

# 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

- 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

- 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

- 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:///Users/ans/Projects/javafx-sdk-17/doc`
      - `file:///Users/ans/Projects/javafx-sdk-17/doc/javafx.base`
      - `file:///Users/ans/Projects/javafx-sdk-17/doc/javafx.controls`
      - `file:///Users/ans/Projects/javafx-sdk-17/doc/javafx.fxml`
      - `file:///Users/ans/Projects/javafx-sdk-17/doc/javafx.graphics`
      - `file:///Users/ans/Projects/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 um 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`

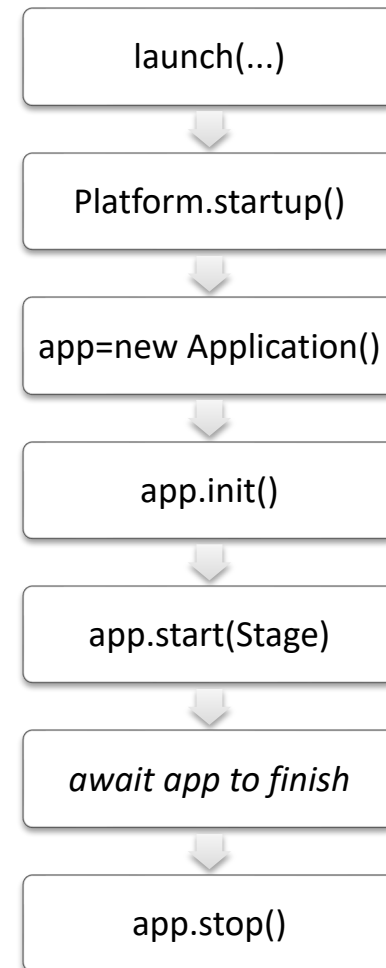
```
public class Main {  
    public static void main(String [] args) {  
        Application.launch(JavaFXMain.class,args);  
    }  
}
```

---

```
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

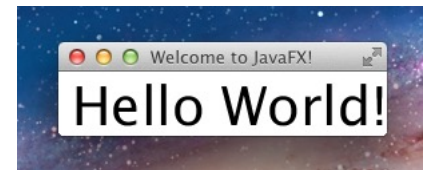


# Scene Graph: Stage, Scene e root node

- 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

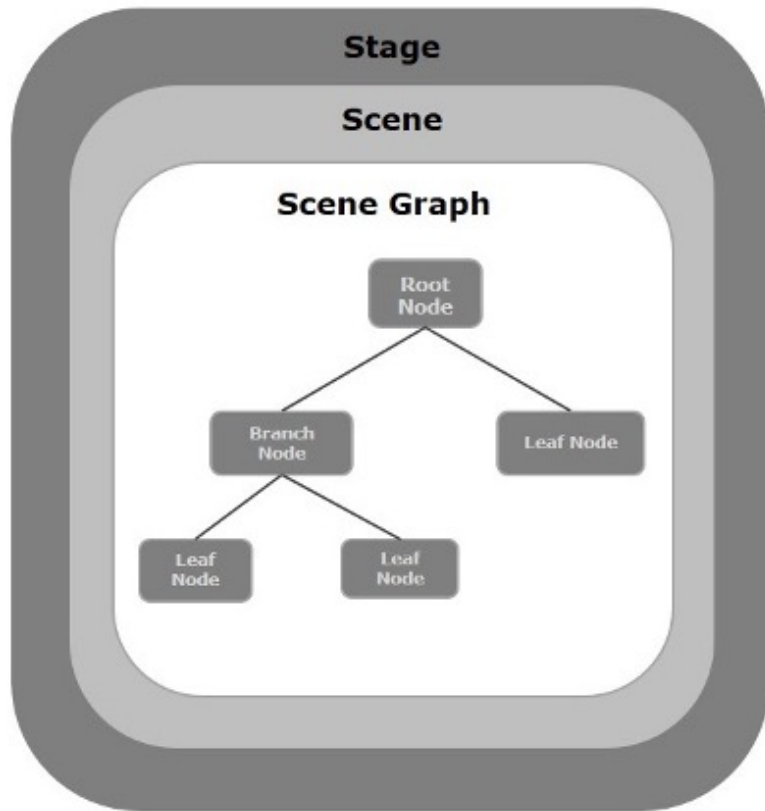
- A interface gráfica que caracteriza uma aplicação *JavaFX* deve ser criada no método `start` do objeto `Application`
- Os objetos que constituem a interface são organizados através de uma árvore de objetos, designada *Scene Graph*, a qual é constituído por:
  - Stage
  - Scene
    - Representa os componentes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



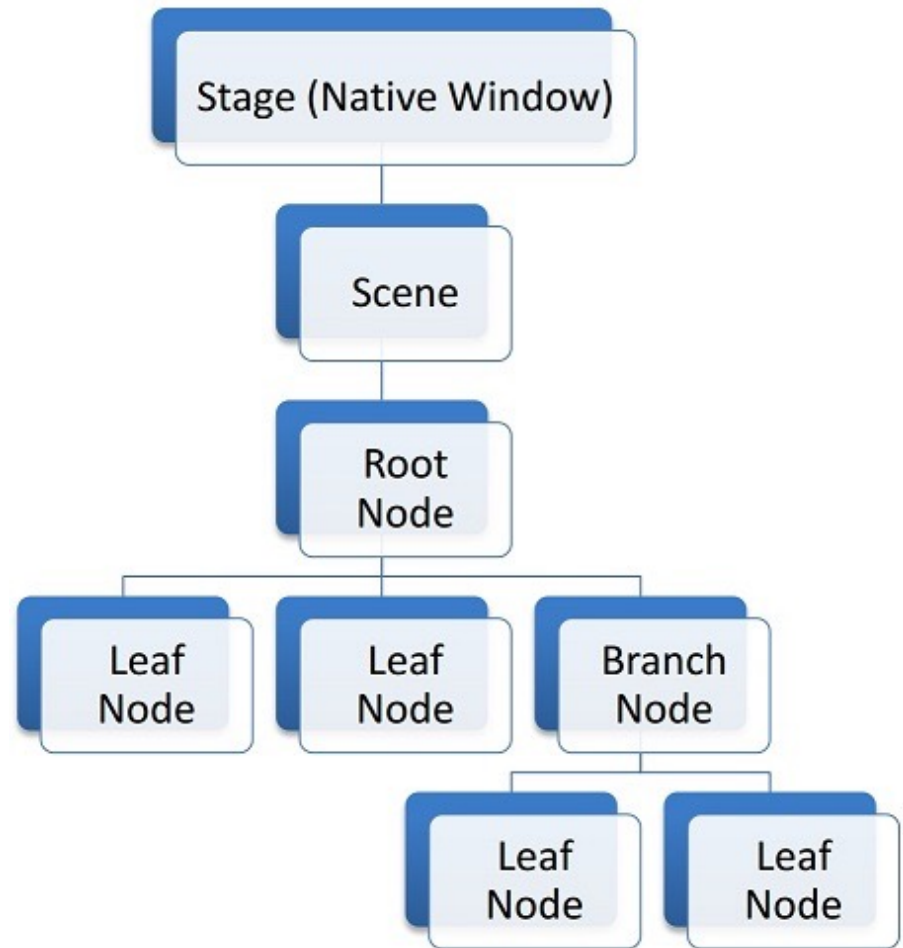
# Scene Graph: Stage, Scene e root node

- 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://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

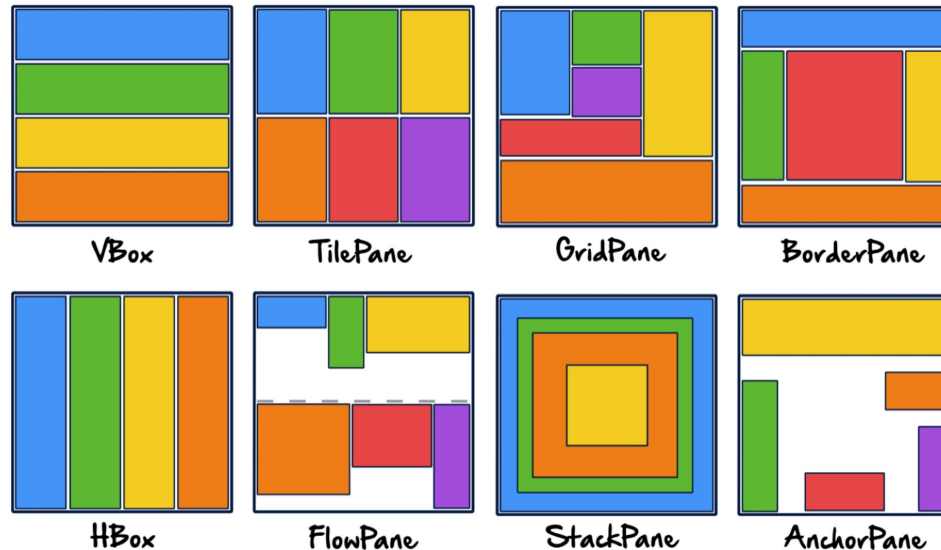
- Region
  - Control
    - TextInputControl
      - **TextArea**, **TextField**
    - ComboBoxBase
      - **ColorPicker**, **ComboBox**, **DatePicker**
    - Labeled
      - ButtonBase
        - **Button**
        - MenuButton
          - SplitMenuButton
        - ToggleButton
          - **RadioButton**
        - **CheckBox**
        - Hyperlink
      - Cell, **Label**, TitledPane
    - Accordion, ButtonBar, **ChoiceBox**, HTML editor, **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 o 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

# 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(); }  
  
    @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 {  
    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 */ }  
}
```