX");  
        stage.setMinWidth(400);  
        stage.show();  
    }  
}
```

RootPane

```
public class RootPane extends BorderPane {  
    ModelManager model;  
  
    public RootPane(ModelManager model) {  
        this.model = model;  
  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() {  
        StackPane stackPane = new StackPane(  
            new BeginUI(model), new MessageDefinitionUI(model),  
            new NumberDefinitionUI(model), new ShowDataUI(model) );  
        this.setCenter(stackPane);  
  
        this.setBottom(new StatusBar(model));  
    }  
  
    private void registerHandlers() { }  
  
    private void update() { }  
}
```

UI da aplicação



BeginUI

```
public class BeginUI extends BorderPane {  
    ModelManager model;  
    Button btnNext;  
  
    public BeginUI(ModelManager model) {  
        this.model = model;  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() {  
        this.setStyle("-fx-background-color: #FFA080;");  
        btnNext = new Button("Next");  
        this.setCenter(btnNext);  
    }  
  
    private void registerHandlers() {  
        model.addPropertyChangeListener(ModelManager.PROP_STATE, evt -> { update(); });  
        btnNext.setOnAction(actionEvent -> { model.next(); });  
    }  
  
    private void update() {  
        this.setVisible(model.getState() == State.BEGIN);  
    }  
}
```

MessageDefinitionUI

```
public class MessageDefinitionUI extends BorderPane {  
    ModelManager model;  
    Button btnNext,btnPrevious;  
    TextField tfMsg;  
  
    public MessageDefinitionUI(ModelManager model) {  
        this.model = model;  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() {  
        this.setStyle(  
            "-fx-background-color: #A0FF80;");  
  
        btnNext = new Button("Next");  
        btnPrevious = new Button("Previous");  
        HBox hbox = new HBox(btnPrevious,btnNext);  
        hbox.setSpacing(10);  
        hbox.setAlignment(Pos.CENTER);  
        tfMsg = new TextField();  
        tfMsg.setMaxWidth(200);  
        VBox vbox = new VBox(tfMsg,hbox);  
        vbox.setSpacing(20);  
        vbox.setAlignment(Pos.CENTER);  
        this.setCenter(vbox);  
  
    }  
(...)
```

```
(...)  
  
private void registerHandlers() {  
    model.addPropertyChangeListener(  
        ModelManager.PROP_STATE, evt -> {  
            update();  
        });  
    btnNext.setOnAction(actionEvent -> {  
        model.next();  
    });  
    btnPrevious.setOnAction(actionEvent -> {  
        model.previous();  
    });  
    tfMsg.setOnKeyTyped(keyEvent -> {  
        model.changeMessage(tfMsg.getText());  
    });  
}  
  
private void update() {  
    this.setVisible(  
        model.getState()==State.MESSAGE_DEFINITION  
    );  
    tfMsg.setText(model.getMessage());  
    tfMsg.requestFocus();  
}
```

NumberDefinitionUI

```
public class NumberDefinitionUI extends BorderPane {  
    ModelManager model;  
    Button btnNext,btnPrevious;  
    TextField tfNumber;  
  
    public NumberDefinitionUI(ModelManager model) {  
        this.model = model;  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() {  
        this.setStyle(  
            "-fx-background-color: #A080FF;");  
  
        btnNext = new Button("Next");  
        btnPrevious = new Button("Previous");  
        HBox hbox = new HBox(btnPrevious,btnNext);  
        hbox.setSpacing(10);  
        hbox.setAlignment(Pos.CENTER);  
        tfNumber = new TextField();  
        tfNumber.setMaxWidth(200);  
        VBox vbox = new VBox(tfNumber,hbox);  
        vbox.setSpacing(20);  
        vbox.setAlignment(Pos.CENTER);  
        this.setCenter(vbox);  
    }  
(...)
```

```
(...)  
  
private void registerHandlers() {  
    model.addPropertyChangeListener(  
        ModelManager.PROP_STATE, evt -> {  
            update();  
        });  
    btnNext.setOnAction(actionEvent -> {  
        model.next();  
    });  
    btnPrevious.setOnAction(actionEvent -> {  
        model.previous();  
    });  
    tfNumber.setOnKeyTyped(keyEvent -> {  
        try {  
            model.changeNumber(  
                Integer.parseInt(tfNumber.getText())  
            );  
        } catch (Exception e) {  
            model.changeNumber(-1);  
        }  
    });  
}  
  
private void update() {  
    this.setVisible(  
        model.getState()==State.NUMBER_DEFINITION);  
    tfNumber.setText(""+model.getNumber());  
    tfNumber.requestFocus();  
}
```

ShowDataUI

```
public class ShowDataUI extends BorderPane {  
    ModelManager model;  
    Button btnPrevious;  
  
    public ShowDataUI(ModelManager model) {  
        this.model = model;  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() {  
        this.setStyle("-fx-background-color: #FFA0FF;");  
        btnPrevious = new Button("Previous");  
        this.setCenter(btnPrevious);  
    }  
  
    private void registerHandlers() {  
        model.addPropertyChangeListener(ModelManager.PROP_STATE, evt -> { update(); });  
        btnPrevious.setOnAction(actionEvent -> { model.previous(); });  
    }  
  
    private void update() {  
        this.setVisible(model.getState() == State.SHOW_DATA);  
    }  
}
```

StatusBar

```
public class StatusBar extends HBox {  
    ModelManager model;  
    Label lbMsg, lbNumber;  
  
    public StatusBar(ModelManager model) {  
        this.model = model;  
        createViews(); registerHandlers(); update();  
    }  
    private void createViews() {  
        Label lbMsgTitle = new Label("Message: ");  
        lbMsgTitle.setPrefWidth(Integer.MAX_VALUE);  
        lbMsgTitle.setAlignment(Pos.CENTER_RIGHT);  
        lbMsg = new Label();  
        lbMsg.setPrefWidth(Integer.MAX_VALUE);  
        lbMsg.setAlignment(Pos.CENTER_LEFT);  
        Label lbNumberTitle = new Label("Number: ");  
        lbNumberTitle.setPrefWidth(Integer.MAX_VALUE);  
        lbNumberTitle.setAlignment(Pos.CENTER_RIGHT);  
        lbNumber = new Label();  
        lbNumber.setPrefWidth(Integer.MAX_VALUE);  
        lbNumber.setAlignment(Pos.CENTER_LEFT);  
        this.getChildren().addAll(lbMsgTitle, lbMsg, lbNumberTitle, lbNumber);  
    }  
    private void registerHandlers() { model.addPropertyChangeListener(ModelManager.PROP_DATA, evt -> { update(); }); }  
    private void update() {  
        lbMsg.setText(model.getMessage());  
        lbNumber.setText("" + model.getNumber());  
    }  
}
```

Programação Avançada

JavaFX

FXML e Controllers

FXML

- Para além da forma de criação de *Scene Graphs* de forma programática, é possível definir a árvore de objetos que constituem a interface gráfica através de ficheiros XML – FXML
 - A árvore de *nodes* é definida através de estruturas XML, com nomes iguais aos Nodes usados programaticamente bem como dos seus atributos
 - As dependências são realizadas definindo estruturas dentro de outras estruturas

FXML

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

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.FlowPane?>

<FlowPane xmlns="http://javafx.com/javafx"
           xmlns:fx="http://javafx.com/fxml"
           prefHeight="400.0" prefWidth="600.0">

    <Label text="Advanced Programming" />

</FlowPane>
```

FXMLLoader

- A leitura dos ficheiros `fxml` pode ser realizada recorrendo ao método `FXMLLoader.load()`
 - O método `load` retorna a raiz da árvore de *nodes*

```
FXMLLoader loader = new FXMLLoader();
loader.setLocation(getClass().getResource("fxml/screen1.fxml"));
Parent root = loader.load();
Scene scene = new Scene(root);
```

- Alternativa:

```
Scene scene = new Scene( new FXMLLoader(
                                getClass().getResource("fxml/screen1.fxml")
                                ).load() );
```

Controller

- Para processar os eventos da árvore de objetos definida através de um ficheiro FXML pode ser definido um *Controller*
 - Um *Controller* é uma classe Java que poderá ser associada de duas formas
 - Ficheiro XML
 - Incluir no elemento *root* a especificação do controlador através do atributo `fx:controller`
`fx:controller="pt.isec.pa.prepfxml.ui.Screen1"`
 - Pode-se obter a referência para o *controller* através do objeto FXMLLoader
`Screen1 screen1 = loader.getController();`
 - *Runtime*
 - Usar o método `setController` da instância FXMLLoader para indicar o objeto *controller*
`loader.setController(new Screen1());`
 - Notas: nos exemplos anteriores a classe `Screen1` pode ser definida com:
`package pt.isec.pa.prepfxml.ui.Screen1`
`class Screen1 { }`

Controller e Initializable

- Para que possam ser feitas iniciações a variáveis ou execução de outro tipo de comportamento, após o ficheiro FXML ser lido com sucesso, a classe *controller* pode implementar a interface **Initializable**

- Esta interface obriga à implementação do método

```
@Override
```

```
public void initialize(URL url,  
                      ResourceBundle resourceBundle) {  
    //TODO  
}
```

Referências para os *nodes*

- Ao ser criada a árvore de objetos através de ficheiros FXML perde-se a possibilidade de guardar em variáveis as referências para os *nodes* criados
- Para resolver este problema, o FXML permite fazer uma associação fácil entre os elementos XML e variáveis de tipos adequados criadas no *controller* através do atributo `fx:id`

- FXML:

```
...
<Label fx:id="myLabel" text="Advanced Programming" />
...
```

- *Controller*:

```
public class Screen1 implements Initializable {
    @FXML public Label myLabel;

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        myLabel.setText("AdvProg-DEIS-ISEC");
    }
}
```

Processamento de eventos

- O processamento de eventos que ocorram sobre os *nodes* podem ser configurados usando atributos com nomes correspondentes aos métodos `setOn*`, por exemplo: `onAction`, `onMouseClicked`, ...
 - No atributos indica-se o nome do método definido para esse efeito no *controller* precedido de '#'
 - FXML:

```
...
<Button text="Button 1" onAction="#onButton1" />
...
...
```

- *Controller*:

```
public class Screen1 implements Initializable {
    @FXML public Label myLabel;

    @FXML
    public void onButton1(ActionEvent event) {
        myLabel.setText("PA-DEIS-ISEC");
    }
}
```

Inclusão de outros ficheiros FXML

- Um ficheiro FXML pode ser formado pela composição de vários ficheiros FXML, fazendo a sua inclusão através do elemento

```
<fx:include source="fxmL" />
```

- Cada FXML terá o seu *controller*



- screen1.fxml
 - ...
 - <VBox xmlns="http://javafx.com/javafx" xmlns:fx="http://javafx.com/fxml" fx:controller="pt.isec.pa.prepFXML.ui.Screen1" prefHeight="400.0" prefWidth="600.0">
 <fx:include source="mytoolbar.fxml" maxHeight="40"/>
 <Label fx:id="myLabel" text="..." />
 <Button text="Button 1" onAction="#onButton1" />
 </VBox>
- mytoolbar.fxml
 - ...
 - <ToolBar xmlns="http://javafx.com/javafx" xmlns:fx="http://javafx.com/fxml" fx:controller="pt.isec.pa.prepFXML.ui.MyToolbar">
 <Button text="Button 1" />
 <Button text="Button 2" />
 <Button text="Button 3" />
 </ToolBar>

Inclusão de outros ficheiros FXML

- A partir do código de um controlador pode-se alterar a árvore de objetos e passar o controlo para outro controlador
 - Deverá ser obtida a referência do objeto Scene para alterar o elemento *root*

```
@FXML  
public void onButton1(ActionEvent event)  
        throws IOException {  
    Button btn = (Button) event.getSource();  
    Scene scene = btn.getScene();  
    scene.setRoot( new FXMLLoaderLoader(  
            getClass().getResource("fxml/screen2.fxml")  
        ).load() );  
}
```

Obtenção do objeto Scene

- Para facilitar o acesso ao objeto Scene poder-se-á atribuir um `fx:id` ao elemento raiz que depois será usado para esse fim

- `screen2.fxml`:

```
<AnchorPane xmlns="http://javafx.com/javafx"
    xmlns:fx="http://javafx.com/fxml"
    fx:controller="pt.isec.pa.prepfxml.ui.Screen2"
    prefHeight="400.0" prefWidth="600.0"
    fx:id="root" >

    <Label text="PA" alignment="CENTER"
        fx:id="lbCenter"
        AnchorPane.bottomAnchor="0.0"
        AnchorPane.topAnchor="0.0"
        AnchorPane.rightAnchor="0.0"
        AnchorPane.leftAnchor="0.0" />

</AnchorPane>
```

- `Screen2.java`:

```
public class Screen2 {
    @FXML public AnchorPane root;
    @FXML public Label lbCenter;

    public void function() {
        ...
        Scene scene = root.getScene();
        ...
    }
}
```

Acesso aos dados da aplicação

- A passagem de uma referência para o modelo de dados a usar nos diversos ecrãs poderá não ser uma tarefa simples
 - Quando os controladores são criados de forma explícita e, posteriormente, atribuído ao *loader* (*setController*) a tarefa fica facilitada porque se possui uma referência para o controlador
 - Pode-se passar informação pelo *constructor* do controlador
 - Podem-se usar métodos *set** ou similares para alterar os dados
 - Depois de realizar o *load()* aproveitar o método *getController()* para obter a referência do *controller*
 - A partir dessa referência, chamar métodos *set** ou similares para passar informação
- Se para o contexto da aplicação a utilização de *Singleton* ou *Multiton* for admissível então o acesso aos dados é facilitado

UserData

- Os dados podem ser passados entre *controllers* através da informação *user data*
 - passar através do *user data* da raiz da árvore de *nodes* (`loader.getRoot()`) a referência para o modelo de dados ou similar

```
root.setUserData(myModel);
```

 - Há que ter o cuidado para depois ir passando o objeto *user data* entre árvores de objetos que forem sendo carregadas
- passar através do *user data* do objeto *Scene*
 - Esta forma garante que todas as árvores que forem atribuídas a este *Scene* têm acesso ao mesmo *user data*
 - Há que ter os seguintes cuidados:
 - No método *initialize* do *controller* o método *getScene* devolve `null`
 - Pode-se usar a propriedade associada ao objeto *Scene* para detetar a sua disponibilização
 - Poderão existir outros métodos em que o *getScene* não devolve `null`, mas poderá não ter sido ainda atribuída informação de *user data*
 - Para evitar esta situação pode-se primeiro criar o objeto *Scene* com um *fake pane* (ex.: `new Pane()`) para se poder atribuir informação de *user data* e depois atribuir a raiz pretendida para o *Scene Graph*
 - Esta situação aplica-se apenas à fase inicial de atribuição da primeira árvore de objetos

Passagem de dados entre controllers

```
public class MainJFX extends Application {  
    DataManager model;  
    @Override  
    public void init() throws Exception {  
        super.init();  
        model = new DataManager();  
    }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        FXMLLoader loader = new FXMLLoader();  
        loader.setLocation(  
            getClass().getResource("fxml/screen1.fxml"));  
        Parent root = loader.load();  
        Scene scene = new Scene(root);  
        scene.setUserData(model);  
        stage.setScene(scene);  
        stage.setTitle("PA-DEIS-ISEC");  
        stage.show();  
    }  
}
```

```
public class Screen1 {  
    @FXML public Label myLabel;  
    public void onButton1(ActionEvent event)  
        throws IOException {  
        Button btn = (Button) event.getSource();  
        Scene scene = btn.getScene();  
        scene.setRoot(new FXMLLoader(  
            getClass().getResource(  
                "fxml/screen2.fxml")).load());  
    }  
}
```

```
public class Screen2 {  
    @FXML public AnchorPane root;  
    @FXML public Label lbCenter;  
  
    public void onButton(ActionEvent event) {  
        Scene scene = root.getScene();  
        DataManager model = (DataManager)  
            scene.getUserData();  
        lbCenter.setText(model.getValueS());  
    }  
}
```

Passagem de dados entre controllers

```
public class MainJFX extends Application {  
    DataManager model;  
    @Override  
    public void init() throws Exception {  
        super.init();  
        model = new DataManager();  
    }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        FXMLLoader loader = new FXMLLoader();  
        loader.setLocation(  
            getClass().getResource("fxml/screen1.fxml"));  
        Parent root = loader.load();  
        Scene scene = new Scene(new Pane());  
        scene.setUserData(model);  
        scene.setRoot(root);  
        stage.setScene(scene);  
        stage.setTitle("PA-DEIS-ISEC");  
        stage.show();  
    }  
}
```

```
public class Screen1 {  
    @FXML public Label myLabel;  
    public void onButton1(ActionEvent event)  
        throws IOException {  
        Button btn = (Button) event.getSource();  
        Scene scene = btn.getScene();  
        scene.setRoot(new FXMLLoader(  
            getClass().getResource(  
                "fxml/screen2.fxml")).load());  
    }  
}
```

```
public class Screen2 {  
    @FXML public AnchorPane root;  
    @FXML public Label lbCenter;  
  
    public void onButton(ActionEvent event) {  
        Scene scene = root.getScene();  
        DataManager model = (DataManager)  
            scene.getUserData();  
        lbCenter.setText(model.getValueS());  
    }  
}
```

Passagem de dados entre *controllers*

- Mesmo usando a solução indicada anteriormente existe um problema (que já tinha sido referido)
- No momento em que o método `initialize` do *controller* é executado, a raiz da árvore de nós ainda não foi associada ao objeto *Scene*
 - Se o modelo de dados fosse passado através do *user data* da raiz da árvore de nós o problema era similar porque o método `setUserData` só será executado depois do `load()` retornar, ou seja, já depois do método `initialize` ser executado
- A solução passa por aceder ao modelo de dados passado através do *user data* só depois de a raiz da árvore de objetos ser atribuída a um objeto *Scene*

Propriedades JavaFX

- O *JavaFX*, através de um conjunto de classes disponível no package `javafx.beans`, fornece as funcionalidades do padrão *Observer/Observable* para as propriedades dos objetos *JavaFX*
 - Assim, é possível manifestar o interesse nas alterações sofridas por uma qualquer propriedade de um objeto *JavaFX* (*width*, *height*, *text*, ...)
 - As propriedades são acedidas através de métodos com nomes no formato `<prop>Property()`
 - Exemplos:
 - `root.widthProperty()`
 - `root.heightProperty()`
 - `label.textProperty()`
 - `root.sceneProperty()`

Propriedades JavaFX

- Os métodos referidos para acesso às propriedades retornam objetos que implementam várias interfaces Java entre as quais se podem salientar:
 - **Observable**
 - disponibiliza operações típicas do padrão *observer/observable*
 - addListener
 - removeListener
 - **Property<T>**
 - disponibiliza gerar relacionamentos/dependências entre propriedades
 - bind
 - bindBidirectional
 - unbind
 - unbindBidirectional
 - **WritableValue<T>**
 - permite obter e alterar o valor
 - getValue
 - setValue
 - ...

Passagem de dados entre controllers

```
public class MainJFX extends Application {  
    DataManager model;  
    @Override  
    public void init() throws Exception {  
        super.init();  
        model = new DataManager();  
    }  
  
    @Override  
    public void start(Stage stage) throws Exception {  
        Scene scene = new Scene(new Pane());  
        scene.setUserDatas(model);  
  
        FXMLLoader loader = new FXMLLoader();  
        loader.setLocation(  
            getClass().getResource("fxml/screen1.fxml"));  
  
        Parent root = loader.load();  
        scene.setRoot(root);  
        stage.setScene(scene);  
        stage.setTitle("PA-DEIS-ISEC");  
        stage.show();  
    }  
}
```

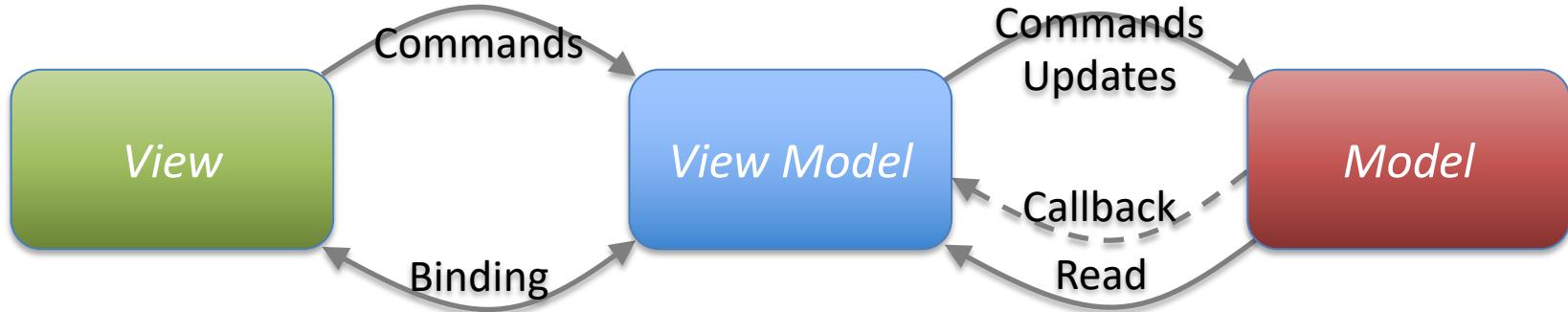
```
public class Screen1 implements Initializable {  
    @FXML public VBox root;  
  
    DataManager model;  
  
    @Override  
    public void initialize(URL url,  
        ResourceBundle resourceBundle) {  
        root.sceneProperty()  
            .addListener( (obs, oldS, newS) -> {  
                model=(DataManager) newS.getUserData();  
                registerHandlers();  
                update();  
            });  
    }  
  
    private void registerHandlers() { }  
  
    private void update() { }  
}
```

Criação de novas propriedades

- No contexto de qualquer classe podem ser criadas novas propriedades, derivando novas classes a partir das interfaces mencionadas anteriormente
- No *package javafx.beans* são fornecidas
 - classes abstratas para os tipos mais usuais
 - IntegerProperty, DoubleProperty, StringProperty, BooleanProperty, ListProperty<E>, ...
 - classes instanciáveis para os tipos mais usuais
 - SimpleIntegerProperty, SimpleDoubleProperty, SimpleStringProperty, ...

Model-View-ViewModel (MVVM revisitado)

- A utilização das propriedades permite a implementação do padrão arquitetural MVVM de forma facilitada
 - Neste padrão
 - A View apenas tem acesso ao modelo de dados através da classe ViewModel
 - As atualizações das vistas são feitas preferencialmente de forma automática através de operações de *bind* entre as propriedades da classe ViewModel e as propriedades das Views



Exemplo MVVM

```
class Screen1MVVM {  
    DataManager model;  
    DoubleProperty valueD;  
    StringProperty valueS;  
  
    public Screen1MVVM(DataManager model) {  
        this.model = model;  
        valueD =  
            new SimpleDoubleProperty(model.getValueD());  
        valueS =  
            new SimpleStringProperty(model.getValueS());  
  
        model.addListener(evt ->  
            Platform.runLater(() -> {  
                valueD.set(model.getValueD());  
                valueS.set(model.getValueS());  
            }));  
        valueS.addListener((obs, oldV, newV) ->  
            Platform.runLater(() -> {  
                model.setValueS(newV);  
            }));  
        valueD.addListener((obs, oldV, newV) ->  
            Platform.runLater(() -> {  
                model.setValueD(newV.doubleValue());  
            }));  
    }  
}
```

```
public class Screen1 extends VBox {  
    Screen1MVVM viewModel;  
    TextField tfD, tfS;  
  
    public Screen1(DataManager model) {  
        this.viewModel = new Screen1MVVM(model);  
        createViews();  
        registerHandlers();  
        update();  
    }  
  
    private void createViews() {  
        // configuration of tfD and tfS omitted  
    }  
  
    private void registerHandlers() {  
        tfD.textProperty()  
            .bindBidirectional(  
                viewModel.valueD,  
                new NumberStringConverter("#.#####"));  
        tfS.textProperty()  
            .bindBidirectional(viewModel.valueS);  
    }  
  
    private void update() { }  
}
```

Exemplo MVVM com FXML

```
class Screen2MVVM {  
    DataManager model;  
    IntegerProperty valueI;  
    StringProperty valueS;  
  
    public Screen2MVVM(DataManager model) {  
        this.model = model;  
        valueI =  
            new SimpleIntegerProperty(model.getValueI());  
        valueS =  
            new SimpleStringProperty(model.getValueS());  
  
        model.addListener(evt ->  
            Platform.runLater(() -> {  
                valueI.set(model.getValueI());  
                valueS.set(model.getValueS());  
            }));  
        valueS.addListener((obs, oldV, newV) ->  
            Platform.runLater(() -> {  
                model.setValueS(newV);  
            }));  
        valueI.addListener((obs, oldV, newV) ->  
            Platform.runLater(() -> {  
                model.setValueI(newV.intValue());  
            }));  
    }  
}
```

```
public class Screen2 implements Initializable {  
    @FXML public VBox root;  
    @FXML public TextField tfI;  
    @FXML public TextField tfS;  
    Screen2MVVM viewModel;  
    @Override  
    public void initialize(  
        URL url, ResourceBundle resourceBundle) {  
        root.sceneProperty()  
            .addListener( (obs, oldS, newS) -> {  
                viewModel = new Screen2MVVM(  
                    (DataManager) newScene.getUserData() );  
                registerHandlers();  
                update();  
            });  
    }  
    private void registerHandlers() {  
        tfI.textProperty()  
            .bindBidirectional(  
                viewModel.valueI,  
                new NumberStringConverter("#"));  
        tfS.textProperty()  
            .bindBidirectional(viewModel.valueS);  
    }  
    private void update() { }  
}
```