Propriedades e Ligações Dinâmicas em JavaFX

Java Interface

- Na forma mais simplista podemos dizer que uma **interface** é um conjunto de métodos *abstratos* agregados sob uma estrutura.
- Ultrapassa a limitação da herança múltipla em POO.
- Define um protocolo geral de ação/comportamento a ser detalhado e implementado por classes (java class) no futuro. Estas comprometem-se a garantir esse comportamento.
- Amplamente utilizado no tratamento de eventos, nas interfaces gráficas, quer do Swing quer do JavaFX.

Java Interface

```
interface <NomeI> {
    … metodo1(…);
    … metodo2(…);
    … … …
    … metodoN(…);
}
```

Definidos com prefixo opcional "abstract" ou em alternativa "default". Neste último caso teremos de fornecer uma implementação.

```
class <NomeC> implements <NomeI> {
    … … …
    … metodo1(…) {…}
    … metodo2(…) {…}
    … … …
    … metodoN(…) {…}
    … … …
}
```

Utilidade?

Java Interface — Exemplo:

Definidos com prefixo opcional "abstract" ou em alternativa "default". Neste último caso teremos de fornecer uma implementação.

```
public interface FiguraGeometrica
{
    abstract double perimetro();

    abstract double area();


    default double racio() {
        return   perimetro() / area();
    }

    default String getNome() {
        return this.getClass().getSimpleName();
    }

    default boolean maiorQue(FiguraGeometrica other) {
        return this.area() > other.area();
    }
}
```

Java Interface — Exemplo:

```java
public class Circulo implements FiguraGeometrica
{
    private final double raio;


    public Circulo(double raio) {
        if ( raio < 0 )  raio = 0;
        this.raio= raio;
    }

    public double getRaio() {
        return raio;
    }


    @Override
    public double perimetro() {
        return 2.0*Math.PI*raio;
    }

    @Override
    public double area() {
        return Math.PI * raio*raio;
    }
}
```

```java
public class Retangulo implements FiguraGeometrica
{
    private final double comprimento;
    private final double largura;


    public Retangulo(double comprimento, double largura) {
        this.comprimento= comprimento;
        this.largura= largura;
    }

    public double getComprimento() {
        return comprimento;
    }

    public double getLargura() {
        return largura;
    }


    @Override
    public double perimetro() {
        return 2*comprimento + 2*largura;
    }

    @Override
    public double area() {
        return comprimento * largura;
    }
}
```

Java Interface & GUI Events

- No tratamento de eventos de interfaces gráficas, uma component (ex: `Button`) informa um *listener* de algo que sucedeu.
- Um *listener* é a implementação (`implements`) de uma `Java interface`, (ex: `ActionListener`) especificando a resposta ao evento correspondente.
- O *listener* terá de ficar "registado" na component correspondente: `componente.addListener(…)`
- Vejamos a seguir um pequeno exemplo …

Java Interface & GUI Events

Java Swing

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class ButCuriosidade
{
    public static void main(String[] args) {
        JButton myBut= new JButton("Curiosidade");

        myBut.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent ae) {
                System.out.println("A curiosidade é perigosa!");
            }
        });

        JFrame window= new JFrame("Frame Curiosidade (IHC 2015/16)");
        window.getContentPane().add(myBut);
        window.setSize(300, 182);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocationRelativeTo(null);
        window.setVisible(true);
    }
}
```

java.awt.event

Class ActionEvent

java.lang.Object
- java.util.EventObject
  - java.awt.AWTEvent
    - java.awt.event.ActionEvent

java.awt.event

Interface ActionListener

All Superinterfaces:

EventListener

Java Interface & GUI Events

JavaFX

```
public class SimpleEvents extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button myButton = new Button();
        myButton.setText("Say 'Hello World'");
        myButton.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(myButton);
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

javafx.event

Interface EventHandler<T extends Event>

Type Parameters:

T - the event class this handler can handle

All Superinterfaces:

java.util.EventListener

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
```

Java Lambdas

- Nova funcionalidade do Java 8 que implementa aspetos de *programação funcional*.

- Simplifica o código, nomeadamente em situações de *classes anónimas interiores*.

- Permite uma definição dinâmica de funções e sua passagem como argumento de outra função/método.

- Assim, um método pode receber tipos primitivos, objetos e funções (*lambdas*).

Java Lambdas

- Em Java, uma expressão lambda é composta por três partes: (1) a lista de argumentos; (2) o símbolo “->” e (3) o corpo do lambda, por exemplo:

(1)	(2)	(3)
Argument List	**Arrow Token**	**Body**
`(int x, int y)`	`->`	`x + y`

```
(String s) -> { System.out.println(s); }

() -> 47
```

Java Lambdas

Exemplo concreto:

```
public class RunnableTest {

    public static void main(String[] args) {
        System.out.println("=== RunnableTest ===");

        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello world one!");
            }

        };

        Runnable r2 = () -> System.out.println("Hello world two!");

        r1.run();
        r2.run();
    }
}
```

java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>,
RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState,
ForkJoinWorkerThread, FutureTask,
RenderableImageProducer, SwingWorker,
Thread, TimerTask

`public interface` **`Runnable`**

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while

Java Events with Lambdas

Java Swing

```
import java.awt.event.ActionEvent;
import javax.swing.JButton;
import javax.swing.JFrame;

public class ButCuriosidade
{
    public static void main(String[] args) {
        JButton myBut= new JButton("Curiosidade");

        myBut.addActionListener((ActionEvent ae) -> {
            System.out.println("A curiosidade é perigosa!");
        });

        JFrame window= new JFrame("Frame Curiosidade (IHC 2015/16)");
        window.getContentPane().add(myBut);
        window.setSize(300, 182);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocationRelativeTo(null);
        window.setVisible(true);
    }
}
```

Java Events com Lambdas

Java Swing

Functional Interface !

```
public static void main(String[] args) {
    JButton myBut= new JButton("Curiosidade");

    myBut.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent ae) {
            System.out.println("A curiosidade é perigosa!");
        }
    });
```

```
public static void main(String[] args) {
    JButton myBut= new JButton("Curiosidade");

    myBut.addActionListener((ActionEvent ae) -> {
        System.out.println("A curiosidade é perigosa!");
    });
```

Java Events com Lambdas

JavaFX

Functional Interface !

```
Button myButton = new Button();

myButton.setText("Say 'Hello World'");
myButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

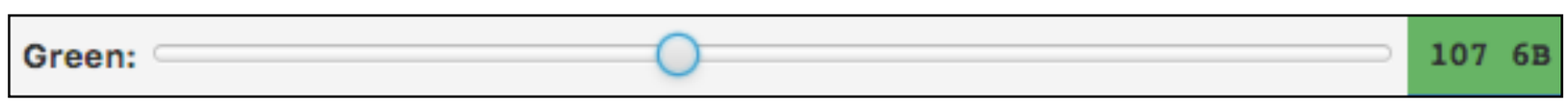
```
Button myButton = new Button();

myButton.setText("Say 'Hello World'");
myButton.setOnAction((ActionEvent event) -> {
    System.out.println("Hello World!");
});
```

Java Events com Lambdas

JavaFX

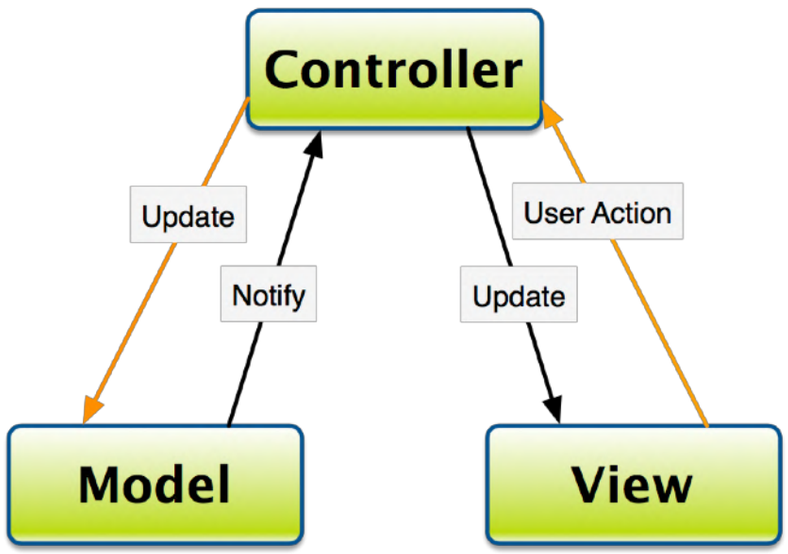
```
@Override
public void initialize(URL url, ResourceBundle rb) {
    slideRed.valueProperty().addListener(new ChangeListener() {
        @Override
        public void changed(ObservableValue observable, Object oldValue, Object newValue) {
            pintar();
        }
    });
    slideGreen.valueProperty().addListener((observable, oldValue, newValue) -> pintar());
    slideBlue.valueProperty ().addListener((observable, oldValue, newValue) -> pintar());
}
```



Slider

Observadores e Observados

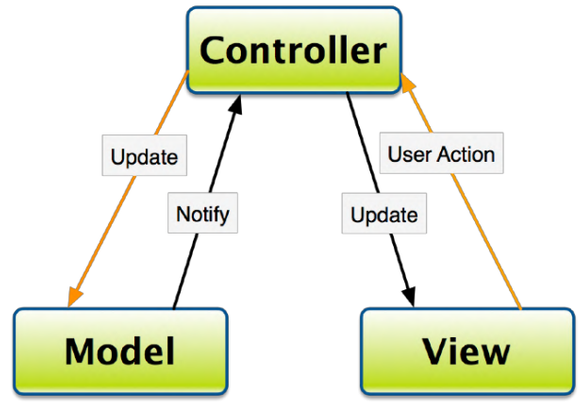
- O Java oferece infraestrutura de suporte ao desenvolvimento **MVC**, ideal para projetos com GUI



- Os pares Observados—Observadores são uma das técnicas de suporte em JavaFX, em particular nas propriedades

Observadores e Observados

- O Java oferece infraestrutura de suporte ao desenvolvimento **MVC**, ideal para projetos com GUI



- Os pares Observados—Observadores são uma das técnicas de suporte em JavaFX, em particular nas propriedades

- Existe um objeto com característica de observado e outro com característica de observador, emparelhado com o primeiro.

- Quando o estado do observado é alterado, o observador é notificado diretamente.

Observadores e Observados

O `Observer` é uma *interface*
e o `Observable` é uma *class*

```
public class Main
{
    public static void main(String[] args) {
        ObservableValue ovalue= new ObservableValue(1);
        ovalue.addObserver(new GeneralObserver());

        ovalue.setValor(2);
        ovalue.setValor(2*ovalue.getValor());
    }
}
```

```
import java.util.Observable;
import java.util.Observer;


public class GeneralObserver implements Observer
{
    @Override
    public void update(Observable o, Object arg) {
        System.out.printf("I saw a %s object changing its state!\n", o.getClass().getName());
        System.out.printf("Argument: %s\n\n", arg);
    }
}
```

```
import java.util.Observable;

public class ObservableValue extends Observable
{
    private int valor= 0;

    public ObservableValue(int n) {
        this.valor= n;
    }

    public void setValor(int n) {
        this.valor= n;
        setChanged();
        notifyObservers();
    }

    public int getValor() {
        return valor;
    }
}
```

Observadores e Observados

```java
public class Main2
{
    public static void main(String[] args) {
        ObservableValue x= new ObservableValue(1);
        Totoloto totA= new Totoloto("Euromilhões ...");
        Totoloto totB= new Totoloto("Lotaria .......");
        DetailedObserver policia= new DetailedObserver();

        x.addObserver(policia);
        totA.addObserver(policia);
        totB.addObserver(policia);

        x.setValor(3457);
        totA.jogar();
        totB.jogar();
    }
}
```

```java
import java.util.Observable;
import java.util.Observer;

public class DetailedObserver implements Observer
{
    @Override
    public void update(Observable ob, Object o) {
        if ( ob.getClass().getName().equals("Totoloto") ) {
            Totoloto loto= (Totoloto)o;
            System.out.printf("Saw Totoloto %s with value %d\n",
                    loto.getNome(),
                    loto.getNumero()
            );
        }
    }
}
```

```java
public class Totoloto extends Observable
{
    private final String nome;
    private final Random sorte;
    private int numero;

    public Totoloto(String nome) {
        this.nome= nome;
        this.numero= 0;
        this.sorte= new Random();
    }

    public void jogar() {
        numero= sorte.nextInt(49) + 1;
        setChanged();
        notifyObservers(this);
    }

    public String getNome() {
        return nome;
    }

    public int getNumero() {
        return numero;
    }
}
```

Propriedades e ligações (bindings)

- Baseado nos princípios anteriores, a **JavaFX** disponibiliza os conceitos de “propriedade observável” e “ligação entre propriedades”
- Elementos disponíveis nas packages:

```
javafx.beans.property.*
javafx.beans.binding.*
```

- Assim, podemos criar dependências automáticas entre as propriedades de diferentes objetos.
- É introduzida uma **nova convenção de nomes**, em especial para a definição de métodos de acesso às propriedades de objetos.

Propriedades — Exemplo:

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

class Fatura
{
    private DoubleProperty valor = new SimpleDoubleProperty(0);

    public final double getValor() {
        return valor.get();
    }
    public final void setValor(double value) {
        valor.set(value);
    }

    public DoubleProperty valorProperty() {
        return valor;
    }
}
```

Novidade!

Propriedades — Exemplo:

```
import javafx.beans.value.ObservableValue;
import javafx.beans.value.ChangeListener;

public class MainFatura
{
    public static void main(String[] args) {
        Fatura electricidade = new Fatura();

        electricidade.valorProperty().addListener(new ChangeListener() {
            @Override
            public void changed(ObservableValue o, Object oldV, Object newV) {
                System.out.println("A fatura elétrica mudou!");
                System.out.println("Tipologia ..... "+o.getClass().getName());
                System.out.println("Valor antigo:... "+oldV);
                System.out.println("Valor novo:..... "+newV);
            }
        });

        electricidade.setValor(100.00);
    }
}
```

Ligações entre Propriedades

```
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;


public class BindingExample
{
    public static void main(String[] args) {
        IntegerProperty a = new SimpleIntegerProperty(0);
        IntegerProperty b = new SimpleIntegerProperty(0);

        a.bind(b);
        b.set(4);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());

        a.unbind();
        a.set(3);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());

        a.bindBidirectional(b);
        b.set(7);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());

        a.set(9);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());
    }
}
```

Ligações entre Propriedades

```
import javafx.beans.binding.Bindings;
import javafx.beans.binding.NumberBinding;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;


public class MainSumBinding
{
public static void main(String[] args) {
        IntegerProperty a = new SimpleIntegerProperty(1);
        IntegerProperty b = new SimpleIntegerProperty(2);
        IntegerProperty c = new SimpleIntegerProperty(3);

        NumberBinding sum = a.add(b);
        NumberBinding som = Bindings.add(sum,c).multiply(3.75);
        System.out.println("Sum is: "+sum.getValue());
        System.out.println("Som is: "+som.getValue());
        a.set(3);
        b.set(-7);
        c.set(5);
        System.out.println("Sum is: "+sum.getValue());
        System.out.println("Som is: "+som.getValue());
    }
}
```

JavaFX: TableView e TableColumn

- Em muitas aplicações é necessário recorrer à apresentação de dados em tabelas.

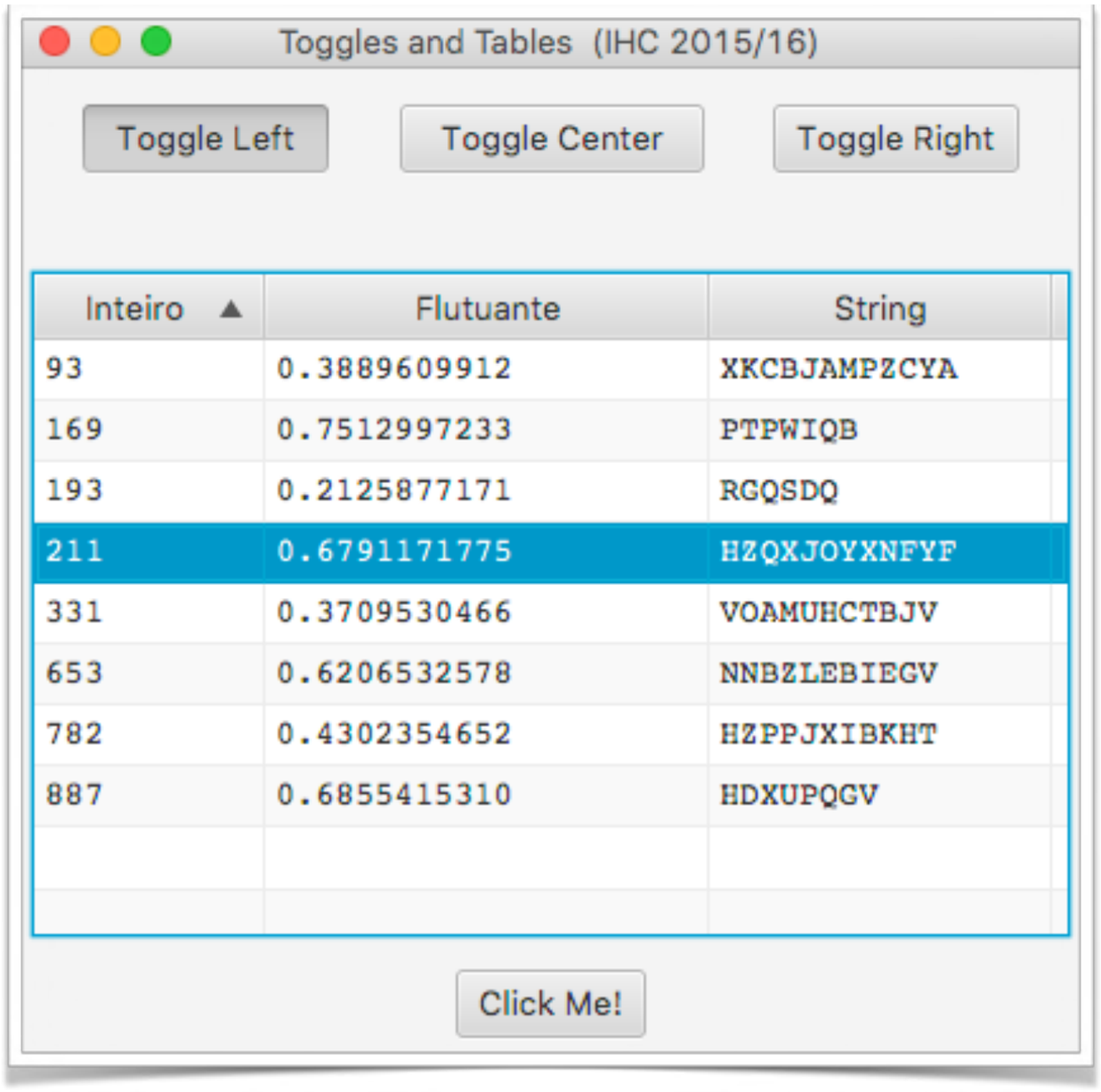
- No JavaFX temos o TableView que agrega vários TableColumn.

- Podem ser incorporados como qualquer outra componente, do Scene Builder.

- Dada a sua natureza exigem, no entanto, um modelo de dados representativo da informação apresentada na tabela.

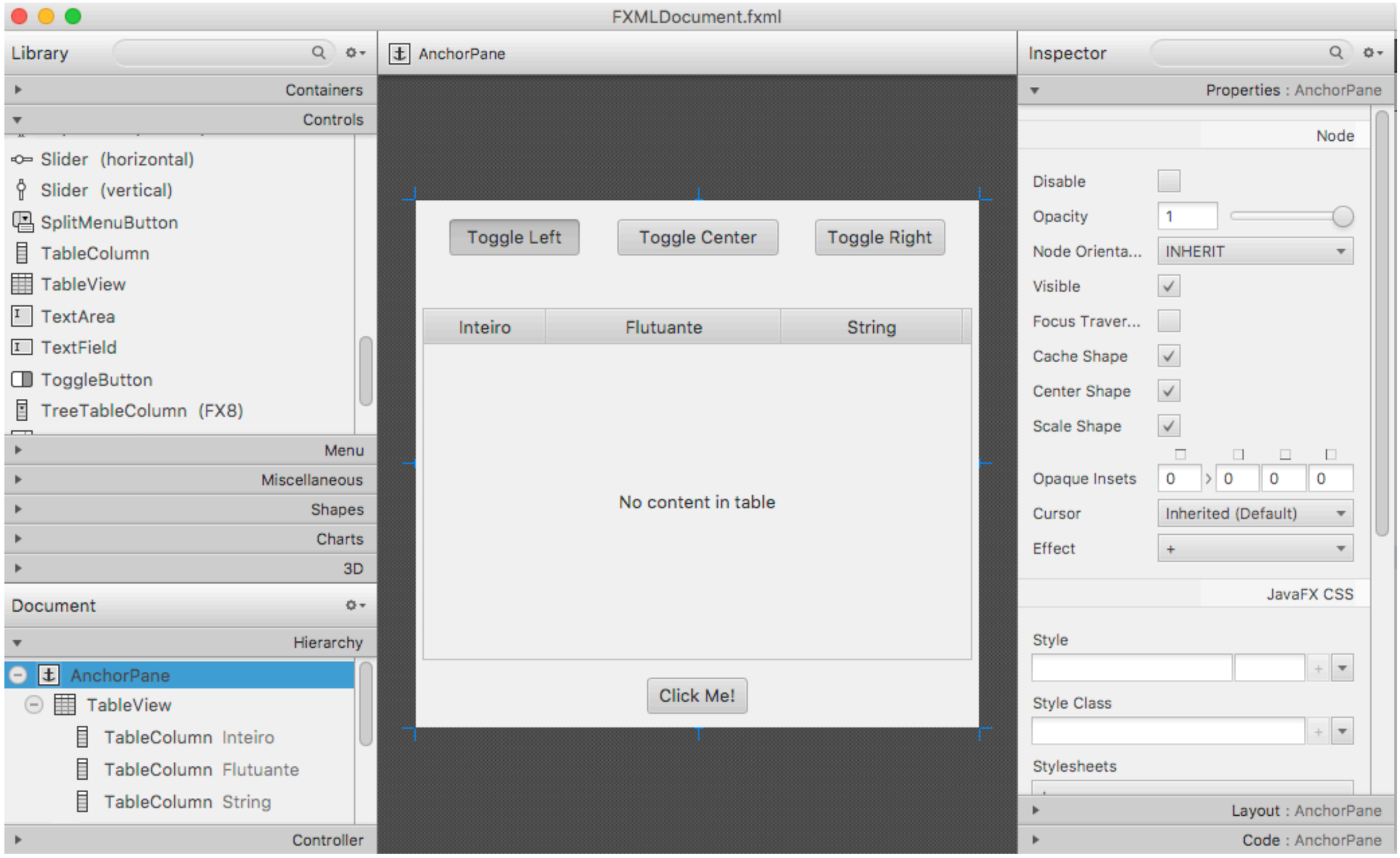
JavaFX: TableView e TableColumn

- Exemplo simples:



JavaFX: TableView e TableColumn

- O design da View, no Scene Builder



JavaFX: TableView e TableColumn

- O modelo de dados da tabela, no Controller:

```
public class FXMLDocumentController implements Initializable
{
    @FXML private Button button;

    @FXML private ToggleButton tgbLeft;
    @FXML private ToggleButton tgbCenter;
    @FXML private ToggleButton tgbRight;
    @FXML private ToggleGroup group;

    @FXML private TableView table;


    @FXML
    private void handleButtonAction(ActionEvent event) {
        Random r= new Random();
        ObservableList<Dado> data = FXCollections.observableArrayList(
                new Dado(r), new Dado(r), new Dado(r), new Dado(r),
                new Dado(r), new Dado(r), new Dado(r), new Dado(r)
        );
        ObservableList<TableColumn> colunas= table.getColumns();
        for (TableColumn c : colunas) {
            c.setCellValueFactory(new PropertyValueFactory<>(c.getText()));
            setSomeCellDefinitions(c);
        }
        table.setItems(data);
    }
```

```
public interface ObservableList<E>
extends List<E>, Observable

A list that allows listeners to track changes when they occur.

Since:
JavaFX 2.0

See Also:
ListChangeListener, ListChangeListener.Change
```

JavaFX: TableView e TableColumn

- A classe “Dado” representa uma linha de dados:

```
public class Dado {
    private int inteiro;
    private double flutuante;
    private String string;

    public Dado(Random r) {
        this.inteiro= 1+r.nextInt(1000);
        this.flutuante= r.nextDouble();
        this.string= randomString(r);
    }

    public Dado(int inteiro, double flutuante, String string) {
        this.inteiro = inteiro;
        this.flutuante = flutuante;
        this.string = string;
    }

    public int getInteiro() {
        return inteiro;
    }

    public double getFlutuante() {
        return flutuante;
    }

    public String getString() {
        return string;
    }
```

```
    public String getString() {
        return string;
    }

    public void setInteiro(int inteiro) {
        this.inteiro = inteiro;
    }

    public void setFlutuante(double flutuante) {
        this.flutuante = flutuante;
    }

    public void setString(String string) {
        this.string = string;
    }


    public static String randomString(Random r) {
        int n= 5 + r.nextInt(10);
        StringBuilder sb= new StringBuilder();
        for (int i = 0; i < n; i++) {
            char ci= (char)((short)'A' + r.nextInt(26));
            sb.append(ci);
        }
        return sb.toString();
    }
```

JavaFX: TableView e TableColumn

- Formatações adicionais nas células:

```
private void setSomeCellDefinitions(TableColumn c) {
    c.setCellFactory(new Callback<TableColumn, TableCell>() {
        @Override
        public TableCell call(TableColumn param) {
            TableCell cell = new TableCell() {
                @Override
                public void updateItem(Object item, boolean empty) {
                    if (item != null) {
                        /**/
                        String s= item.toString(); if ( s.length() > 12 ) s= s.substring(0, 12);
                        /**/
                        setText(s);
                        setStyle("-fx-font-family: Courier");
                    }
                }
            };
            if (tgbLeft.isSelected()) {
                cell.setAlignment(Pos.TOP_LEFT);
            } else if (tgbCenter.isSelected()) {
                cell.setAlignment(Pos.TOP_CENTER);
            } else {
                cell.setAlignment(Pos.TOP_RIGHT);
            }
            return cell;

        }
    });
}
```