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

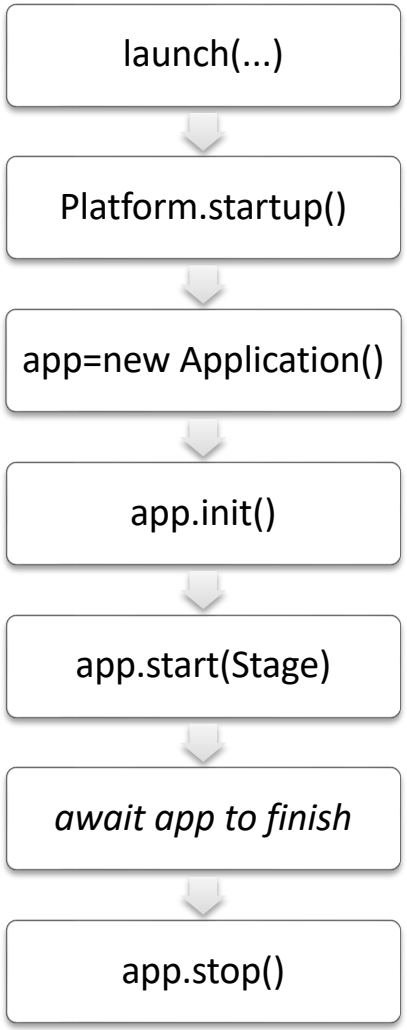
- Quando a classe que deriva de `Application` é diferente da classe onde está definida a função `main`

```
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*

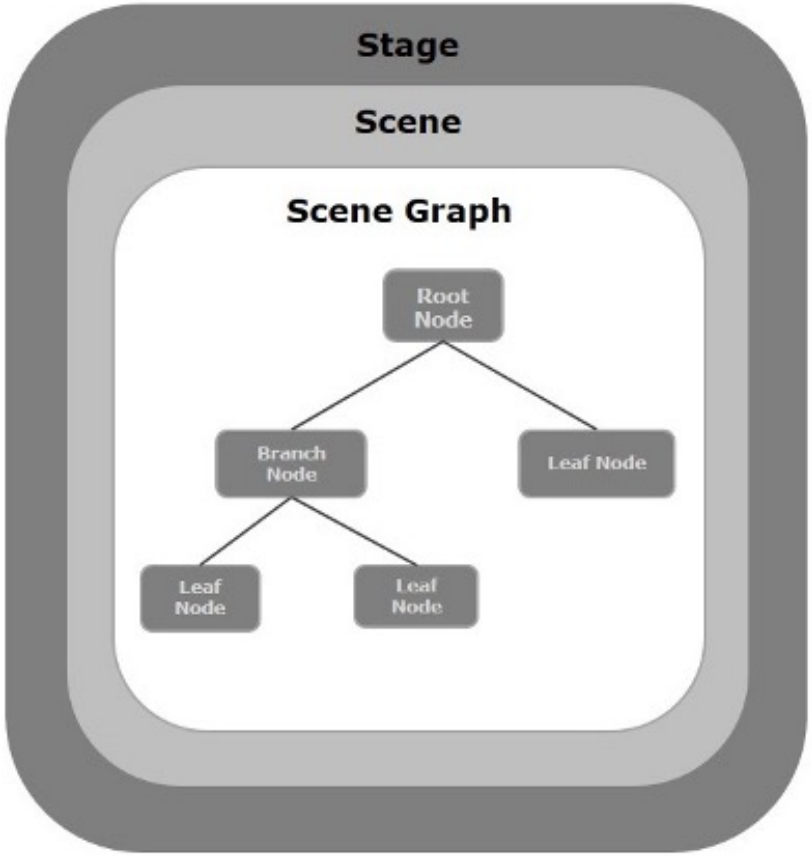
Scene Graph: Stage, Scene e root node

- 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*

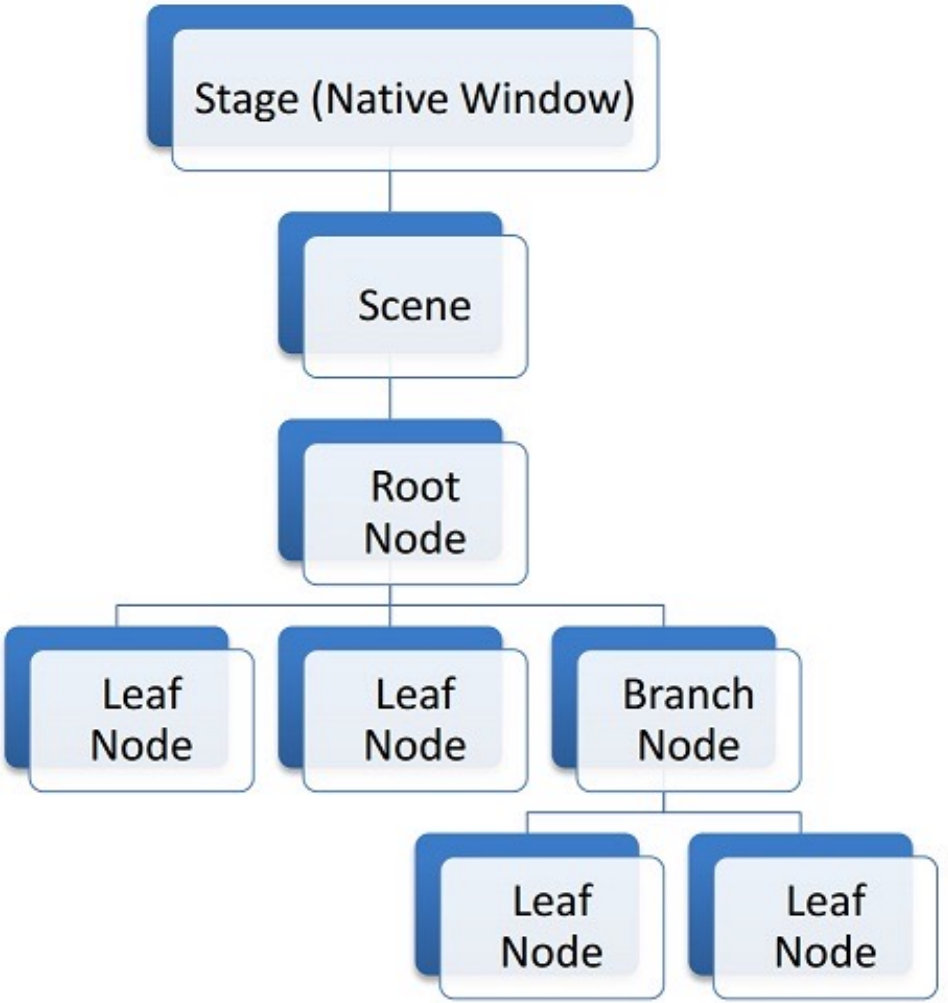
Scene Graph: Stage, Scene e root node

- 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

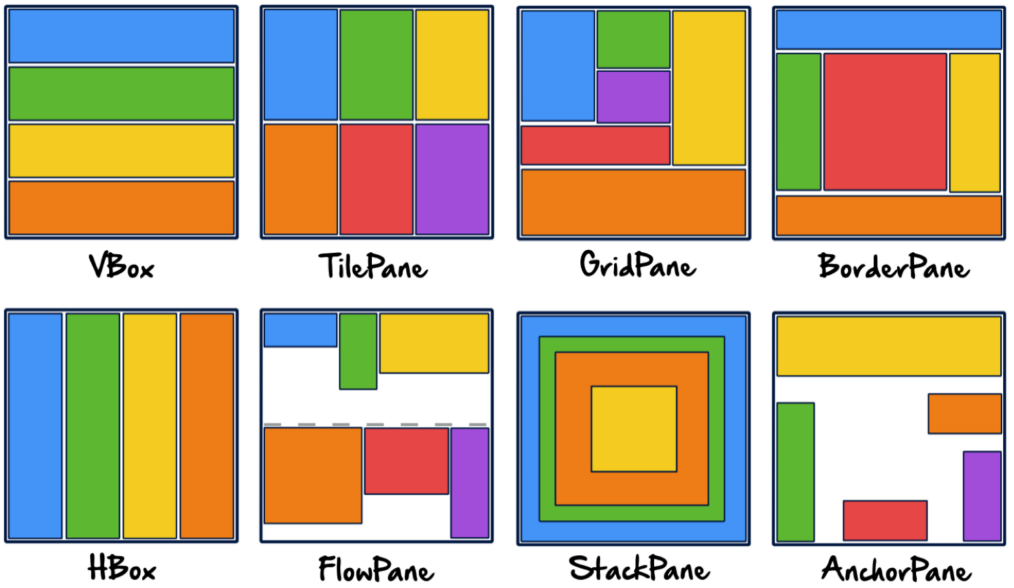
- Region
  - Control
    - TextInputControl
      - **TextArea**, **TextField**
    - ComboBoxBase
      - **ColorPicker**, ComboBox, **DatePicker**
    - Labeled
      - ButtonBase
        - **Button**
        - MenuButton
          - SplitMenuButton
        - ToggleButton
          - **RadioButton**
        - **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*

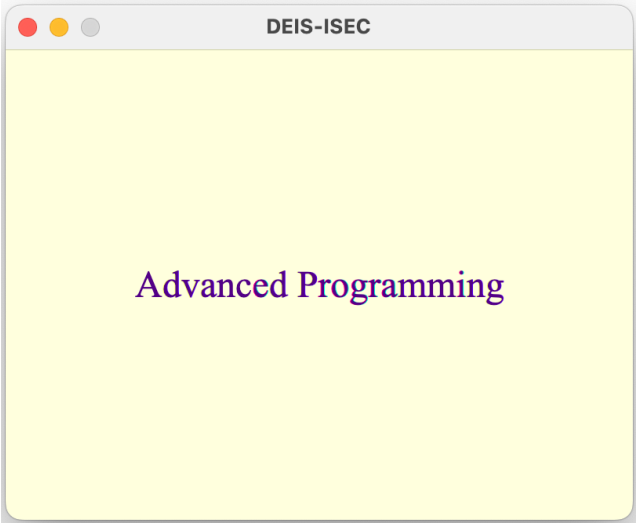
- Dependendo do tipo de objeto de *layout* usado, estão disponíveis diversos parâmetros de formatação e formas de adicionar os componentes que gere (*children*)
  - `getChildren().add, getChildren().addAll`
  - `setPadding`
  - `setAlignment`
  - `setSpacing`
  - `setTopAnchor, setBottomAnchor, setLeftAnchor, 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
  - 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 o método `addEventFilter` podem-se tratar diferentes tipos de eventos e podem ser configurados vários processamentos para o mesmo evento

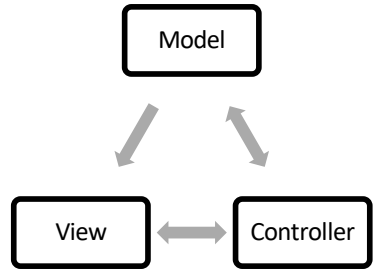
Organização do projeto

- 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
  - ...

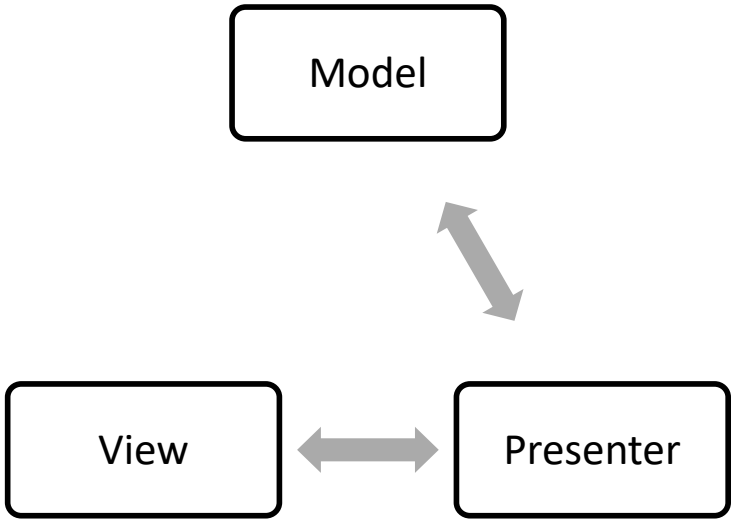
MVC

- 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



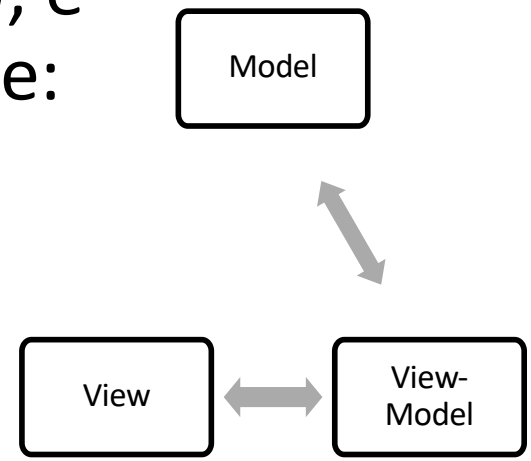
MVP

- 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 atual 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



MVC@PA

- Por vezes é difícil fazer a distinção entre o "controlador" e as "vistas" uma vez que a programação das informações e configuração do aspeto 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`
      - Reponsável 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.getScene().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

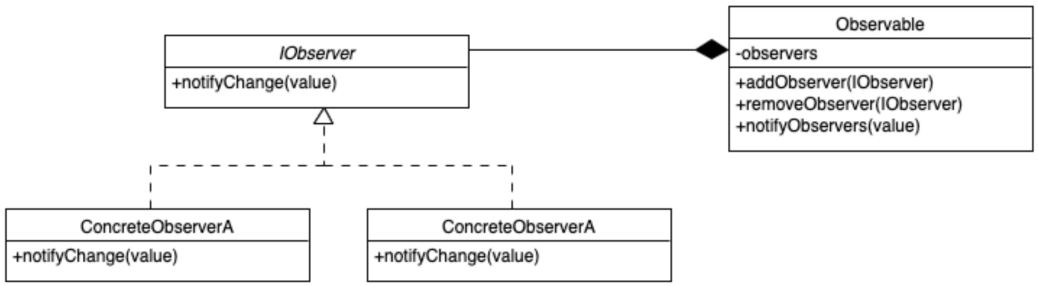
Observer/Observable

```
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);
    }
}
```

```
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);
    }
}
```

```
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");
    }
}
```

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



Observer/Observable

- No `package java.util` é fornecida uma implementação do padrão *Observer*/*Observable* no entanto está marcado como *deprecated*

- Existe um outro padrão similar ao *Observer*/*Observable* designado por *Publisher*/*Subscriber* que em termos práticos oferece funcionalidades similares