CH24

Event Handling

The modern approach to handling events is the **delegation event model**. In this model, a **source** generates an event and sends it to the **listener(s)**. Once the listen receives the event, it processes it and returns. In this model, the logic that processes the events is separate from the UI logic that generates events.

Events

An event is an object that describes a state change in a source. Examples include pressing a button and clicking the mouse, a timer expiring, an operation completes.

Event Sources

A source is an object that generates an event. A source must register listeners in order for listeners to receive notifications about events. The general form of registration methods is:

public void add*Type*Listener(*Type*Listen el)

where *type* is the name of the event. For example, *addKeyListener()* registers a keyboard event listener; *addMouseMotionListener()* registers a mouse motion listener. Events are only **cast** to listeners with the appropriate registration.

Some sources only allow one listener to be registered.

Listeners can unregister by calling *removeTypeListener()*.

Event Listeners

A listener is an object that is notified when an event occurs. The listener must supply the event handlers.

Event Classes

The root of the event class hierarchy is *java.util.EventObject*, superclass to all events. It defines two methods: *getSource()* and *toString()*.

JavaFX notes

JavaFX applications do not require a *main()* if they are created a JAR file which embeds the JavaFX launcher. *main()* is only used as a fallback if it doesn’t launch properly.

The main class extends *javafx.application.Application*. A javafx application does the following when launched:

1. constructs an instance of the *Application*
2. calls *init()*, the initialization method. Can override to perform initialization prior to application starting.
3. calls *start(javafx.Stage.Stage)*, the main entry point for all javafx applications.
4. calls *stop()* when application finishes running – either the application calls *Platform.exit()*, or all windows have been closed

Stage, Scene, Scene Graph

The javafx UI contains a stage and a scene. The stage is the top level javafx container; the scene is the container of all contents. The scene is represented by a hierarchical scene graph of nodes – the root node and child nodes.

*Stage*

javafx can have multiple windows open; each is a stage. A primary *Stage* is constructed by the platform; the application can construct additional *Stages*. A stage can only display one scene at a time, but the scen can change at runtime.

Notable methods:

* *show()* – must be called for the window to open
  + *showAndWait()* -- *show()* that returns when the window is closed
* *setTitle(String str)* – sets what displays on the window
* *setScene(Scene s)* – sets the scene on this stage
* *setX(), setY()*
* *setWidth(), setHeight()*
* *setMaxHeight(double d), setMaxWidth(d), setMinHeight(d), setMinWidth(d)*

A stage must have a stsyle before being made visible. It is set with *initstyle()*. The styles are:

* *StageStyle.DECORATED* – white background with platform decorations (top bar; minimize, maximize, close buttons)
* *StageStyle*.UNDECORATED – white background and no decorations
* *StageStyle*.*TRANSPARENT*
* *StageStyle.UTILITY* – white background and minimal decorations.
* *StageStyle*.*UNIFIED* – like a decorated stage but with no border between decoration and main areas

Fullscreen – switch a stage into fullscreen mode by calling *setFullScreen()*.

Stage modality – determines if the stage’s window will block other windows created by the same javafx application. Set modality with *initModality()*.

Owner – a stage can be owned by another stage via *initOwner()*.

*Scene*

The *Scene* constructor takes a *Parent* object as the root. *Parent* encompasses all nodes with children. A scene graph can have zero or one parent. A scene needs to be attached to a stage to be displayed.

General form of Scene construction:

parent parentObj = new Parent();

Scene scene = new scene(parentObj);

Stage stage = new stage();

stage.setScene(scene);

Scene Graph & Node

All visual components need to be attached to a scene to be displayed. The total object graph attached to a scene is called the scene graph. The components are called nodes. All nodes are subclasses of *javafx.scene.Node*. Only subclasses of *Parent* can be directly added to a scene graph; non *Parent* components need a parent before adding.

A node may occur at most once in a scene graph. Each node can be given a unique ID.

If a child node already has a parent and gets assigned another one, the former parent will be removed.

There are two types of nodes: branch nodes and leaf nodes. Branch nodes can contain other nodes (including branch nodes); leaf nodes cannot.

Branch nodes are subclasses of class *Parent*; leaf nodes are subclasses of *Group*, *Region*, and *Control* (or their subclasses).

FXML

FXML is an XML format for JavaFX layouts.

Loading

The *javafx.fxml.FXMLLoader* class is responsible for loading an FXML files and creating the components it declares.

Example:

FXMLLoader loader = new FXMLLoader();

loader.setLocation(new URL(“C:/path/to/FXML/file.fxml”));

VBox vb = loader.<VBox>load();

Example:

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

<?import javafx.scene.layout.VBox?>

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

<VBox>

<children>

<Label text="Hello world FXML"/>

</children>

</VBox>

Notes:

* the first line is standard
* *load()* returns an object type of its generic parameter, which corresponds to the element in the FXML file
* FXML requires import statements, which start and end with ? (otherwise the same in java)
* The *Label* instance will be added to the *children* property of the *Vbox* instance

FXML can create objects

FXML can create instances of both JavaFX GUI components or regular java classes. It can do so in several ways:

* via FXML elements and constructors with no arguments

The element names are that of their classes. This is only possible if the class has a constructor that takes no arguments. The above is an example.

* via a custom *valueOf()*

In this method, first create a static *valueOf()* method in a custom class. Insert a *value* attribute in the FXML element to create objects.

Example: in Java:

**package** mypackage;

**class** MyClass {

**public** **static** MyClass valueOf(String val) {

**return** **new** MyClass(val);

}

**private** String value = **null**;

**public** MyClass(String s) {

**this**.value = s;

}

}

in FXML:

<?xml version=”1.0” encoding=”UTF-8”?>

<?import mypackage.MyClass?>

<MyClass value = “example”/>

The *<MyClass>* element causes the *FXMLLoader* to call *MyClass.valueOf()*. The object returned by *valueOf()* is inserted into the GUI.

* via factory methods

This is slightly different from the last method:

**package** mypackage;

**class** MyClass {

**public** **static** MyClass instance() {

**return** **new** MyClass();

}

}

then in FXML

<?xml version=”1.0” encoding=”UTF-8”?>

<?import mypackage.MyClass?>

<MyClass fx:factory=”instance”/>

*instance()* must has no arguments.

FXML Properties

Most javaFX objects have properties. Properties are set in XML with either an XML element attribute or nested elements. Example:

<VBox spacing=”20”>

<children>

<Label text=”line”/>

</children>

</VBox>

* In the <VBox> tag, 20 is passed to setSpacing() of the VBox created.
* the <children> tag corresponds to VBox.getChildren(); all nested elements are converted to JavaFX components and added to the VBox object’s children.
  + classes have default properties; for them, the tag can be ignored. For instance, this is equivalent to the above (*children* is the default proerty of *VBox*):

<VBox spacing=”20”>

<Label text=”line”/>

</VBox>

* “line” is passed to the *setText()* method of the created *Label*.

properties have getters and setters, e.g. *getText()* and *setText()*.

Element IDs

In the FXML file, assign IDs to elements via the *id* attribute. It is declared as:

fx:id=”name”

The element can then be referenced by ID in the FXML or CSS document.

Event handling

FXML can set event handlers on elements. To do so, use a *script* element.

Example:

<VBox xmlns:fx="http://javafx.com/fxml">

<Label fx:id="label1" text="Button not clicked"/>

<Button fx:id="button1" text="Click me!" onAction="reactToClick()"/>

<fx:script>

function reactToClick() {

label1.setText("Button clicked"); //reference by ID

}

</fx:script>

</VBox>

*<Button>* declares an event handler via its *onAction* attribute, which calls *reactToClick()* which is defined in *<script>*.

CSS styling

A CSS stylesheet can apply to four different scopes of a JavaFX application:

1. the whole application

The default CSS stylesheet is automatically applied to all JavaFX components if no styling is provided. (this is pre-scripted)

1. scene specific

This stylesheet created for a *Scene* object, and is applied to all components on the *Scene*’s scene graph. it takes precedence over the default stylesheet. It is added with

scene.getStylesheets().add(“scene-style.css”);

1. parent specific

This stylesheet is set on all subclasses of *Parent*. It takes precedence over the default stylesheet and the scene stylesheet. Also added with *getStylesheets().add(“stylesheet.css”)*;

1. component specific

This is not a stylesheet, but a String containing the CSS style properties, stored in a component’s *style* property. Example:

Button button = new Button(“a button”);

button.setStyle(“-fx-background-color: #0000ff; -fx-border-color: #ff0000;”);

JavaFX components decalred inside FXML can be styled with the *style* element. Example:

<VBox>

<Button text="Click me!"/>

<style>

-fx-padding: 10;

-fx-border-width: 3;

</style>

</Button>

</VBox>

Specific Class Examinations

Some node objects:

*javafx.scene.layout.Pane* (Parent)

(Also called containers) *Pane* is the base class for layout panes and used as the parent scene. The individual components on a pane are called its **children**. Children are added with *Pane.getChildren.add()* or *Pane.getChildren.addAll()*.

It has subclasses for various purposes:

* *Pane* itself can be used. It provides absolute positioning for the children and the class does nothing to reposition them.
* *GridPane* – creates a grid of rows and columns. Children occupy one or more row or column on the grid, and are added with *GridPane.add(col, row)* or *GridPane.add(col, row, colSpan, rowSpan)*. The children can overlap and the one added latest will be on top.
* *TilePane* – lays out children on uniform tiles. Nodes are either tiled in rows or columns.
* *StackPane* – lays its children in a back-to-front stack. Can also align the children relative to the parent (center, top-left, bottom-right, etc.).
* *AnchorPane* – allows children to be anchored at an offset to the pane’s edges. Can also be styled with CSS.
* *BorderPane* – lays out children in top, left, bottom, right or center.
* *HBox & VBox –* displays children in a single row/column.

*javafx.scene.Group* (Parent)

*Group* contains an *ObservableList* of its children rendered in order.

*Group* is not resizable, its size is the smallest that can hold all its children. Compared with *Pane* which can be resized.

*javafx.scene.shape.Shape* (leaf)

Class for geometric shapes. Some subclasses include:

* *Arc*
* *Circle*
* *Ellipse*
* *Rectangle*
* *Text* (in detail later)

*javafx.scene.control.Control* (leaf)

Base class for all user interface controls. Its most commonly used subclasses are:

* *Button*
* *Label* (in detail later)
* *ListView*
* *TextField*

its other subclasses are:

* *Accordion*
* *CheckBox*
* *ChoiceBox*
* *ColorPicker*
* *ComboBox*
* *DatePicker*
* *Menu*
* *MenuBar*
* *PasswordField*
* *ProgressBar*
* *RadioButton*
* *Slider*
* *Spinner*
* *SplitMenuButton*
* *SplitPane*
* *TableView*
* *TabPane*
* *TextArea*
* *TitledPane*
* *ToggleButton*
* *ToolBar*
* *TreeTableView*
* *TreeView*

*javafx.scene.Chart*

Javafx has a number of built-in charts. A chart contains 3 parts: the title, legend and chartContent, populated by the specific *Chart* subclasses:

* *AreaChart*
* *BarChart*
* *BubbleChart*
* *LineChart*
* *PieChart*
* *ScatterChart*
* *StackedAreaChart*
* *StackedBarChart*

It is possible to set the mouse cursor in a scene. To do so, call *setCursor()*.

First example

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;

public class HelloWorld extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Hello World!");

Button btn = new Button();

btn.setText("Say 'Hello World'");

btn.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

System.out.println("Hello World!");

}

});

StackPane root = new StackPane(); //this is the root node

root.getChildren().add(btn);

primaryStage.setScene(new Scene(root, 300, 250));

primaryStage.show();

}

}

*javafx.scene.control.Button* implements a button.

Example 2: log-in form

**import** javafx.application.Application;

**import** javafx.scene.Scene;

**import** javafx.scene.control.Button;

**import** javafx.scene.layout.\*;

**import** javafx.scene.text.\*;

**import** javafx.scene.control.\*;

**import** javafx.stage.Stage;

**import** javafx.geometry.\*;

**public** **class** Main **extends** Application {

**public** **static** **void** main(String[] args) {

*launch*(args);

}

**public** **void** start(Stage primaryStage) {

primaryStage.setTitle("Hello World!");

GridPane gp = **new** GridPane();

gp.setAlignment(Pos.***CENTER***);

gp.setHgap(10); //horizontalgap

gp.setVgap(10); //vertical gap

gp.setPadding(**new** Insets(25,25,25,25));

//Insets describes the inside offsets of a rectangular area, with arguments //corresponding to top,right,bottom,left

Scene scene = **new** Scene(gp, 300, 275);

//create Scene with gp as root node, 300 and 275 as width and height

//when the window is resized, the nodes change with it

primaryStage.setScene(scene);

//add text, labels, fields

Text txt = **new** Text("Welcome"); //text node

txt.setFont(Font.*font*("serif", FontWeight.***NORMAL***, 20));

//GradPane.add(Node child, int colIndex, int rowIndex, int colSpan, int rowSpan)

gp.add(txt, 0, 0, 2, 1);

Label username = **new** Label("User Name:");

gp.add(username, 0, 1);

TextField tf = **new** TextField();

gp.add(tf, 1, 1);

Label password = **new** Label("Password");

gp.add(password, 0, 2);

PasswordField pwField = **new** PasswordField();

gp.add(pwField, 1, 2);

//add a button

Button btn = **new** Button("Submit");

//HBox lays out its children in a row, with option for paddings and borders

HBox hbBtn = **new** HBox(10); //spacing 10

hbBtn.setAlignment(Pos.***BOTTOM\_LEFT***);

hbBtn.getChildren().add(btn);

gp.add(hbBtn, 1, 4);

**final** Text actionTarget = **new** Text();

gp.add(actionTarget, 1, 5);

//gp.setGridLinesVisible(true);

primaryStage.show();

}

}

*javafx.scene.image.ImageView*

The *ImageView* control can display an image inside a JavaFX GUI. It must be added to a scene graph to become visible.

The *ImageView* constructor requires an instance of *javafx.scene.image.Image* as parameter, which represents the image to be displayed.

*ImageView* is not a subclass of *Parent*, so it cannot be added directly to the scene graph.

Example: creating an *ImageView* and adding it to the scene graph via a *VBox*.

public class ImgViewer extends Application {

public void start(Stage PrimaryStage) throws Exception {

FileInputStream input = new FileInputStream(“img-37.jpg”);

Image img = new Image(input);

ImageView imgView = new ImageView(img);

HBox hbox = new HBox(imgView);

Scene scene = new Scene(hbox, 200, 100);

PrimaryStage.setScene(Scene);

PrimaryStage.show();

}

}

*ImageView* can also be used in *Label* or *Button*, causing them to display the image along side the text.

*javafx.scene.text.Text*

*Text* displays a text. *Text* is a subclass for *Shape* and *Node*.

Text is set via *setText()*.

Font, size, weight are set via *setFont()*.

Text color is set via *setFill()* and *setStroke()*.

Text position is set via *setX(), setY()* and *setTextOrigin()* (sets the text relative to the Text control).

Example:

Text txt = new Text();

txt.setText(“text to display”);

txt.setFont(Font.font(“Arial, foneWeight.BOLD, 36));

txt.setFill(“Color.BLACK”);