

# Metode avansate de programare

---

## Curs 6 GUI JavaFX



*"I hear and I forget, I see and I remember, I do and I understand."*

*- Confucius*

# Cuprins

A decorative element on the left side of the slide, consisting of three vertical lines in red, green, and blue, and a horizontal teal bar.

- Ce este JavaFX
- Graful de scene
- Lucrul cu componentele grafice
- Gestionarea poziționării
- Tratarea evenimentelor

# Ce este JavaFX ?

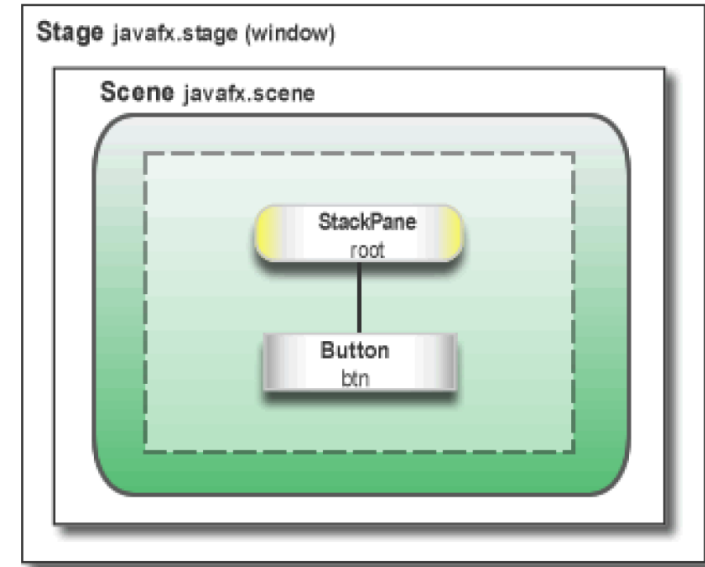
- **Clase si interfete** care asigura suport pentru crearea de aplicatii Java care se pot **proiecta, implementa, testa** pe diferite **platforme**.
- Asigura suport pentru utilizarea de componente Web cum ar fi apeluri de scripturi **JavaScript** sau cod **HTML5**
- Contine componente grafice UI pentru crearea de interfete grafice si gestionarea aspectului lor prin fisiere **CSS**
- Asigura suport pentru grafica interactiva 3D
- Asigur suport pentru manipulare de continut multimedia
- Portabilitate: desktop, browser, dispozitive mobile, TV, console jocuri, Blu-ray, etc.
- Asigura interoperabilitate Swing

# JavaFX APIs -Scene Graph

## scene-graph-based programming model

O aplicatie JavaFX conține:

- un obiect Stage (fereastră)
- unul sau mai multe obiecte Scene



**Graful de scene**(Scene Graph) este o structură arborescentă de componente grafice ale interfeței utilizator.

Un element din **graful de scene** este un **Node**.

- Fiecare nod are un **id**, un **stil grafic** asociat și o **suprafață ocupată** (*ID, style class, bounding volume, etc.*)
- Cu excepția nodului rădăcină, fiecare nod are un singur părinte și 0 sau mai mulți fii.
- Un nod mai poate avea asociate diverse proprietăți (efecte (blur, shadow), opacitate, transformari) și evenimente (*event handlers* (mouse, tastatură))
- Nodurile pot fi interne (Parent) sau frunza

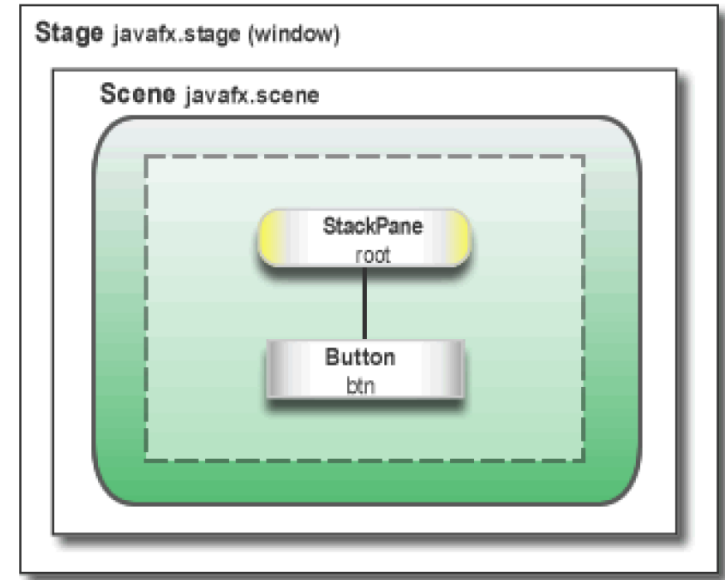
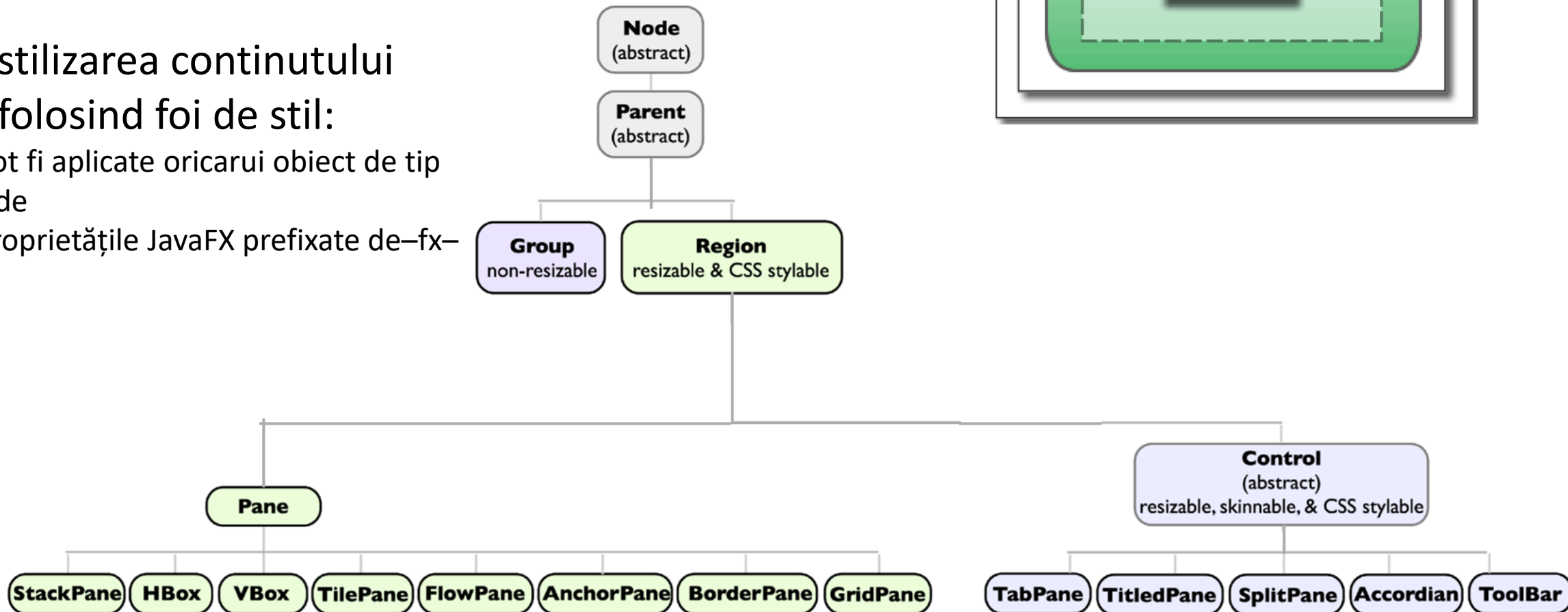
# Arhitectura JavaFX

## ■ controale

- definite in pachetul `javafx.scene.control`
- pot fi grupate in containere / panouri

## ■ stilizarea continutului folosind foi de stil:

- pot fi aplicate oricarui obiect de tip Node
- proprietățile JavaFX prefixate de `-fx-`



# Aplicații java FX

- O aplicație JavaFX este o instanță a clasei **Application**  
**public abstract class** Application **extends** Object;
- Instantierea unui obiect Application se face prin executarea metodei statice *launch()*  
**public static void** launch(String... args);  
args **parametrii aplicației**(parametrii metodei *main*).
- JavaFX runtime execută următoarele operațiuni:
  1. Creează un obiect Application
  2. Apelează metoda **init** a obiectului Application
  3. Apelează metoda **start** a obiectului Application
  4. Așteaptă sfârșitul aplicației
- Parametrii aplicației sunt obținuți prin metoda *getParameters()*

# Scheletul unei aplicatii JavaFX

```
public class Main extends Application {  
    @Override  
    public void start(Stage stage) {  
        Parent root= initRoot();  
        Scene scene = new Scene(root, 550, 500);  
        stage.setTitle("Welcome to JavaFX!!");  
        stage.setScene(scene);  
        stage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

# Exemplu 1 Group

```
public class Main extends Application {

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.PINK);
        stage.setTitle("Welcome to JavaFX!");

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

    public static void main(String[] args) {
        launch(args); //se creaza un obiect de tip Application
    }
}
```



# Adăugarea nodurilor

*// Cream un nod de tip Group*

```
Group group = new Group();
```

*// Cream un nod de tip Rectangle*

```
Rectangle r = new Rectangle(25,25,50,50);
```

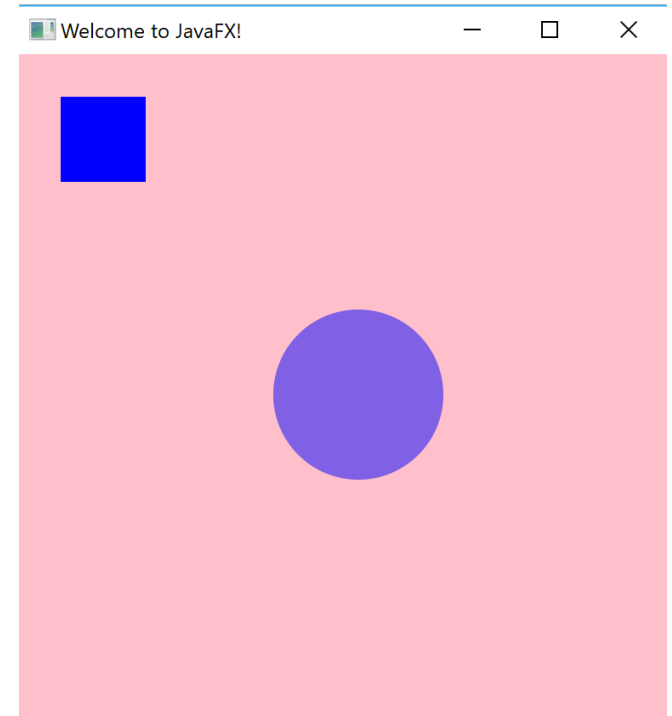
```
r.setFill(Color.BLUE);
```

```
group.getChildren().add(r);
```

*// Cream un nod de tip Circle*

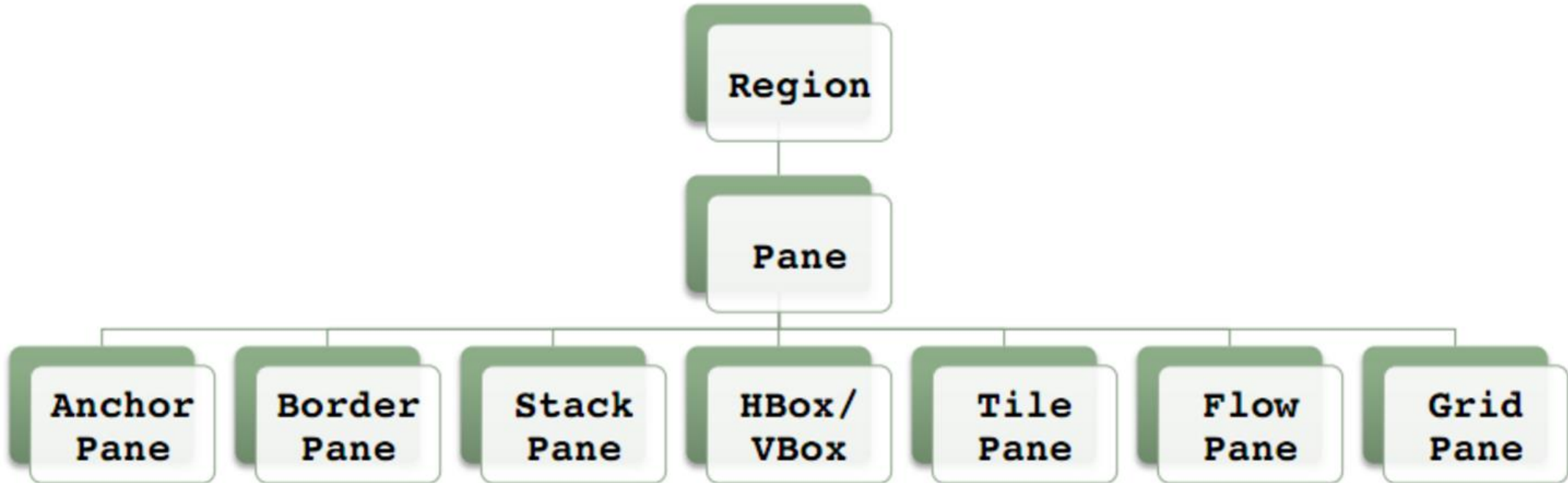
```
Circle c = new Circle(200,200,50, Color.web("blue", 0.5f));
```

```
group.getChildren().add(c);
```



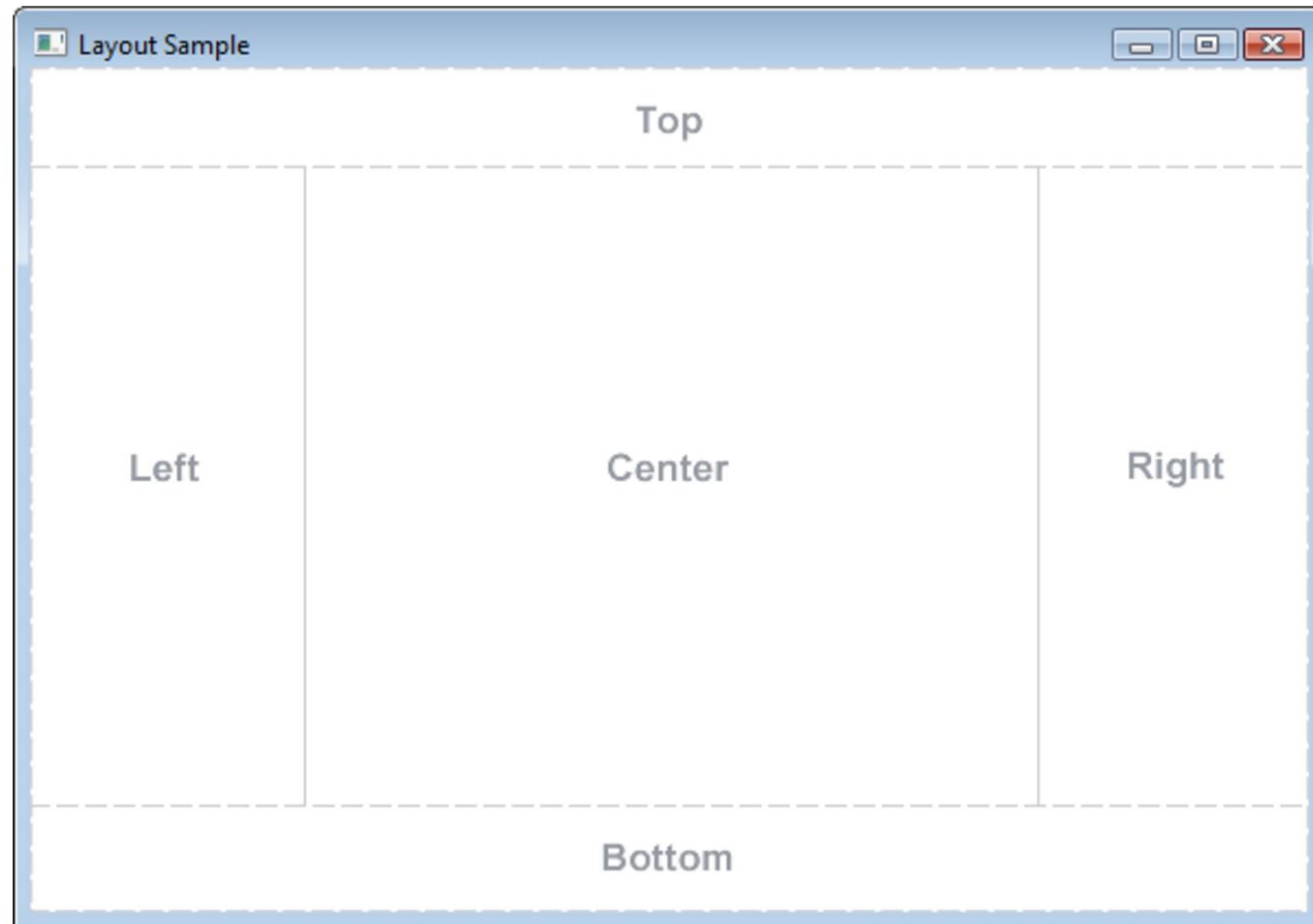
# Gestionarea poziționării componentelor UI

Componete de poziționare – containere de tip Panou (Pane)



# Gestionarea poziționării componentelor UI

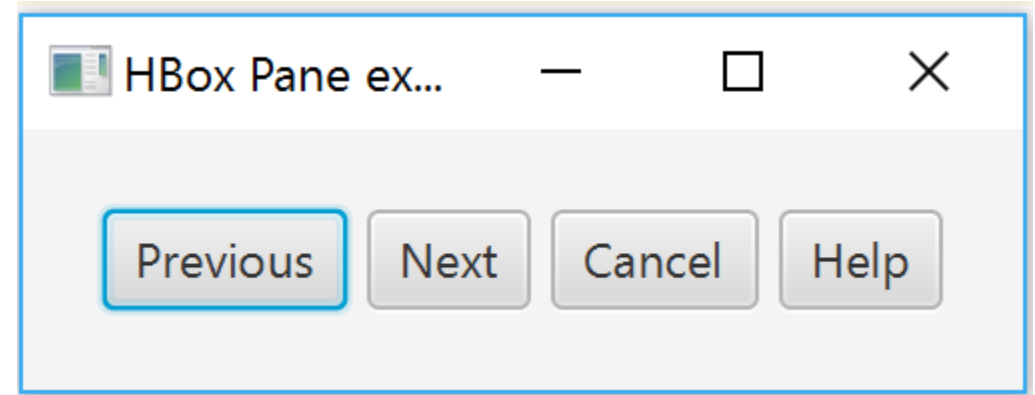
- **BorderPane**



# Gestionarea poziționării componentelor UI

## ■ HBOX

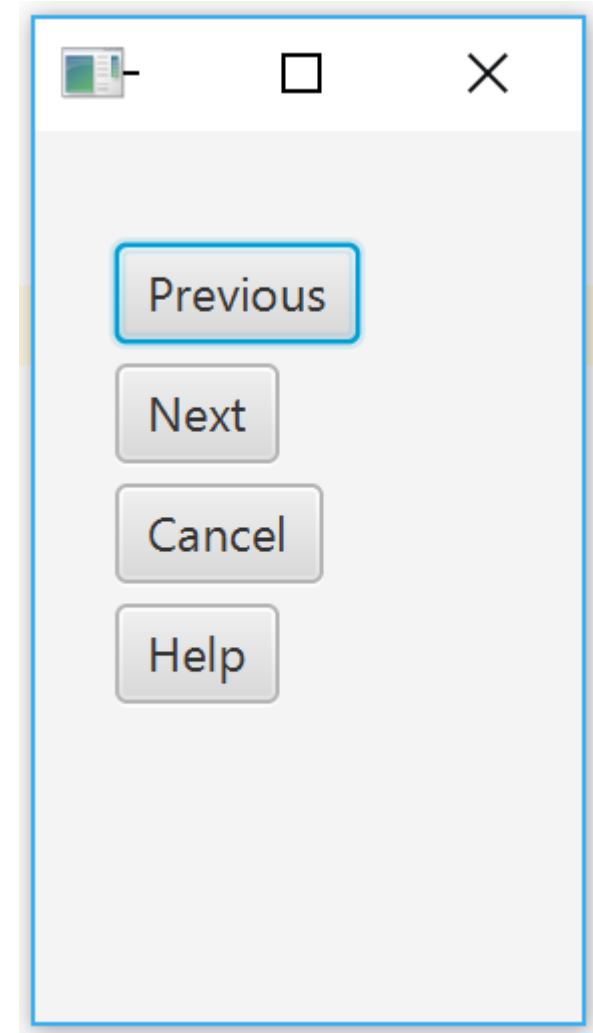
```
HBox root = new HBox(5);  
root.setPadding(new Insets(100));  
root.setAlignment(Pos.BASELINE_RIGHT);  
  
Button prevBtn = new Button("Previous");  
Button nextBtn = new Button("Next");  
Button cancBtn = new Button("Cancel");  
Button helpBtn = new Button("Help");  
  
root.getChildren().addAll(prevBtn, nextBtn,  
cancBtn, helpBtn);
```



# Gestionarea poziționării componentelor UI

- **VBOX**

```
VBox root = new VBox(5);  
root.setPadding(new Insets(20));  
root.setAlignment(Pos.BASELINE_LEFT);  
  
Button prevBtn = new Button("Previous");  
Button nextBtn = new Button("Next");  
Button cancBtn = new Button("Cancel");  
Button helpBtn = new Button("Help");  
  
root.getChildren().addAll(prevBtn, nextBtn,  
cancBtn, helpBtn);  
Scene scene = new Scene(root, 150, 200);
```



# Gestionarea poziționării componentelor UI

## ■ AnchorPane

```
AnchorPane root = new AnchorPane();

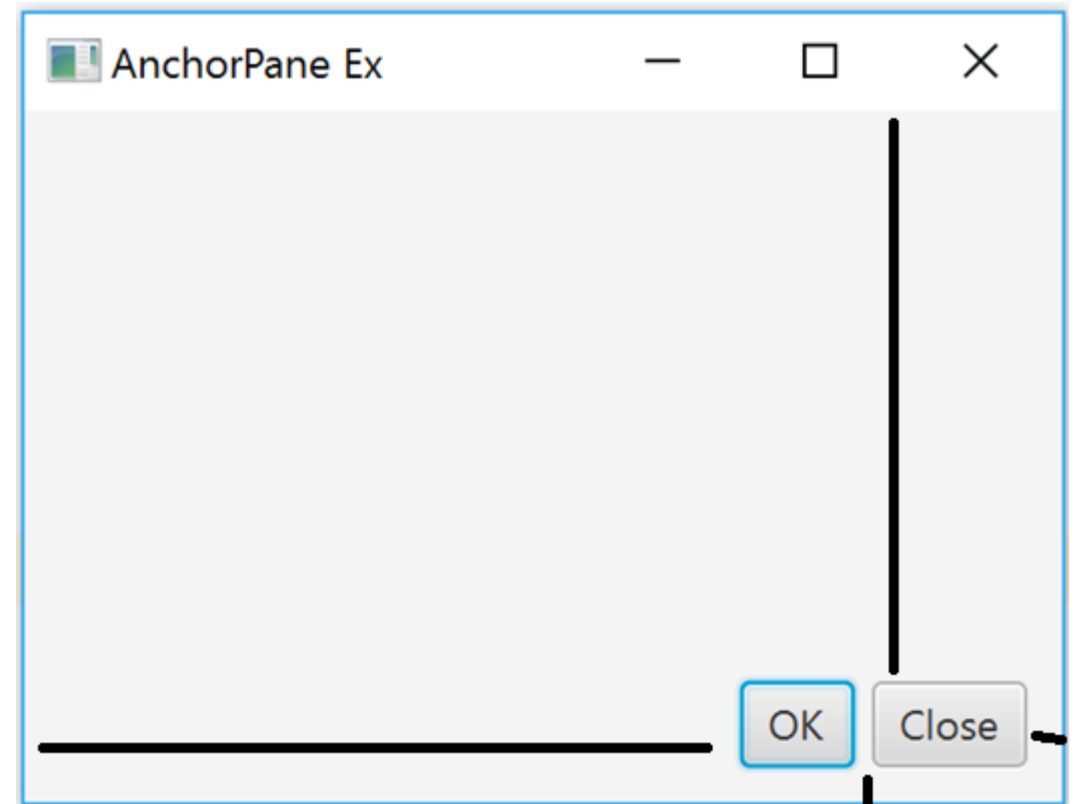
Button okBtn = new Button("OK");
Button closeBtn = new Button("Close");
HBox hbox = new HBox(5, okBtn, closeBtn);

root.getChildren().addAll(hbox);

AnchorPane.setRightAnchor(hbox, 10d);
AnchorPane.setBottomAnchor(hbox, 10d);

Scene scene = new Scene(root, 300, 200);

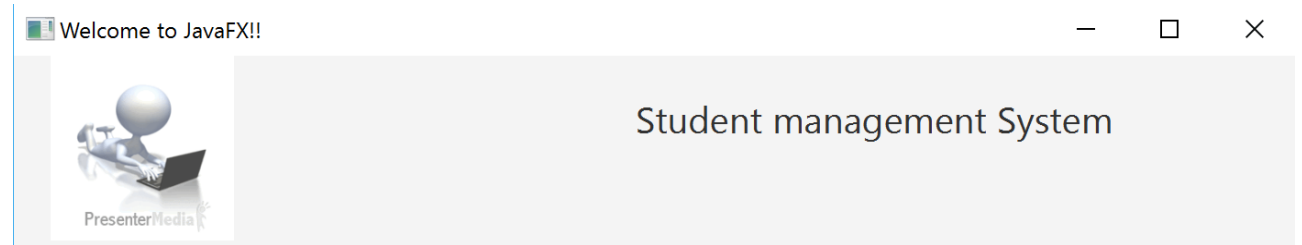
stage.setTitle("AnchorPane Ex");
stage.setScene(scene);
stage.show();
```



# Gestionarea pozitionarii componentelor UI

## AnchorPane StudentView1.java Example

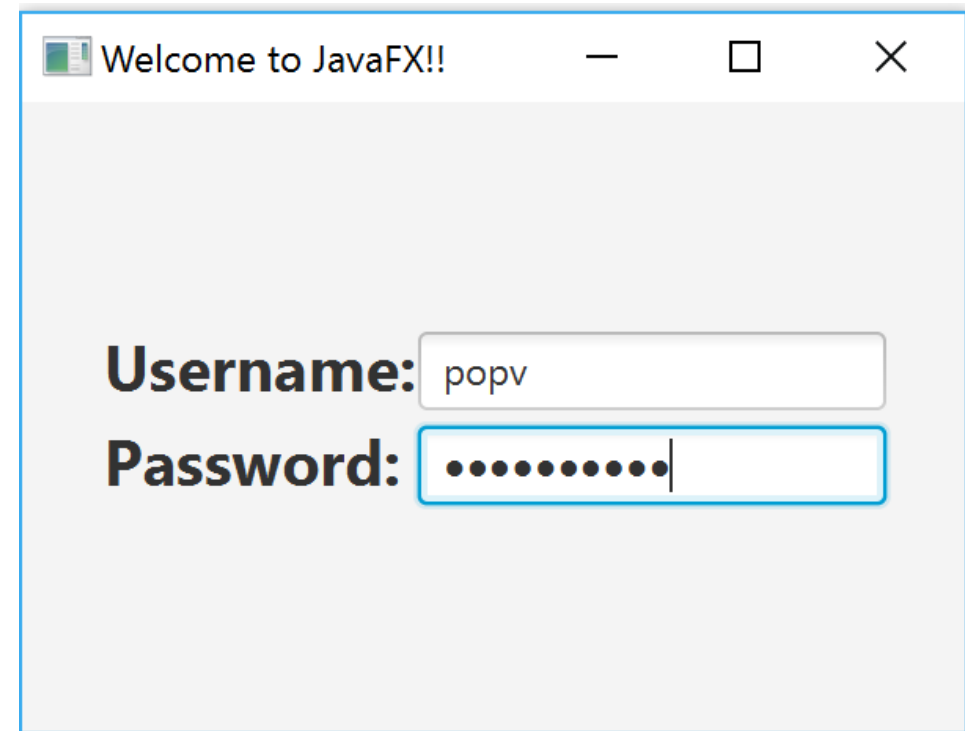
```
private Node initTop() {  
    AnchorPane anchorPane=new AnchorPane();  
  
    Label l=new Label("Student management System");  
    l.setFont(new Font(20));  
  
    AnchorPane.setTopAnchor(l,20d);  
    AnchorPane.setRightAnchor(l,100d);  
    anchorPane.getChildren().add(l);  
  
    Image img = new Image("logo.gif");  
    ImageView imgView = new ImageView(img);  
    imgView.setFitHeight(100);  
    imgView.setFitWidth(100);  
    imgView.setPreserveRatio(true);  
  
    AnchorPane.setLeftAnchor(imgView,20d);  
    AnchorPane.setRightAnchor(imgView,10d);  
    anchorPane.getChildren().add(imgView);  
  
    return anchorPane;  
}
```



# Gestionarea pozitionarii componentelor UI

## ■ GridPane

```
GridPane gr=new GridPane();  
gr.setPadding(new Insets(20));  
gr.setAlignment(Pos.CENTER);  
  
gr.add(createLabel("Username:"),0,0);  
gr.add(createLabel("Password:"),0,1);  
  
gr.add(new TextField(),1,0);  
gr.add(new PasswordField(),1,1);  
  
Scene scene = new Scene(gr, 300, 200);  
stage.setTitle("Welcome to JavaFX!!");  
stage.setScene(scene);  
stage.show();
```





# Componente grafice de control - CGC

- Componentele grafice de control – elemente de bază ale unei aplicații cu interfața grafică utilizator.
- O componentă grafică de control este un nod în graful scenei
- CGC-urile pot fi manipulate de către un utilizator.
- Java FX Controls: [https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui\\_controls.htm#JFXUI336](https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm#JFXUI336)

[Label](#)

[Button](#)

[Radio Button](#)

[Toggle Button](#)

[Checkbox](#)

[Choice Box](#)

[Text Field](#)

[Password Field](#)

[Scroll Bar](#)

[Scroll Pane](#)

[List View](#)

[Table View](#)

[Tree View](#)

[Combo Box](#)

[Separator](#)

[Slider](#)

[Progress Bar and Progress Indicator](#)

[Hyperlink](#)

[Tooltip](#)

[HTML Editor](#)

[Titled Pane and](#)

[Accordion](#)

[Menu](#)

[Color Picker](#)

[Pagination Control](#)

[File Chooser](#)

[Customization of UI](#)

[Controls](#)

# Șablonul Observer (In brief, Observer Pattern = publisher + subscriber.)

- Șablonul *Observer* definește o relație de dependență 1 la n între obiecte:
  - când un obiect își schimbă starea, toți dependenții lui sunt **notificați** și **actualizați automat**.
- Roluri obiecte: *subiect(observat)* și *observator*
- **Utilitate**: mai multe clase(*observatori*) depind de comportamentul unei alte clase(*subiect*), în situații de tipul:
  - o clasă implementează/reprezintă logica, componenta de bază, iar alte clase doar folosesc rezultate ale acesteia (monitorizare).
  - o clasă efectuează acțiuni care apoi pot fi reprezentate în mai multe feluri de către alte clase (view-uri )
  - Practic în toate aceste situații clasele Observer **observă** modificările/acțiunile clasei Subject. Observarea se implementează prin **notificări inițiate din metodele clasei Subject**.

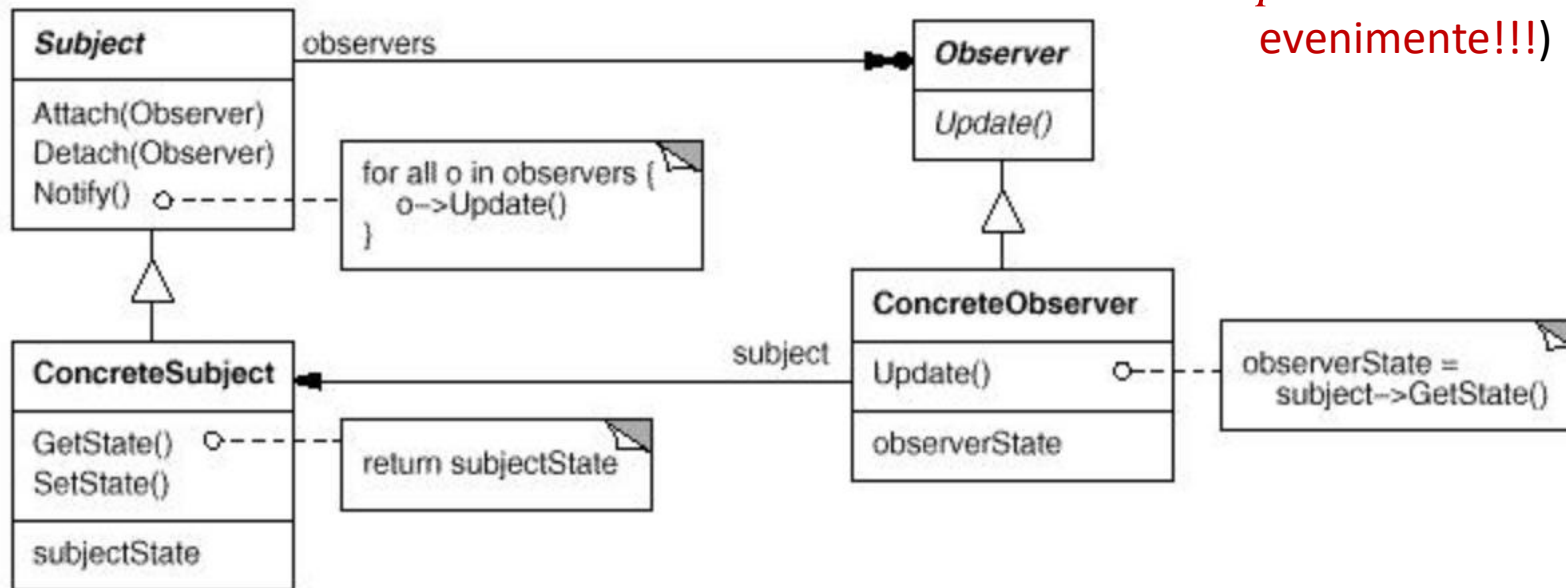
# Șablonul Observer continuare

## Subiect:

- menține o listă de referințe cu observatori fără să știe ce fac observatorii cu datele
- oferă metode de înregistrare/deînregistrare a unui *Observer*
- când apar modificări (e.g. se schimbă starea sa, valorile unor variabile etc) **notifică toți observatorii**

## Observator:

- definește o interfață *Observer* despre schimbări în subiect
- toți observatorii pentru un anumit subiect trebuie să implementeze această interfață
- oferă una sau mai multe metode care să poată fi invocate de către *Subiect* pentru a notifica o schimbare. Ca argumente se poate primi chiar instanța subiectului sau *obiecte speciale care reprezintă evenimentul ce a provocat schimbarea*. (Vezi exemplu seminar cu evenimente!!!)



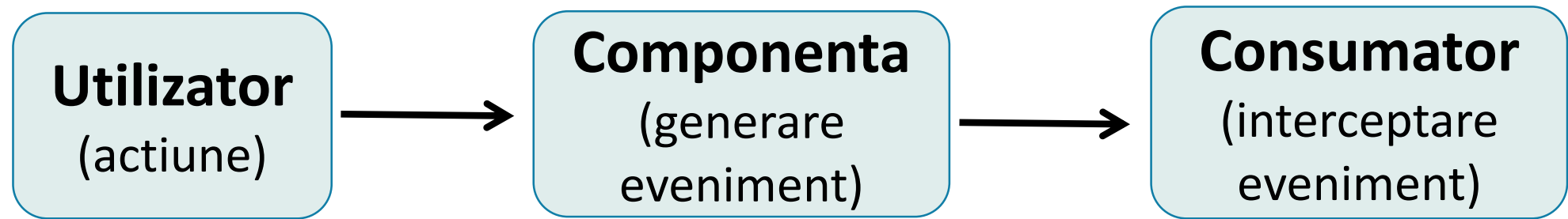
# Event Driven Programming

- **Eveniment:** Orice acțiune efectuată de utilizator generează un eveniment
  - apasarea sau eliberarea unei taste - de la tastatură,
  - deplasarea mouse-ului,
  - apăsarea sau eliberarea unui buton de mouse,
  - deschiderea sau închiderea unei ferestre,
  - efectuarea unui clic de mouse pe o componentă din interfață,
  - intrarea/părăsirea cursorului de mouse în zona unei componente, etc.).
- Există și evenimente care nu sunt generate de utilizatorul aplicației.
- **Un eveniment poate să fie tratat prin execuția unui modul de program.**

# Tratarea evenimentelor - Delegation Event Model.

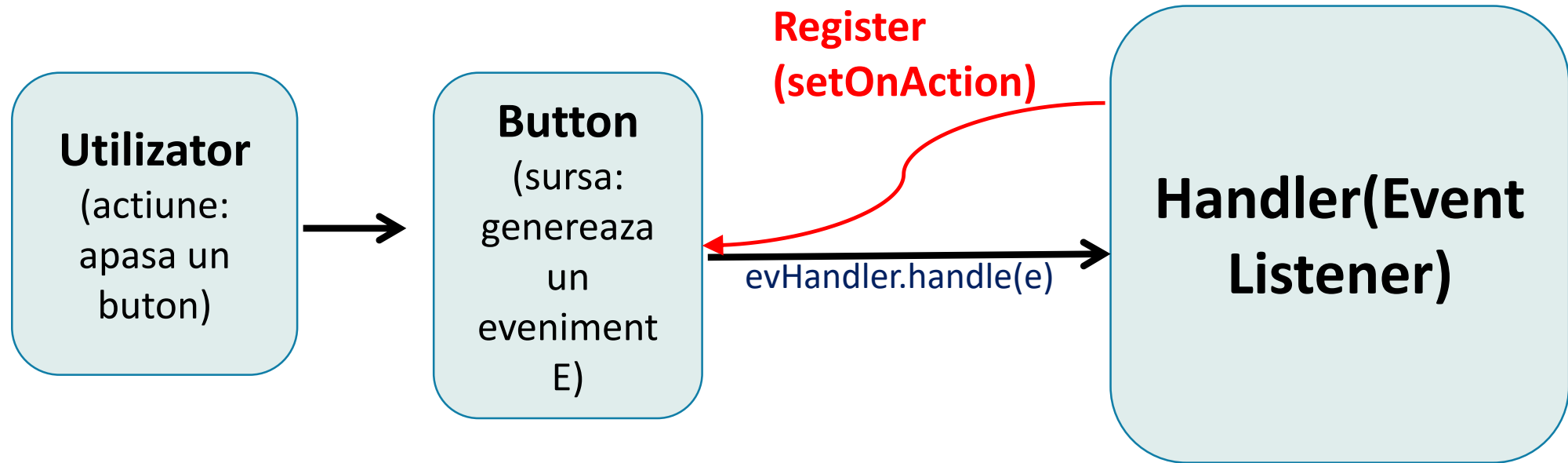
- Distingem trei categorii de obiecte utilizate la tratarea evenimentelor:
  - **surse de evenimente (Event Source)** - acele obiecte care generează evenimente;
  - **evenimentele propriu-zise (Event)**, care sunt tot obiecte (generate de surse și recepționate de consumatori).
  - **consumatori sau ascultători de evenimente** - acele obiecte care recepționează și tratează evenimentele.

# Tratarea evenimentelor

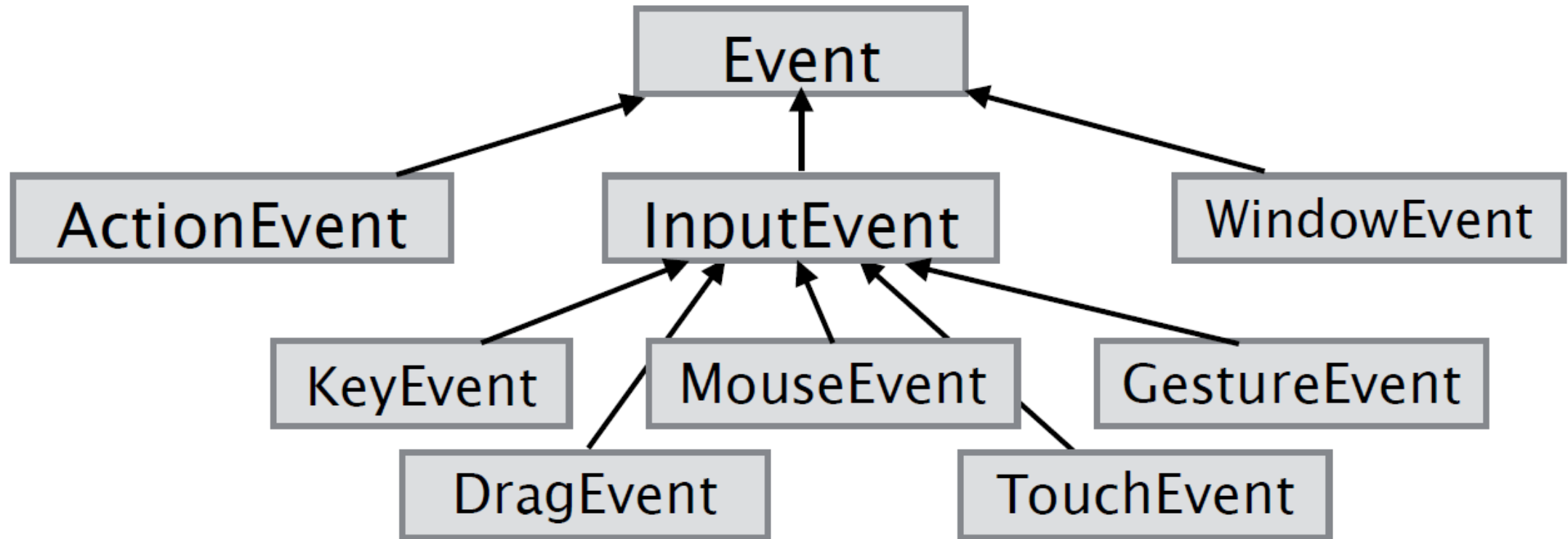


- Fiecare consumator trebuie să fie înregistrat la sursa de eveniment. Prin acest procedeu se asigură că sursa cunoaște toți consumatorii la care trebuie să transmită evenimentele pe care le generează.
- **Modelul "delegarii"** presupune că sursa (un obiect) transmite evenimentele generate de ea către obiectele consumatori, care s-au înregistrat la sursa respectiva a evenimentului.
- Un obiect consumator recepționează evenimente numai de la obiectele sursă la care s-a înregistrat!!!

# Tratarea evenimentelor



# Tipuri de evenimente



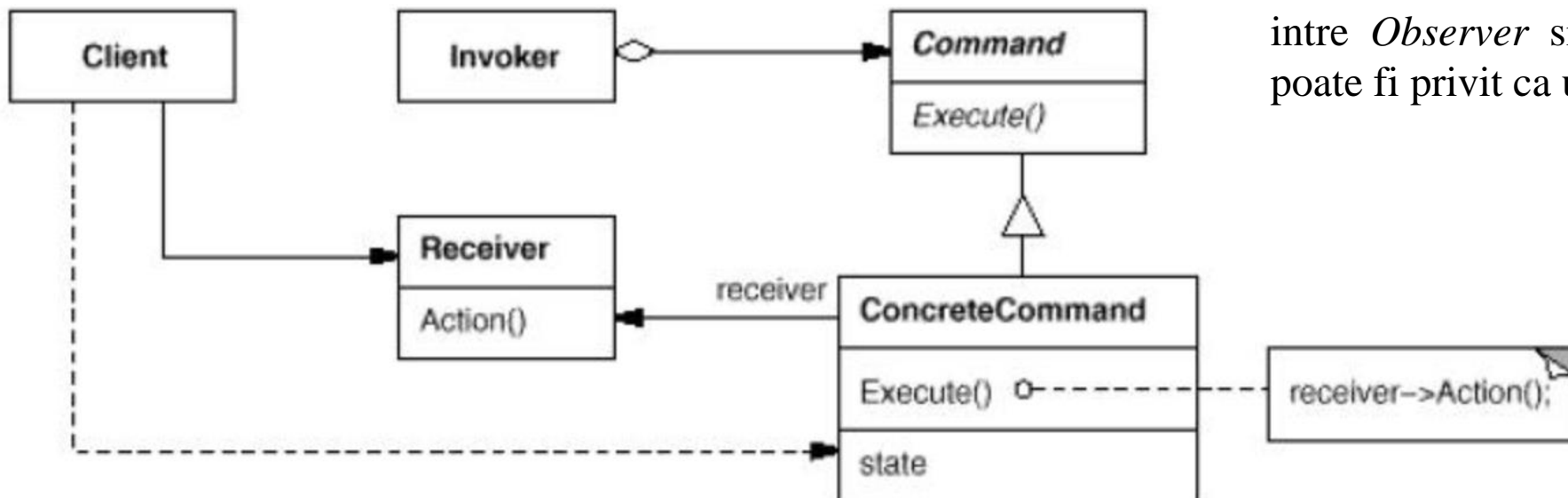


# Șablonul command

- Când se folosește: atunci când dorim să încapsulăm o comandă într-un obiect
- *Utilitate:*
  - **decuplarea** între entitatea care dispune executarea comenzii și entitatea care o execută. Efectul unei comenzi poate fi schimbat dinamic. Decuplarea este necesară pentru a permite, spre exemplu, implementarea clasei buton, în absența informațiilor despre efectul apăsării lui. De asemenea, se permite asocierea aceluiași efect cu 2 surse diferite (element de meniu și buton de pe toolbar, cu efect de *paste*).
  - posibilitatea de a **întârzia** executarea unei comenzi, până la momentul potrivit. Exemplu: caching, pentru evitarea generării prea multor comenzi intermediare, și rularea mai multor comenzi deodată.
  - **memorarea** efectului unei comenzi, pentru a permite **revenirea** la o stare anterioară (undo/redo)
  - posibilitatea definirii de comenzi **compuse** (macro-uri), care grupează alte comenzi

# Șablonul command

- *Command*
  - obiectul comanda
- *ConcreteCommand*
  - implementarea particulara a comenzii
  - apeleaza metode ale obiectului receptor
- *Invoker*
  - declanseaza comanda
- Receiver*
  - realizeaza, efectiv, operatiile aferente comenzii generate
- Client*
  - defineste obiectul comanda si efectul ei



**Observatie:** Nu exista o delimitare clara intre *Observer* si *Command*. Un observator poate fi privit ca un obiect comanda.

# Tratarea evenimentelor - event handler

```
@FunctionalInterface  
Interface EventHandler<T extends Event> extends EventListener{  
    void handle(T event);  
}
```

## Tratarea evenimentului click pe buton

```
Button btn = new Button("Ding!");
```

```
// handle the button clicked event
```

```
btn.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) {  
        Toolkit.getDefaultToolkit().beep();  
    }  
});
```

Se poate asocia o singura metoda handler evenimentului click pe buton!!!

Ce sablon de proiectare este folosit?

```
btn.setOnAction(e->Toolkit.getDefaultToolkit().beep());
```

# Comparatie cu JButton - Java Swing

- Ascultători de evenimente (listener)

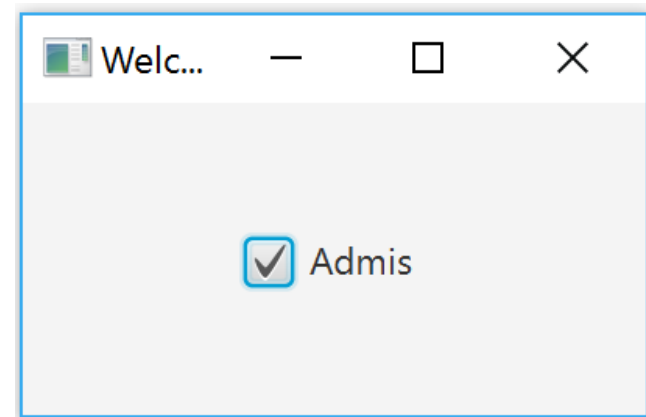
```
JButton b = new JButton("Click me!"); // (Java Swing)
b.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(java.awt.event.ActionEvent e) {
        System.out.println("listener1");
    }
});

b.addActionListener(e -> System.out.println("listener2"));
```

# CheckBox Events

```
CheckBox myCheckBox=new CheckBox("Admis ");
StackPane st=new StackPane(myCheckBox);

// Handle CheckBox event.
myCheckBox.setOnAction((event) -> {
    boolean selected = myCheckBox.isSelected();
    System.out.println("A fost selectat admis ");
});
```



# ObservableValue<T>

- Interfata generica ObservableValue<T> este utilizata pentru a încapsula diverse tipuri de valori și a asigura un mecanism de schimbare a acestora prin notificari.

```
public interface ObservableValue<T> extends Observable;
```

- *Metode:*

```
T getValue(); //furnizeaza valoarea acoperita
```

```
void addListener(ChangeListener<? super T> listener);
```

```
void removeListener(ChangeListener<? super T> listener); // furnizare mecanism  
de inregistrare/stergere ascultatori
```

- *Exemple de implementari:*

```
public class SimpleStringProperty extends StringPropertyBase;
```

```
public class SimpleObjectProperty<T> extends ObjectPropertyBase<T>;
```

```
public class SimpleDoubleProperty extends DoublePropertyBase;
```

# Property - Observable - Listener

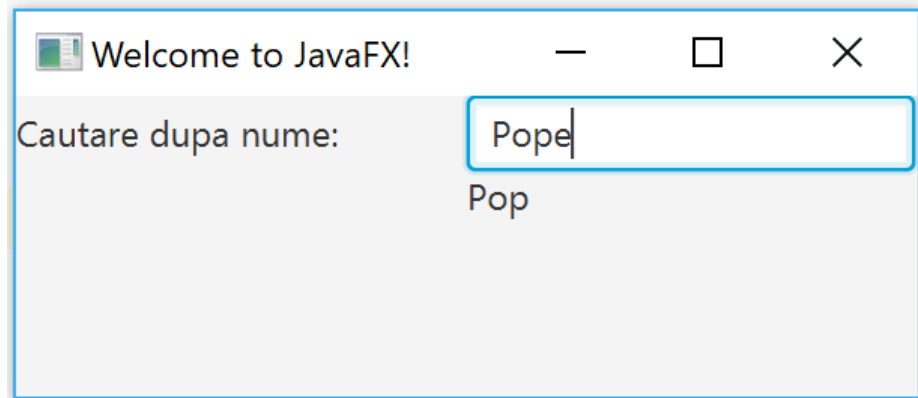
```
BooleanProperty booleanProperty = new SimpleBooleanProperty(true);
// Add change listener
booleanProperty.addListener(new ChangeListener<Boolean>() {
    @Override
    public void changed(ObservableValue<? extends Boolean> observable,
        Boolean oldValue, Boolean newValue) {
        System.out.println("changed " + oldValue + "->" + newValue);
        //myFunc();
    }
});
Button btn = new Button();
btn.setText("Switch boolean flag");
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        booleanProperty.set(!booleanProperty.get()); //switch
        System.out.println("Switch to " + booleanProperty.get());
    }
});
// Bind to another property variable
btn underlineProperty().bind(booleanProperty);
```

Se pot adauga oricati ascultatori!  
(Design pattern: ???)

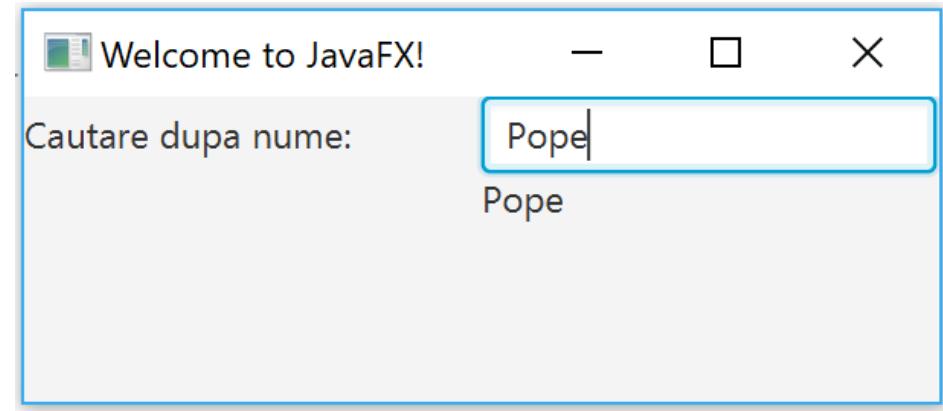
# TextField- events

```
TextField txt=new TextField();
```

```
txt.setOnKeyPressed(new  
EventHandler<KeyEvent>() {  
    @Override  
    public void handle(KeyEvent event) {  
        l.setText(txt.getText());  
    }  
});
```



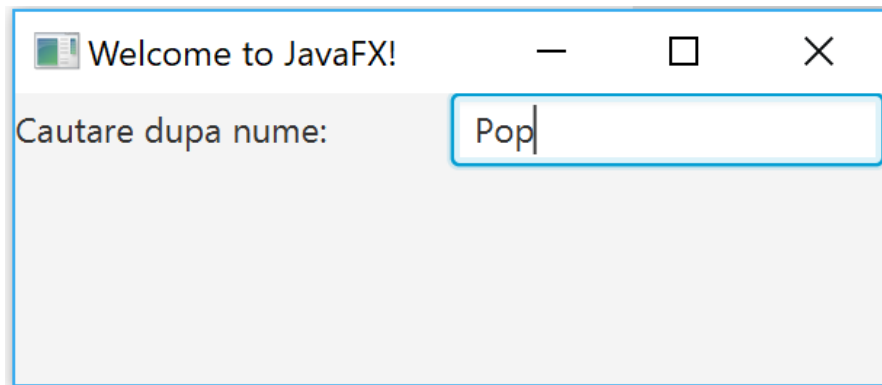
```
txt.textProperty().addListener(new  
ChangeListener<String>() {  
    @Override  
    public void changed(ObservableValue<?  
extends String> observable, String oldValue,  
String newValue) {  
        l.setText(newValue);  
    }  
});
```



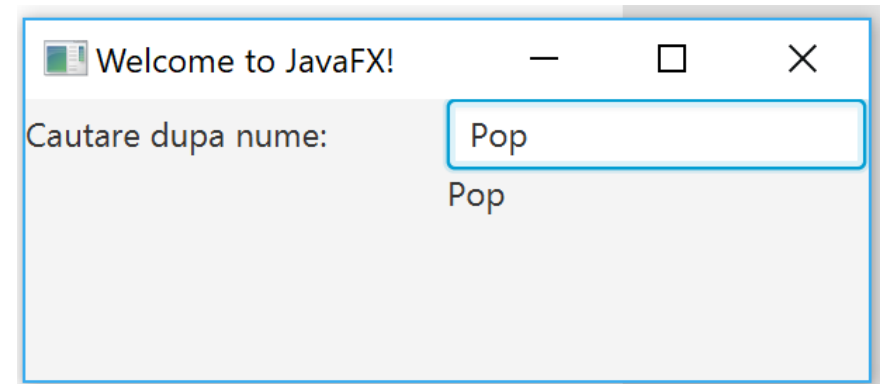


# TextField- events

```
//Handle TextField enter key event.  
txt.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        l.setText(txt.getText());  
    }  
});
```



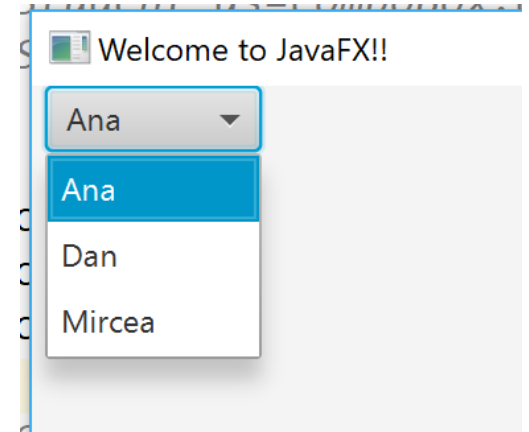
Enter ->



# ComboBox

```
ComboBox<String> comboBox2=new ComboBox<>();  
comboBox2.getItems().setAll("Ana", "Dan", "Mircea");  
comboBox2.getSelectionModel().selectFirst();
```

```
//listen to selectedItemProperty changes  
comboBox2.getSelectionModel().selectedItemProperty().addListener(n  
ew ChangeListener<String>() {  
    @Override  
    public void changed(ObservableValue<? extends String>  
observable, String oldValue, String newValue)  
    {  
        System.out.println(oldValue);  
    }  
});
```



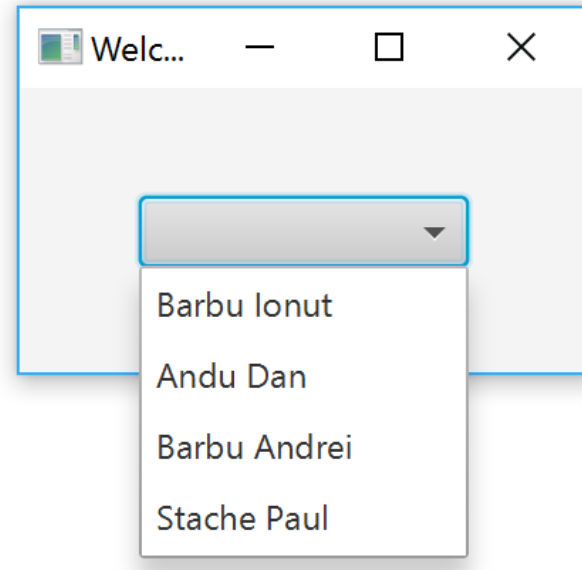
# ComboBox with data object – handle events

## Initializing the ComboBox

```
ComboBox<Student> comboBox=new ComboBox<Student>();  
ObservableList<Student>  
s=FXCollections.observableArrayList(getStudList());  
comboBox.setItems(s);
```

## ComboBox Rendering

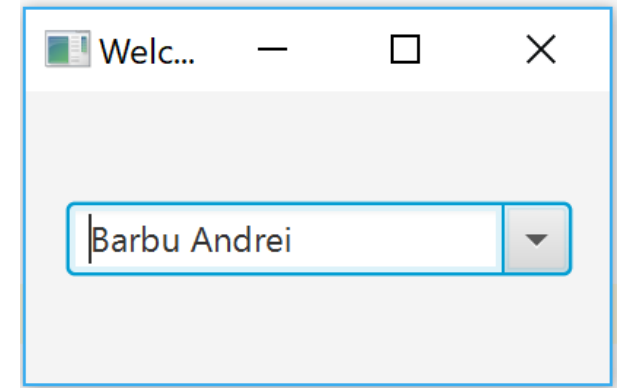
```
// Define rendering of the list of values in ComboBox drop down.  
comboBox.setCellFactory(new Callback<ListView<Student>, ListCell<Student>>() {  
    @Override  
    public ListCell<Student> call(ListView<Student> param) {  
        return new ListCell<Student>(){  
            @Override  
            protected void updateItem(Student item, boolean empty) {  
                super.updateItem(item, empty);  
  
                if (item == null || empty) {  
                    setText(null);  
                } else {  
                    setText(item.getFirstName() + " " + item.getLastName())  
                }  
            }  
        };  
    }  
});
```



# ComboBox handle events

```
// Define rendering of selected value shown in ComboBox.
comboBox.setConverter(new StringConverter<Student>() {
    @Override
    public String toString(Student s) {
        if (s == null) {
            return null;
        } else {
            return s.getFirstName() + " " + s.getLastName();
        }
    }
})

    @Override
    public Student fromString(String studentString) {
        return null; // No conversion fromString needed.
    }
});
```



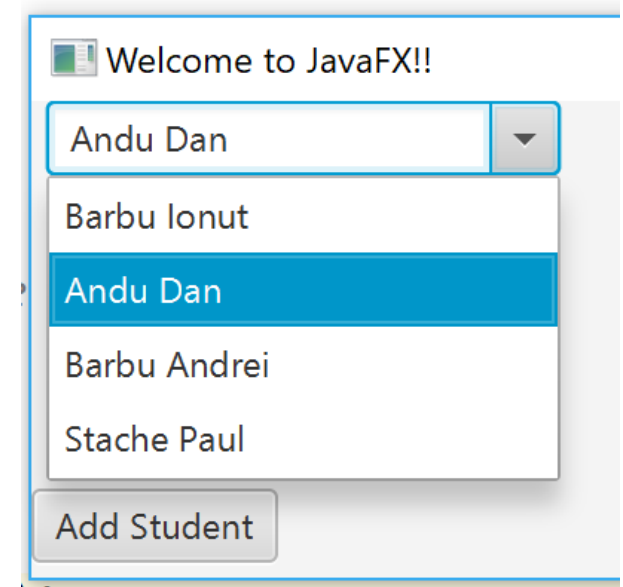
# ComboBox handle events

*//handle selection event*

```
comboBox.setOnAction(ev->{  
    Student as=comboBox.getSelectionModel().getSelectedItem();  
    System.out.println(as.toString());  
});
```

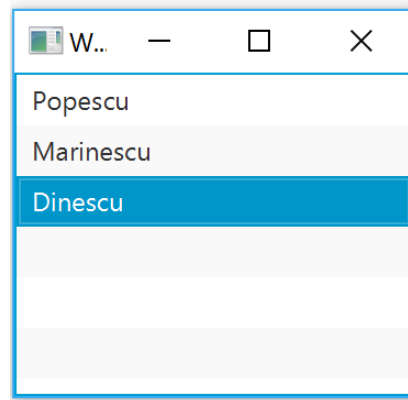
*//listen to selectedItemProperty changes*

```
comboBox.getSelectionModel().selectedItemProperty().addListener(new  
ChangeListener<Student>() {  
    @Override  
    public void changed(ObservableValue<? extends Student>  
observable, Student oldValue, Student newValue) {  
        System.out.println(newValue.toString());  
    }  
});
```



# List View

```
List<String> lview=new List<>(FXCollections.observableArrayList());  
lview.getItems().addAll("Popescu", "Marinescu", "Dinescu");
```



```
List<String> l=Arrays.asList("Popescu", "Marinescu", "Dinescu");  
ObservableList<String>  
observableList=FXCollections.observableArrayList(l);  
lview.setItems(observableList);
```

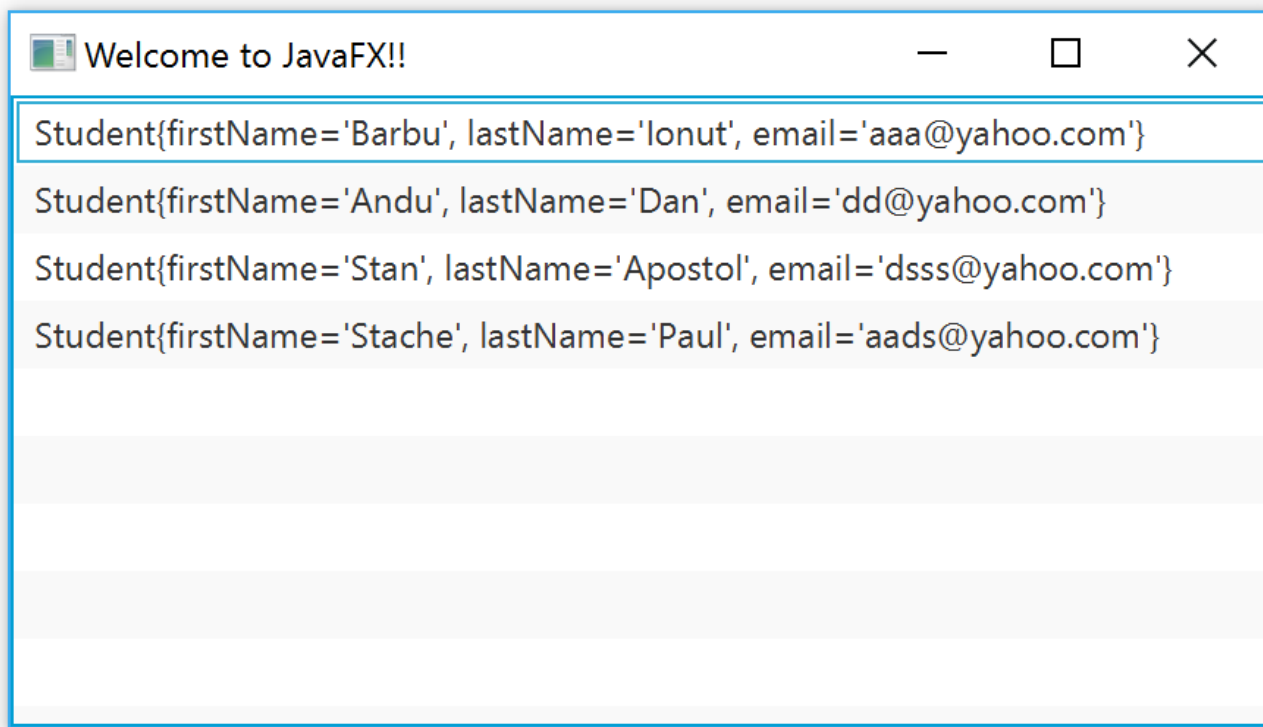
# List View

- Asemantor cu Combobox, dar **List View** nu are **ActionEvent**, in schimb are **selectedItemProperty**

```
myListView.getSelectionModel().selectedItemProperty().addListener((observable, oldValue,
newValue) -> {
    System.out.println("List View Selection Changed (selected: " + newValue.toString() + ")");
});
```

# List View for custom object

```
List View<Student> listView=new List View<>(students);
```

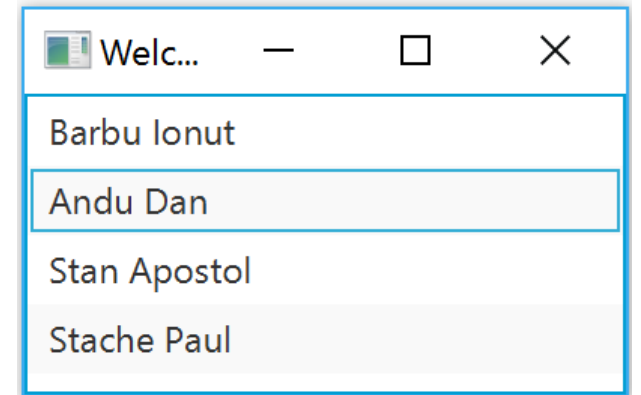




# List View for custom object

## cellFactory method

```
ListView<Student> listView=new ListView<>(students);
```



```
//cellFactory method
```

```
listView.setCellFactory(new Callback<ListView<Student>, ListCell<Student>>()  
{  
    @Override  
    public ListCell<Student> call(ListView<Student> param) {  
        ListCell<Student> listCell=new ListCell<Student>(){  
            // how to update text in this cell?  
        };  
        return listCell;  
    }  
});
```

# List View for custom object

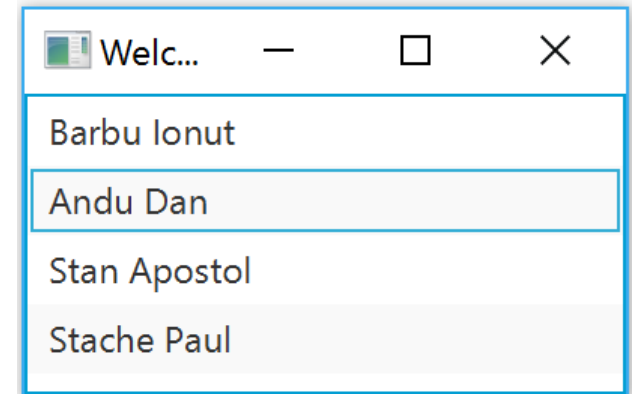
## cellFactory method

```
ListView<Student> listView=new ListView<>(students);
```

*Override updateItem() method from ListCell*

*//rendering data*

```
listView.setCellFactory(list -> new ListCell<Student>(){  
    @Override  
    protected void updateItem(Student item, boolean empty) {  
        super.updateItem(item, empty);  
        if (item == null || empty) {  
            setText(null);  
        } else {  
            setText(item.getFirstName() + " " + item.getLastName());  
        }  
    }  
});
```



# List View add new value

```
private ObservableList<Student> studs= FXCollections.observableArrayList();
```

- ```
List View<Student> list=new List View<>();  
list.setItems(studs);
```
- ```
studs.add(new Student("45","andrei","nistor","gdhgh"));
```

# List View selection

```
//itemul selectat
```

```
Student s=listView.getSelectionModel().getSelectedItem();
```

```
listView.getSelectionModel().selectedItemProperty().addListener(new ChangeListener<Student>()  
{  
    @Override  
    public void changed(ObservableValue<? extends Student> observable, Student oldValue,  
Student newValue) {  
        System.out.println(newValue.toString());  
    }  
});
```

# ListView set focus

```
ListView<Student> listView=new ListView<>(students);
```

```
listView.getFocusModel().focus(2);
```



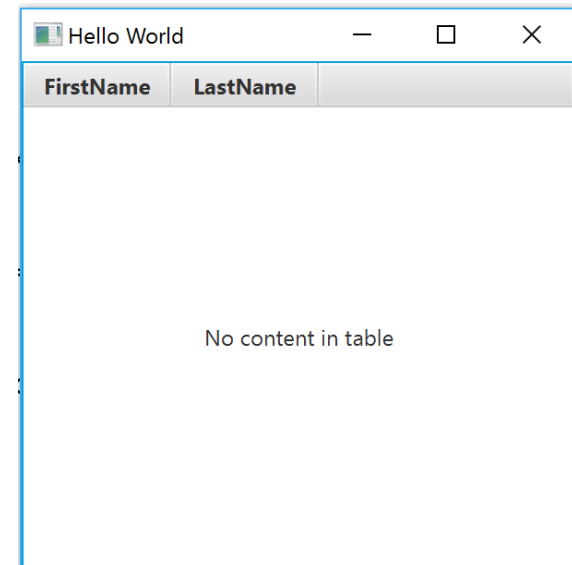
# TableView

- Create

```
TableView<Student> tableView=new TableView<Student>();
```

```
TableColumn<Student,String> columnName=new TableColumn<>("FirstName");  
TableColumn<Student,String> columnLastName=new TableColumn<>("LastName");
```

```
tableView.getColumns().addAll(columnName,columnLastName);
```



\_\_\_\_\_

- Binding data

```
List<Student> l=new ArrayList<Student>();
l.add(new Student("Barbu", "Ionut", "aaa@yahoo.com"));
l.add(new Student("Andu", "Dan", "dd@yahoo.com"));
l.add(new Student("Stan", "Apostol", "dsss@yahoo.com"));
l.add(new Student("Stache", "Paul", "aads@yahoo.com"));
```

```
ObservableList<Student> students = FXCollections.observableArrayList(1);
```

```
TableView<Student> tableView=new TableView<Student>();
```

```
TableColumn<Student,String> columnName=new TableColumn<>("FirstName");
TableColumn<Student,String> columnLastName=new TableColumn<>("LastName");
tableView.getColumns().addAll(columnName,columnLastName);
```

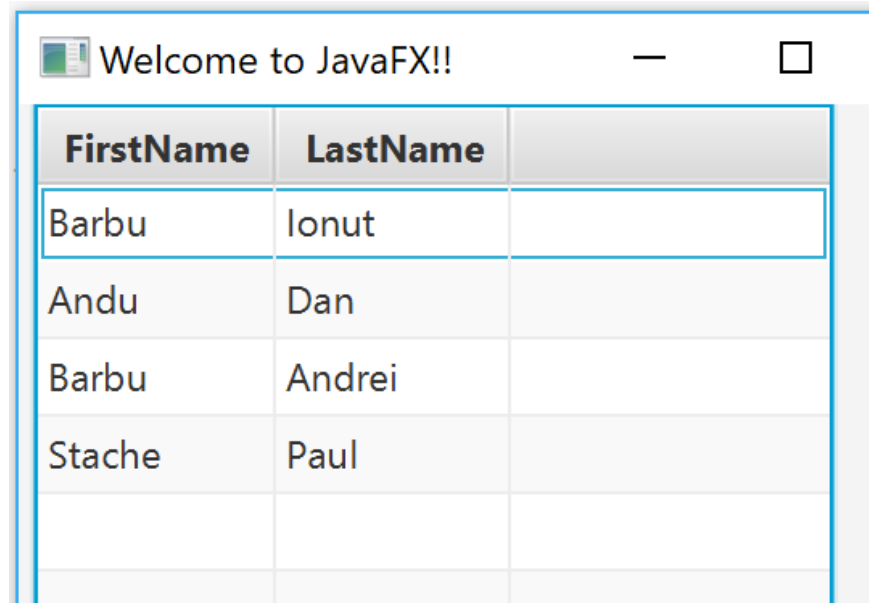
<pre>tableView.setItems(students);</pre>
--

[illegible]

# TableView

- **setCellValueFactory** method

```
columnName.setCellValueFactory(new PropertyValueFactory<Student, String>("firstName"));  
columnLastName.setCellValueFactory(new PropertyValueFactory<Student, String>("lastName"));
```



The screenshot shows a JavaFX window titled "Welcome to JavaFX!!" with a standard title bar (minimize, maximize, close buttons). Inside the window is a TableView with two columns: "FirstName" and "LastName". The table contains four rows of data:

FirstName	LastName
Barbu	Ionut
Andu	Dan
Barbu	Andrei
Stache	Paul



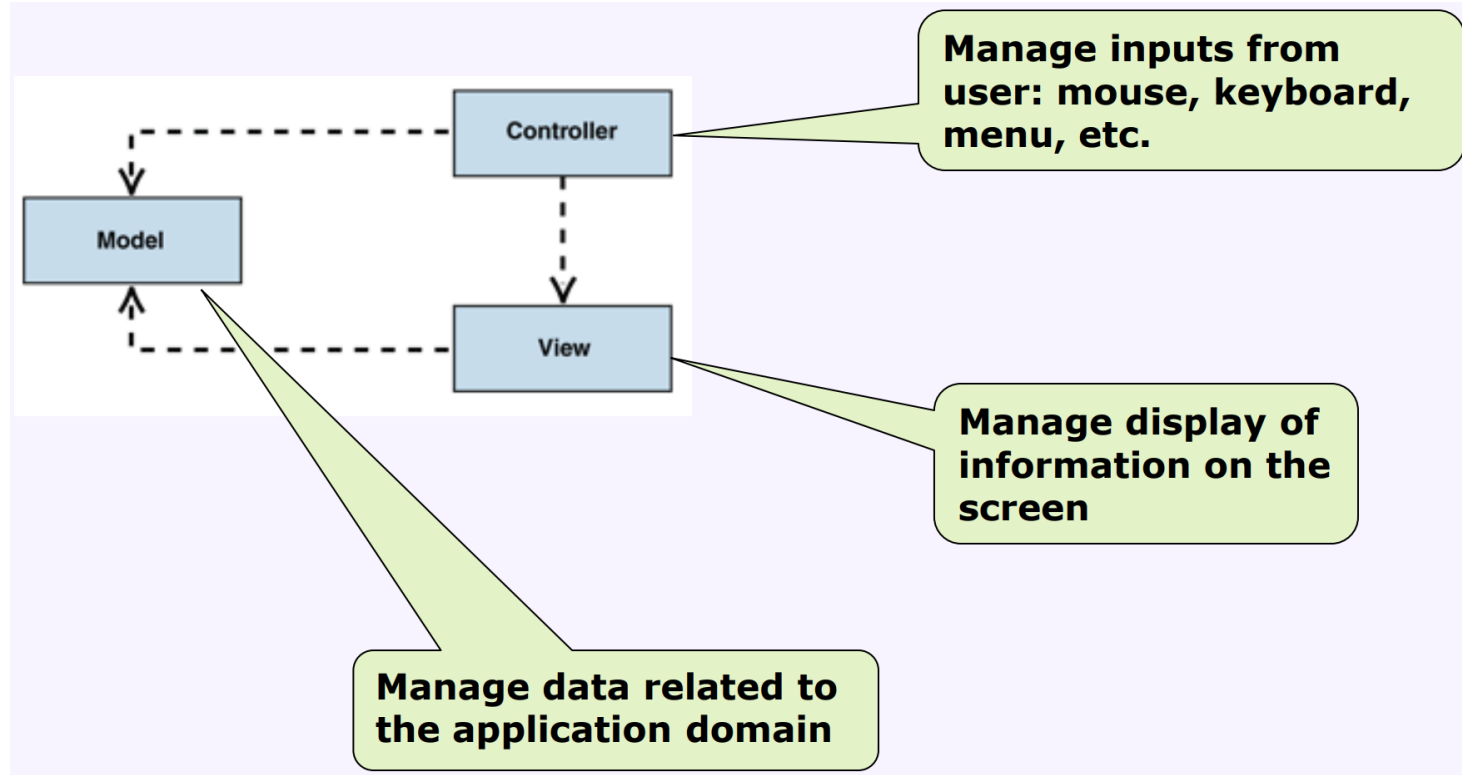
# TableView

- Listen for table selection changes

```
tableView.getSelectionModel().selectedItemProperty().addListener(new  
ChangeListener<Student>() {  
    @Override  
    public void changed(ObservableValue<? extends Student> observable, Student  
oldValue, Student newValue) {  
        System.out.println("A fost selectat"+ newValue.toString());  
    }  
});
```

# Model View Controller (MVC)

- JavaFX este dezvoltata dupa filozofia **Model View Controller** (MVC) separand partea de logica de partea de vizualizare si manipulare.



# Documentatie

- Tutoriale JavaFX
- <http://docs.oracle.com/javafx/index.html>
- **JavaFX API**
- <http://docs.oracle.com/javafx/2.0/api/index.html>
- <http://code.makery.ch/blog/javafx-8-event-handling-examples/>
- <http://code.makery.ch/library/javafx-8-tutorial/part2/>