JAVAFX PROGRAMMING COOKBOOK

Hot Recipes for JavaFX Development

JavaFX™

ANDREAS POMAROLLI

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

JavaFX Programming Cookbook

Contents

Preface

JavaFX is a software platform for creating and delivering desktop applications, as well as rich internet applications (RIAs) that can run across a wide variety of devices. JavaFX is intended to replace Swing as the standard GUI library for Java SE, but both will be included for the foreseeable future. JavaFX has support for desktop computers and web browsers on Microsoft Windows, Linux, and Mac OS X.

JavaFX 2.0 and later is implemented as a native Java library, and applications using JavaFX are written in native Java code. JavaFX Script has been scrapped by Oracle, but development is being continued in the Visage project. JavaFX 2.x does not support the Solaris operating system or mobile phones; however, Oracle plans to integrate JavaFX to Java SE Embedded 8, and Java FX for ARM processors is in developer preview phase. (Source: https://en.wikipedia.org/wiki/JavaFX)

In this ebook, we provide a compilation of JavaFX programming examples that will help you kick-start your own web projects. We cover a wide range of topics, from Concurrency and Media, to Animation and FXML. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

Andreas has graduated from Computer Science and Bioinformatics at the University of Linz. During his studies he has been involved with a large number of research projects ranging from software engineering to data engineering and at least web engineering.

His scientific focus includes the areas of software engineering, data engineering, web engineering and project management. He currently works as a software engineer in the IT sector where she is mainly involved with projects based on Java, Databases and Web Technologies.

Chapter 1

JavaFX Tutorial for Beginners

JavaFX is an open source Java-based framework for developing rich client applications. It is comparable to other frameworks on the market such as Adobe Flex and Microsoft Silverlight.

JavaFX is also seen as the successor of Swing in the arena of graphical user interface (GUI) development technology in Java platform. The JavaFX library is available as a public Java application programming interface (API).

The following examples uses Java SE 7 and JavaFX 2.2.

1.1 Introduction

The GUI in JavaFX is constructed as a scene graph. A scene graph is a collection of visual elements, called nodes, arranged in a hierarchical fashion. A scene graph is built using the public JavaFX API. Nodes in a scene graph can handle user inputs and user gestures. They can have effects, transformations, and states.

Types of nodes in a scene graph include simple UI controls such as buttons, text fields, two-dimensional (2D) and three-dimensional (3D) shapes, images, media (audio and video), web content, and charts.

1.2 Your First JavaFX Application

1.2.1 The Code

FxFirstExample.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class FxFirstExample extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text
```

```
                Text text = new Text("Hello JavaFX");
                // Create the VBox
                VBox root = new VBox();
                // Add the Text to the VBox
                root.getChildren().add(text);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                // Create the Scene
                Scene scene = new Scene(root);

                // Set the Properties of the Stage
                stage.setX(100);
                stage.setY(200);
                stage.setMinHeight(300);
                stage.setMinWidth(400);

                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("Your first JavaFX Example");
                // Display the Stage
                stage.show();
        }

}
```

1.2.2 Overriding the start() Method

A JavaFX application is a class that must inherit from the Application class that is in the javafx.application package. So it is necessary to override the `start()` method.

```
@Override
public void start(Stage stage)
{
        // do something
}
```

The `start()` method is the entry point for a JavaFX application. It is called by the JavaFX application launcher. Notice that the `start()` method is passed an instance of the Stage class, which is known as the primary stage of the application. You can create more stages as necessary in your application. However, the primary stage is always created by the JavaFX runtime.

1.2.3 Showing the Stage

Similar to a stage in the real world, a JavaFX `stage` is used to display a Scene. A `scene` has visuals, such as text, shapes, images, controls, animations, and effects, with which the user may interact, as is the case with all GUI-based applications.

A `stage` in JavaFX is a top-level container that hosts a `scene`, which consists of visual elements. The `Stage` class represents a stage in a JavaFX application. The primary `stage` is created by the platform and passed to the `start(Stage s)` method of the `Application` class.

```
@Override
public void start(Stage stage)
{
        // Display the Stage
        stage.show();
}
```

The bounds of a `stage` consist of four properties:

- X
- Y
- Width
- Height

The `x` and `y` properties determine the location of the upper-left corner of the `stage`. The `width` and `height` properties determine its size.

```
// Set the Properties of the Stage
stage.setX(100);
stage.setY(200);
stage.setMinHeight(300);
stage.setMinWidth(400);
```

The primary stage created by the application launcher does not have a `scene`. You have to create a `scene` for your `stage`. You must show the `stage` to see the visuals contained in its `scene`. Use the `show()` method to show the `stage`. Optionally, you can set a title for the stage using the `setTitle()` method.

```
// Create the Scene
Scene scene = new Scene(root);

// Add the scene to the Stage
stage.setScene(scene);
// Set the title of the Stage
stage.setTitle("Your first JavaFX Example");
// Display the Stage
stage.show();
```

1.2.4 Launching the Application

You can use one of the following two options to run a JavaFX Application:

It is not necessary to have a `main()` method in the class to start a JavaFX application. When you run a Java class that inherits from the `Application` class, the java command launches the JavaFX application if the class being run does not contain the `main()` method.

If you include a `main()` method in the JavaFX application class inside the `main()` method, call the `launch()` static method of the `Application` class to launch the application. The `launch()` method takes a String array as an argument, which are the parameters passed to the JavaFX application.

If you are using the first option, you do not need to write a `main()` method for the `FxFirstExample` class. If you are using the second option, the `HelloFXApp` class has to be enriched with the `main()` method.

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

1.2.5 Adding a Scene to the Stage

An instance of the `Scene` class represents a scene. A `stage` contains one `scene`, and a `scene` contains visual contents. The contents of the `scene` are arranged in a tree-like hierarchy. At the top of the hierarchy is the root node. The root node may contain child nodes, which in turn may contain their child nodes, and so on. You must have a root node to create a `scene`. You can use a VBox or another node type as the root node. `VBox` stands for Vertical box, which arranges its children vertically in a column.

The following code snippet adds the `scene` to the `stage`:

```
// Add the scene to the Stage
stage.setScene(scene);
```

1.2.6 The GUI

Figure 1.1: Yout First JavaFX Example

1.3 Controls

1.3.1 Introduction

JavaFX lets you create applications using GUI components. An application with a GUI performs three tasks:

- Accepts inputs from the user through input devices such as a keyboard or a mouse
- Processes the inputs
- Displays outputs

The UI provides a means to exchange information in terms of input and output between an application and its users. Entering text using a keyboard, selecting a menu item using a mouse, clicking a button, or other actions are examples of providing input to a GUI application. The application displays outputs on a computer monitor using text, charts, dialog boxes, and so forth.

Users interact with a GUI application using graphical elements called controls or widgets. Buttons, labels, text fields, text area, radio buttons, and check boxes are a few examples of controls. Devices like a keyboard, a mouse, and a touch screen are used to provide input to controls. Controls can also display output to the users. Controls generate events that indicate an occurrence of some kind of interaction between the user and the control. For example, pressing a button using a mouse or a spacebar generates an action event indicating that the user has pressed the button.

1.3.2 Label

An instance of the Label class represents a label control. As the name suggest, a `Label` is simply a label that is used to identify or describe another component on a screen. It can display a text, an icon, or both. Typically, a `Label` is placed next to (to the right or left) or at the top of the node it describes. A `Label` is not focus traversable. That is, you cannot set the focus to a `Label` using the Tab key. A `Label` control does not generate any interesting events that are typically used in an application.

1.3.2.1 The Code

FxLabelExample.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FxLabelExample extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text Fields
                TextField firstNameFld = new TextField();
                TextField lastNameFld = new TextField();

                // Create the Labels
                Label firstNameLbl = new Label("_First Name:");
                Label lastNameLbl = new Label("_Last Name:");

                // Bind the Label to the according Field
                firstNameLbl.setLabelFor(firstNameFld);
                // Set mnemonic parsing to the Label
                firstNameLbl.setMnemonicParsing(true);

                // Bind the Label to the according Field
                lastNameLbl.setLabelFor(lastNameFld);
                // Set mnemonic parsing to the Label
                lastNameLbl.setMnemonicParsing(true);

                // Create the GridPane
                GridPane root = new GridPane();
                // Add the Labels and Fields to the GridPane
                root.addRow(0, firstNameLbl, firstNameFld);
                root.addRow(1, lastNameLbl, lastNameFld);
                // Set the Size of the GridPane
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the GridPane
                 * Set the border-style of the GridPane
                 * Set the border-width of the GridPane
                 * Set the border-insets of the GridPane
                 * Set the border-radius of the GridPane
                 * Set the border-color of the GridPane
```

```
                */
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Label Example");
                // Display the Stage
                stage.show();
        }
}
```

The above example shows a window with two `Label` controls with text First Name: and Last Name: The `Label` with the text First Name: is an indicator for the user that he should enter a first name in the field that is placed right next to it. A similar argument goes for the Last Name: `Label` control.

1.3.2.2 Adding a Mnemonic to a Label

A `Label` control can have a mnemonic. `Mnemonic` parsing for `Label` controls is set to false by default. When you press the mnemonic key for a Label, the focus is set to the `labelFor` node for that `Label`. The following snippet of code creates a TextField and a `Label`. The `Label` sets a mnemonic, enables mnemonic parsing, and sets the `TextField` as its `labelFor` property.

```
// Create the Text Fields
TextField firstNameFld = new TextField();
TextField lastNameFld = new TextField();

// Create the Labels
Label firstNameLbl = new Label("_First Name:");
Label lastNameLbl = new Label("_Last Name:");

// Bind the Label to the according Field
firstNameLbl.setLabelFor(firstNameFld);
// Set mnemonic parsing to the Label
firstNameLbl.setMnemonicParsing(true);

// Bind the Label to the according Field
lastNameLbl.setLabelFor(lastNameFld);
// Set mnemonic parsing to the Label
lastNameLbl.setMnemonicParsing(true);
```

The topic `TextField` will be discussed in the next section.

A GridPane contains all labels and text fields in the above example. A full description of the class `GridPane`, `VBox` and other panes, which are used in all examples is available in the article JavaFX Layout Example.

1.3.2.3 The GUI

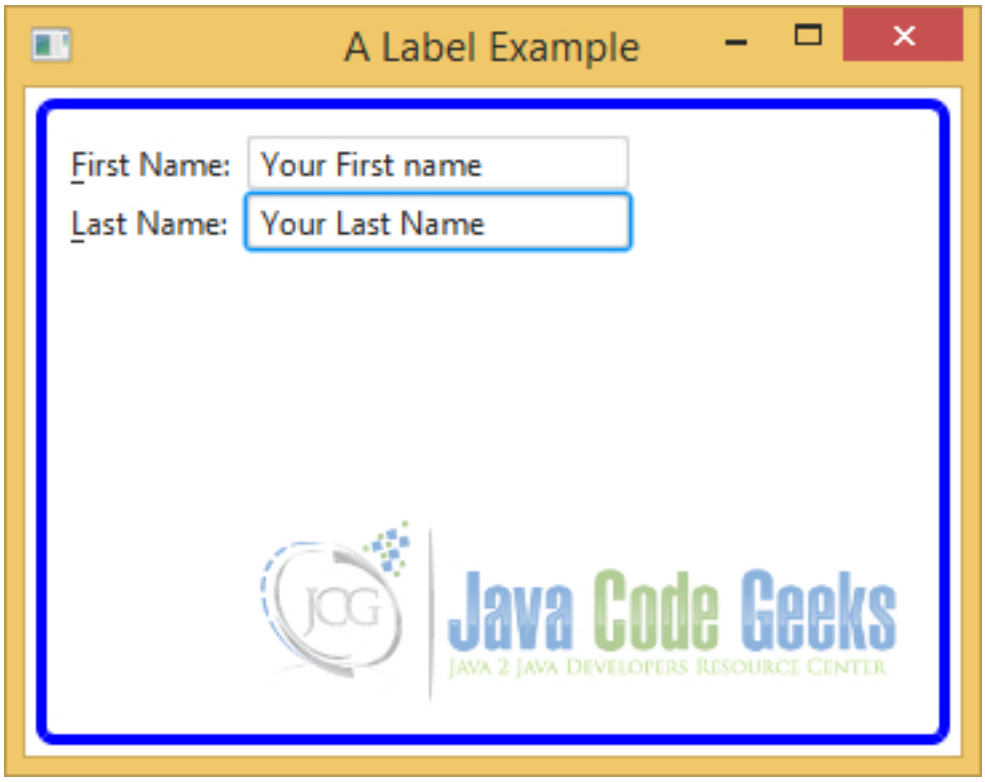

Figure 1.2: A JavaFX Label Example

1.3.3 TextField

A `TextField` is a text input control. It inherits from the A TextInputControl class. It lets the user enter a single line of plain text. If you need a control to enter multiline text, use `TextArea` instead. Newline and tab characters in the text are removed.

1.3.3.1 The Code

FxTextFieldExample.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FxTextFieldExample extends Application
{
        // Create the Message Label
        Label messageLbl = new Label("Enter your Name into the Text Fields.");

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

```
    @Override
    public void start(Stage stage)
    {
            // Create the TextFields
            TextField firstNameFld = new TextField();
            TextField lastNameFld = new TextField();

            // Both fields should be wide enough to display 15 chars
            firstNameFld.setPrefColumnCount(15);
            lastNameFld.setPrefColumnCount(15);

            // Set ActionEvent handlers for both fields
            firstNameFld.setOnAction(new EventHandler<ActionEvent>()
            {
        @Override public void handle(ActionEvent e)
        {
            printMessage("You have changed the First Name!");
        }
    });
            lastNameFld.setOnAction(new EventHandler<ActionEvent>()
            {
        @Override public void handle(ActionEvent e)
        {
            printMessage("You have changed the Last Name !");
        }
    });

            GridPane root = new GridPane();
            // Set the horizontal spacing to 10px
            root.setHgap(10);
            // Set the vertical spacing to 5px
            root.setVgap(5);

            // Add Labels and Fields to the GridPane
            root.addRow(0, messageLbl);
            root.addRow(1, new Label("First Name:"), firstNameFld);
            root.addRow(2, new Label("Last Name:"), lastNameFld);

            // Set the Size of the GridPane
            root.setMinSize(350, 250);

            /*
             * Set the padding of the GridPane
             * Set the border-style of the GridPane
             * Set the border-width of the GridPane
             * Set the border-insets of the GridPane
             * Set the border-radius of the GridPane
             * Set the border-color of the GridPane
            */
            root.setStyle("-fx-padding: 10;" +
                          "-fx-border-style: solid inside;" +
                          "-fx-border-width: 2;" +
                          "-fx-border-insets: 5;" +
                          "-fx-border-radius: 5;" +
                          "-fx-border-color: blue;");

            // Create the Scene
            Scene scene = new Scene(root);
            // Add the scene to the Stage
            stage.setScene(scene);
            // Set the title of the Stage
```

```
                stage.setTitle("A TextField Example");
                // Display the Stage
                stage.show();
        }

        // Helper Method
        public void printMessage(String message)
        {
                // Set the Text of the Label
                messageLbl.setText(message);
        }

}
```

1.3.3.2 Setting the width of a TextField

The `prefColumnCount` property determines the width of the control. The `TextField` in ur example is wide enough to display fiteen letters

```
// Both fields should be wide enough to display 15 chars
firstNameFld.setPrefColumnCount(15);
lastNameFld.setPrefColumnCount(15);
```

1.3.3.3 Adding ActionEvent Handler to a TextField

The `onAction` property is an ActionEvent handler, which is called when the Enter key is pressed in the `TextField`, as shown in the following code:

```
// Set ActionEvent handlers for both fields
firstNameFld.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have changed the First Name!");
        }
});
lastNameFld.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have changed the Last Name !");
        }
});
```

1.3.3.4 The GUI

After starting the application, the following window appears:

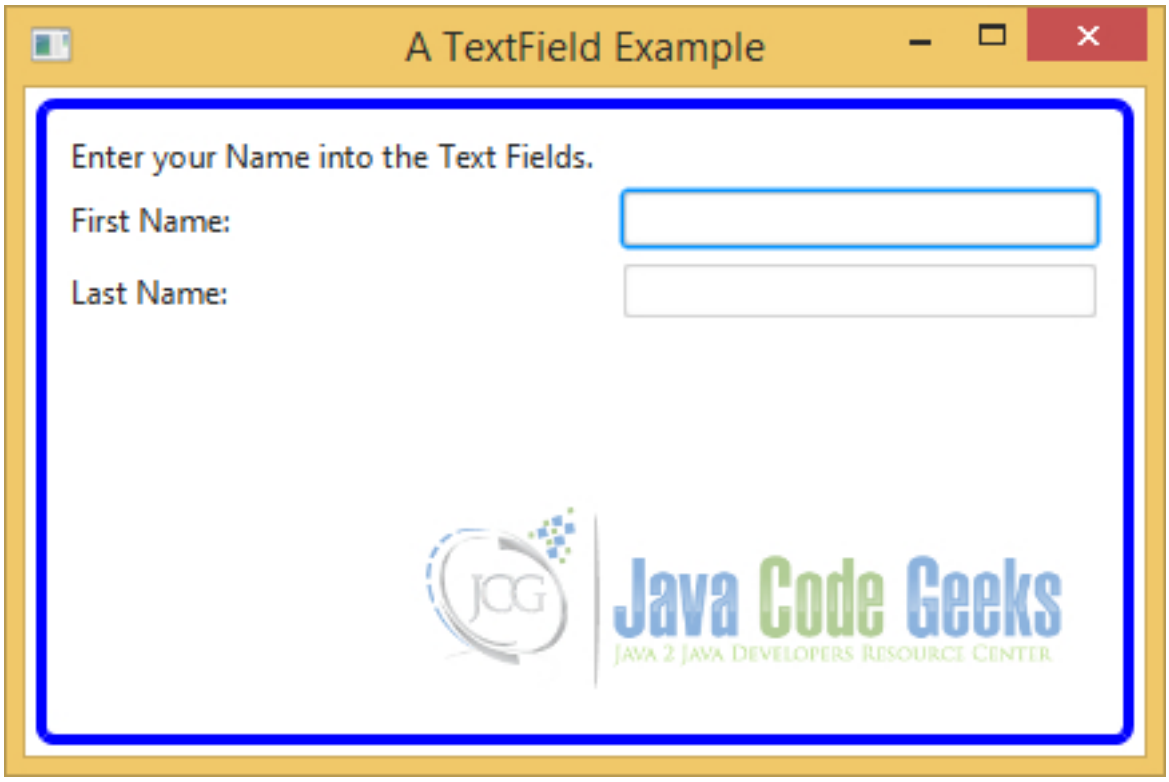

Figure 1.3: A TextField Example before inserting data

After inserting text into the `TextField`, the message will be changed:

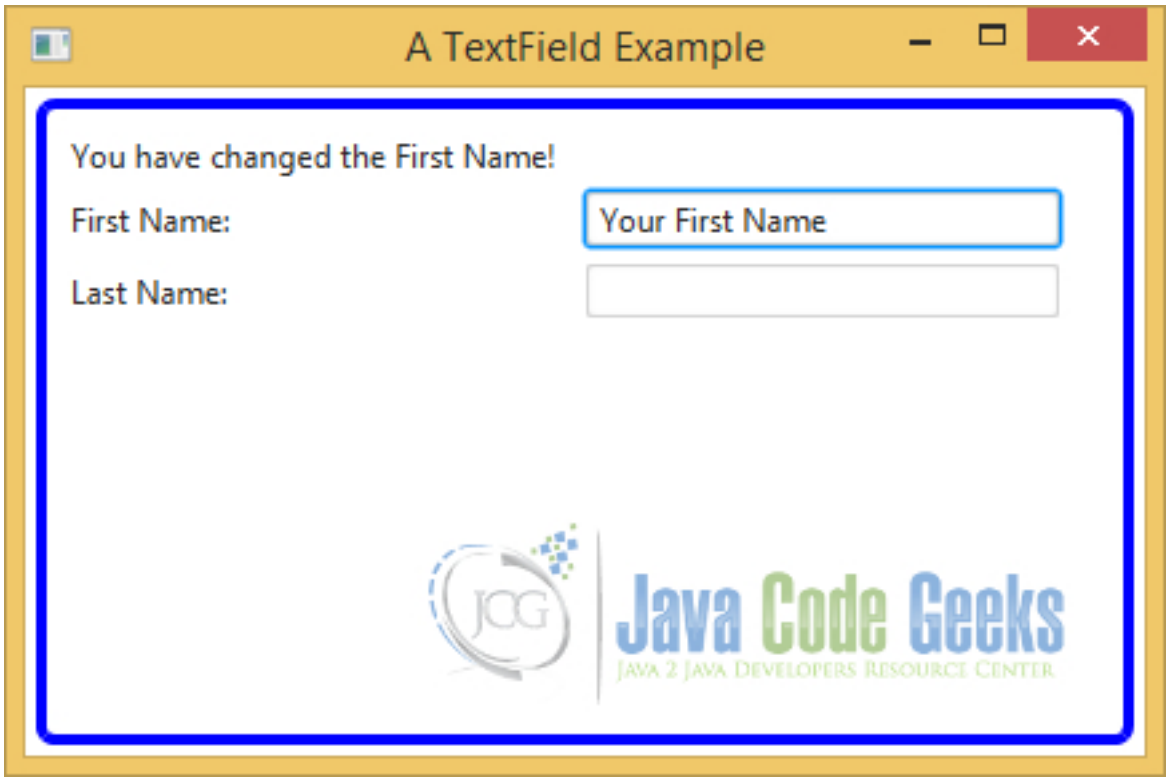

Figure 1.4: A TextField Example after inserting data

1.3.4 Button

A button that executes a command when activated is known as a command button. The Button, Hyperlink, and MenuButton classes represent command buttons. A `MenuButton` lets the user execute a command from a list of commands. Buttons used for presenting different choices to users are known as choice buttons. The ToggleButton, CheckBox, and RadioButton classes represent choice buttons. The third kind of button is a hybrid of the first two kinds. They let users execute a command or make choices.

1.3.4.1 The Code

FxButtonExample.java

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

public class FxButtonExample extends Application
{
        // Create the Message Label
        Label messageLbl = new Label("Press any Button to see the message");

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

```
            Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
            // Create a normal button with N as its mnemonic
            Button newBtn = new Button("_New");
            // Add EventHandler to the Button
            newBtn.setOnAction(new EventHandler<ActionEvent>()
            {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have pressed the new Button");
            }
        });

            // Create a default button with S as its mnemonic
            Button saveBtn = new Button("_Save");
            // Set this Button as the Default
            saveBtn.setDefaultButton(true);
            // Add EventHandler to the Button
            saveBtn.setOnAction(new EventHandler<ActionEvent>()
            {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have pressed the save Button");
            }
        });

            // Create a cancel button with C as its mnemonic
            Button cancelBtn = new Button("_Cancel");
            cancelBtn.setCancelButton(true);
            // Add EventHandler to the Button
            cancelBtn.setOnAction(new EventHandler<ActionEvent>()
            {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have pressed the cancel Button");
            }
        });

            // Create the HBox
            HBox buttonBox = new HBox();
            // Add the children to the HBox
            buttonBox.getChildren().addAll(newBtn, saveBtn, cancelBtn);
            // Set the vertical spacing between children to 15px
            buttonBox.setSpacing(15);

            // Create the VBox
            VBox root = new VBox();
            // Add the children to the VBox
            root.getChildren().addAll(messageLbl, buttonBox);
            // Set the vertical spacing between children to 15px
            root.setSpacing(15);
            // Set the Size of the VBox
            root.setMinSize(350, 250);

            /*
             * Set the padding of the VBox
             * Set the border-style of the VBox
             * Set the border-width of the VBox
```

```
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Button Example");
                // Display the Stage
                stage.show();
        }

        // Helper Method
        public void printMessage(String message)
        {
                // Set the Text of the Label
                messageLbl.setText(message);
        }

}
```

1.3.4.2 Adding ActionEvent Handler to a Button

An instance of the `Button` class represents a command button. Typically, a `Button` has text as its label and an `ActionEvent` handler is registered to it. The `mnemonicParsing` property for the `Button` class is set to true by default.

```
// Create a normal button with N as its mnemonic
Button newBtn = new Button("_New");
// Add EventHandler to the Button
newBtn.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have pressed the new Button");
        }
});
```

1.3.4.3 Setting the mode of a Button

A `Button` can be in one of three modes:

- A normal button
- A default button
- A cancel button

For a normal button, its `ActionEvent` is fired when the button is activated. For a default button, the `ActionEvent` is fired when the Enter key is pressed and no other node in the scene consumes the key press.

```
// Create a default button with S as its mnemonic
Button saveBtn = new Button("_Save");
// Set this Button as the Default
saveBtn.setDefaultButton(true);
// Add EventHandler to the Button
saveBtn.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have pressed the save Button");
        }
});
```

For a cancel button, the `ActionEvent` is fired when the Esc key is pressed and no other node in the scene consumes the key press. By default, a `Button` is a normal button. The default and cancel modes are represented by the `defaultButton` and `cancelButton` properties. You would set one of these properties to true to make a button a default or cancel button. By default, both properties are set to false.

```
// Create a cancel button with C as its mnemonic
Button cancelBtn = new Button("_Cancel");
cancelBtn.setCancelButton(true);
// Add EventHandler to the Button
cancelBtn.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have pressed the cancel Button");
        }
});
```

The following snippet of code creates a normal button and adds an `ActionEvent` handler. When the button is activated, for example, by clicking using a mouse, the `printMessage()` method is called:

```
// Create a normal button with N as its mnemonic
Button newBtn = new Button("_New");
// Add EventHandler to the Button
newBtn.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have pressed the new Button");
        }
});
```

1.3.4.4 The GUI

After starting the application, the following window appears:

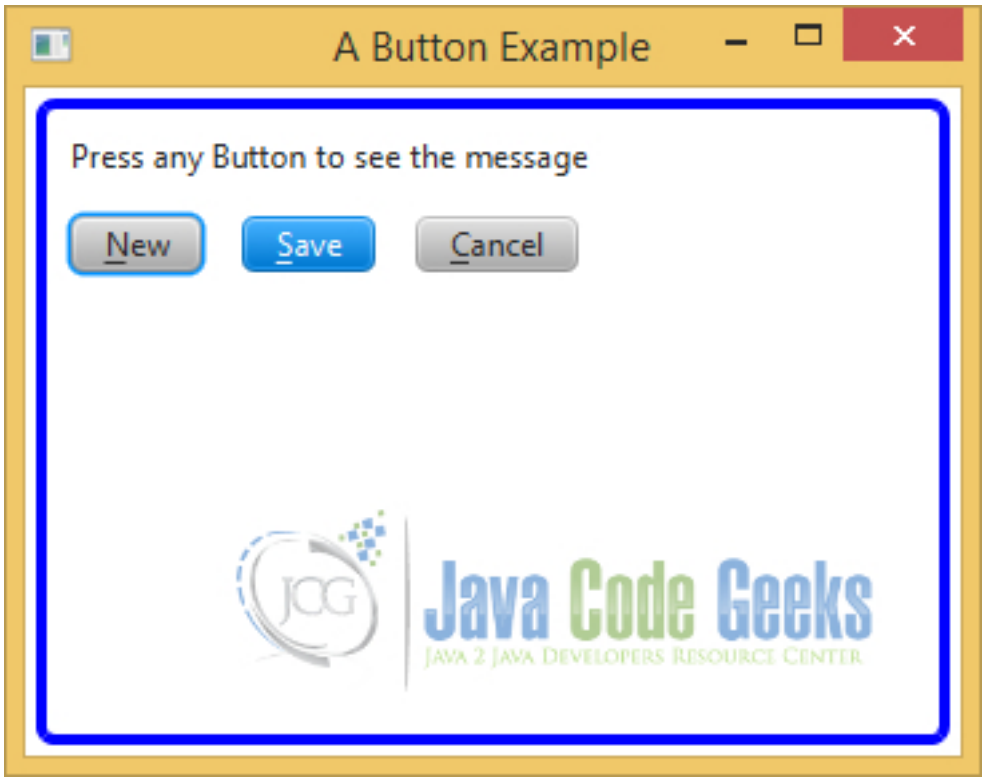

Figure 1.5: A ButtonExample before pressing any Button

After pressing any `Button`, the message will be changed:

Figure 1.6: A ButtonExample after pressing any Button

1.3.5 MenuButton

A `MenuButton` control looks like a button and behaves like a menu. When it is activated, it shows a list of options in the form of a pop-up menu. The list of options in the menu is maintained in an ObservableList<MenuItem> whose reference is returned by the `getItems()` method. To execute a command when a menu option is selected, you need to add the `ActionEvent` handler to the `MenuItem`.

1.3.5.1 The Code

FxMenuButtonExample.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.MenuButton;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxMenuButtonExample extends Application
{
        // Create the Message Label
        Label messageLbl = new Label("Choose your car!");

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

```
        }

        @Override
        public void start(Stage stage)
        {
                // Create the MenuItem ford
                MenuItem ford = new MenuItem("Ford");
                // Add EventHandler to the MenuItem
                ford.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have selected: Ford");
            }
        });

                // Create the MenuItem audi
                MenuItem audi = new MenuItem("Audi");
                // Add EventHandler to the MenuItem
                audi.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have selected: Audi");
            }
        });

                // Create the MenuItem ferrari
                MenuItem ferrari = new MenuItem("Ferrari");
                // Add EventHandler to the MenuItem
                ferrari.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have selected: Ferrari");
            }
        });

                // Create the MenuItem porsche
                MenuItem porsche = new MenuItem("Porsche");
                // Add EventHandler to the MenuItem
                porsche.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have selected: Porsche");
            }
        });

                // Create the MenuButton
                MenuButton cars = new MenuButton("Select");
                // Add menu items to the MenuButton
                cars.getItems().addAll(ford, audi, ferrari, porsche);

                // Create the VBox
                VBox root = new VBox();
                // Add the children to the VBox
                root.getChildren().addAll(cars, messageLbl);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
```

```
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A MenuButton Example");
                // Display the Stage
                stage.show();
        }

        // Helper Method
        public void printMessage(String message)
        {
                // Set the Text of the Label
                messageLbl.setText(message);
        }
}
```

1.3.5.2 The GUI

After starting the application, the following window appears:

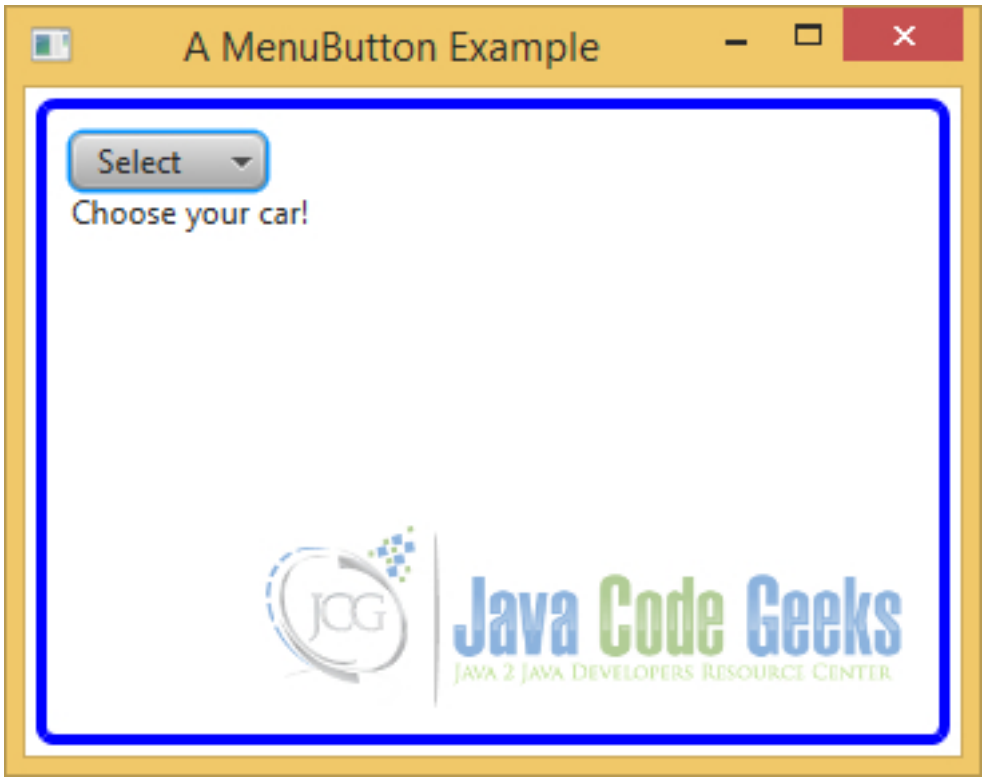

Figure 1.7: A MenuButton Example after pressing any Button

After pressing any `MenuButton`, the message will be changed:

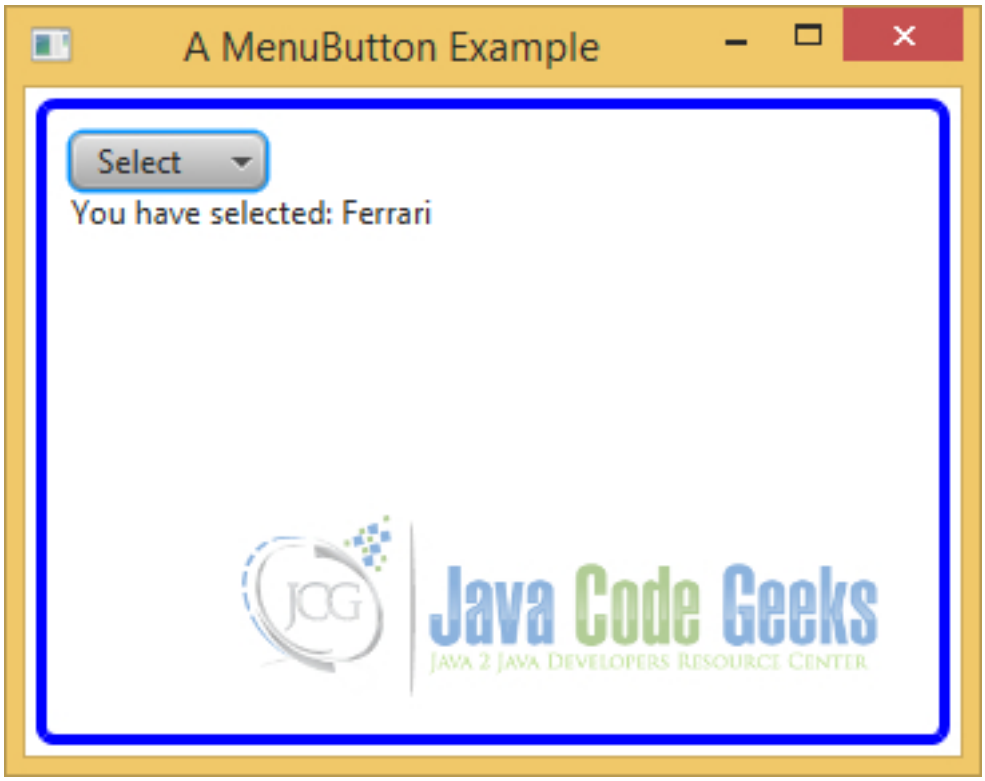

Figure 1.8: A MenuButton Example before pressing any Button

1.3.6 CheckBox

A `CheckBox` is a three-state selection control:

- Checked
- Unchecked
- Undefined

The `undefined` state is also known as an `indeterminate` state. A `CheckBox` supports a selection of three choices:

- True
- False
- Unknown

or

- Yes
- No
- Unknown

Usually, a `CheckBox` has text as a label. Clicking a `CheckBox` transitions it from one state to another cycling through three states. A box is drawn for a `CheckBox`. In the `unchecked` state, the box is empty. A tick mark is present in the box when it is in the `checked` state. In the `undefined` state, a horizontal line is present in the box.

1.3.6.1 The Code

FxCheckBoxExample.java

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxCheckBoxExample extends Application
{
        // Create the Selection Label
        Label selectionMsg = new Label("Choose your Car");

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

        @Override
        public void start(Stage stage)
        {
                // Create a CheckBox to support only two states
                CheckBox fordCbx = new CheckBox("Ford");
                // Create a CheckBox to support three states
                CheckBox audiCbx = new CheckBox("Audi");
                audiCbx.setAllowIndeterminate(true);

                // Add a ChangeListener to the CheckBox fordCbx
                fordCbx.selectedProperty().addListener(new ChangeListener<Boolean>()
                {
                    public void changed(ObservableValue<? extends Boolean> ov,
                                final Boolean value, final Boolean newValue)
                        {
                        if(newValue != null && newValue)
                        {
                                printMessage("Your Selection: Ford");
                        }
                        }
                });

                // Add a ChangeListener to the CheckBox audiCbx
                audiCbx.selectedProperty().addListener(new ChangeListener<Boolean>()
                {
                    public void changed(ObservableValue<? extends Boolean> ov,
                                final Boolean value, final Boolean newValue)
                        {
                        if(newValue != null && newValue)
                        {
                                printMessage("Your Selection: Audi");
                        }
                        }
                });

                // Add a ChangeListener to the CheckBox audiCbx
                audiCbx.indeterminateProperty().addListener(new ChangeListener<Boolean>()
                {
                    public void changed(ObservableValue<? extends Boolean> ov,
```

```
                                final Boolean value, final Boolean newValue)
                        {
                        if(newValue != null && newValue)
                        {
                                printMessage("Your indeterminate Selection: Audi");
                        }
                        }
                });

                // Create the VBox
                VBox root = new VBox();
                // Add the children to the VBox
                root.getChildren().addAll(selectionMsg, fordCbx, audiCbx);
                // Set the vertical spacing between children to 20px
                root.setSpacing(20);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A CheckBox Example");
                // Display the Stage
                stage.show();
        }

        // Helper Method
        public void printMessage(String message)
        {
                // Set the Text of the Label
                selectionMsg.setText(message);
        }
}
```

By default, the `CheckBox` control supports only the two states `checked` and `unchecked`. The `allowIndeterminate` property specifies whether the third state (the undefined state) is available for selection. By default, it is set to false.

```
// Create a CheckBox to support three states
CheckBox audiCbx = new CheckBox("Audi");
audiCbx.setAllowIndeterminate(true);
```

1.3.6.2 The ObservableValue Class

The ObservableValue interface inherits from the Observable interface. An `ObservableValue` wraps a value, which can be observed for changes. It has a `getValue()` method that returns the value it wraps. It generates invalidation events and change events. Invalidation events are generated when the value in the `ObservableValue` is no longer valid. Change events are generated when the value changes. You can register a ChangeListener to an `ObservableValue`. The `changed()` method of the `ChangeListener` is called every time the value of its value changes. The `changed()` method receives three arguments:

- The reference of the ObservableValue
- The old value
- The new value

The following code snippet shows an example of the usage of an `ObservableValue`:

```
// Add a ChangeListener to the CheckBox audiCbx
audiCbx.indeterminateProperty().addListener(new ChangeListener<Boolean>()
{
        public void changed(ObservableValue<? extends Boolean> ov,
                        final Boolean value, final Boolean newValue)
        {
                if(newValue != null && newValue)
                {
                        printMessage("Your indeterminate Selection: Audi");
                }
        }
});
```

1.3.6.3 The GUI

After starting the application, the following window appears:

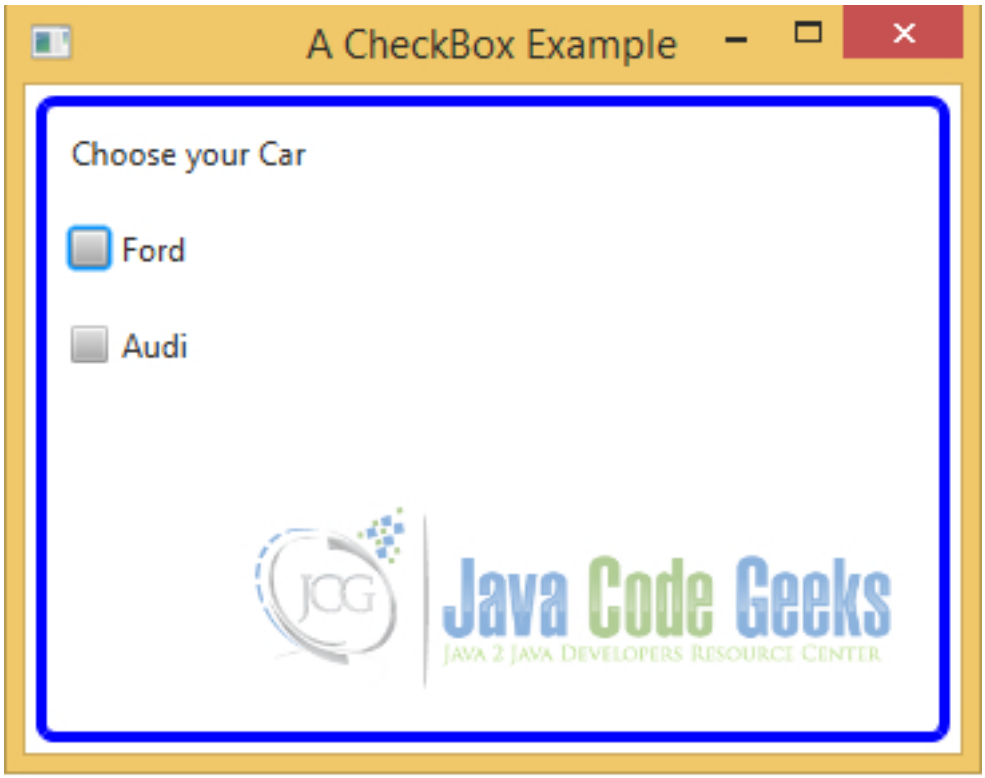

Figure 1.9: A CheckBox Example before Selection

If you make a click on a specific `CheckBox`, the following message apperas in our example:

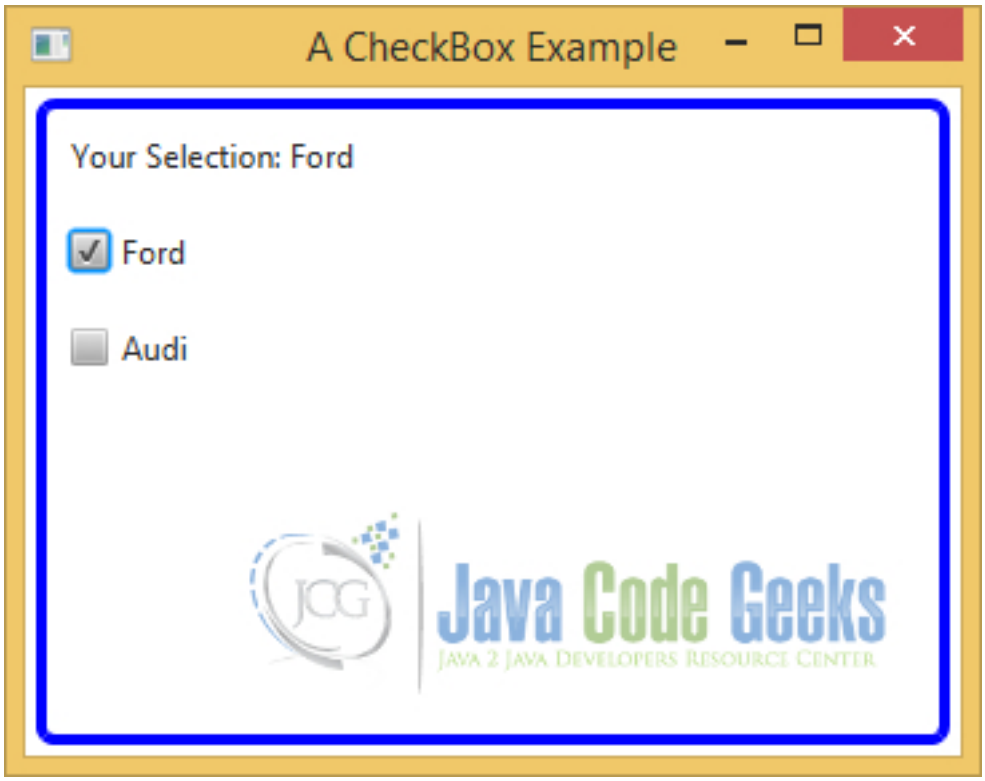

Figure 1.10: A CheckBox Example after Selection

1.3.7 ToggleButton

`ToggleButton` is a two-state button control. The two states are:

- Selected
- Unselected

Its `selected` property indicates whether it is selected. The `selected` property is true when it is in the selected state. Otherwise, it is false. When it is in the selected state, it stays depressed. You can toggle between the selected and unselected states by pressing it, and hence it got the name `ToggleButton`. For `ToggleButtons`, mnemonic parsing is enabled by default.

1.3.7.1 The Code

FxToggleButtonExample.java

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Toggle;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
```

```
public class FxToggleButtonExample extends Application
{
        // Create the Message Label
        Label selectionMsg = new Label("Your selection: None");

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

        @Override
        public void start(Stage stage)
        {
                // Create four ToggleButtons
                ToggleButton fordBtn = new ToggleButton("Ford");
                ToggleButton audiBtn = new ToggleButton("Audi");
                ToggleButton ferrariBtn = new ToggleButton("Ferrari");
                ToggleButton porscheBtn = new ToggleButton("Porsche");

                // Create a ToggleGroup
                final ToggleGroup group = new ToggleGroup();
                // Add all ToggleButtons to a ToggleGroup
                group.getToggles().addAll(fordBtn, audiBtn, ferrariBtn, porscheBtn);

                // Create a ChangeListener for the ToggleGroup
                group.selectedToggleProperty().addListener(new ChangeListener<Toggle>(){
                    public void changed(ObservableValue<? extends Toggle> ov,
                            final Toggle toggle, final Toggle new_toggle)
                    {
                        String toggleBtn = ((ToggleButton)new_toggle).getText();
                        selectionMsg.setText("Your selection: " + toggleBtn);
            }});

                // Create the Label for the Selection
                Label selectLbl = new Label("Select the car you like:");

                // Create a HBox
                HBox buttonBox = new HBox();
                // Add ToggleButtons to an HBox
                buttonBox.getChildren().addAll(fordBtn, audiBtn, ferrariBtn, porscheBtn);
                // Set the spacing between children to 10px
                buttonBox.setSpacing(10);

                // Create the VBox
                VBox root = new VBox();
                // Add the Labels and HBox to the VBox
                root.getChildren().addAll(selectionMsg, selectLbl, buttonBox);
                // Set the spacing between children to 10px
                root.setSpacing(10);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
```

```
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A ToggleButton Example");
                // Display the Stage
                stage.show();
        }
}
```

1.3.7.2 The GUI

After starting the application, the following window appears:

Figure 1.11: A ToggleButton Example before pressing any Button

After pressing any `ToggleButton`, the message will be changed:

Figure 1.12: A ToggleButton Example after pressing any Button

1.3.8 RadioButton

An instance of the `RadioButton` class represents a radio button. It inherits from the `ToggleButton` class. Therefore, it has all of the features of a toggle button. A radio button is rendered differently compared to a toggle button. Like a toggle button, a radio button can be in one of the two states:

- Selected
- Unselected

Its `selected` property indicates its current state. Like a toggle button, its mnemonic parsing is enabled by default. Like a toggle button, it also sends an `ActionEvent` when it is selected and unselected.

1.3.8.1 The Code

FxRadioButtonExample.java

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.Toggle;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
```

```
public class FxRadioButtonExample extends Application
{
        // Create the Selection Label
        Label selectionMsg = new Label("Your selection: None");

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

        @Override
        public void start(Stage stage)
        {
                // Create the RadioButtons
                RadioButton fordBtn = new RadioButton("Ford");
                RadioButton audiBtn = new RadioButton("Audi");
                RadioButton ferrariBtn = new RadioButton("Ferrari");
                RadioButton porscheBtn = new RadioButton("Porsche");

                // Create a ToggleGroup
                ToggleGroup group = new ToggleGroup();
                // Add all RadioButtons to a ToggleGroup
                group.getToggles().addAll(fordBtn, audiBtn, ferrariBtn, porscheBtn);

                // Add a listener to the ToggleGroup
                group.selectedToggleProperty().addListener(new ChangeListener<Toggle>(){
                    public void changed(ObservableValue<? extends Toggle> ov,
                            final Toggle toggle, final Toggle new_toggle)
                    {
                        String toggleBtn = ((ToggleButton)new_toggle).getText();
                        selectionMsg.setText("Your selection: " + toggleBtn);
            }});

                // Select the default car as ferrari
                ferrariBtn.setSelected(true);

                // Create the Selection Label
                Label msg = new Label("Select the car you like the most:");

                // Create a HBox
                HBox buttonBox = new HBox();
                // Add RadioButtons to an HBox
                buttonBox.getChildren().addAll(fordBtn, audiBtn, ferrariBtn, porscheBtn);
                // Set the spacing between children to 10px
                buttonBox.setSpacing(10);

                // Create the VBox
                VBox root = new VBox();
                // Add Labels and RadioButtons to an VBox
                root.getChildren().addAll(selectionMsg, msg, buttonBox);
                // Set the spacing between children to 10px
                root.setSpacing(10);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
```

```
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A RadioButton Example");
                // Display the Stage
                stage.show();
        }
}
```

There is a significant difference in the use of radio buttons compared to the use of toggle buttons. Recall that when toggle buttons are used in a group, there may not be any selected toggle button in the group. When radio buttons are used in a group, there must be one selected radio button in the group. Unlike a toggle button, clicking a selected radio button in a group does not unselect it. To enforce the rule that one radio button must be selected in a group of radio buttons, one radio button from the group is selected programmatically by default.

1.3.8.2 The GUI

After starting the application, the following window appears:

Figure 1.13: A RadioButton Example before pressing any Button

After pressing any `RadioButton`, the message will be changed:

Figure 1.14: A RadioButton Example after pressing any Button

1.3.9 ChoiceBox

A ChoiceBox is used to let a user select an item from a small list of items. The items may be any type of objects. A `ChoiceBox` is a parameterized class. The parameter type is the type of the items in its list. If you want to store mixed types of items in a `ChoiceBox`, you can use its raw type, as shown in the following code:

1.3.9.1 The Code

FxChoiceBoxExample.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FxChoiceBoxExample extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
```

```
                // Create the Label for the Car
                Label carLbl = new Label("Car:");

                // Create a ChoiceBox for cars
                ChoiceBox<String> cars = new ChoiceBox<>();
                // Add the items to the ChoiceBox
                cars.getItems().addAll("Ford", "Audi", "Ferrari", "Porsche");

                // Create the Selection Message Label
                Label selectionMsgLbl = new Label("Your selection:");

                // Create the Selection Value Label
                Label selectedValueLbl = new Label();
                // Bind the value property to the text property of the Label
                selectedValueLbl.textProperty().bind(cars.valueProperty());

                // Display controls in a GridPane
                GridPane root = new GridPane();
                // Set the spacing between columns and rows
                root.setVgap(10);
                root.setHgap(10);

                // Add the Labels and the ChoiceBox to the GridPane
                root.addRow(0, carLbl, cars);
                root.addRow(1, selectionMsgLbl, selectedValueLbl);

                // Set the Size of the GridPane
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the GridPane
                 * Set the border-style of the GridPane
                 * Set the border-width of the GridPane
                 * Set the border-insets of the GridPane
                 * Set the border-radius of the GridPane
                 * Set the border-color of the GridPane
                */
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A ChoiceBox Example");
                // Display the Stage
                stage.show();
        }
}
```

1.3.9.2 The GUI

After starting the application, the following window appears:

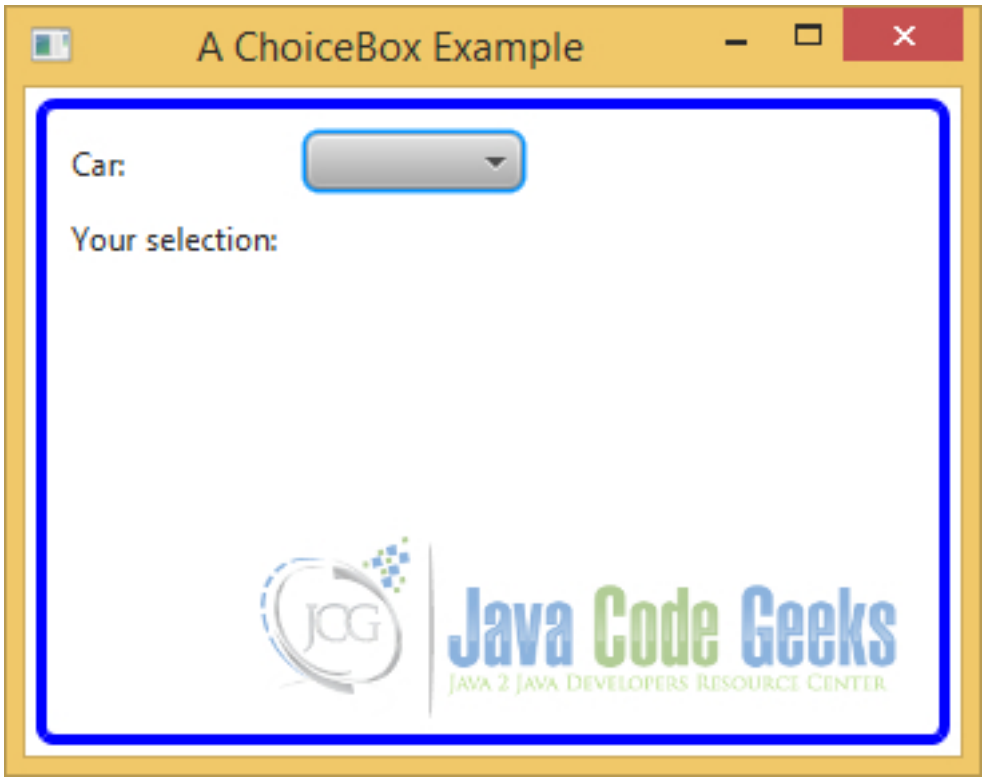

Figure 1.15: A ChoiceBox Example before Selection

After selecting an item, the message will be changed:

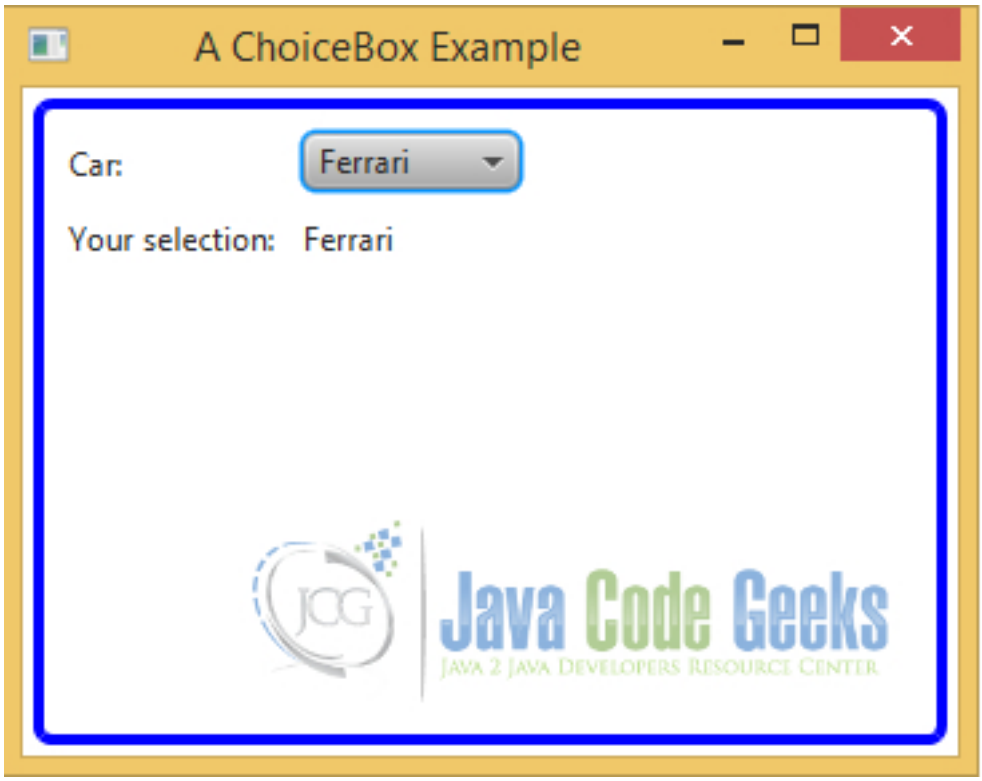

Figure 1.16: A ChoiceBox Example after Selection

1.3.10 ComboBox

A ComboBox is used to let a user select an item from a list of items. You can think of `ComboBox` as an advanced version of `ChoiceBox`. A `ComboBox` is highly customizable. The `ComboBox` class inherits from ComboBoxBase class, which provides the common functionality for all ComboBox-like controls, such as `ComboBox`, ColorPicker, and DatePicker. If you want to create a custom control that will allow users to select an item from a pop-up list, you need to inherit your control from the `ComboBoxBase` class.

1.3.10.1 The Code

FxComboBoxExample.java

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxComboBoxExample extends Application
{
        // Create the Selection Label
        Label selectionLbl = new Label("Your selection: None");

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

```
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Label for the Car
                Label carLbl = new Label("Car:");

                // Create a ComboBox for cars
                ComboBox<String> cars = new ComboBox<>();
                // Add the items to the ComboBox
                cars.getItems().addAll("Ford", "Audi", "Ferrari", "Porsche");
                // Select the first car from the list
                cars.getSelectionModel().selectFirst();

                // Add a ChangeListener to the ComboBox
                cars.getSelectionModel().selectedItemProperty().addListener(new  ↩
                    ChangeListener<String>()
                {
                    public void changed(ObservableValue<? extends String> ov, final String  ↩
                        oldValue, final String newValue)
                    {
                        selectionLbl.setText("Your selection: " + newValue);
                    }
                });

                // Create the HBox
                HBox carbox = new HBox();
                // Add the children to the HBox
                carbox.getChildren().addAll(carLbl, cars);
                // Set the vertical spacing between children to 10px
                carbox.setSpacing(10);

                // Create the VBox
                VBox root = new VBox();
                // Add the children to the VBox
                root.getChildren().addAll(carbox, selectionLbl);
                // Set the vertical spacing between children to 10px
                root.setSpacing(10);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
```

```
                // Set the title of the Stage
                stage.setTitle("A ComboBox Example");
                // Display the Stage
                stage.show();
        }

}
```

1.3.10.2 The GUI

After starting the application, the following window appears:

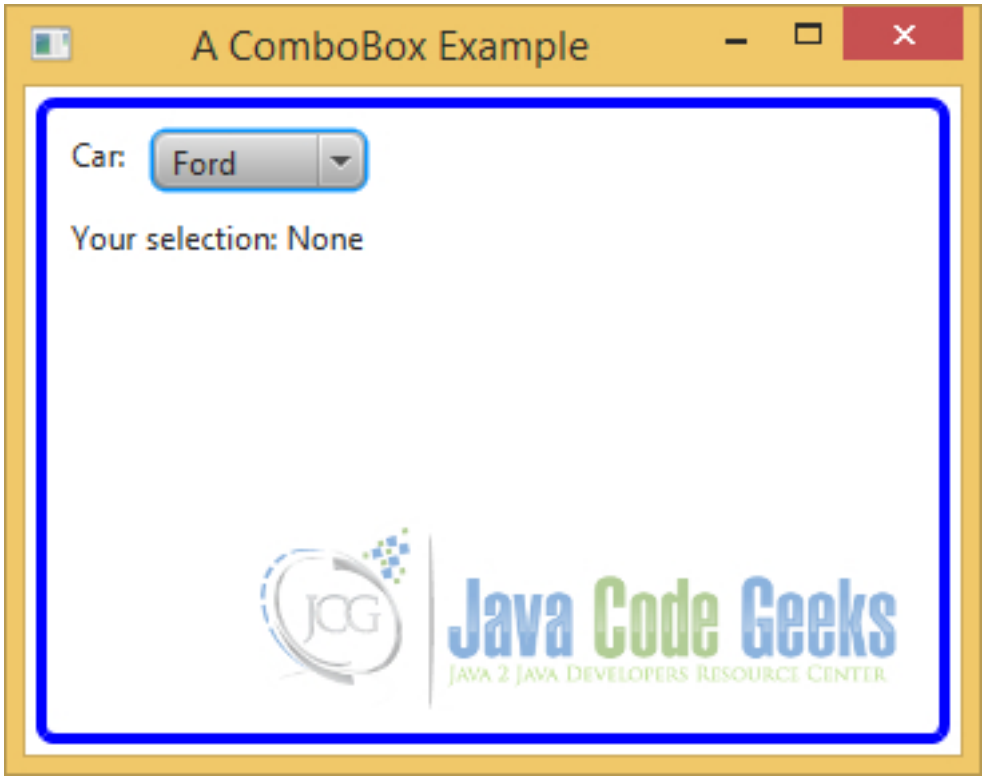

Figure 1.17: A ComboBox Example before Selection

After selectang an item, the message will be changed:

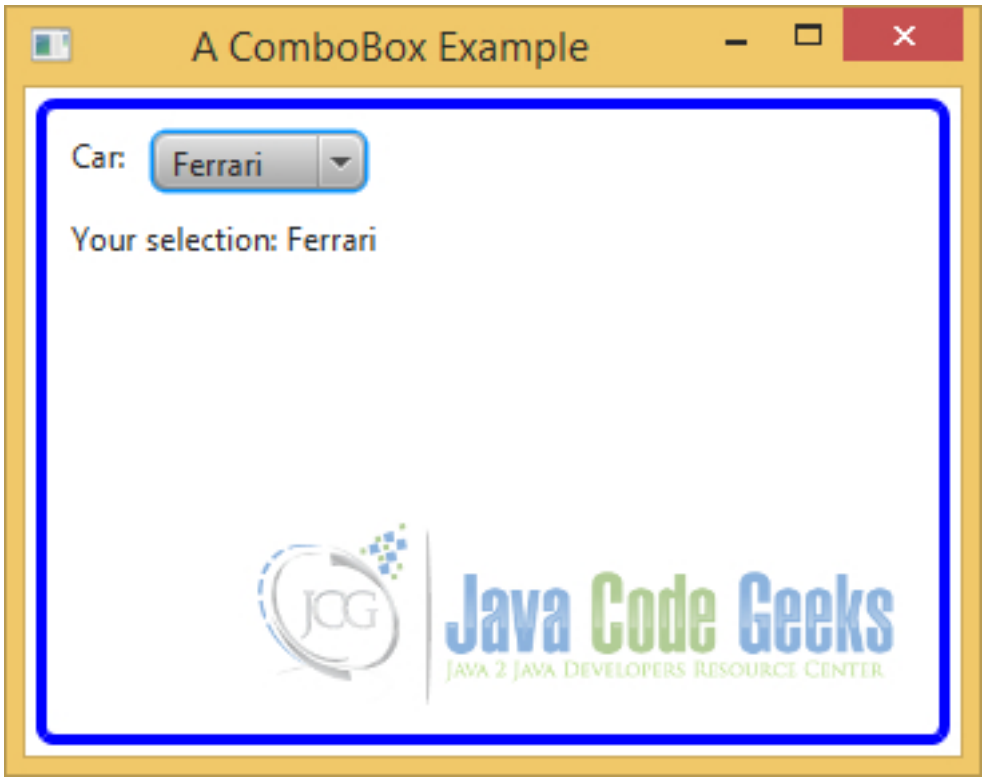

Figure 1.18: A ComboBox Example after Selection

1.3.11 ListView

A ListView is used to allow a user to select one item or multiple items from a list of items. Each item in `ListView` is represented by an instance of the ListCell class, which can be customized. The items list in a `ListView` may contain any type of objects. `ListView` is a parameterized class. The parameter type is the type of the items in the list.

1.3.11.1 The Code

FxListViewExample.java

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FxListViewExample extends Application
{
        // Create the Selection Label
        Label selectionLbl = new Label("Your selection: None");

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

```
        @Override
        public void start(Stage stage)
        {
                // Create the ListView
                ListView<String> cars = new ListView<String>();
                // Add Items to the ListView
                cars.getItems().addAll("Ford", "Audi", "Ferrari", "Porsche");
                // Select the first car from the list
                cars.getSelectionModel().selectFirst();
                // Add ChangeListener to the ListView
                cars.getSelectionModel().selectedItemProperty().addListener(new ↩
                    ChangeListener<String>()
                {
                    public void changed(ObservableValue<? extends String> ov, final String ↩
                        oldValue, final String newValue)
                    {
                        selectionLbl.setText("Your selection: " + newValue);
                    }
                });

                // Create the GridPane
                GridPane root = new GridPane();
                // Set the horizontal and vertical spacing between columns and rows
                root.setVgap(10);
                root.setHgap(10);
                // Add ListView and Label to the GridPane
                root.addRow(0, cars);
                root.addRow(1, selectionLbl);
                // Set the Size of the GridPane
                root.setMinSize(300, 200);

                /*
                 * Set the padding of the GridPane
                 * Set the border-style of the GridPane
                 * Set the border-width of the GridPane
                 * Set the border-insets of the GridPane
                 * Set the border-radius of the GridPane
                 * Set the border-color of the GridPane
                */
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A ListView Example");
                // Display the Stage
                stage.show();
        }

}
```

1.3.11.2 The GUI

After starting the application, the following window appears:

Figure 1.19: A ListView Example before Selection

After selecting an item in the list, the message will be changed:

Figure 1.20: A ListView Example after Selection

1.3.12 TextArea

A TextArea is a text input control. It inherits from the A `TextInputControl` class. It lets the user enter multiline plain text. If you need a control to enter a single line of plain text, use `TextField` instead. If you want to use rich text, use the HTMLEditor control. Unlike the `TextField`, newline and tab characters in the text are preserved. A newline character starts a new paragraph in a `TextArea`.

1.3.12.1 The Code

FxComboBoxExample.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxTextAreaExample extends Application
{
        // Create the Message TextArea
        TextArea messageArea = new TextArea();

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

        @Override
        public void start(Stage stage)
        {
                // Create the TextField for the Input
                final TextField input = new TextField();
                input.setPromptText("Input your message here");

                // Set the Prompt and Size of the TextArea
                messageArea.setPromptText("Your Message:");
                messageArea.setPrefColumnCount(20);
                messageArea.setPrefRowCount(10);

                // Create the Print Button
                Button printBtn = new Button("Print Message");
                // Add an EvenetHandler to the Button
                printBtn.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                messageArea.appendText(input.getText()+ "\n");
            }
        });

                // Create the VBox
                VBox root = new VBox();
                // Add Labels, TextArea and TextField to the VBox
                root.getChildren().addAll(new Label("Input:"), input, new Label("Messages:" ↩
                    ), messageArea, printBtn);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
```

```
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A TextArea Example");
                // Display the Stage
                stage.show();
        }
}
```

1.3.12.2 The GUI

After starting the application, the following window appears:

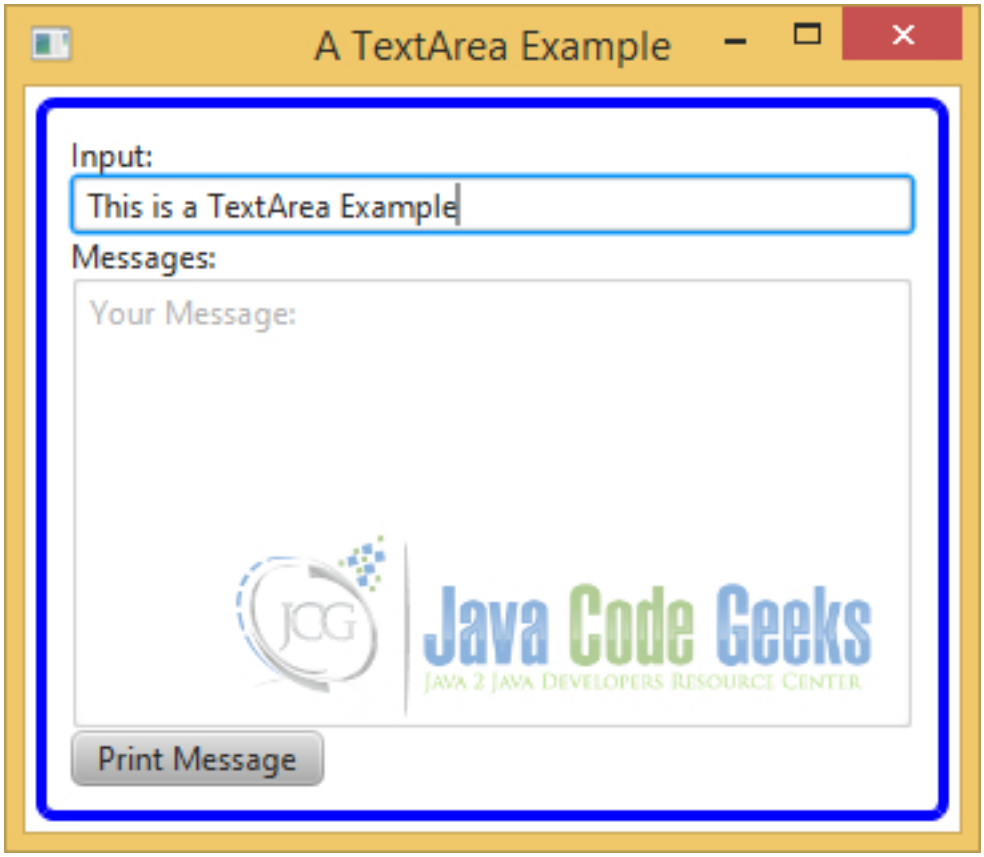

Figure 1.21: A TextArea Example before Inserting a Text

After inserting a text in the `TextField`, the `TextArea` contains the inserted data:

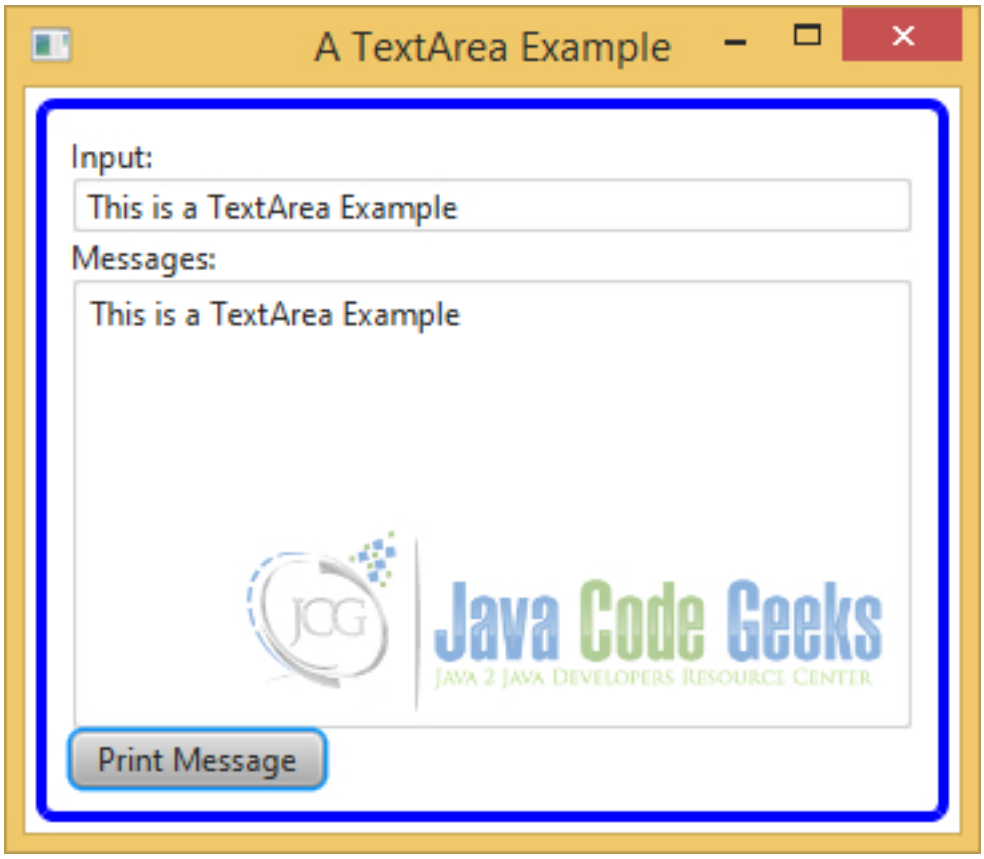

Figure 1.22: A TextArea Example after Inserting a Text

1.3.13 Menu

A Menu is used to provide a list of actionable items to the user in a compact form. You can also provide the same list of items using a group of buttons, where each button represents an actionable item. It is a matter of preference which one you use: a menu or a group of buttons.

There is a noticeable advantage of a using a menu. It uses much less space on the screen, compared to a group of buttons, by folding (or nesting) the group of items under another item.

For example, if you have used a file editor, the menu items such as New, Open, Save, and Print are nested under a top-level File menu.

A user needs to click the File menu to see the list of items that are available under it. Typically, in cases of a group of buttons, all items are visible to the user all the time, and it is easy for users to know what actions are available. Therefore, there is little tradeoff between the amount of space and usability when you decide to use a menu or buttons. Typically, a menu bar is displayed at the top of a window.

Using a `menu` is a multistep process. The following sections describe the steps in detail. The following is the summary of steps:

- 1. Create a menu bar and add it to a container.
- 2. Create menus and add them to the menu bar.
- 3. Create menu items and add them to the menus.
- 4. Add ActionEvent handlers to the menu items to perform actions when they are clicked.

1.3.13.1 The Code

FxMenuExample.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxMenuExample extends Application
{
        // Create the Message Label
        Label messageLbl = new Label("Press any Menu Item to see the message");

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

        @Override
        public void start(Stage stage)
        {
                // Create some menus
                Menu fileMenu = new Menu("File");
                Menu editMenu = new Menu("Edit");

                // Create the MenuItem New
                MenuItem newItem = new MenuItem("New");
                newItem.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have pressed the New Menu Item");
            }
        });

                // Create the MenuItem Open
                MenuItem openItem = new MenuItem("Open");
                openItem.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have pressed the Open Menu Item");
            }
        });

                // Add menu items to the File menu
                fileMenu.getItems().addAll(newItem, openItem);

                // Create the MenuItem Copy
                MenuItem copyItem = new MenuItem("Copy");
                copyItem.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
```

```
                printMessage("You have pressed the Copy Menu Item");
            }
        });

                // Create the MenuItem Paste
                MenuItem pasteItem = new MenuItem("Paste");
                pasteItem.setOnAction(new EventHandler<ActionEvent>()
                {
            @Override public void handle(ActionEvent e)
            {
                printMessage("You have pressed the Paste Menu Item");
            }
        });

                // Add menu items to the Edit menu
                editMenu.getItems().addAll(copyItem, pasteItem);

                // Create a menu bar
                MenuBar menuBar = new MenuBar();
                // Add menus to a menu bar
                menuBar.getMenus().addAll(fileMenu, editMenu);

                // Create the Menu Box
                HBox menu = new HBox();
                // Add the MenuBar to the Menu Box
                menu.getChildren().add(menuBar);

                // Create the VBox
                VBox root = new VBox();
                // Add the children to the VBox
                root.getChildren().addAll(menu,messageLbl);
                // Set the Size of the VBox
                root.setMinSize(350, 250);

                /*
                 * Set the padding of the VBox
                 * Set the border-style of the VBox
                 * Set the border-width of the VBox
                 * Set the border-insets of the VBox
                 * Set the border-radius of the VBox
                 * Set the border-color of the VBox
                */
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Menu Example");
                // Display the Stage
                stage.show();
        }

        // Helper Method
        public void printMessage(String message)
        {
```

```
                // Set the Text of the Label
                messageLbl.setText(message);
        }

}
```

1.3.13.2 Using Menu Bars

A menu bar is a horizontal bar that acts as a container for menus. An instance of the MenuBar class represents a menu bar. You can create a `MenuBar` using its default constructor:

```
// Create a menu bar
MenuBar menuBar = new MenuBar();
// Add menus to a menu bar
menuBar.getMenus().addAll(fileMenu, editMenu);
```

1.3.13.3 Using Menus

A menu contains a list of actionable items, which are displayed on demand, for example, by clicking it. The list of menu items is hidden when the user selects an item or moves the mouse pointer outside the list. A menu is typically added to a menu bar or another menu as a submenu. An instance of the Menu class represents a menu. A menu displays text and a graphic.

```
// Create some menus
Menu fileMenu = new Menu("File");
Menu editMenu = new Menu("Edit");
```

1.3.13.4 Using Menu Items

A MenuItem represents an actionable option. When it is clicked, the registered `ActionEvent` handlers are called.

The following snippet of code creates an New MenuItem and adds an ActionEvent handler that prints out the message:

```
// Create the MenuItem New
MenuItem newItem = new MenuItem("New");
newItem.setOnAction(new EventHandler<ActionEvent>()
{
        @Override public void handle(ActionEvent e)
        {
                printMessage("You have pressed the New Menu Item");
        }
});
```

A `MenuItem` is added to a menu. A menu stores the reference of its items in an `ObservableList` whose reference can be obtained using the `getItems()` method:

```
// Add menu items to the Edit menu
editMenu.getItems().addAll(copyItem, pasteItem);
```

1.3.13.5 The GUI

After starting the application, the following window appears:

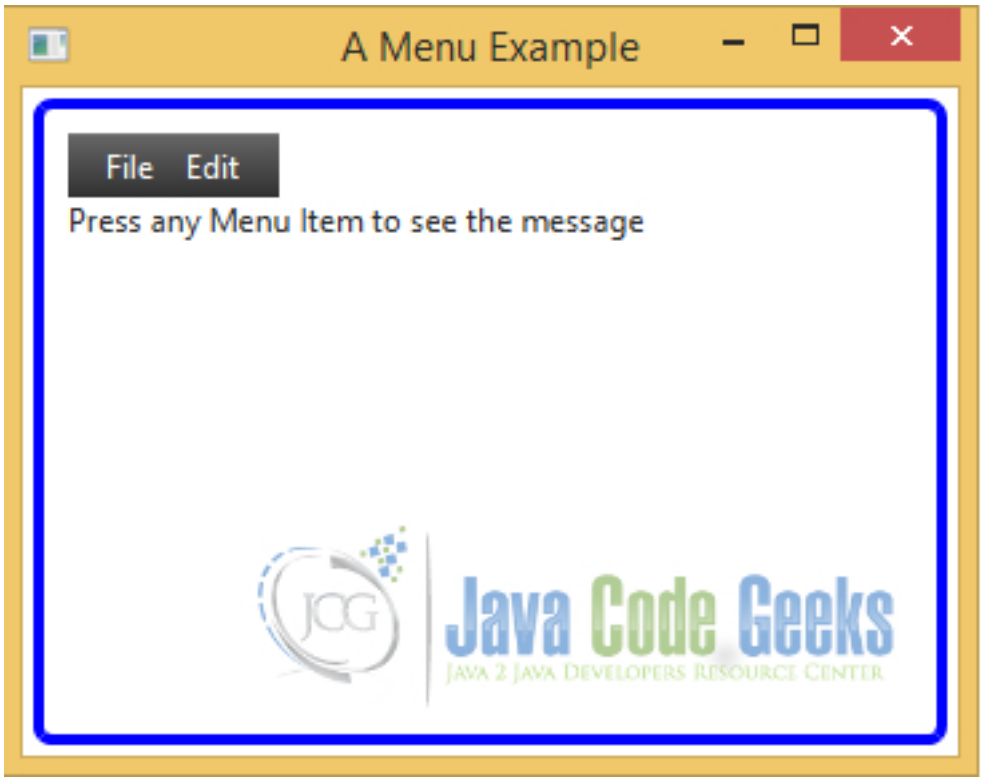

Figure 1.23: A Menu Example before choosing an Item

After selecting a specific `MenuItem`, the message will be changed:

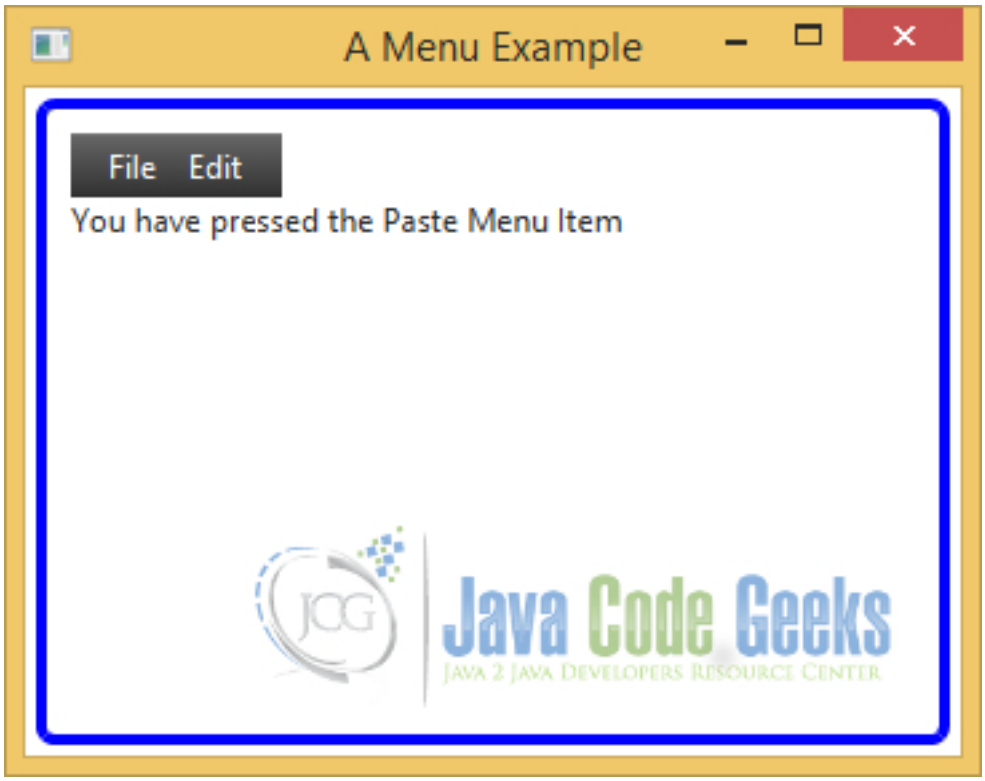

Figure 1.24: A Menu Example after choosing an Item

1.4 Download Java Source Code

This was a JavaFX tutorial with a short Inroduction and an Example about the most important classes.

Download

You can download the full source code of this tutorial here: **JavaFxTutorial.zip**

Chapter 2

The JavaFX Media API

This is an article about the JavaFX Media API. JavaFX supports playing audio and video through the JavaFX Media API. HTTP live streaming of static media files and live feeds are also supported.

A number of media formats are supported, including AAC, AIFF, WAV, and MP3. FLV containing VP6 video and MP3 audio and MPEG-4 multimedia container with H.264/AVC video formats are also supported.

The support for a specific media format is platform dependent. Some media playback features and formats do not require any addition installations; some require third-party software to be installed.

The following examples use Java SE 7 and JavaFX 2.2.

2.1 Introduction

The Media API consists of the following classes:

- AudioClip
- Media
- MediaPlayer
- MediaView
- MediaErrorEvent
- MediaException

AudioClip is used to play a short audio clip with minimal latency. Typically, this is useful for sound effects, which are usually short audio clips.

Use the Media, MediaPlayer, and MediaView classes for playing audios and videos of longer length.

The `Media` and `MediaPlayer` classes are used to play audios as well as videos. An instance of the `Media` class represents a media resource, which could be an audio or video. It provides the information about the media, for example, the duration of the media.

An instance of the `MediaPlayer` class provides controls for playing a media.

An instance of the `MediaView` class provides the view of a media being played by a `MediaPlayer`. A `MediaView` is used for viewing a video.

Several things can go wrong when you attempt to play a media, for example, the media format may not be supported or the media content may be corrupt.

An instance of the MediaException class represents a specific type of media error that may occur during media playback. When a media-related error occurs, a MediaErrorEvent is generated. You can handle the error by adding an appropriate event handler to the media objects.

2.2 Playing Audio Clips

2.2.1 The Code

FxMediaExample1.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.AudioClip;
import javafx.stage.Stage;

public class FxMediaExample1 extends Application
{
        private AudioClip audioClip;

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

        @Override
        public void init()
        {
                // Create an AudioClip, which loads the audio data synchronously
                final URL resource = getClass().getResource("Test.mp3");
                audioClip = new AudioClip(resource.toExternalForm());
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Buttons
                Button playButton = new Button("Play");
                Button stopButton = new Button("Stop");

                // Create the Sliders
                final Slider cycleSlider = new Slider(1, 5, 1);
                cycleSlider.setMajorTickUnit(1);
                cycleSlider.setShowTickLabels(true);

                final Slider volumeSlider = new Slider(0.0, 1.0, 0.5);
                volumeSlider.setMajorTickUnit(0.1);
                volumeSlider.setShowTickLabels(true);

                final Slider rateSlider = new Slider(0, 8, 4);
                rateSlider.setMajorTickUnit(1);
                rateSlider.setShowTickLabels(true);

                final Slider balanceSlider = new Slider(-1.0, 1.0, 0.0);
                balanceSlider.setMajorTickUnit(0.2);
                balanceSlider.setShowTickLabels(true);
```

```
            final Slider panSlider = new Slider(-1.0, 1.0, 0.0);
            panSlider.setMajorTickUnit(0.2);
            panSlider.setShowTickLabels(true);

            final Slider prioritySlider = new Slider(0.0, 10.0, 0.0);
            prioritySlider.setMajorTickUnit(1);
            prioritySlider.setShowTickLabels(true);

            // Create the Event Handlers for the Button
            playButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            audioClip.play();
        }
    });

            stopButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            audioClip.stop();
        }
    });

            // Bind the Properties
            audioClip.cycleCountProperty().bind(cycleSlider.valueProperty());
            audioClip.volumeProperty().bind(volumeSlider.valueProperty());
            audioClip.rateProperty().bind(rateSlider.valueProperty());
            audioClip.balanceProperty().bind(balanceSlider.valueProperty());
            audioClip.panProperty().bind(panSlider.valueProperty());
            audioClip.priorityProperty().bind(prioritySlider.valueProperty());

            // Create the GridPane
            GridPane sliderPane = new GridPane();
            // Set horizontal and vertical Spacing
            sliderPane.setHgap(5);
            sliderPane.setVgap(10);

            // Add the details to the GridPane
            sliderPane.addRow(0, new Label("CycleCount:"), cycleSlider);
            sliderPane.addRow(1, new Label("Volume:"), volumeSlider);
            sliderPane.addRow(2, new Label("Rate:"), rateSlider);
            sliderPane.addRow(3, new Label("Balance:"), balanceSlider);
            sliderPane.addRow(4, new Label("Pan:"), panSlider);
            sliderPane.addRow(5, new Label("Priority:"), prioritySlider);

            // Create the HBox
            HBox buttonBox = new HBox(5, playButton, stopButton);

            VBox root = new VBox(5,sliderPane, buttonBox);
            // Set the Sie of the VBox
            root.setPrefWidth(300);
            root.setPrefHeight(350);

            // Set the Style-properties of the HBox
            root.setStyle("-fx-padding: 10;" +
                          "-fx-border-style: solid inside;" +
                          "-fx-border-width: 2;" +
                          "-fx-border-insets: 5;" +
                          "-fx-border-radius: 5;" +
                          "-fx-border-color: blue;");
```

```
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("An AucioClip Example");
                // Display the Stage
                stage.show();
        }
}
```

An instance of the `AudioClip` class is used to play a short audio clip with minimal latency. Typically, this is useful for playing short audio clips, for example, a beep sound when the user makes an error or producing short sound effects in gaming applications.

The `AudioClip` class provides only one constructor that takes a URL in string form, which is the URL of the audio source. The audio clip is immediately loaded into memory in raw, uncompressed form. This is the reason why you should not use this class for long-playing audio clips.

The source URL could use the HTTP, file, and JAR protocols. This means that you can play an audio clip from the Internet, the local file system, and a JAR file.

The following snippet of code creates an `AudioClip`:

```
// Create an AudioClip, which loads the audio data synchronously
final URL resource = getClass().getResource("Test.mp3");
audioClip = new AudioClip(resource.toExternalForm());
```

When an `AudioClip` object is created, the audio data are loaded into the memory and they are ready to be played immediately. Use the `play()` method to play the audio and the `stop()` method to stop the playback:

```
/ Create the Event Handlers for the Button
playButton.setOnAction(new EventHandler <ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        audioClip.play();
    }
});

stopButton.setOnAction(new EventHandler <ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        audioClip.stop();
    }
});
```

The `AudioClip` class supports setting some audio properties when the clip is played:

- cycleCount
- volume
- rate
- balance
- pan
- priority

All of the above properties, except the `cycleCount`, can be set on the `AudioClip` class. Subsequent calls to the `play()` method will use them as defaults. The `play()` method may also override the defaults for a specific playback. The `cycleCount` property must be specified on the `AudioClip` and all subsequent playbacks will use the same value. The `cycleCount` specifies the number of times the clip is played when the `play()` method is called. It defaults to 1, which plays the clip only once.

The `volume` specifies the relative volume of the playback. The valid range is 0.0 to 1.0. A value of 0.0 represented muted, whereas 1.0 represents full volume.

The `rate` specifies the relative speed at which the audio is played. The valid range is 0.125 to 8.0. A value of 0.125 means the clip is played eight times slower, and the value of 8.0 means the clip will play eight times faster. The `rate` affects the playtime and the pitch. The default rate is 1.0, which plays the clip at the normal `rate`.

The `balance` specifies the relative volume for the left and right channels. The valid range is -1.0 to 1.0. A value of -1.0 sets the playback in the left channel at normal volume and mutes the right channel. A value of 1.0 sets the playback in the right channel at normal volume and mutes the left channel. The default value is 0.0, which sets the playback in both channels at normal volume.

The `pan` specifies distribution of the clip between the left and right channels. The valid range is -1.0 to 1.0. A value of -1.0 shifts the clip entirely to the left channel. A value of 1.0 shifts the clip entirely to the right channel. The default value is 0.0, which plays the clip normally. Setting the value for `pan` for a mono clip has the same effect of setting the balance. You should change the default for this property only for audio clips using stereo sound.

The `priority` specifies the priority of the clip relative to other clips. It is used only when the number of playing clips exceeds the system limits. The playing clips with the lower priority will be stopped. It can be set to any integer. The default priority is set to zero.

The `play()` method is overloaded. It has three versions:

- void play()
- void play(double volume)
- void play(double volume, double balance, double rate, double pan,int priority)

The no-args version of the method uses all of the properties set on the `AudioClip`. The other two versions can override the specified properties for a specific playback.

Suppose the volume for the `AudioClip` is set to 1.0. Calling `play()` will play the clip at the volume 1.0 and calling `play(0.20)` will play the clip at volume 0.20, leaving the volume property for the `AudioClip` unchanged at 1.0. That is, the `play()` method with parameters allows you to override the `AudioClip` properties on a per-playback basis.

The `AudioClip` class contains an `isPlaying()` method to check if the clip is still playing. It returns true is the clip is playing. Otherwise, it returns false.

2.2.2 The GUI

A simple Audio Player Example:

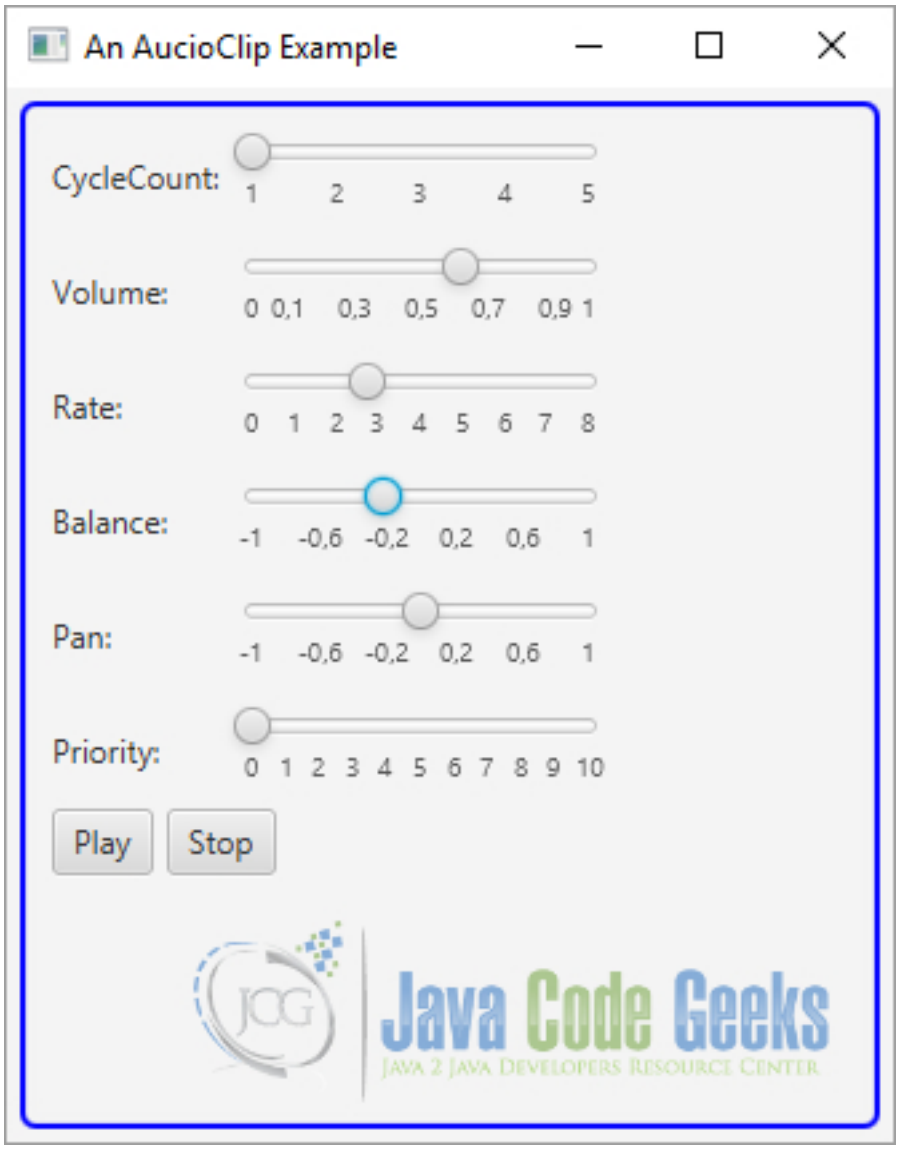

Figure 2.1: A simple JavaFX AudioClip Example

2.3 Playing Media

2.3.1 The Code

FxMediaExample2.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
```

```
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxMediaExample2 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp4");
                String mediaStringUrl = mediaUrl.toExternalForm();

                // Create a Media
                Media media = new Media(mediaStringUrl);

                // Create a Media Player
                final MediaPlayer player = new MediaPlayer(media);
                // Automatically begin the playback
                player.setAutoPlay(true);

                // Create a 400X300 MediaView
                MediaView mediaView = new MediaView(player);
                mediaView.setFitWidth(400);
                mediaView.setFitHeight(300);
                mediaView.setSmooth(true);

                // Create the DropShadow effect
                DropShadow dropshadow = new DropShadow();
                dropshadow.setOffsetY(5.0);
                dropshadow.setOffsetX(5.0);
                dropshadow.setColor(Color.WHITE);

                mediaView.setEffect(dropshadow);

                // Create the Buttons
                Button playButton = new Button("Play");
                Button stopButton = new Button("Stop");

                // Create the Event Handlers for the Button
                playButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                if (player.getStatus() == Status.PLAYING)
                {
                        player.stop();
                        player.play();
                }
                else
                {
                        player.play();
                }
            }
        });
```

```
                stopButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                player.stop();
            }
        });

                // Create the HBox
                HBox controlBox = new HBox(5, playButton, stopButton);

                // Create the VBox
                VBox root = new VBox(5,mediaView,controlBox);

                // Set the Style-properties of the HBox
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A simple Media Example");
                // Display the Stage
                stage.show();
        }
}
```

JavaFX provides a unified API to work with audio and videos. You use the same classes to work with both. The Media API internally treats them as two different types of media that is transparent to the API users.

The Media API contains three core classes to play back media:

- Media
- MediaPlayer
- MediaView

2.3.2 Creating a Media Object

An instance of the `Media` class represents a media resource, which could be an audio or a video. It provides the information related to the media, for example, the duration, metadata, data, and so forth.

If the media is a video, it provides the width and height of the video. A `Media` object is immutable. It is created by supplying a string which contains the path of the media resource, as in the following code:

```
// Locate the media content in the CLASSPATH
URL mediaUrl = getClass().getResource("Test.mp4");
String mediaStringUrl = mediaUrl.toExternalForm();

// Create a Media
Media media = new Media(mediaStringUrl);
```

The `Media` class contains the following properties, which are read-only:

- duration
- width
- height
- error
- onError

The `duration` specifies the duration of the media in seconds. It is a Duration object. If the duration is unknown, it is `Duration.UNKNOWN`.

The `width` and `height` give the width and height of the source media in pixels, respectively. If the media does not have width and height, they are set as zero.

The `error` and `onError` properties are related. The `error` property represents the `MediaException` that occurs during the loading of the media. The `onError` is a Runnable object that you can set to get notified when an error occurs. The `run()` method of the `Runnable` is called when an error occurs.

2.3.3 Creating a MediaPlayer Object

A `MediaPlayer` provides the controls, for example, play, pause, stop, seek, play speed, volume adjustment, for playing the media. The `MediaPlayer` provides only one constructor that takes a `Media` object as an argument:

```
// Create a Media Player
final MediaPlayer player = new MediaPlayer(media);
```

You can get the reference of the media from the `MediaPlayer` using the `getMedia()` method of the `MediaPlayer` class.

Like the `Media` class, the `MediaPlayer` class also contains `error` and `onError` properties to report errors. When an error occurs on the `MediaPlayer`, the same error is also reported on the `Media` object.

2.3.4 Creating a MediaView Node

A `MediaView` is a Node. It provides the view of a media being played by a `MediaPlayer`. Note that an audio clip does not have visuals. If you try creating a `MediaView` for an audio content, it would be empty. To watch a video, you create a `MediaView` and add it to a Scene Graph.

The `MediaView` class provides two constructors:

one no-args constructor and one that takes a `MediaPlayer` as an argument:

- public MediaView()
- public MediaView(MediaPlayer mediaPlayer)

The no-args constructor creates a `MediaView` that is attached to any `MediaPlayer`. You will need to set a `MediaPlayer` using the setter for the `mediaPlayer` property:

```
// Create a 400X300 MediaView
MediaView mediaView = new MediaView();
mediaView.setMediaPlayer(player);
mediaView.setFitWidth(400);
mediaView.setFitHeight(300);
```

The other constructor lets you specify a `MediaPlayer` for the `MediaView`:

```
// Create a 400X300 MediaView
MediaView mediaView = new MediaView(player);
mediaView.setFitWidth(400);
mediaView.setFitHeight(300);
```

2.3.5 Customizing the MediaView

If the media has a view (e.g., a video), you can customize the size, area, and quality of the video using the following properties:

- fitHeight
- fitWidth
- preserveRatio
- smooth
- viewport
- x
- y

The `fitWidth` and `fitHeight` properties specify the resized width and height of the video, respectively. By default, they are zero, which means that the original width and height of the media will be used.

```
mediaView.setfitWidth(400);
mediaView.setFitHeight(300);
```

The `preserveRatio` property specifies whether to preserve the aspect ratio of the media while resizing. By default, it is false.

The `smooth` property specifies the quality of the filtering algorithm to be used in resizing the video. The default value is platform dependent. If it is set to true, a better-quality filtering algorithm is used.

```
mediaView.setSmooth(true);
```

A `viewport` is a rectangular region to view part of a graphic. The `viewport`, `x`, and `y` properties together let you specify the rectangular area in the video that will be shown in the `MediaView`.

A `MediaView` is a `Node`. Therefore, to give a better visual experience to the audience, you can also apply effects and transformations to the `MediaView`.

2.3.6 Combining Media, MediaPlayer, and MediaView

The content of a media can be used simultaneously by multiple `Media` objects. However, one `Media` object can be associated with only one media content in its lifetime.

A `Media` object can be associated with multiple `MediaPlayer` objects. However, a `MediaPlayer` is associated with only one `Media` in its lifetime.

A `MediaView` may optionally be associated with a `MediaPlayer`. Of course, a `MediaView` that is not associated with a `MediaPlayer` does not have any visuals. The `MediaPlayer` for a `MediaView` can be changed.

Changing the `MediaPlayer` for a `MediaView` is similar to changing the channel on a television. The view for the `MediaView` is provided by its current `MediaPlayer`. You can associate the same `MediaPlayer` with multiple MediaViews:

Different MediaViews may display different parts of the same media during the playback.

2.3.7 The GUI

The following image shows the GUI of the `MediaPlayer`:

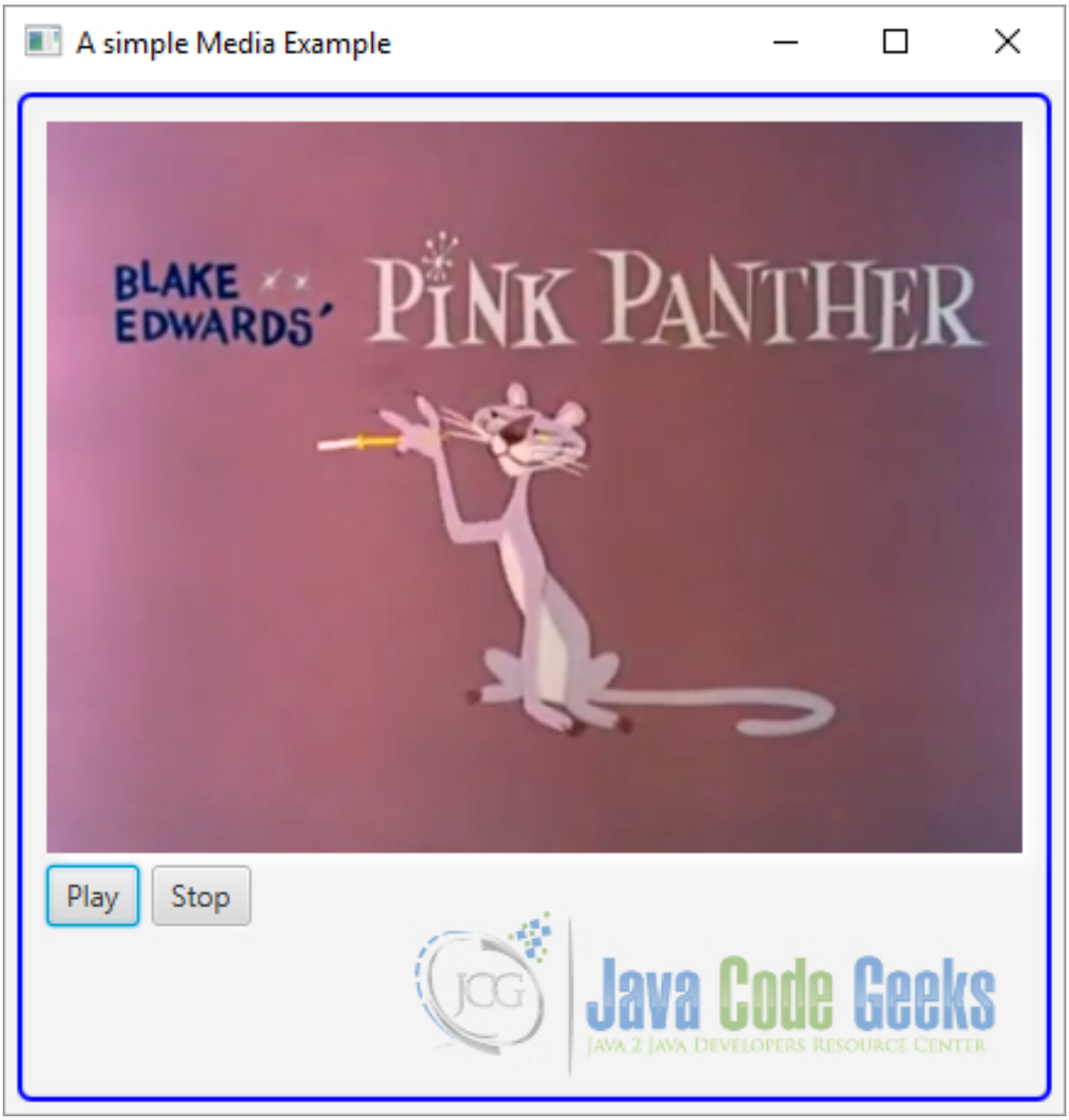

Figure 2.2: A JavaFX MediaView Example

2.4 Handling Playback Errors

2.4.1 The Code

FxMediaExample3.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.effect.DropShadow;
```

```
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaErrorEvent;
import javafx.scene.media.MediaException;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxMediaExample3 extends Application
{
        // Create the Area for Logging
        private TextArea messageArea = new TextArea();

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

        @Override
        public void start(Stage stage)
        {
                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp4");
                String mediaStringUrl = mediaUrl.toExternalForm();

                // Create a Media
                final Media media = new Media(mediaStringUrl);

                // Create a Media Player
                final MediaPlayer player = new MediaPlayer(media);
                // Automatically begin the playback
                player.setAutoPlay(true);

                // Create a 400X300 MediaView
                final MediaView mediaView = new MediaView(player);
                mediaView.setFitWidth(400);
                mediaView.setFitHeight(300);
                mediaView.setSmooth(true);

                // Create the DropShadow effect
                DropShadow dropshadow = new DropShadow();
                dropshadow.setOffsetY(5.0);
                dropshadow.setOffsetX(5.0);
                dropshadow.setColor(Color.WHITE);

                mediaView.setEffect(dropshadow);

                // Create the Buttons
                Button playButton = new Button("Play");
                Button stopButton = new Button("Stop");

                // Create the Event Handlers for the Button
                playButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                if (player.getStatus() == Status.PLAYING)
                {
                        player.stop();
```

```
                        player.play();
                }
                else
                {
                        player.play();
                }
        }
});

                stopButton.setOnAction(new EventHandler <ActionEvent>()
                {
        public void handle(ActionEvent event)
        {
                player.stop();
        }
});

                // Create Handlers for handling Errors
                player.setOnError(new Runnable()
                {
        public void run()
        {
                // Handle asynchronous error in Player object.
                printMessage(player.getError());
        }
});

                media.setOnError(new Runnable()
                {
        public void run()
        {
                // Handle asynchronous error in Media object.
                printMessage(media.getError());
        }
});

                mediaView.setOnError(new EventHandler <MediaErrorEvent>()
                {
                        public void handle(MediaErrorEvent event)
                        {
                // Handle asynchronous error in MediaView.
                printMessage(event.getMediaError());
        }
                });

                // Create the HBox
                HBox controlBox = new HBox(5, playButton, stopButton);

                // Create the VBox
                VBox root = new VBox(5,mediaView,controlBox,messageArea);

                // Set the Style-properties of the HBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
```

```
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("Handling Media Errors");
                // Display the Stage
                stage.show();
        }

        private void printMessage(MediaException error)
        {
                MediaException.Type errorType = error.getType();
                String errorMessage = error.getMessage();
                messageArea.appendText("\n" + "Type:" + errorType + ", error mesage:" +  ↩
                    errorMessage);
        }
}
```

An instance of the `MediaException` class, which inherits from the RuntimeException class, represents a media error that may occur in a `Media`, `MediaPlayer`, and `MediaView`.

`Media` playback may fail for a number of reasons. The API users should be able to identify specific errors. The `MediaException` class defines a static enum `MediaException.Type` whose constants identify the type of error. The `MediaException` class contains a `getType()` method that returns one of the constants of the `MediaException.Type` enum.

The constants in the `MediaException.Type` enum are listed below:

- MEDIA_CORRUPTED
- MEDIA_INACCESSIBLE
- MEDIA_UNAVAILABLE
- MEDIA_UNSPECIFIED
- MEDIA_UNSUPPORTED
- OPERATION_UNSUPPORTED
- PLAYBACK_HALTED
- PLAYBACK_ERROR
- UNKNOWN

The `MEDIA_CORRUPTED` error type indicates that the media is corrupted or invalid.

The `MEDIA_INACCESSIBLE` error type indicates that the media is inaccessible. However, the media may exist.

The `MEDIA_UNAVAILABLE` error type indicates that that media does not exist or it is unavailable.

The `MEDIA_UNSPECIFIED` error type indicates that the media has not been specified.

The `MEDIA_UNSUPPORTED` error type indicates that the media is not supported by the platform.

The `OPERATION_UNSUPPORTED` error type indicates that the operation performed on the media is not supported by the platform.

The `PLAYBACK_HALTED` error type indicates an unrecoverable error that has halted the playback.

The `PLAYBACK_ERROR` error type indicates a playback error that does not fall into any other described categories.

The `UNKNOWN` error type indicates that an unknown error has occurred.

The `Media` and `MediaPlayer` classes contain an error property that is a `MediaException`. All three classes contain an `onError` property, which is an event handler that is invoked when an error occurs. The types of the onError properties in these classes are not consistent.

It is a `Runnable` for the `Media` and `MediaPlayer` classes and the `MediaErrorEvent` for the `MediaView` class.

The following snippet of code shows how to handle errors on a `Media`, `MediaPlayer`, and `MediaView`.

```
// Create Handlers for handling Errors
player.setOnError(new Runnable()
{
    public void run()
    {
        // Handle asynchronous error in Player object.
        printMessage(player.getError());
    }
});

media.setOnError(new Runnable()
{
    public void run()
    {
        // Handle asynchronous error in Media object.
        printMessage(media.getError());
    }
});

mediaView.setOnError(new EventHandler <MediaErrorEvent>()
{
    public void handle(MediaErrorEvent event)
    {
        // Handle asynchronous error in MediaView.
        printMessage(event.getMediaError());
    }
});
```

`Media` error handlers are invoked on the JavaFX Application Thread. Therefore, it is safe to update the Scene Graph from the handlers.

It is recommended that you enclose the creation of the `Media`, `MediaPlayer`, and `MediaView` objects in a try-catch block and handle the exception appropriately. The `onError` handlers for these objects are involved after the objects are created. If an error occurs during the creation of these objects, those handlers will not be available.

2.4.2 The GUI

The following GUI shows a `MediaPlayer` with Error Handling:

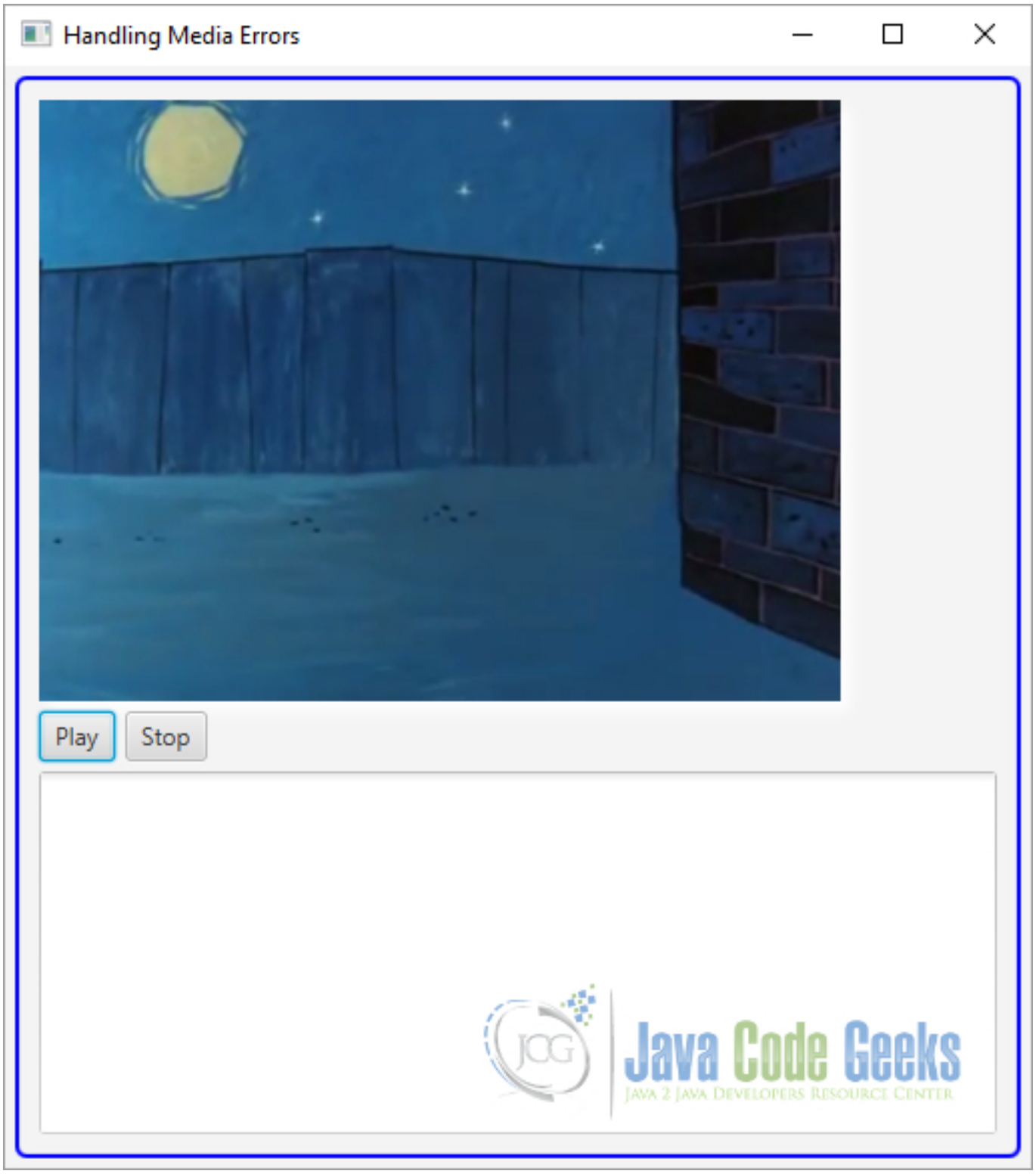

Figure 2.3: An JavaFX Media Example with Error Handling

2.5 State Transitions of the MediaPlayer

2.5.1 The Code

FxMediaExample4.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaErrorEvent;
import javafx.scene.media.MediaException;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxMediaExample4 extends Application
{
        // Create the Area for Logging
        private TextArea messageArea = new TextArea();

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

        @Override
        public void start(Stage stage)
        {
                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp4");
                String mediaStringUrl = mediaUrl.toExternalForm();

                // Create a Media
                final Media media = new Media(mediaStringUrl);

                // Create a Media Player
                final MediaPlayer player = new MediaPlayer(media);
                // Automatically begin the playback
                player.setAutoPlay(true);

                // Create a 400X300 MediaView
                final MediaView mediaView = new MediaView(player);
                mediaView.setFitWidth(400);
                mediaView.setFitHeight(300);
                mediaView.setSmooth(true);

                // Create the DropShadow effect
                DropShadow dropshadow = new DropShadow();
                dropshadow.setOffsetY(5.0);
```

```
            dropshadow.setOffsetX(5.0);
            dropshadow.setColor(Color.WHITE);

            mediaView.setEffect(dropshadow);

            // Create the Buttons
            Button playButton = new Button("Play");
            Button stopButton = new Button("Stop");

            // Create the Event Handlers for the Button
            playButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            if (player.getStatus() == Status.PLAYING)
            {
                    player.stop();
                    player.play();
            }
            else
            {
                    player.play();
            }
        }
    });

            stopButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            player.stop();
        }
    });

            // Create Handlers for handling Errors
            player.setOnError(new Runnable()
            {
        public void run()
        {
            // Handle asynchronous error in Player object.
            printMessage(player.getError());
        }
    });

            media.setOnError(new Runnable()
            {
        public void run()
        {
            // Handle asynchronous error in Media object.
            printMessage(media.getError());
        }
    });

            mediaView.setOnError(new EventHandler <MediaErrorEvent>()
            {
                    public void handle(MediaErrorEvent event)
                    {
            // Handle asynchronous error in MediaView.
            printMessage(event.getMediaError());
        }
            });
```

```
            // Add a ChangeListener to the player
            player.statusProperty().addListener(new ChangeListener<MediaPlayer.Status ↩
                >()
            {
                    // Log the Message
                public void changed(ObservableValue<? extends MediaPlayer.Status> ov,
                        final MediaPlayer.Status oldStatus, final MediaPlayer.Status  ↩
                            newStatus)
                {
                    messageArea.appendText("\nStatus changed from " + oldStatus + " to  ↩
                        " + newStatus);
                }
            });

            // Add a Handler for PLAYING status
            player.setOnPlaying(new Runnable()
            {
        public void run()
        {
            messageArea.appendText("\nPlaying now");
        }
            });

            // Add a Handler for STOPPED status
            player.setOnStopped(new Runnable()
            {
        public void run()
        {
            messageArea.appendText("\nStopped now");
        }
            });

            // Create the HBox
            HBox controlBox = new HBox(5, playButton, stopButton);

            // Create the VBox
            VBox root = new VBox(5,mediaView,controlBox,messageArea);

            // Set the Style-properties of the HBox
            root.setStyle("-fx-padding: 10;" +
                          "-fx-border-style: solid inside;" +
                          "-fx-border-width: 2;" +
                          "-fx-border-insets: 5;" +
                          "-fx-border-radius: 5;" +
                          "-fx-border-color: blue;");

            // Create the Scene
            Scene scene = new Scene(root);
            // Add the scene to the Stage
            stage.setScene(scene);
            // Set the title of the Stage
            stage.setTitle("A State Transition Example");
            // Display the Stage
            stage.show();
    }

    private void printMessage(MediaException error)
    {
            MediaException.Type errorType = error.getType();
            String errorMessage = error.getMessage();
            messageArea.appendText("\n" + "Type:" + errorType + ", error mesage:" +  ↩
                errorMessage);
```

```
        }
}
```

A `MediaPlayer` always has a status. The current status of a `MediaPlayer` is indicated by the read-only `status` property. The status changes when an action is performed on the `MediaPlayer`. It cannot be set directly. The status of a `MediaPlayer` is defined by one of the eight constants in the `MediaPlayer.Status` enum:

- UNKNOWN
- READY
- PLAYING
- PAUSED
- STALLED
- STOPPED
- HALTED
- DISPOSE

The `MediaPlayer` transitions from one status to another when one of the following methods is called:

- play()
- pause()
- stop()
- dispose()

When a `MediaPlayer` is created, its status is `UNKNOWN`. Once the media is prerolled and it is ready to be played, the `MediaPlayer` transitions from `UNKNOWN` to `READY`. Once the `MediaPlayer` exits the `UNKNOWN` status, it cannot reenter it in its lifetime.

The `MediaPlayer` transitions to the `PLAYING` status when the `play()` method is called. This status indicates that the media is playing. Note if the `autoPlay` property is set to true, the `MediaPlayer` may enter the `PLAYING` status without calling the `play()` method explicitly after it is created.

When the `MediaPlayer` is playing, it may enter the `STALLED` status if it does not have enough data in its buffer to play. This status indicates that the `MediaPlayer` is buffering data. When enough data are buffered, it goes back to the `PLAYING` status.

When a `MediaPlayer` is stalled, calling the `pause()` and `stop()` methods, it transitions to the `PAUSED` and `STOPPED` status, respectively. In that case, the buffering continues. However, the `MediaPlayer` does not transition to the `PLAYING` status once enough data are buffered. Rather, it stays in the `PAUSED` or `STOPPED` status.

Calling the `paused()` method transitions the `MediaPlayer` to the `PAUSED` status. Calling the `stop()` method transitions the `MediaPlayer` to the `STOPPED` status.

In cases of an unrecoverable error, the `MediaPlayer` transitions to the `HALTED` terminal status. This status indicates that the `MediaPlayer` cannot be used again. You must create a new `MediaPlayer` if you want to play the media again.

The `dispose()` method frees all of the resources associated with the `MediaPlayer`. However, the `Media` object used by the `MediaPlayer` can still be used. Calling the `dispose()` method transitions the `MediaPlayer` to the terminal status `DISPOSED`.

It is common to display the status of the `MediaPlayer` in an application. Add a ChangeListener to the `status` property to listen for any status changes.

Typically, you will be interested in receiving a notification when the status of the `MediaPlayer` changes. There are two ways to get the notifications:

- By adding a ChangeListener to the status property
- By setting status change handlers

The first method is suitable if you are interested in listening for any type of status change. The following snippet of code shows this method:

```
// Add a ChangeListener to the player
player.statusProperty().addListener(new ChangeListener<MediaPlayer.Status>()
{
        // Log the Message
    public void changed(ObservableValue<? extends MediaPlayer.Status> ov,
            final MediaPlayer.Status oldStatus, final MediaPlayer.Status newStatus)
    {
        messageArea.appendText("\nStatus changed from " + oldStatus + " to " + newStatus);
    }
});
```

The second method is suitable if you are interested in handling a specific type of status change. The `MediaPlayer` class contains the following properties that can be set to `Runnable` objects:

- onReady
- onPlaying
- onRepeat
- onStalled
- onPaused
- onStopped
- onHalted

The `run()` method of the `Runnable` object is called when the `MediaPlayer` enters into the specific status.

For example, the `run()` method of the `onPlaying` handler is called when the player enters the `PLAYING` status.

The following snippet of code shows how to set handlers for a specific type of status change:

```
// Add a Handler for PLAYING status
player.setOnPlaying(new Runnable()
{
        public void run()
        {
                messageArea.appendText("\nPlaying now");
        }
});

// Add a Handler for STOPPED status
player.setOnStopped(new Runnable()
{
        public void run()
        {
        messageArea.appendText("\nStopped now");
        }
});
```

2.5.2 The GUI

The following GUI shows a `MediaPlayer` with State Transitions:

Figure 2.4: An JavaFX Media Example with State Transitions

2.6 Controlling Media Properties

2.6.1 The Code

FxMediaExample5.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.beans.InvalidationListener;
import javafx.beans.Observable;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.control.TextArea;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxMediaExample5 extends Application
{
        // Create the Area for Logging
        private TextArea messageArea = new TextArea();

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

        @Override
        public void start(Stage stage)
        {
                // Create the Sliders
                final Slider cycleSlider = new Slider(1, 5, 1);
                cycleSlider.setMajorTickUnit(1);
                cycleSlider.setShowTickLabels(true);

                final Slider volumeSlider = new Slider(0.0, 1.0, 0.5);
                volumeSlider.setMajorTickUnit(0.1);
                volumeSlider.setShowTickLabels(true);

                final Slider rateSlider = new Slider(0, 8, 4);
                rateSlider.setMajorTickUnit(1);
                rateSlider.setShowTickLabels(true);

                final Slider balanceSlider = new Slider(-1.0, 1.0, 0.0);
                balanceSlider.setMajorTickUnit(0.2);
                balanceSlider.setShowTickLabels(true);

                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp4");
                String mediaStringUrl = mediaUrl.toExternalForm();
```

```
            // Create a Media
            final Media media = new Media(mediaStringUrl);

            // Create a Media Player
            final MediaPlayer player = new MediaPlayer(media);
            // Automatically begin the playback
            player.setAutoPlay(true);

            // Create a 400X300 MediaView
            final MediaView mediaView = new MediaView(player);
            mediaView.setFitWidth(400);
            mediaView.setFitHeight(300);
            mediaView.setSmooth(true);

            // Create the DropShadow effect
            DropShadow dropshadow = new DropShadow();
            dropshadow.setOffsetY(5.0);
            dropshadow.setOffsetX(5.0);
            dropshadow.setColor(Color.WHITE);

            mediaView.setEffect(dropshadow);

            // Create the Buttons
            Button playButton = new Button("Play");
            Button stopButton = new Button("Stop");

            // Create the Event Handlers for the Button
            playButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            if (player.getStatus() == Status.PLAYING)
            {
                    player.stop();
                    player.play();
            }
            else
            {
                    player.play();
            }
        }
    });

            stopButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            player.stop();
        }
    });

            // Create the Listener for the Sliders
            cycleSlider.valueProperty().addListener(new InvalidationListener()
            {
                    @Override
                    public void invalidated(Observable ov)
                    {
                            if (cycleSlider.isValueChanging())
                            {
                                    messageArea.appendText("\nCycle Count changed to: " ↩
                                        + (int)cycleSlider.getValue());
```

```
                                player.setCycleCount((int)cycleSlider.getValue());
                        }
                }
        });

        volumeSlider.valueProperty().addListener(new InvalidationListener()
        {
                @Override
                public void invalidated(Observable ov)
                {
                        if (volumeSlider.isValueChanging())
                        {
                                messageArea.appendText("\nVolume changed to: " +  ↩
                                    volumeSlider.getValue());
                                player.setVolume(volumeSlider.getValue());
                        }
                }
        });

        rateSlider.valueProperty().addListener(new InvalidationListener()
        {
                @Override
                public void invalidated(Observable ov)
                {
                        if (rateSlider.isValueChanging())
                        {
                                messageArea.appendText("\nRate changed to: " +  ↩
                                    rateSlider.getValue());
                                player.setRate(rateSlider.getValue());
                        }
                }
        });

        balanceSlider.valueProperty().addListener(new InvalidationListener()
        {
                @Override
                public void invalidated(Observable ov)
                {
                        if (balanceSlider.isValueChanging())
                        {
                                messageArea.appendText("\nBalance changed to: " +  ↩
                                    balanceSlider.getValue());
                                player.setVolume(balanceSlider.getValue());
                        }
                }
        });

        // Add Handlers for End and Repeat
        player.setOnEndOfMedia(new Runnable()
        {
    public void run()
    {
        messageArea.appendText("\nEnd of media !");
    }
        });

        player.setOnRepeat(new Runnable()
        {
    public void run()
    {
        messageArea.appendText("\nRepeating media !");
    }
```

```
                });

                // Create the GridPane
                GridPane sliderPane = new GridPane();
                // Set horizontal and vertical Spacing
                sliderPane.setHgap(5);
                sliderPane.setVgap(10);

                // Add the details to the GridPane
                sliderPane.addRow(0, new Label("CycleCount:"), cycleSlider);
                sliderPane.addRow(1, new Label("Volume:"), volumeSlider);
                sliderPane.addRow(2, new Label("Rate:"), rateSlider);
                sliderPane.addRow(3, new Label("Balance:"), balanceSlider);

                // Create the HBox
                HBox controlBox = new HBox(5, playButton, stopButton);

                // Create the VBox
                VBox root = new VBox(5,mediaView,controlBox,sliderPane,messageArea);

                // Set the Style-properties of the HBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Media Properties Example");
                // Display the Stage
                stage.show();
        }
}
```

2.6.2 Repeating Media Playback

A media can be played repeatedly for a specified number of times or even indefinitely. The `cycleCount` property specifies the number of times a playback will be repeated. By default, it is set to 1. Set it to `MediaPlayer.INDEFINITE` to repeat the playback indefinitely until the player is paused or stopped. The readonly `currentCount` property is set to the number of completed playback cycles. It is set to 0 when the media is playing the first cycle. At the end of the first cycle, it is set to 1; it is incremented to 2 at the end of second cycle, and so on.

The following code would set a playback cycle of four times:

```
// The playback should repeat 4 times
player.setCycleCount(4);
```

You can receive a notification when the end of media for a cycle in playback is reached. Set a `Runnable` for the `onEndOfMedia` property of the `MediaPlayer` class to get the notification. Note that if a playback continues for four cycles, the end of media notification will be sent four times.

```
// Add Handlers for End and Repeat
player.setOnEndOfMedia(new Runnable()
{
        public void run()
```

```
        {
                messageArea.appendText("\nEnd of media !");
        }
});
```

You can add an `onRepeat` event handler that is called when the end of media for a playback cycle is reached and the playback is going to repeat. It is called after the `onEndOfMedia` event handler:

```
player.setOnRepeat(new Runnable()
{
        public void run()
        {
                messageArea.appendText("\nRepeating media !");
        }
});
```

2.6.3 Controlling the Playback Rate

The `rate` property of the `MediaPlayer` specifies the rate of the playback. The valid range is 0.0 to 8.0. For example, a rate of 2.0 plays the media two times faster than the normal rate. The default value is 1.0, which plays the media at the normal rate. The read-only `currentRate` property is the current rate of playback.

The following code would set the rate at three times the normal rate:

```
player.setRate(3.0);
```

2.6.4 Controlling the Playback Volume

Three properties in the `MediaPlayer` class control the volume of the audio in the media:

- volume
- mute
- balance

The `volume` specifies the volume of the audio. The range is 0.0 to 1.0. A value of 0.0 makes the audio inaudible, whereas a value of 1.0 plays it at full volume. The default value is 1.0.

The `mute` specifies whether the audio is produced by the `MediaPlayer`. By default, its value is false and the audio is produced. Setting it to true does not produce audio. Note that setting the mute property does not affect the volume property.

Suppose the volume is set to 1.0 and the muted is set to true. There is no audio being produced. When the mute is set to false, the audio will use the volume property that is 1.0 and it will play at full volume. The following code would set the volume at half:

```
player.setVolumne(0.5);
player.setMute(true)
```

The `balance` specifies the relative volume for the left and right channels. The valid range is -1.0 to 1.0. A value of -1.0 sets the playback in the left channel at normal volume and mutes the right channel. A value of 1.0 sets the playback in the right channel at normal volume and mutes the left channel. The default value is 0.0, which sets the playback in both channels at normal volume.

2.6.5 The GUI

The following GUI shows a `MediaPlayer` with Properties:

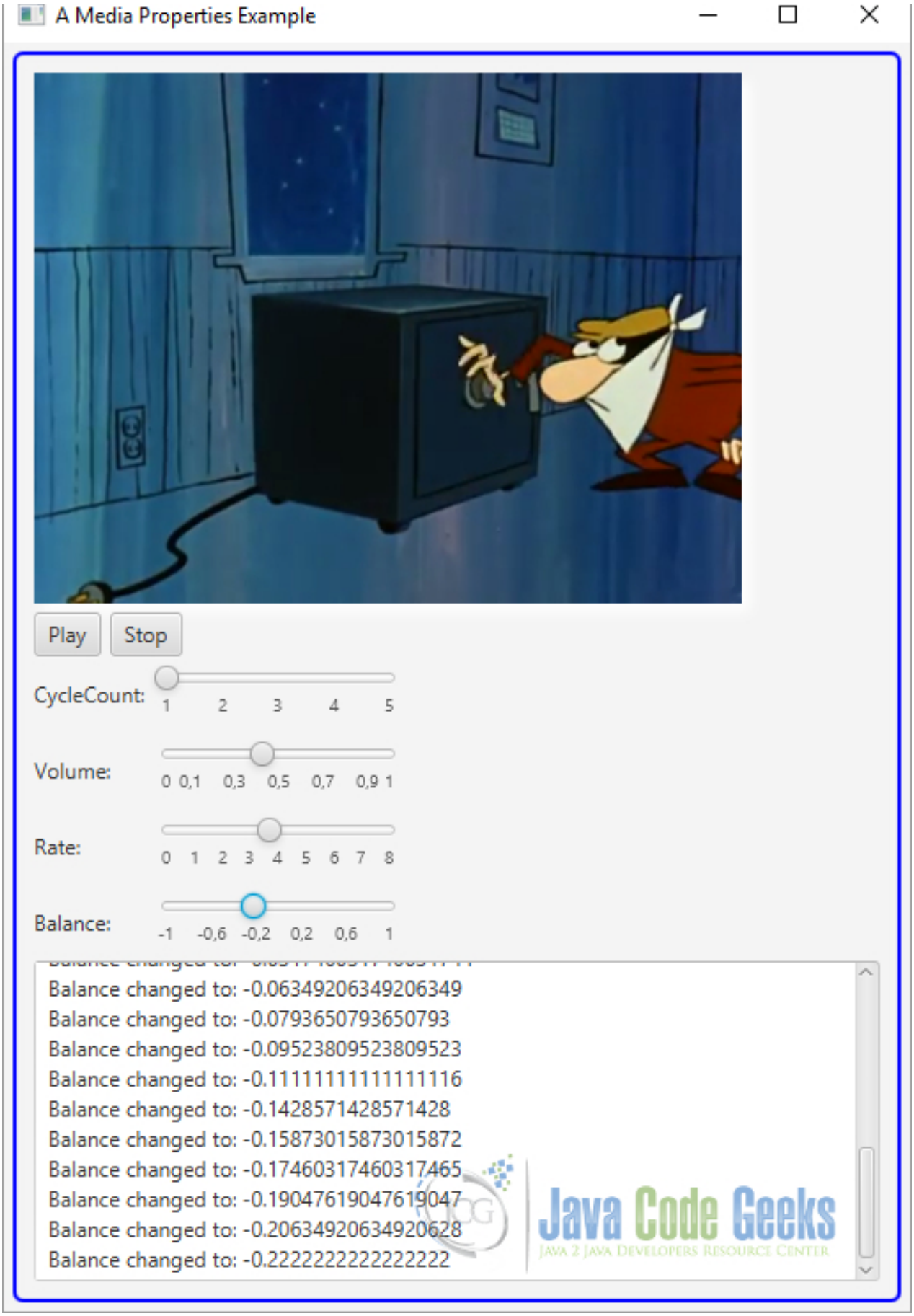

Figure 2.5: An JavaFX Media Example with Properties

2.7 Tracking Media Time

2.7.1 The Code

FxMediaExample6.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxMediaExample6 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp4");
                String mediaStringUrl = mediaUrl.toExternalForm();

                // Create a Media
                final Media media = new Media(mediaStringUrl);

                // Create a Media Player
                final MediaPlayer player = new MediaPlayer(media);
                // Automatically begin the playback
                player.setAutoPlay(true);

                // Set the Times of the Player
                player.setStartTime(Duration.minutes(1));
                player.setStopTime(Duration.minutes(2));

                // Create a 400X300 MediaView
                final MediaView mediaView = new MediaView(player);
                mediaView.setFitWidth(400);
                mediaView.setFitHeight(300);
                mediaView.setSmooth(true);

                // Create the DropShadow effect
                DropShadow dropshadow = new DropShadow();
                dropshadow.setOffsetY(5.0);
                dropshadow.setOffsetX(5.0);
                dropshadow.setColor(Color.WHITE);
```

```
                mediaView.setEffect(dropshadow);

                // Create the Buttons
                Button playButton = new Button("Play");
                Button stopButton = new Button("Stop");

                // Create the Event Handlers for the Button
                playButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                if (player.getStatus() == Status.PLAYING)
                {
                        player.stop();
                        player.play();
                }
                else
                {
                        player.play();
                }
            }
        });

                stopButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                player.stop();
            }
        });

                // Create the HBox
                HBox controlBox = new HBox(5, playButton, stopButton);

                // Create the VBox
                VBox root = new VBox(5,mediaView,controlBox);

                // Set the Style-properties of the HBox
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Tracking Media Example");
                // Display the Stage
                stage.show();
        }
}
```

Displaying the media duration and the elapsed time for a playback are important feedback for the audience. A good understanding of these duration types is important in developing a good media playback dashboard.

Different types of `Duration` can be associated with a media:

- The current duration of a media playing media

- The duration of the media playback
- The duration of the media play for one cycle
- The start offset time
- The end offset time
- onStopped
- onHalted
- DISPOSE

By default, a media plays for its original duration. For example, if the duration of the media is 30 minutes, the media will play for 30 minutes in one cycle. The `MediaPlayer` lets you specify the length of the playback, which can be anywhere in the `Duration` of the media. For example, for each playback cycle, you can specify that only the middle 10 minutes (11th to 12th) of the media should be played. The length of the media playback is specified by the following two properties of the `MediaPlayer` class:

- startTime
- stopTime

Both properties are of the `Duration` type. The `startTime` and `stopTime` are the time offsets where the media should start and stop playing for each cycle, respectively. By default, the startTime is set to `Duration.ZERO` and the `stopTime` is set to the duration of the media.

The following snippet of code sets these properties, so the media will be played from the first minute to the second minute:

```
// Set the Times of the Player
player.setStartTime(Duration.minutes(1));
player.setStopTime(Duration.minutes(2));
```

2.7.2 The GUI

The following GUI shows a `MediaPlayer` with Time Tracking:

Figure 2.6: An JavaFX Media Example with Media Time Tracking

2.8 Marking Positions in the Media

2.8.1 The Code

FxMediaExample7.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.collections.ObservableMap;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaMarkerEvent;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.util.Duration;
import javafx.util.Pair;

public class FxMediaExample7 extends Application
{
        // Create the Area for Logging
        private TextArea messageArea = new TextArea();

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

        @Override
        public void start(Stage stage)
        {
                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp4");
                String mediaStringUrl = mediaUrl.toExternalForm();

                // Create a Media
                final Media media = new Media(mediaStringUrl);

                // Create the Markers
                ObservableMap<String, Duration> markers = media.getMarkers();
                markers.put("START", Duration.ZERO);
                markers.put("INTERVAL", media.getDuration().divide(2.0));
                markers.put("END", media.getDuration());

                // Create a Media Player
                final MediaPlayer player = new MediaPlayer(media);
                // Automatically begin the playback
                player.setAutoPlay(true);

                // Create a 400X300 MediaView
                final MediaView mediaView = new MediaView(player);
                mediaView.setFitWidth(400);
                mediaView.setFitHeight(300);
                mediaView.setSmooth(true);

                // Create the DropShadow effect
                DropShadow dropshadow = new DropShadow();
                dropshadow.setOffsetY(5.0);
                dropshadow.setOffsetX(5.0);
                dropshadow.setColor(Color.WHITE);

                // Set the Effect on the MediaView
                mediaView.setEffect(dropshadow);

                // Create the Buttons
```

```
            Button playButton = new Button("Play");
            Button stopButton = new Button("Stop");

            // Create the Event Handlers for the Button
            playButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            if (player.getStatus() == Status.PLAYING)
            {
                    player.stop();
                    player.play();
            }
            else
            {
                    player.play();
            }
        }
    });

            stopButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            player.stop();
        }
    });

            // Add a marker event handler
            player.setOnMarker(new EventHandler <MediaMarkerEvent>()
            {
        public void handle(MediaMarkerEvent event)
        {
                    Pair<String, Duration> marker = event.getMarker();
                    String markerText = marker.getKey();
                    Duration markerTime = marker.getValue();
                    messageArea.appendText("\nReached the marker " + markerText + " at  ↩
                        " + markerTime);
        }
            });

            // Create the HBox
            HBox controlBox = new HBox(5, playButton, stopButton);

            // Create the VBox
            VBox root = new VBox(5,mediaView,controlBox,messageArea);

            // Set the Style-properties of the HBox
            root.setStyle("-fx-padding: 10;" +
                          "-fx-border-style: solid inside;" +
                          "-fx-border-width: 2;" +
                          "-fx-border-insets: 5;" +
                          "-fx-border-radius: 5;" +
                          "-fx-border-color: blue;");

            // Create the Scene
            Scene scene = new Scene(root);
            // Add the scene to the Stage
            stage.setScene(scene);
            // Set the title of the Stage
            stage.setTitle("A Markers Example");
            // Display the Stage
```

```
                stage.show();
        }
}
```

You can associate markers with specific point on the media timeline. Markers are simply text that are useful in a number of ways. You can use them to insert advertisements.

For example, you can insert a URL as the marker text. When the marker is reached, you can pause playing the media and play another media. Note that playing another media involves creating new `Media` and `MediaPlayer` objects. You can reuse a `MediaView`. When you are playing the advertisement video, associate the `MediaView` with the new `MediaPlayer`. When the advertisement playback is finished, associate the MediaView back to the main `MediaPlayer`.

The `Media` class contains a `getMarkers()` method that returns an ObservableMap. You need to add the (key, value) pairs in the map to add markers.

The following snippet of code adds three markers to a media:

```
// Create the Markers
ObservableMap<String, Duration> markers = media.getMarkers();
markers.put("START", Duration.ZERO);
markers.put("INTERVAL", media.getDuration().divide(2.0));
markers.put("END", media.getDuration());
```

The `MediaPlayer` fires a MediaMarkerEvent when a marker is reached. You can register a handler for this event in the `onMarker` property of the `MediaPlayer`.

The following snippet of code shows how to handle the `MediaMarkerEvent`. The `getMarker()` method of the event returns a Pair whose key and value are the marker text and marker duration, respectively.

```
// Add a marker event handler
player.setOnMarker(new EventHandler <MediaMarkerEvent>()
{
        public void handle(MediaMarkerEvent event)
        {
                Pair<String, Duration> marker = event.getMarker();
                String markerText = marker.getKey();
                Duration markerTime = marker.getValue();
                messageArea.appendText("\nReached the marker " + markerText + " at " +  ↩
                    markerTime);
        }
});
```

2.8.2 The GUI

The following GUI shows a `MediaPlayer` with Markers:

Figure 2.7: An JavaFX Media Example with Position Markers

2.9 Showing Media Metadata

2.9.1 The Code

FxMediaExample8.java

```
import java.net.URL;
import javafx.application.Application;
import javafx.collections.ObservableMap;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxMediaExample8 extends Application
{
        // Create the Area for Logging
        private TextArea messageArea = new TextArea();

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

        @Override
        public void start(Stage stage)
        {
                // Locate the media content in the CLASSPATH
                URL mediaUrl = getClass().getResource("Test.mp3");
                String mediaStringUrl = mediaUrl.toExternalForm();

                // Create a Media
                final Media media = new Media(mediaStringUrl);

                // Create a Media Player
                final MediaPlayer player = new MediaPlayer(media);
                // Automatically begin the playback
                player.setAutoPlay(false);

                // Create a 400X300 MediaView
                final MediaView mediaView = new MediaView(player);
                mediaView.setFitWidth(400);
                mediaView.setFitHeight(300);
                mediaView.setSmooth(true);

                // Create the DropShadow effect
                DropShadow dropshadow = new DropShadow();
                dropshadow.setOffsetY(5.0);
                dropshadow.setOffsetX(5.0);
                dropshadow.setColor(Color.WHITE);
```

```
            mediaView.setEffect(dropshadow);

            // Create the Buttons
            Button playButton = new Button("Play");
            Button stopButton = new Button("Stop");

            // Create the Event Handlers for the Button
            playButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            if (player.getStatus() == Status.PLAYING)
            {
                    player.stop();
                    player.play();
            }
            else
            {
                    player.play();
            }
        }
    });

            stopButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            player.stop();
        }
    });

            // Display the metadata data on the console
            player.setOnReady(new Runnable()
            {
        public void run()
        {
                    ObservableMap<String, Object> metadata = media.getMetadata();
                    for(String key : metadata.keySet())
                    {
                            messageArea.appendText("\n" + key + " = "  + metadata.get( ↩
                                key));
                    }
        }
            });

            // Create the HBox
            HBox controlBox = new HBox(5, playButton, stopButton);

            // Create the VBox
            VBox root = new VBox(5,mediaView,controlBox,messageArea);

            // Set the Style-properties of the HBox
            root.setStyle("-fx-padding: 10;" +
                          "-fx-border-style: solid inside;" +
                          "-fx-border-width: 2;" +
                          "-fx-border-insets: 5;" +
                          "-fx-border-radius: 5;" +
                          "-fx-border-color: blue;");

            // Create the Scene
            Scene scene = new Scene(root);
            // Add the scene to the Stage
```

```
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Metadata Example");
                // Display the Stage
                stage.show();
        }
}
```

Some metadata may be embedded into a media that describe the media. Typically, the metadata contains the title, artist name, album name, genre, year, and so forth.

The following snippet of code displays the metadata for the media when the `MediaPlayer` enters the `READY` status. Do not try reading the metadata just after creating the `Media` object, as the metadata may not be available.

```
// Display the metadata data on the console
player.setOnReady(new Runnable()
{
        public void run()
        {
                ObservableMap<String, Object> metadata = media.getMetadata();
                for(String key : metadata.keySet())
                {
                        messageArea.appendText("\n" + key + " = "  + metadata.get(key));
                }
        }
});
```

You cannot be sure whether there are metadata in a media or the type of metadata a media may contain. In your application, you can just look for the title, artist, album, and year. Alternatively, you could read all of the metadata and display them in a two-column table. Sometimes the metadata may contain an embedded image of the artist. You would need to check the class name of the value in the map to use the image.

2.9.2 The GUI

The following GUI shows a `MediaPlayer` with Metadata:

Figure 2.8: An JavaFX Media Example with Metadata

2.10 Download Java Source Code

This was an example of `javafx.scene.media`

Download

You can download the full source code of this example here: **JavaFxMediaExample.zip**

Chapter 3

The JavaFX Concurrent Framework

This is an article about the JavaFX Concurrent Framework API. Java 5 added a comprehensive concurrency framework to the Java programming language through the libraries in the `java.util.concurrent` package. The JavaFX Concurrency Framework is very small.

It is built on top of the Java language Concurrency Framework keeping in mind that it will be used in a GUI environment.

The following examples use Java SE 8 and JavaFX 2.2.

3.1 Introduction

The framework consists of one interface, four classes, and one enum.

An instance of the Worker interface represents a Task that needs to be performed in one or more background threads. The state of the `Task` is observable from the JavaFX Application Thread.

The `Task`, Service, and ScheduledService classes implement the `Worker` interface. They represent different types of tasks. They are abstract classes. An instance of the `Task` class represents a one-shot task.

A `Task` cannot be reused. An instance of the `Service` class represents a reusable task. The `ScheduledService` class inherits from the `Service` class. A `ScheduledService` is a `Task` that can be scheduled to run repeatedly after a specified interval.

The constants in the `Worker.State` enum represent different states of a `Worker`.

An instance of the WorkerStateEvent class represents an event that occurs as the state of a `Worker` changes. You can add event handlers to all three types of tasks to listen to the change in their states.

3.2 Understanding the Worker Interface

The `Worker<V>` interface provides the specification for any task performed by the JavaFX Concurrency Framework. A `Worker` is a `Task` that is performed in one or more background threads. The generic parameter `V` is the data type of the result of the `Worker`.

The state of the `Task` is observable. The state of the `Task` is published on the JavaFX Application `Thread`, making it possible for the `Task` to communicate with the Scene Graph, as is commonly required in a GUI application.

3.2.1 Utility Classes

Let us create the reusable GUI and non-GUI parts of the programs to use in examples in the following sections.

The `WorkerStateGUI` class builds a GridPane to display all properties of a `Worker`.

It is used with a `Worker<ObservableList<Long>>`. It displays the properties of a `Worker` by UI elements to them. You can bind properties of a `Worker` to the UI elements by passing a Worker to the constructor or calling the `bindToWorker()` method.

WorkerStateGUI.java

```
import javafx.beans.binding.When;
import javafx.collections.ObservableList;
import javafx.concurrent.Worker;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.TextArea;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;

public class WorkerStateGUI extends GridPane
{
        // Create the Labels
        private final Label title = new Label("");
        private final Label message = new Label("");
        private final Label running = new Label("");
        private final Label state = new Label("");
        private final Label totalWork = new Label("");
        private final Label workDone = new Label("");
        private final Label progress = new Label("");

        // Create the TextAreas
        private final TextArea value = new TextArea("");
        private final TextArea exception = new TextArea("");

        // Create the ProgressBar
        private final ProgressBar progressBar = new ProgressBar();

        public WorkerStateGUI()
        {
                addGUI();
        }

        public WorkerStateGUI(Worker<ObservableList<Long>> worker)
        {
                addGUI();
                bindToWorker(worker);
        }

        private void addGUI()
        {
                value.setPrefColumnCount(20);
                value.setPrefRowCount(3);
                exception.setPrefColumnCount(20);
                exception.setPrefRowCount(3);

                this.setHgap(5);
                this.setVgap(5);

                addRow(0, new Label("Title:"), title);
                addRow(1, new Label("Message:"), message);
                addRow(2, new Label("Running:"), running);
                addRow(3, new Label("State:"), state);
                addRow(4, new Label("Total Work:"), totalWork);
                addRow(5, new Label("Work Done:"), workDone);
                addRow(6, new Label("Progress:"), new HBox(2, progressBar, progress));
```

```
                addRow(7, new Label("Value:"), value);
                addRow(8, new Label("Exception:"), exception);
        }

        public void bindToWorker(final Worker<ObservableList<Long>> worker)
        {
                // Bind Labels to the properties of the worker
                title.textProperty().bind(worker.titleProperty());
                message.textProperty().bind(worker.messageProperty());
                running.textProperty().bind(worker.runningProperty().asString());
                state.textProperty().bind(worker.stateProperty().asString());

                totalWork.textProperty().bind(
                        new When(worker.totalWorkProperty().isEqualTo(-1)).then("Unknown")
                        .otherwise(worker.totalWorkProperty().asString()));

                workDone.textProperty().bind(
                        new When(worker.workDoneProperty().isEqualTo(-1)).then("Unknown")
                        .otherwise(worker.workDoneProperty().asString()));

                progress.textProperty().bind(
                        new When(worker.progressProperty().isEqualTo(-1)).then("Unknown")
                        .otherwise(worker.progressProperty().multiply(100.0).asString("%.2f ↩
                            %%")));

                progressBar.progressProperty().bind(worker.progressProperty());
                value.textProperty().bind(worker.valueProperty().asString());

                worker.exceptionProperty().addListener(new ChangeListener<Throwable>()
                {
                        public void changed(ObservableValue<? extends Throwable> prop,
                            final Throwable oldValue, final Throwable newValue)
                        {
                                if (newValue != null)
                                {
                                        exception.setText(newValue.getMessage());
                                }
                                else
                                {
                                        exception.setText("");
                                }
                        }
                });
        }
}
```

The `PrimeUtil` class is a utility class to check whether a number is a prime number.

PrimeUtil.java

```
public class PrimeUtil
{
        public static boolean isPrime(long num)
        {
                if (num <= 1 || num % 2 == 0)
                {
                        return false;
                }

                int upperDivisor = (int)Math.ceil(Math.sqrt(num));

                for (int divisor = 3; divisor <= upperDivisor; divisor += 2)
```

```
                {
                        if (num % divisor == 0)
                        {
                                return false;
                        }
                }

                return true;
        }
}
```

3.2.2 State Transitions for a Worker

During the life cycle, a `Worker` transitions through different states. The constants in the `Worker.State` enum represent the valid states of a `Worker`.

- Worker.State.READY
- Worker.State.SCHEDULED
- Worker.State.RUNNING
- Worker.State.SUCCEEDED
- Worker.State.CANCELLED
- Worker.State.FAILED

When a `Worker` is created, it is in the `READY` state. It transitions to the `SCHEDULED` state, before it starts executing. When it starts running, it is in the `RUNNING` state. Upon successful completion, a `Worker` transitions from the `RUNNING` state to the `SUCCEEDED` state. If the `Worker` throws an exception during its execution, it transitions to the `FAILED` state. A `Worker` may be cancelled using the `cancel()` method.

It may transition to the `CANCELLED` state from the `READY`, `SCHEDULED`, and `RUNNING` states. These are the normal state transitions for a one-shot `Worker`.

A reusable `Worker` may transition from the `CANCELLED`, `SUCCEEDED`, and `FAILED` states to the `READY` state.

3.2.3 Properties of a Worker

The `Worker` interface contains nine read-only properties that represent the internal state of the `Task`.

- title
- message
- running
- state
- progress
- workDone
- totalWork
- value
- exception

When you create a `Worker`, you will have a chance to specify these properties. The properties can also be updated as the task progresses.

The `title` property represents the title for the task.

The `message` property represents a detailed message during the task processing.

The `running` property tells whether the `Worker` is running. It is true when the Worker is in the `SCHEDULED` or `RUNNING` states. Otherwise, it is false.

The `state` property specifies the state of the `Worker`. Its value is one of the constants of the `Worker.State` enum.

The `totalWork`, `workDone`, and `progress` properties represent the progress of the task. The `totalWork` is the total amount of work to be done. The `workDone` is the amount of work that has been done. The `progress` is the ratio of `workDone` and `totalWork`.

The `value` property represents the result of the task. Its value is non-null only when the `Worker` finishes successfully reaching the `SUCCEEDED` state.

A task may fail by throwing an exception. The exception property represents the exception that is thrown during the processing of the task. It is non-null only when the state of the `Worker` is `FAILED`.

Typically, when a `Task` is in progress, you want to display the task details in a `Scene` Graph.

The Concurrency Framework makes sure that the properties of a `Worker` are updated on the JavaFX `Application Thread`. Therefore, it is fine to bind the properties of the UI elements in a Scene Graph to these properties.

3.3 Using the Task Class

An instance of the `Task<V>` class represents a one-time task. Once the task is completed, cancelled, or failed, it cannot be restarted.

The `Task<V>` class implements the `Worker<V>` interface. Therefore, all properties and methods specified by the `Worker<V>` ` interface are available in the `Task<V>` class.

The `Task<V>` class inherits from the FutureTask<V> class, which is part of the Java Concurrency Framework.

The `FutureTask<V>` implements the Future<V>, RunnableFuture<V>, and Runnable interfaces.

Therefore, a `Task<V>` also implements all these interfaces.

3.3.1 The Code

PrimeFinderTask.java

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.concurrent.Task;

public class PrimeFinderTask extends Task<ObservableList<Long>>
{
        // Define the Limits
        private long lowerLimit = 1;
        private long upperLimit = 30;
        private long sleepTimeInMillis = 500;

        public PrimeFinderTask()
        {
        }

        public PrimeFinderTask(long lowerLimit, long upperLimit)
        {
                this.lowerLimit = lowerLimit;
```

```
            this.upperLimit = upperLimit;
        }

        public PrimeFinderTask(long lowerLimit,long upperLimit,long sleepTimeInMillis)
        {
            this(lowerLimit, upperLimit);
            this.sleepTimeInMillis = sleepTimeInMillis;
        }

        // The task implementation
        @Override
        protected ObservableList<Long> call()
        {
            // An observable list to represent the results
            final ObservableList<Long> results = FXCollections.<Long> ↩
                observableArrayList();

            // Update the title
            this.updateTitle("Prime Number Finder Task");
            long count = this.upperLimit - this.lowerLimit + 1;
            long counter = 0;

            // Find the prime numbers
            for (long i = lowerLimit; i <= upperLimit; i++)
            {
                // Check if the task is cancelled
                if (this.isCancelled())
                {
                    break;
                }

                // Increment the counter
                counter++;

                // Update message
                this.updateMessage("Checking " + i + " for a prime number");

                // Sleep for some time
                try
                {
                    Thread.sleep(this.sleepTimeInMillis);
                }
                catch (InterruptedException e)
                {
                    // Check if the task is cancelled
                    if (this.isCancelled())
                    {
                        break;
                    }
                }

                // Check if the number is a prime number
                if (PrimeUtil.isPrime(i))
                {
                    // Add to the list
                    results.add(i);

                    // Publish the read-only list to give the GUI
                    // access to the partial results
                    updateValue(FXCollections.<Long>unmodifiableObservableList( ↩
                        results));
                }
```

```
                        // Update the progress
                        updateProgress(counter, count);
                }

                return results;
        }

        @Override
        protected void cancelled()
        {
                super.cancelled();
                updateMessage("The task was cancelled.");
        }

        @Override
        protected void failed()
        {
                super.failed();
                updateMessage("The task failed.");
        }

        @Override
        public void succeeded()
        {
                super.succeeded();
                updateMessage("The task finished successfully.");
        }
}
```

The above program is an implementation of the `Task<ObservableList<Long>>`. It checks for prime numbers between the specified `lowerLimit` and `upperLimit`. It returns all the numbers in the range. Notice that the task thread sleeps for a short time before checking a number for a prime number. This is done to give the user an impression of a long-running task.

It is not needed in a real world application. The `call()` method handles an `InterruptedException` and finishes the task if the task was interrupted as part of a cancellation request. The call to the method `updateValue()` needs little explanation.

```
updateValue(FXCollections.<Long>unmodifiableObservableList(results));
```

Every time a prime number is found, the results list is updated. The foregoing statement wraps the results list in an unmodifiable observable list and publishes it for the client. This gives the client access to the partial results of the task. This is a quick and dirty way of publishing the partial results. If the `call()` method returns a primitive value, it is fine to call the `updateValue()` method repeatedly.

The following program contains the complete code to build a GUI using your `PrimeFinderTask` class.

FxConcurrentExample1.java

```
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.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import static javafx.concurrent.Worker.State.READY;
import static javafx.concurrent.Worker.State.RUNNING;

public class FxConcurrentExample1 extends Application
{
        // Create the Buttons
```

```
        Button startButton = new Button("Start");
        Button cancelButton = new Button("Cancel");
        Button exitButton = new Button("Exit");

        // Create the task
        PrimeFinderTask task = new PrimeFinderTask();

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

        @Override
        public void start(final Stage stage)
        {
                // Create the Event-Handlers for the Buttons
                startButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                startTask();
            }
        });

                exitButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                stage.close();
            }
        });

                cancelButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                task.cancel();
            }
        });

                // Enable/Disable the Start and Cancel buttons
                startButton.disableProperty().bind(task.stateProperty().isNotEqualTo(READY) ↩
                    );
                cancelButton.disableProperty().bind(task.stateProperty().isNotEqualTo( ↩
                    RUNNING));

                // Create the GridPane
                GridPane pane = new WorkerStateGUI(task);

                // Create the ButtonBox
                HBox buttonBox = new HBox(5, startButton, cancelButton, exitButton);

                // Create the BorderPane
                BorderPane root = new BorderPane();
                root.setCenter(pane);
                root.setBottom(buttonBox);

                // Set the Style-properties of the BorderPane
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
```

```
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root,500,400);
                // Add the scene to the Stage
                stage.setScene(scene);
                // Set the title of the Stage
                stage.setTitle("A Prime Number Finder Task");
                // Display the Stage
                stage.show();
        }

        public void startTask()
        {
                // Schedule the task on a background thread
                Thread backgroundThread = new Thread(task);
                backgroundThread.setDaemon(true);
                backgroundThread.start();
        }
}
```

3.3.2 Creating a Task

Creating a `Task<V>` is easy. You need to subclass the `Task<V>` class and provide an implementation for the abstract method `call()`. The `call()` method contains the logic to perform the `Task`.

The following snippet of code shows the skeleton of a `Task` implementation:

```
// The task implementation
@Override
protected ObservableList<Long> call()
{
        // An observable list to represent the results
        final ObservableList<Long> results = FXCollections.<Long>observableArrayList();

        // Update the title
        this.updateTitle("Prime Number Finder Task");
        long count = this.upperLimit - this.lowerLimit + 1;
        long counter = 0;

        // Find the prime numbers
        for (long i = lowerLimit; i <= upperLimit; i++)
        {
                // Check if the task is cancelled
                if (this.isCancelled())
                {
                        break;
                }

                // Increment the counter
                counter++;

                // Update message
                this.updateMessage("Checking " + i + " for a prime number");

                // Sleep for some time
                try
                {
                        Thread.sleep(this.sleepTimeInMillis);
                }
```

```
                catch (InterruptedException e)
                {
                        // Check if the task is cancelled
                        if (this.isCancelled())
                        {
                                break;
                        }
                }

                // Check if the number is a prime number
                if (PrimeUtil.isPrime(i))
                {
                        // Add to the list
                        results.add(i);

                        // Publish the read-only list to give the GUI
                        // access to the partial results
                        updateValue(FXCollections.<Long>unmodifiableObservableList(results) ↩
                            );
                }

                // Update the progress
                updateProgress(counter, count);
        }

        return results;
}
```

3.3.3 Updating Task Properties

Typically, you would want to update the properties of the `Task` as it progresses. The properties must be updated and read on the JavaFX `Application Thread`, so they can be observed safely in a GUI environment. The `Task<V>` class provides special methods to update some of its properties.

- protected void updateMessage(String message)
- protected void updateProgress(double workDone, double totalWork)
- protected void updateProgress(long workDone, long totalWork)
- protected void updateTitle(String title)
- protected void updateValue(V value)

You provide the values for the `workDone` and the `totalWork` properties to the `updateProgress()` method. The progress property will be set to `workDone/totalWork`. The method throws a runtime exception if the workDone is greater than the `totalWork` or both are less than -1.0.

Sometimes, you may want to publish partial results of a task in its value property. The `updateValue()` method is used for this purpose. The final result of a task is the return value of its `call()` method.

All `updateXxx()` methods are executed on the JavaFX `Application Thread`. Their names indicate the property they update. They are safe to be called from the `call()` method of the `Task`.

If you want to update the properties of the `Task` from the `call()` method directly, you need to wrap the code inside a `Platform.runLater()` call.

3.3.4 Listening to Task Transition Events

The `Task` class contains the following properties to let you set event handlers for its state transitions:

- onCancelled
- onFailed
- onRunning
- onScheduled
- onSucceeded

3.3.5 Cancelling a Task

Use one of the following two `cancel()` methods to cancel a task:

- public final boolean cancel()
- public boolean cancel(boolean mayInterruptIfRunning)

The first version removes the `Task` from the execution queue or stops its execution.

The second version lets you specify whether the thread running the `Task` be interrupted.

Make sure to handle the InterruptedException inside the `call()` method. Once you detect this exception, you need to finish the `call()` method quickly. Otherwise, the call to `cancel(true)` may not cancel the task reliably. The `cancel()` method may be called from any thread.

The following methods of the `Task` are called when it reaches a specific state:

- protected void scheduled()
- protected void running()
- protected void succeeded()
- protected void cancelled()
- protected void failed()

Their implementations in the `Task` class are empty. They are meant to be overridden by the subclasses.

3.3.6 Running a Task

A `Task` is `Runnable` as well as a `FutureTask`. To run it, you can use a background thread or an ExecutorService.

```
// Schedule the task on a background thread
Thread backgroundThread = new Thread(task);
backgroundThread.setDaemon(true);
backgroundThread.start();
```

3.3.7 The GUI

The following image shows the window after starting the program

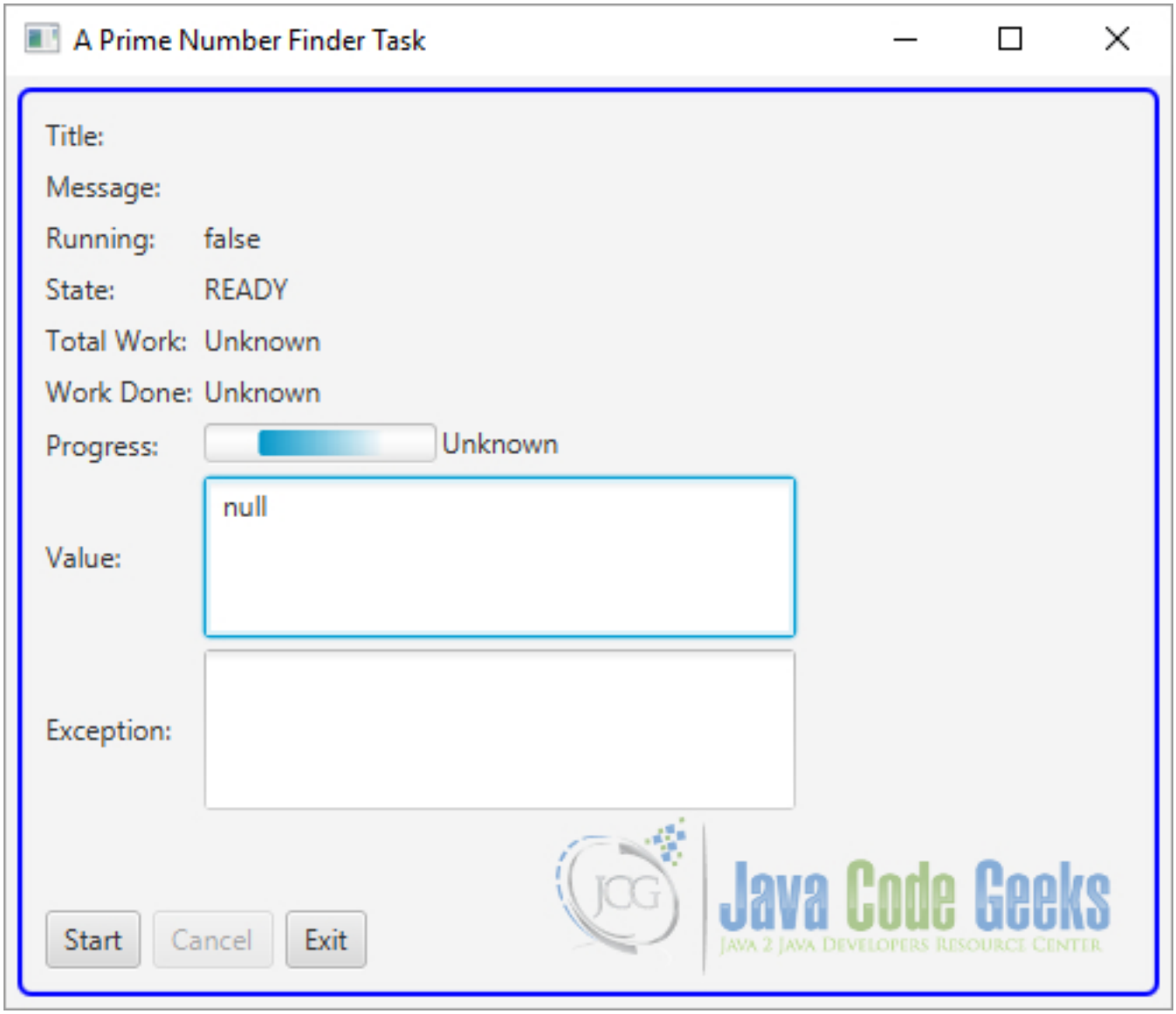

Figure 3.1: The output of the Program before starting the Task

The following Figure shows the window when the task is running. You will need to click the Start button to start the task.

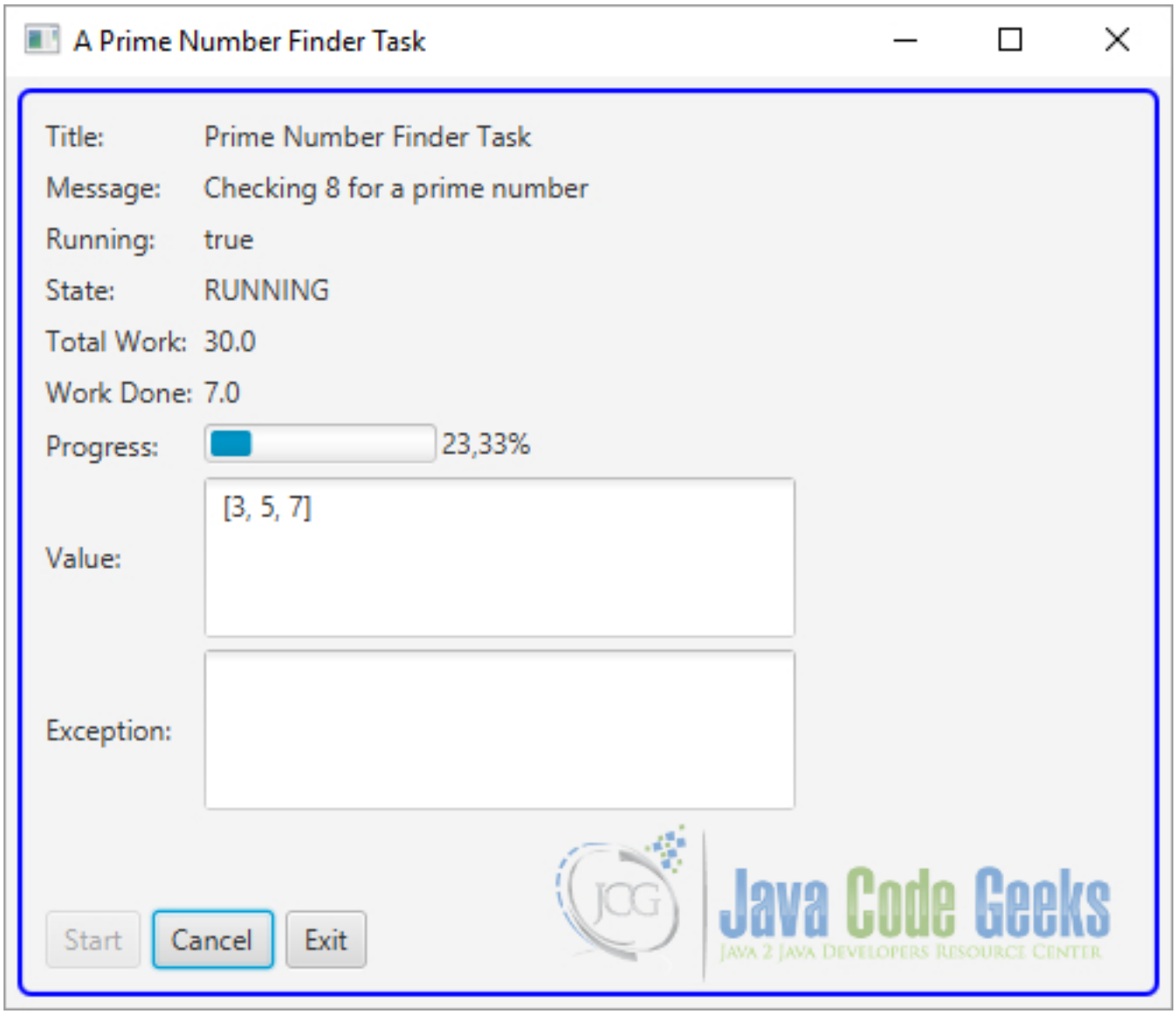

Figure 3.2: The output of the Program during the execution of the Task

Clicking the Cancel button cancels the task. Once the task finishes, it is cancelled or it fails; you cannot restart it and both the Start and Cancel buttons are disabled.

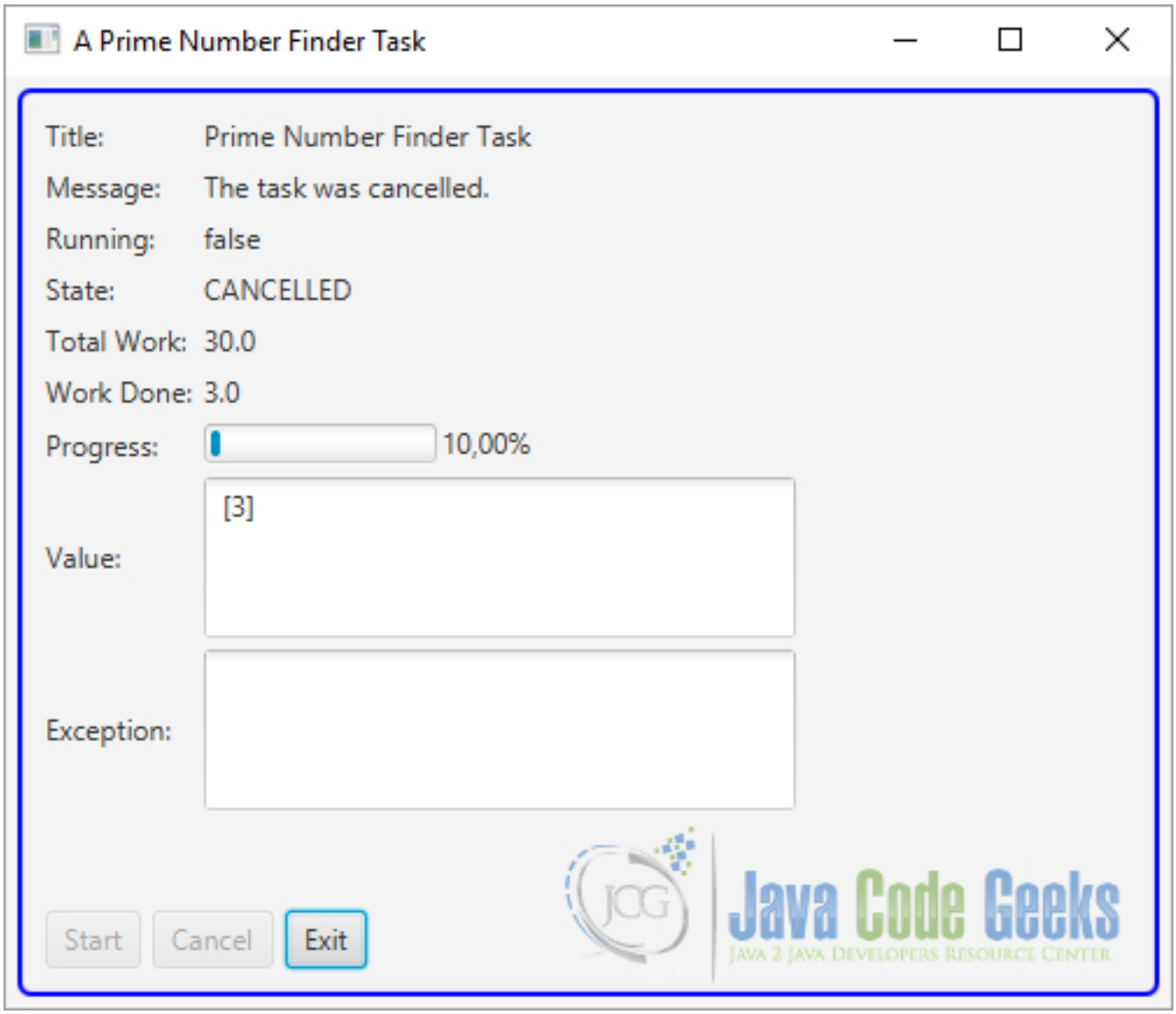

Figure 3.3: The output of the Program after canceling the Task

Notice that when the task finds a new prime number, it is displayed on the window immediately.

After execution of the `Task`, the result will be shown:

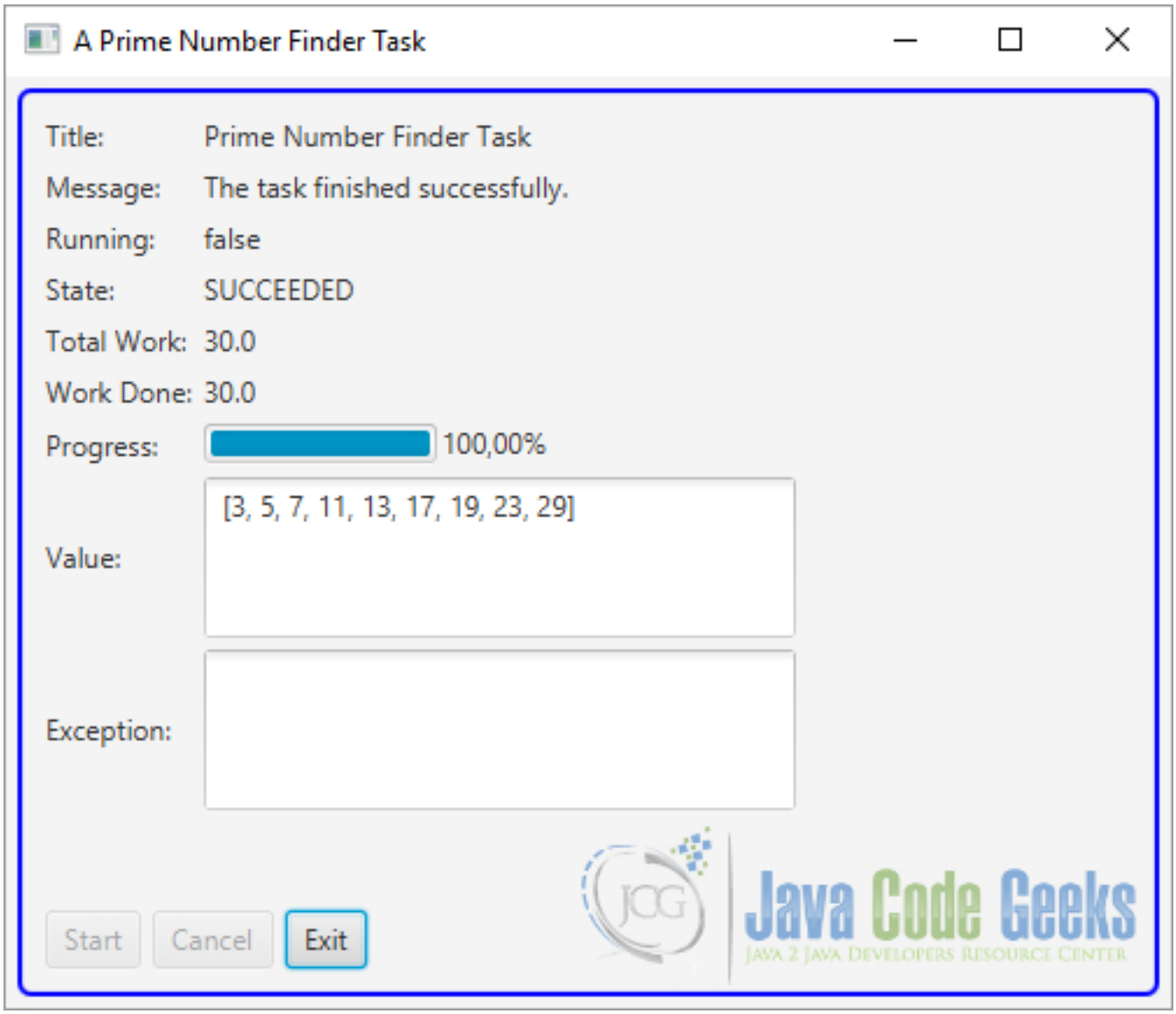

Figure 3.4: The output of the Program after finishing the Task

3.4 Using the Service Class

The `Service<V>` class is an implementation of the `Worker<V>` interface. It encapsulates a `Task<V>`. It makes the `Task<V>` reusable by letting it be started, cancelled, reset, and restarted.

3.4.1 The Code

The following program shows how to use a `Service`. The `Service` object is created and stored as an instance variable. The `Service` object manages a `PrimeFinderTask` object, which is a `Task` to find prime numbers between two numbers.

Four buttons are added: Start/Restart, Cancel, Reset, and Exit. The Start Button is labeled Restart after the `Service` is started for the first time. The buttons do what their labels indicate. Buttons are disabled when they are not applicative.

FxConcurrentExample2.java

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.binding.Bindings;
import javafx.collections.ObservableList;
import javafx.concurrent.Service;
import javafx.concurrent.Task;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import static javafx.concurrent.Worker.State.RUNNING;
import static javafx.concurrent.Worker.State.SCHEDULED;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class FxConcurrentExample2 extends Application
{
        // Create the Buttons
        Button startButton = new Button("Start");
        Button cancelButton = new Button("Cancel");
        Button exitButton = new Button("Exit");
        Button resetButton = new Button("Reset");
        boolean onceStarted = false;

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

        @Override
        public void start(Stage stage)
        {
                // Create the Event-Handlers for the Buttons
                startButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                if (onceStarted)
                {
                        service.restart();
                }
                else
                {
                        service.start();
                        onceStarted = true;
                        startButton.setText("Restart");
                }
            }
        });

                exitButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                Platform.exit();
            }
        });

                cancelButton.setOnAction(new EventHandler <ActionEvent>()
```

```
            {
        public void handle(ActionEvent event)
        {
            service.cancel();
        }
    });

            resetButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            service.reset();
        }
    });

            // Enable/Disable the Reset and Cancel buttons
            cancelButton.disableProperty().bind(service.stateProperty().isNotEqualTo( ↩
                RUNNING));
            resetButton.disableProperty().bind(Bindings.or(service.stateProperty(). ↩
                isEqualTo(RUNNING),
                            service.stateProperty().isEqualTo(SCHEDULED)));

            // Create the GridPane
            GridPane pane = new WorkerStateGUI(service);

            // Create the ButtonBox
            HBox buttonBox = new HBox(5, startButton, cancelButton, resetButton,  ↩
                exitButton);

            // Create the BorderPane
            BorderPane root = new BorderPane();
            root.setCenter(pane);
            root.setBottom(buttonBox);

            // Set the Style-properties of the BorderPane
            root.setStyle("-fx-padding: 10;" +
                            "-fx-border-style: solid inside;" +
                            "-fx-border-width: 2;" +
                            "-fx-border-insets: 5;" +
                            "-fx-border-radius: 5;" +
                            "-fx-border-color: blue;");

            // Create the Scene
            Scene scene = new Scene(root,500,400);
            // Add the scene to the Stage
            stage.setScene(scene);
            // Set the title of the Stage
            stage.setTitle("A Prime Number Finder Task");
            // Display the Stage
            stage.show();
    }

    // Create the service
    Service<ObservableList<Long>> service = new Service<ObservableList<Long>>()
    {
            @Override
            protected Task<ObservableList<Long>> createTask()
            {
                    return new PrimeFinderTask();
            }
    };
```

```
}
```

3.4.2 Creating a Service

Remember that a `Service<V>` encapsulates a `Task<V>`. Therefore, you need a `Task<V>` to have a `Service<V>`.

The `Service<V>` class contains an abstract protected `createTask()` method that returns a `Task<V>`.

To create a service, you need to subclass the `Service<V>` class and provide an implementation for the `createTask()` method.

The following snippet of code creates a Service that encapsulates a `PrimeFinderTask`, which you have created earlier:

```
// Create the service
Service<ObservableList<Long>> service = new Service<ObservableList<Long>>()
{
        @Override
        protected Task<ObservableList<Long>> createTask()
        {
                return new PrimeFinderTask();
        }
};
```

The `createTask()` method of the service is called whenever the service is started or restarted.

3.4.3 Updating Service Properties

The `Service` class contains all properties that represent the internal state of a `Worker`. It adds an executor property, which is a `java.util.concurrent.Executor`.

The property is used to run the `Service`. If it is not specified, a daemon thread is created to run the `Service`.

Unlike the `Task` class, the `Service` class does not contain `updateXxx()` methods for updating its properties. Its properties are bound to the corresponding properties of the underlying `Task<V>`.

When the `Task` updates its properties, the changes are reflected automatically to the `Service` and to the client.

3.4.4 Cancelling the Service

Use the `cancel()` methods to cancel a `Service`. The method sets the state of the `Service` to `CANCELLED`.

The following snippet of code shows an example:

```
service.cancel();
```

3.4.5 Starting the Service

Calling the `start()` method of the `Service` class starts a `Service`. The method calls the `createTask()` method to get a `Task` instance and runs the `Task`. The `Service` must be in the `READY` state when its `start()` method is called.

The following snippet of code shows an example:

```
service.start();
```

3.4.6 Resetting the Service

Calling the `reset()` method of the `Service` class resets the `Service`. Resetting puts all the `Service` properties back to their initial states. The state is set to `READY`.

Resetting a `Service` is allowed only when the `Service` is in one of the finish states: `SUCCEEDED`, `FAILED`, `CANCELLED`, or `READY`. Calling the `reset()` method throws a runtime exception if the `Service` is in the `SCHEDULED` or `RUNNING` state.

The following snippet of code shows an example:

```
service.reset();
```

3.4.7 Restarting the Service

Calling the `restart()` method of the `Service` class restarts a `Service`. It cancels the task if it exists, resets the service, and starts it. It calls the three methods on the `Service` object in sequence.

- cancel()
- reset()
- start()

The following snippet of code shows an example:

```
service.restart();
```

3.4.8 The GUI

The following window shows the program after starting:

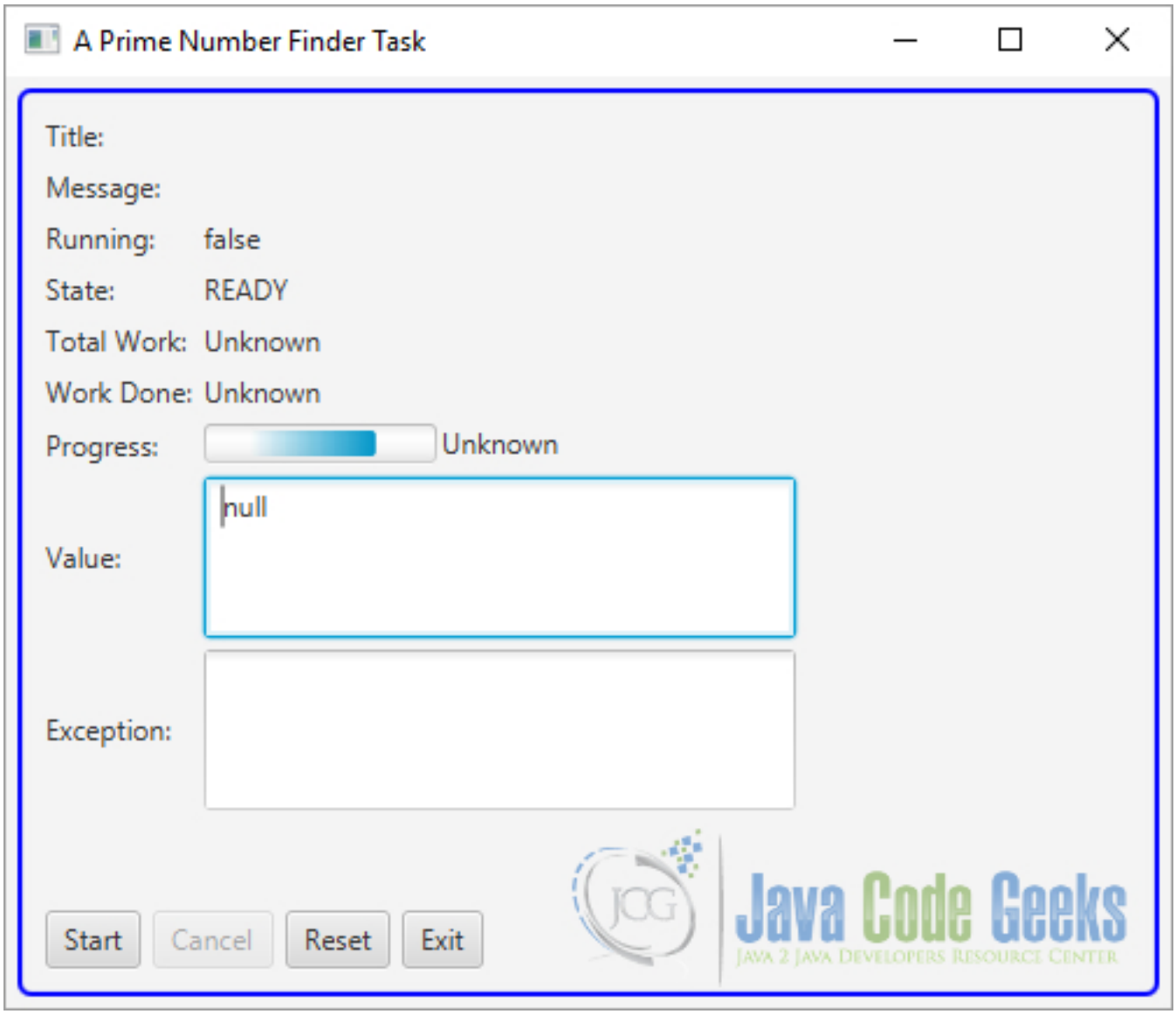

Figure 3.5: The output of the Program before starting the Task

The following GUI shows the program after pressing the Start `Button`:

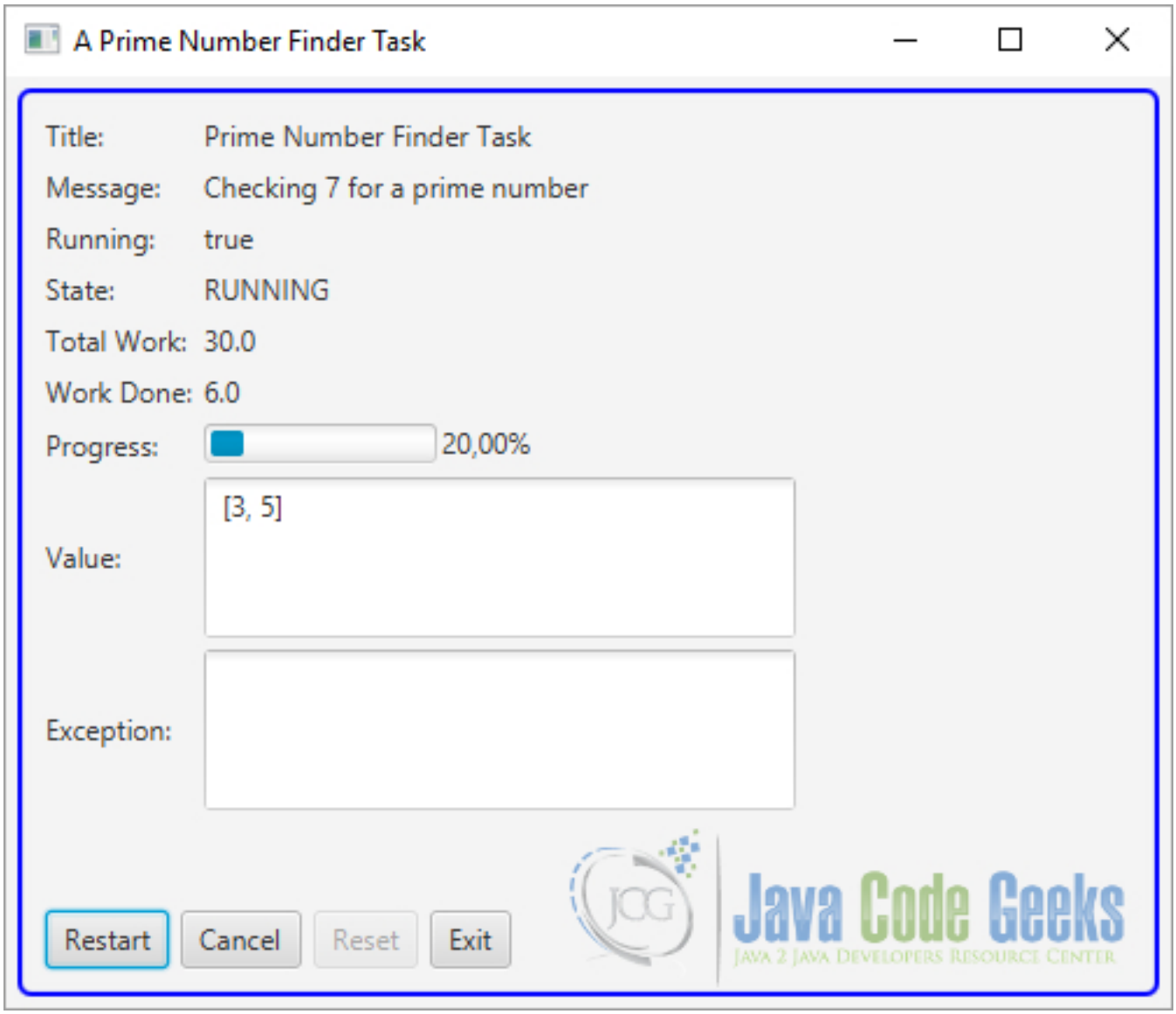

Figure 3.6: The output of the Program during the execution of the Task

After pressing the Cancel `Button`, the following window will appear:

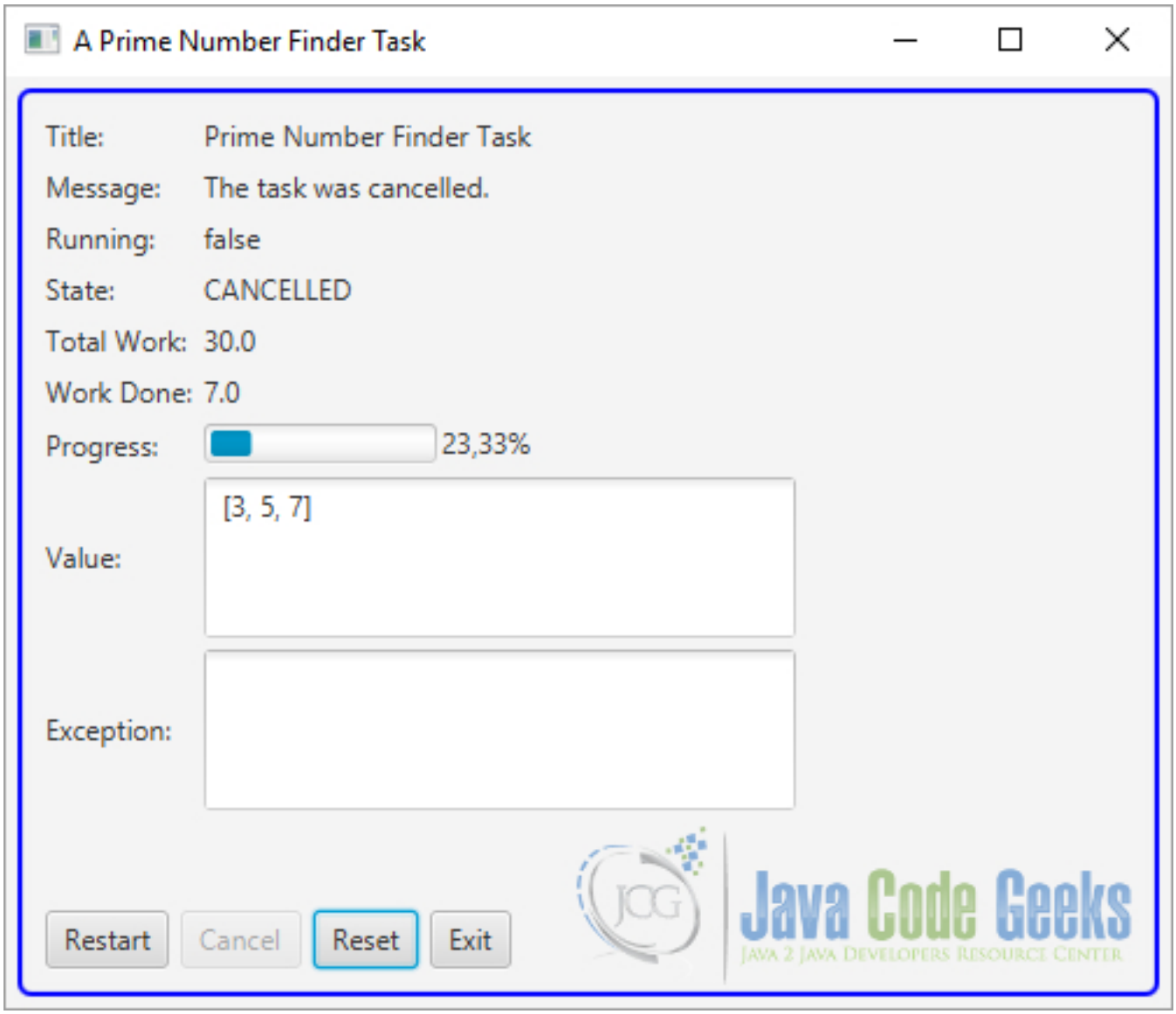

Figure 3.7: The output of the Program after cancelling the running Task

The following GUI shows the program after pressing the Restart `Button`:

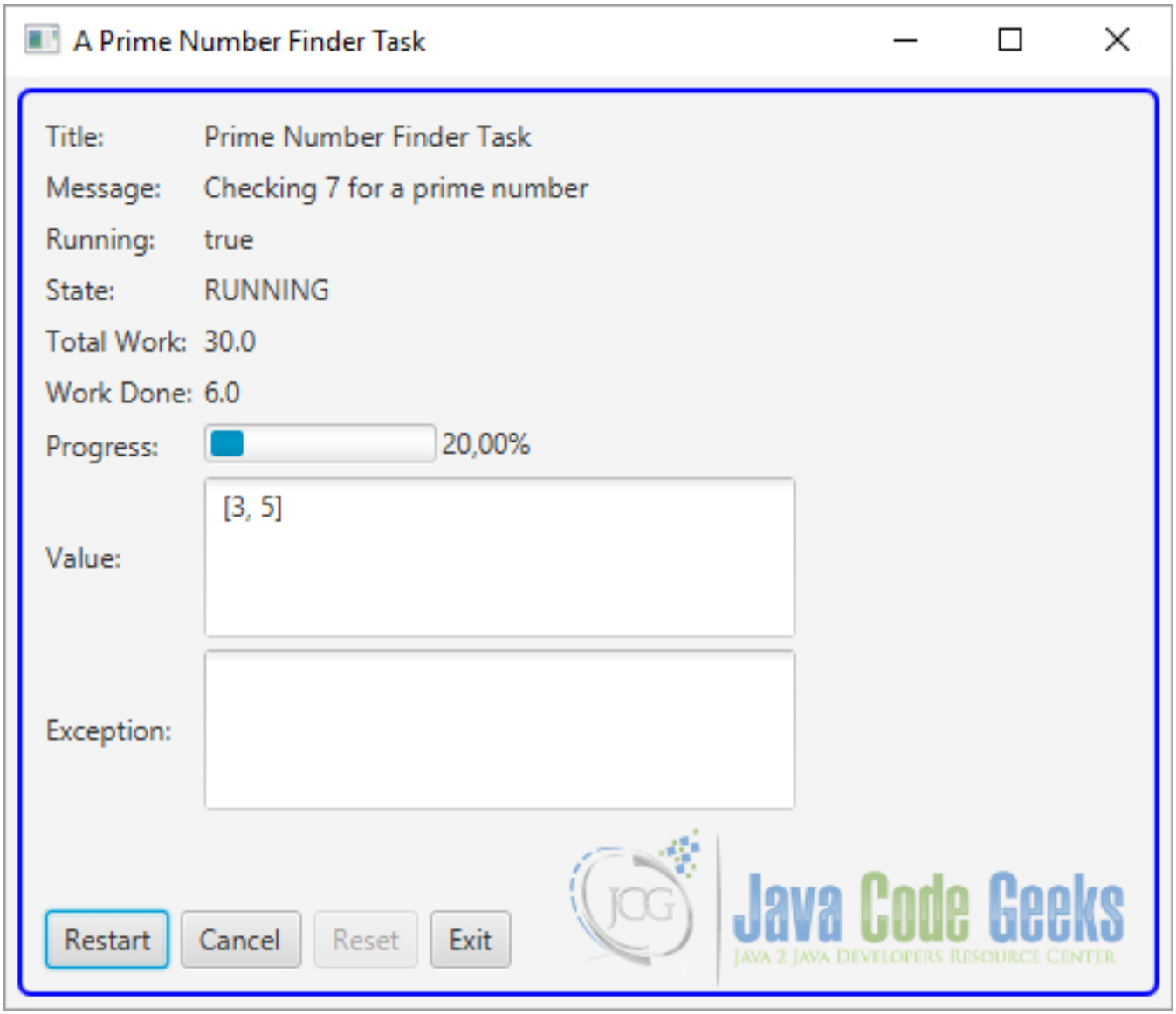

Figure 3.8: The output of the Program during the execution of the Task

3.5 Using the ScheduledService Class

The `ScheduledService<V>` is a `Service<V>`, which automatically restarts. It can restart when it finishes successfully or when it fails. Restarting on a failure is configurable. The `ScheduledService<V>` class inherits from the `Service<V>` class. The `ScheduledService` is suitable for tasks that use polling.

3.5.1 The Code

FxConcurrentExample3.java

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.binding.Bindings;
import javafx.collections.ObservableList;
import javafx.concurrent.ScheduledService;
```

```
import javafx.concurrent.Task;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import static javafx.concurrent.Worker.State.RUNNING;
import static javafx.concurrent.Worker.State.SCHEDULED;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxConcurrentExample3  extends Application
{
        // Create the Buttons
        Button startButton = new Button("Start");
        Button cancelButton = new Button("Cancel");
        Button exitButton = new Button("Exit");
        Button resetButton = new Button("Reset");
        boolean onceStarted = false;

        // Create the scheduled service
        ScheduledService<ObservableList<Long>> service =
                new ScheduledService<ObservableList<Long>>()
                {
                        @Override
                        protected Task<ObservableList<Long>> createTask()
                        {
                                return new PrimeFinderTask();
                        }
                };

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

        @Override
        public void start(Stage stage)
        {
                // Configure the scheduled service
                service.setDelay(Duration.seconds(5));
                service.setPeriod(Duration.seconds(30));
                service.setMaximumFailureCount(5);

                // Create the Event-Handlers for the Buttons
                startButton.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                if (onceStarted)
                {
                        service.restart();
                }
                else
                {
                        service.start();
                        onceStarted = true;
                        startButton.setText("Restart");
                }
            }
```

```
        });

            exitButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            Platform.exit();
        }
        });

            cancelButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            service.cancel();
        }
        });

            resetButton.setOnAction(new EventHandler <ActionEvent>()
            {
        public void handle(ActionEvent event)
        {
            service.reset();
        }
        });

            // Enable/Disable the Reset and Cancel buttons
            cancelButton.disableProperty().bind(service.stateProperty().isNotEqualTo( ↩
                RUNNING));
            resetButton.disableProperty().bind(Bindings.or(service.stateProperty(). ↩
                isEqualTo(RUNNING),
                    service.stateProperty().isEqualTo(SCHEDULED)));

            // Create the GridPane
            GridPane pane = new WorkerStateGUI(service);

            // Create the ButtonBox
            HBox buttonBox = new HBox(5, startButton, cancelButton, resetButton,  ↩
                exitButton);

            // Create the BorderPane
            BorderPane root = new BorderPane();
            root.setCenter(pane);
            root.setBottom(buttonBox);

            // Set the Style-properties of the BorderPane
            root.setStyle("-fx-padding: 10;" +
                          "-fx-border-style: solid inside;" +
                          "-fx-border-width: 2;" +
                          "-fx-border-insets: 5;" +
                          "-fx-border-radius: 5;" +
                          "-fx-border-color: blue;");

            // Create the Scene
            Scene scene = new Scene(root,500,400);
            // Add the scene to the Stage
            stage.setScene(scene);
            // Set the title of the Stage
            stage.setTitle("A Prime Number Finder Task");
            // Display the Stage
            stage.show();
    }
```

```
}
```

3.5.2 Creating a ScheduledService

The process of creating a `ScheduledService` is the same as that of creating a `Service`. You need to subclass the `ScheduledService<V>` class and provide an implementation for the `createTask()` method.

The following snippet of code creates a `ScheduledService` that encapsulates a `PrimeFinderTask`, which you have created earlier:

```
// Create the scheduled service
ScheduledService<ObservableList<Long>> service =
new ScheduledService<ObservableList<Long>>()
{
        @Override
        protected Task<ObservableList<Long>> createTask()
        {
                return new PrimeFinderTask();
        }
};
```

The `createTask()` method of the service is called when the service is started or restarted manually or automatically.

Note that a `ScheduledService` is automatically restarted. You can start and restart it manually by calling the `start()` and `restart()` methods.

3.5.3 Updating ScheduledService Properties

The `ScheduledService<V>` class inherits properties from the `Service<V>` class. It adds the following properties that can be used to configure the scheduling of the `Service`.

- lastValue
- delay
- period
- restartOnFailure
- maximumFailureCount
- backoffStrategy
- cumulativePeriod
- currentFailureCount
- maximumCumulativePeriod

A `ScheduledService<V>` is designed to run several times. The current value computed by the `Service` is not very meaningful. Your class adds a new property `lastValue`, which is of the type `V`, and it is the last value computed by the `Service`.

The `delay` is a duration, which specifies a delay between when the `Service` is started and when it begins running. The `Service` stays in the `SCHEDULED` state for the specified delay. The delay is honored only when the `Service` is started manually calling the `start()` or `restart()` method. When the `Service` is restarted automatically, honoring the delay property depends on the current state of the `Service`.

The `period` is a duration, which specifies the minimum amount of time between the last run and the next run. The default period is zero.

The `restartOnFailure` specifies whether the `Service` restarts automatically when it fails. By default, it is set to true.

The `currentFailureCount` is the number of times the scheduled `Service` has failed. It is reset to zero when the scheduled `Service` is restarted manually.

The `maximumFailureCount` specifies the maximum number of times the `Service` can fail before it is transitioned into the `FAILED` state and it is not automatically restarted again.

The `backoffStrategy` is a `Callback<ScheduledService<?>,Duration>` that computes the duration to add to the period on each failure. Typically, if a `Service` fails, you want to slow down before retrying it.

Suppose a `Service` runs every 10 minutes.

The `rerun` gaps are computed based on the non-zero period and the current failure count.

The `cumulativePeriod` is a duration, which is the time between the current failed run and the next run.

3.5.4 Listening to ScheduledService Transition Events

The `ScheduledService` goes through the same transition states as the `Service`. It goes through the `READY`, `SCHEDULED`, and `RUNNING` states automatically after a successful run. Depending on how the scheduled service is configured, it may go through the same state transitions automatically after a failed run.

You can listen to the state transitions and override the transition-related methods as you can for a `Service`. When you override the transition-related methods in a `ScheduledService` subclass, make sure to call the super method to keep your `ScheduledService` working properly.

3.5.5 The GUI

The following image show the state of the `ScheduledService` when it is not started:

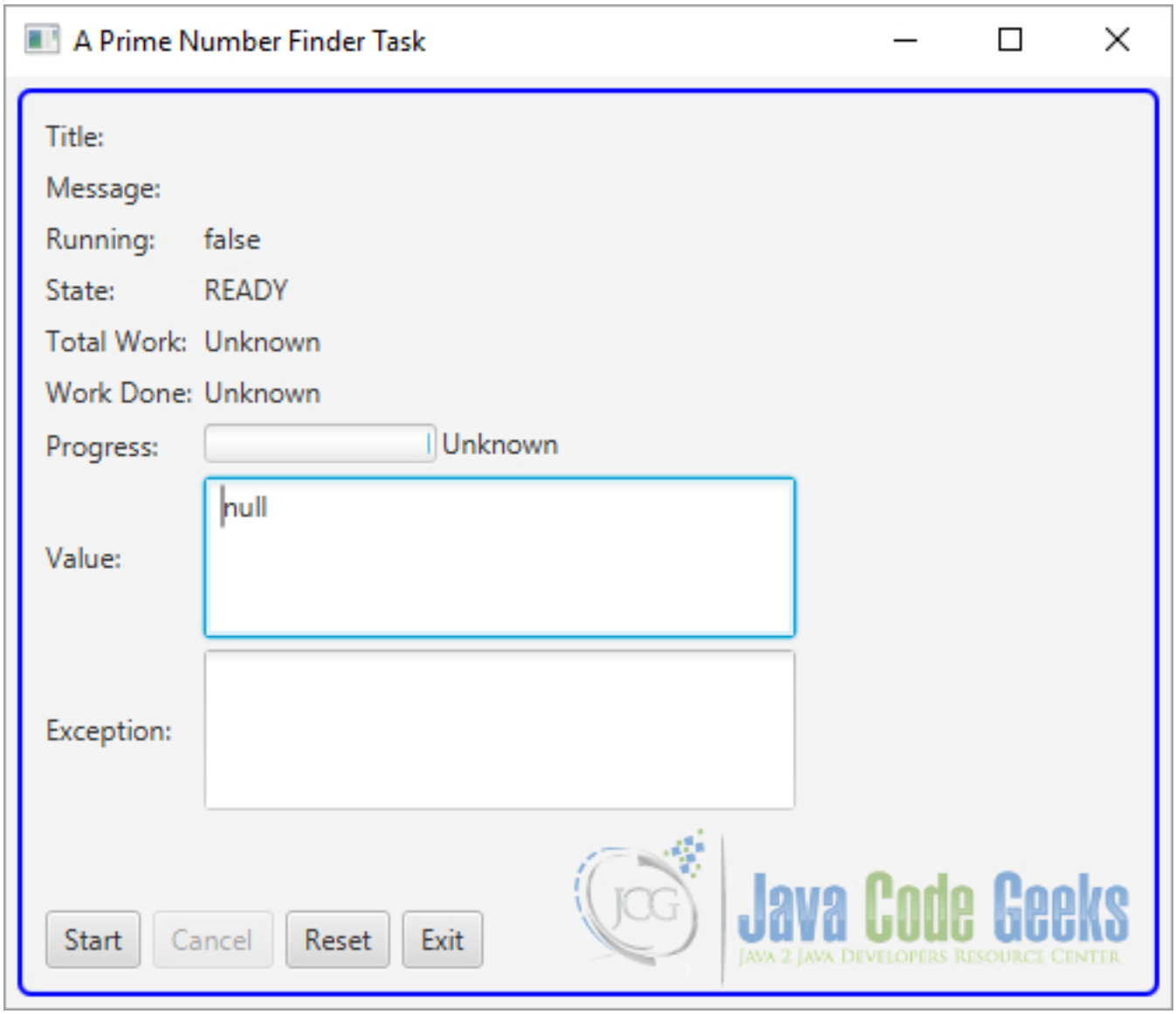

Figure 3.9: The output of the Program before starting the Task

The next image shows the `Service`, when it is running:

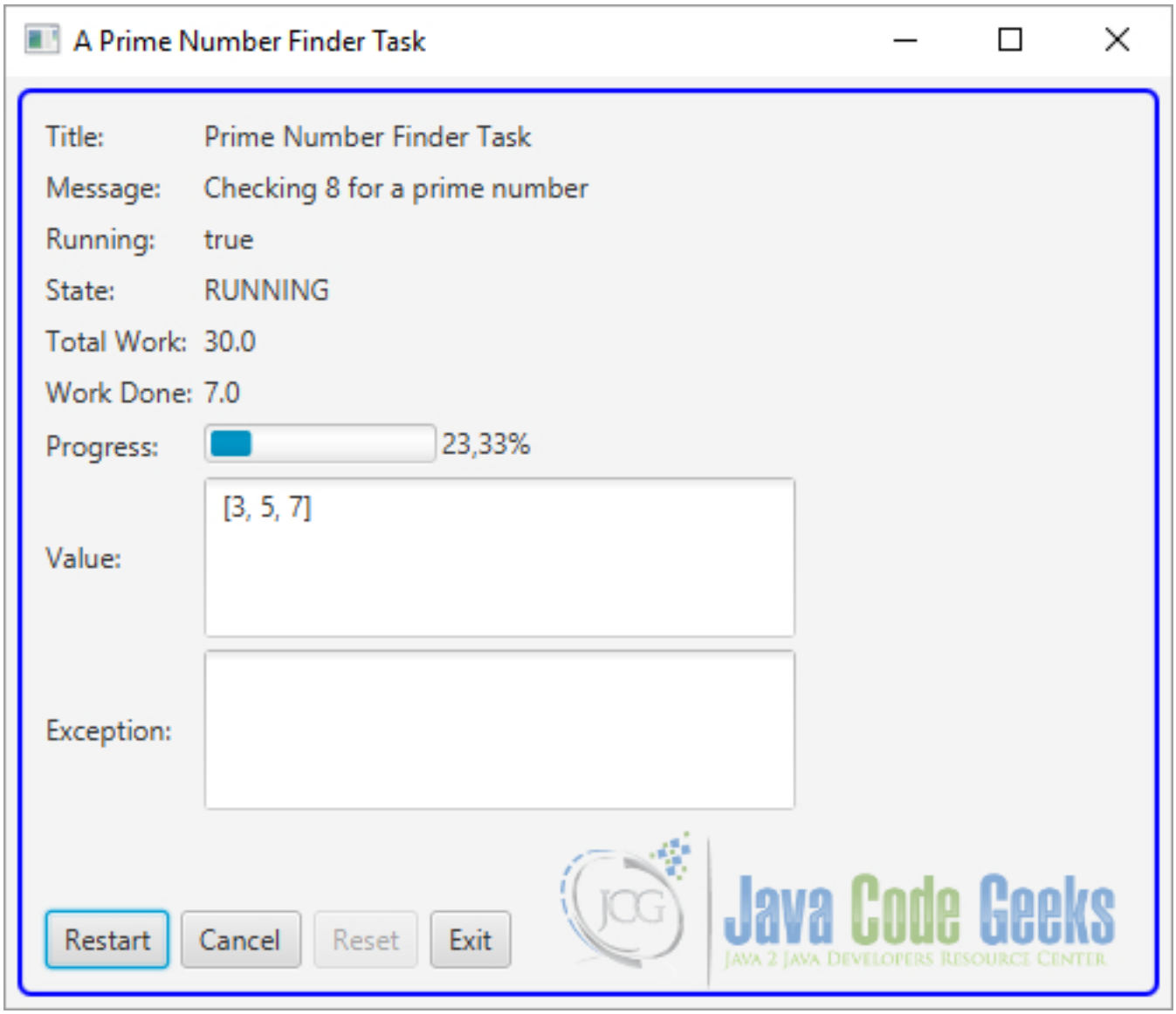

Figure 3.10: The output of the Program during the execution of the Task

The following imgae shows the program after cancelling:

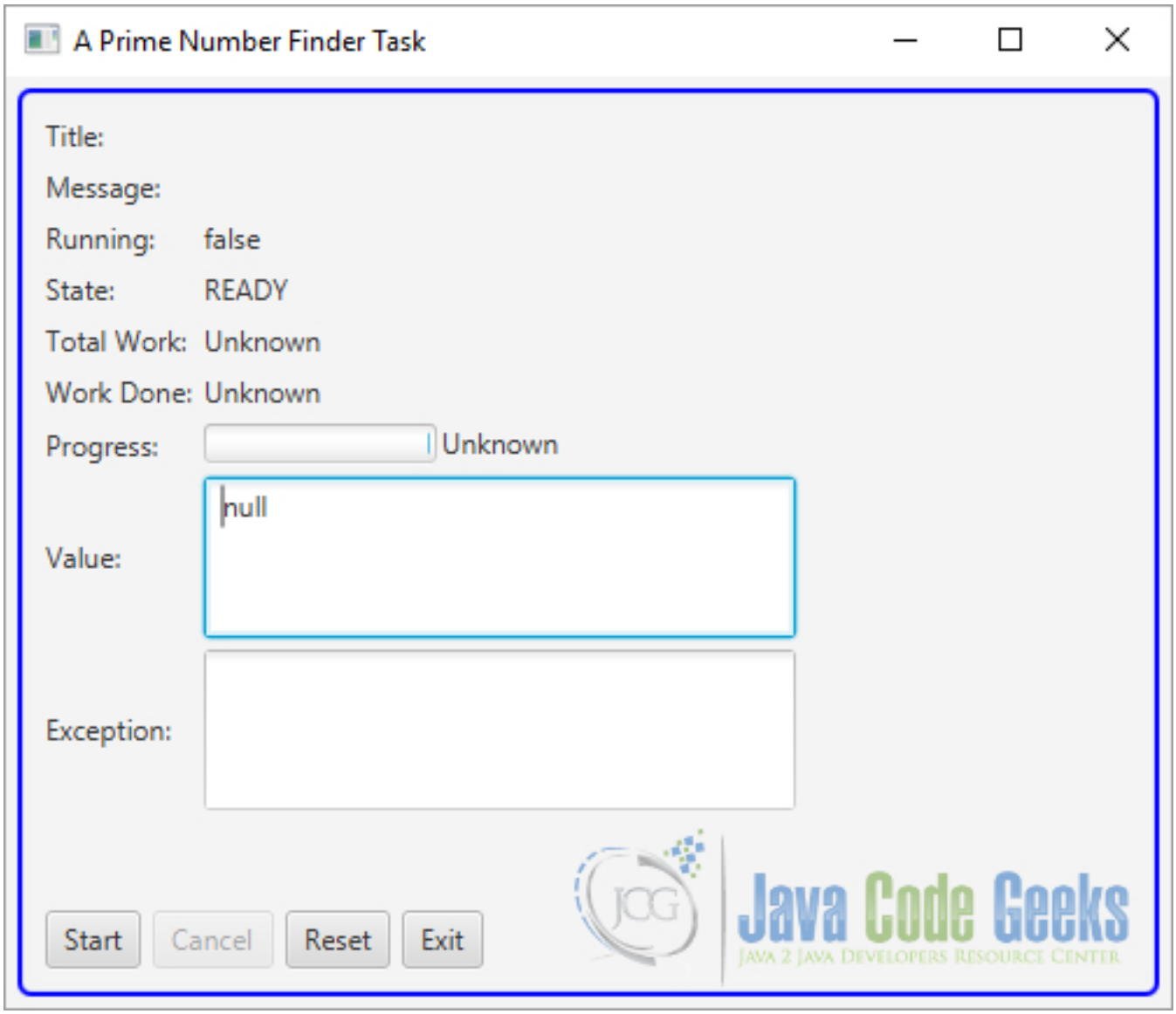

Figure 3.11: The output of the Program after resetting the running Task

The last image shows the application after restarting:

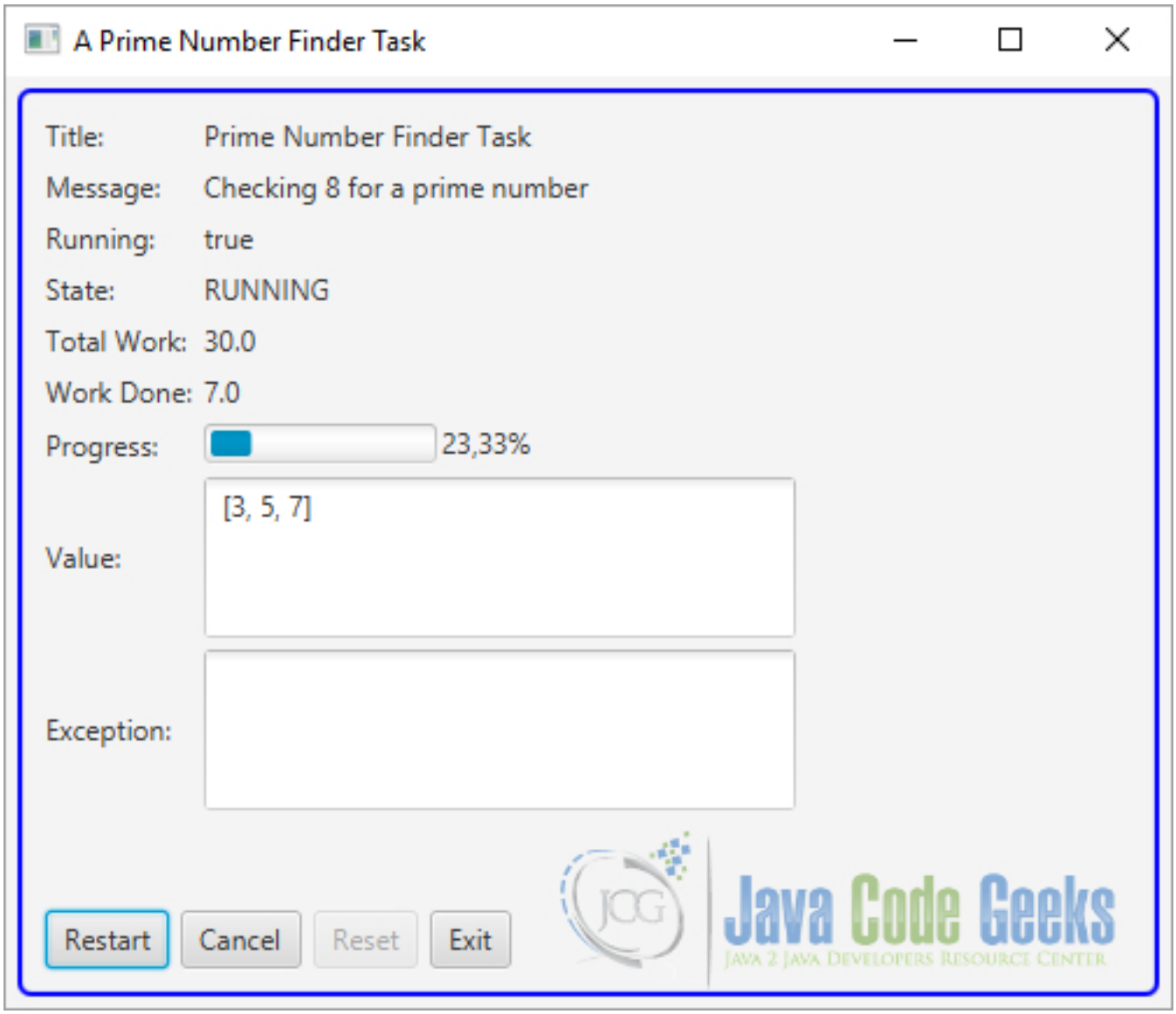

Figure 3.12: The output of the Program after canceling the running Task

3.6 Download Java Source Code

This was an example of `javafx.concurrent`

Download

You can download the full source code of this example here: **JavaFxConcurrentExample.zip**

Chapter 4

JavaFX Animation Example

This is a JavaFX Animation Example. In real world, animation implies some kind of motion, which is generated by displaying images in quick succession. For example, when you watch a movie, you are watching images, which change so quickly that you get an illusion of motion.

In JavaFX, animation is defined as changing the property of a node over time. If the property that changes determines the location of the node, the animation in JavaFX will produce an illusion of motion as found in movies.

Not all animations have to involve motion; for example, changing the fill property of a shape over time is an animation in JavaFX that does not involve motion.

The following table shows an overview of the whole article:

The following examples uses Java SE 7 and JavaFX 2.2.

4.1 Introduction

To understand how `Animation` is performed, it is important to understand some key concepts.

- Timeline
- Key frame
- Key value
- Interpolator

Animation is performed over a period of time. A Timeline denotes the progression of time during animation with an associated key frame at a given instant. A key frame represents the state of the Node being animated at a specific instant on the timeline. A key frame has associated key values. A key value represents the value of a property of the node along with an Interpolator to be used.

4.1.1 The Duration Class

The Duration class is in the `javafx.util package`. It represents a duration of time in milliseconds, seconds, minutes, and hours. It is an immutable class. A `Duration` represents the amount of time for each cycle of an animation. A `Duration` can represent a positive or negative duration.

You can create a `Duration` object in three ways.

- Using the constructor
- Using factory methods

- Using the valueOf() method from a duration in String format

The constructor takes the amount of time in milliseconds.

```
Duration tenMillis = new Duration(10);
```

Factory methods create `Duration` objects for different units of time. They are `millis()`, `seconds()`, `minutes()`, and `hours()`.

```
Duration tenMillis = Duration.millis(10);
Duration tenSeconds = Duration.seconds(10);
Duration tenMinutes = Duration.minutes(10);
Duration tenHours = Duration.hours(10);
```

The `valueOf()` static method takes a String argument containing the duration of time and returns a `Duration` object. The format of the argument is "number[ms|s|m|h]", where number is the amount of time, and ms, s, m, and h denote milliseconds, seconds, minutes, and hours, respectively.

```
Duration tenMillis = Duration.valueOf("10.0ms");
Duration tenMililsNeg = Duration.valueOf("-10.0ms");
```

You can also represent a duration of an unknown amount of time and an indefinite time using the `UNKNOWN` and `INDEFINITE` constants of the `Duration` class, respectively. You can use the `isIndefinite()` and `isUnknown()` methods to check if a duration represents an indefinite or unknown amount of time.

4.1.2 The KeyValue Class

An instance of the KeyValue class represents a key value that is interpolated for a particular interval during animation. It encapsulates three things.

- A target
- An end value for the target
- An interpolator

The target is a `WritableValue`, which qualifies all JavaFX properties to be a target. The end value is the value for the target at the end of the interval. The `Interpolator` is used to compute the intermediate key frames.

A key frame contains one or more key values and it defines a specific point on a `Timeline`. A interval is defined by two or more key frames. Each key frame contains a key value.

An `Animation` may progress forward or backward on the `Timeline`. When an interval starts, the end value of the target is taken from the key value of the end key frame of the interval and its `nterpolator` is used to compute the intermediate key frames.

The `KeyValue` class is immutable. It provides two constructors:

- KeyValue(WritableValue target, T endValue)
- KeyValue(WritableValue target, T endValue, Interpolator interpolator)

The `Interpolator.LINEAR` is used as the default `Interpolator` that interpolates the animated property linearly with time.

The following snippet of code creates a `Text` object and two `KeyValue` objects. The `translateX` property is the target. 0 and 100 are the end values for the target. The default interpolator is used.

```
Text msg = new Text("This is a JavaFX Animation Example");
KeyValue startKeyValue = new KeyValue(msg.translateXProperty(), 0.0);
KeyValue endKeyValue = new KeyValue(msg.translateXProperty(), 100.0);
```

4.1.3 The KeyFrame Class

A key frame defines the target state of a `Node` at a specified point on the `Timeline`. The target state is defined by the key values associated with the key frame.

A key frame encapsulates four things:

- An instant on the timeline
- A set of KeyValues
- A name
- An ActionEvent handler

The instant on the `Timeline` with which the key frame is associated is defined by a `Duration`, which is an offset of the key frame on the timeline.

- The set of KeyValues defines the end value of the target for the key frame.

A key frame may optionally have a name that can be used as a cue point to jump to the instant defined by it during an animation. The `getCuePoints()` method of the `Animation` class returns a Map of cue points on the `Timeline`.

Optionally, you can attach an ActionEvent handler to a KeyFrame. The `ActionEvent` handler is called when the time for the key frame arrives during animation.

An instance of the `KeyFrame` class represents a key frame. The class provides several constructors:

- KeyFrame(Duration time, EventHandler onFinished, KeyValue... values)
- KeyFrame(Duration time, KeyValue... values)
- KeyFrame(Duration time, String name, EventHandler onFinished, Collection values)
- KeyFrame(Duration time, String name, EventHandler onFinished, KeyValue... values)
- KeyFrame(Duration time, String name, KeyValue... values)

The following snippet of code creates two instances of `KeyFrame` that specify the `translateX` property of a `Text` node at 0 seconds and 10 seconds on a `Timeline`:

```
Text msg = new Text("This is a JavaFX Animation Example");
KeyValue startKeyValue = new KeyValue(msg.translateXProperty(), 0.0);
KeyValue endKeyValue = new KeyValue(msg.translateXProperty(), 100.0);
KeyFrame startFrame = new KeyFrame(Duration.ZERO, startKeyValue);
KeyFrame endFrame = new KeyFrame(Duration.seconds(3), endKeyValue);
```

4.2 The Timeline Animation

4.2.1 The Code

FxAnimationExample1.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
```

```
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxAnimationExample1 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text
                Text text = new Text("A Scrolling Text!");
                // Set the Font of the Text
                text.setFont(Font.font(36));

                // Create the VBox
                VBox root = new VBox(text);
                // Set the Size of the Pane
                root.setPrefSize(400, 100);
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("A simple JavaFX Animation Example");
                // Display the Stage
                stage.show();

                // Get the Width of the Scene and the Text
                double sceneWidth = scene.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the start and end Key Frames
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth ↩
                    );
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create a Timeline
                Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(Timeline.INDEFINITE);
                // Run the animation
                timeline.play();
```

```
        }
}
```

A timeline animation is used for animating any properties of a `Node`. An instance of the `Timeline` class represents a timeline animation. Using a timeline animation involves the following steps:

- Construct key frames
- Create a Timeline object with key frames
- Set the animation properties
- Use the play() method to run the animation

You can add key frames to a `Timeline` at the time of creating it or after. The `Timeline` instance keeps all key frames in an ObservableList object. The `getKeyFrames()` method returns the list. You can modify the list of key frames at any time. If the timeline animation is already running, you need to stop and restart it to pick up the modified list of key frames.

The `Timeline` class contains several constructors.

- Timeline()
- Timeline(double targetFramerate)
- Timeline(double targetFramerate, KeyFrame... keyFrames)
- Timeline(KeyFrame... keyFrames)

The no-args constructor creates a `Timeline` with no key frames with animation running at the optimum rate. Other constructors let you specify the target frame rate for the animation, which is the number of frames per second, and the key frames. Note that the order in which the key frames are added to a `Timeline` is not important. `Timeline` will order them based on their time offset.

The logic to perform the animation is in the `start()` method. The method starts with creating a Text object, a VBox with the `Text` object, and setting up a Scene for the Stage. After showing the `Stage`, it sets up an animation. It gets the width of the `Scene` and the `Text` object.

```
// Get the Width of the Scene and the Text
double sceneWidth = scene.getWidth();
double textWidth = text.getLayoutBounds().getWidth();
```

Two key frames are created: one for time = 0 seconds and one for time = 10 seconds. The animation uses the `translateX` property of the `Text` object to change its horizontal position to make it scroll. At 0 seconds, the `Text` is positioned at the `Scene` width, so it is invisible. At 10 seconds, it is placed to the left of the scene at a distance equal to its length, so again it is invisible.

```
// Define the Durations
Duration startDuration = Duration.ZERO;
Duration endDuration = Duration.seconds(10);

// Create the start and end Key Frames
KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth);
KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 * textWidth);
KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

// Create a Timeline
Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
```

By default, the animation will run only one time. That is, the Text will scroll from right to left once and the animation will stop. You can set the cycle count for an animation, which is the number of times the animation needs to run. You run the animation forever by setting the cycle count to `Timeline.INDEFINITE`.

```
// Let the animation run forever
timeline.setCycleCount(Timeline.INDEFINITE);
```

Finally, the animation is started by calling the `play()` method.

```
// Run the animation
timeline.play();
```

4.2.2 The GUI

Figure 4.1: A simple JavaFX Animation Example with a scrolling Text

4.3 Controlling an Animation

The `Animation` class contains properties and methods that can be used to control animation in various ways. The following sections will explain those properties and methods and how to use them to control animation.

4.3.1 Playing, Stopping and Pausing an Animation

4.3.1.1 The Code

FxAnimationExample2.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
```

```
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxAnimationExample2 extends Application
{
        // Create the Timeline
        Timeline timeline = new Timeline();
        // Create the Label
        Label status = new Label("Current State: " + timeline.getStatus());

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

        @Override
        public void start(Stage stage) throws InterruptedException
        {
                // Create the Text
                Text text = new Text("A Scrolling Text!");
                // Set the Font of the Text
                text.setFont(Font.font(36));

                // Create the Buttons
                Button play = new Button("Play");
                Button pause = new Button("Pause");
                Button replay = new Button("RePlay");
                Button stop = new Button("Stop");

                // Create the Event-Handlers for the Buttons
                play.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                timeline.play();
                status.setText("Current State: " + timeline.getStatus());
            }
        });

                replay.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                timeline.playFromStart();
                status.setText("Current State: " + timeline.getStatus());
            }
        });

                pause.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                timeline.pause();
                status.setText("Current State: " + timeline.getStatus());
            }
        });

                stop.setOnAction(new EventHandler <ActionEvent>()
                {
            public void handle(ActionEvent event)
            {
                timeline.stop();
```

```
                status.setText("Current State: " + timeline.getStatus());
            }
        });

                // Create the Button Box
                HBox buttonBox = new HBox();
                // Set Spacing to 10 pixels
                buttonBox.setSpacing(10);
                // Add the Children to the HBox
                buttonBox.getChildren().addAll(play,pause,replay,stop);

                // Create the VBox
                VBox root = new VBox(buttonBox,status,text);
                // Set Spacing to 10 pixels
                root.setSpacing(10);
                // Set the Size of the VBox
                root.setPrefSize(500, 200);
                // Set the Style-properties of the VBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("A JavaFX Animation Control Example");
                // Display the Stage
                stage.show();

                // Get the Scene width and the Text width
                double sceneWidth = scene.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the start and end Key Frames
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth ↩
                    );
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create a Timeline
                timeline = new Timeline(startKeyFrame, endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(Timeline.INDEFINITE);
        }
}
```

4.3.1.2 Playing an Animation

The `Animation` class contains four methods to play an animation:

- play()
- playFrom(Duration time)
- playFrom(String cuePoint)
- playFromStart()

The `play()` method plays an animation from its current position. If the animation was never started or stopped, it will play from the beginning. If the animation was paused, it will play from the position where it was paused. You can use the `jumpTo(Duration time)` and `jumpTo(String cuePoint)` methods to set the current position of the animation to a specific duration or a cue point, before calling the `play()` method.

Calling the `play()` method is asynchronous. The animation may not start immediately. Calling the `play()` method while animation is running has no effect.

```
// Create the Event-Handlers for the Buttons
play.setOnAction(new EventHandler <ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        timeline.play();
        status.setText("Current State: " + timeline.getStatus());
    }
});
```

The `playFrom()` method plays an animation from the specified duration or the specified cue point. Calling this method is equivalent to setting the current position using the `jumpTo()` method and then calling the `play()` method.

The `playFromStart()` method plays the animation from the beginning (duration = 0).

The following snippet of code will play the animation after pressing the Play Button:

```
replay.setOnAction(new EventHandler <ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        timeline.playFromStart();
        status.setText("Current State: " + timeline.getStatus());
    }
});
```

4.3.1.3 Stopping an Animation

Use the `stop()` method to stop a running animation. The method has no effect if the animation is not running. The animation may not stop immediately when the method is called as the method executes asynchronously. The method resets the current position to the beginning. That is, calling `play()` after `stop()` will play the animation from the beginning.

The following code snippet stops an animation after pressing the Stop `Button`:

```
stop.setOnAction(new EventHandler <ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        timeline.stop();
        status.setText("Current State: " + timeline.getStatus());
    }
});
```

4.3.1.4 Pausing an Animation

Use the `pause()` method to pause an animation. Calling this method when animation is not running has no effect. This method executes asynchronously. Calling the `play()` method when the animation is paused plays it from the current position. If you want to play the animation from the start, call the `playFromStart()` method.

The following snippet of code will pause the animation:

```
pause.setOnAction(new EventHandler <ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        timeline.pause();
        status.setText("Current State: " + timeline.getStatus());
    }
});
```

4.3.1.5 Knowing the State of an Animation

An `Animation` can be one of the following three states:

- Running
- Paused
- Stopped

The three states are represented by `RUNNING`, `STOPPED`, and `PAUSED` constants of the `Animation.Status` enum. You do not change the state of an animation directly. It is changed by calling one of the methods of the `Animation` class. The class contains a read-only status property that can be used to know the state of the animation at any time.

The following snippet of code reads the current status of the animation:

```
status.setText("Current State: " + timeline.getStatus());
```

4.3.1.6 The GUI

Figure 4.2: A JavaFX Animation Control Example

4.3.2 Delaying the Start of an Animation

4.3.2.1 The Code

FxAnimationExample3.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxAnimationExample3 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text
                Text text = new Text("A Scrolling Text!");
                // Set the Font of the Text
                text.setFont(Font.font(36));

                // Create the VBox
                VBox root = new VBox(text);
                // Set the Size of the VBox
                root.setPrefSize(500, 100);
                // Set the Style-properties of the VBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("A JavaFX Duration Animation Example");
                // Display the Stage
                stage.show();

                // Get the Width of the Scene and the Text
                double sceneWidth = scene.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the start and end Key Frames
```

```
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth ↩
                    );
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create a Timeline
                Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(Timeline.INDEFINITE);
                // Delay the start of the animation by 2 seconds
                timeline.setDelay(Duration.seconds(2));
                // Run the animation
                timeline.play();
        }
}
```

You can specify a delay in starting the animation using the delay property. The value is specified in `Duration`. By default, it is 0 milliseconds.

In the following example, the animation starts with a delay of two seconds:

```
// Create a Timeline
Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
// Let the animation run forever
timeline.setCycleCount(Timeline.INDEFINITE);
// Delay the start of the animation by 2 seconds
timeline.setDelay(Duration.seconds(2));
// Run the animation
timeline.play();
```

4.3.2.2 The GUI

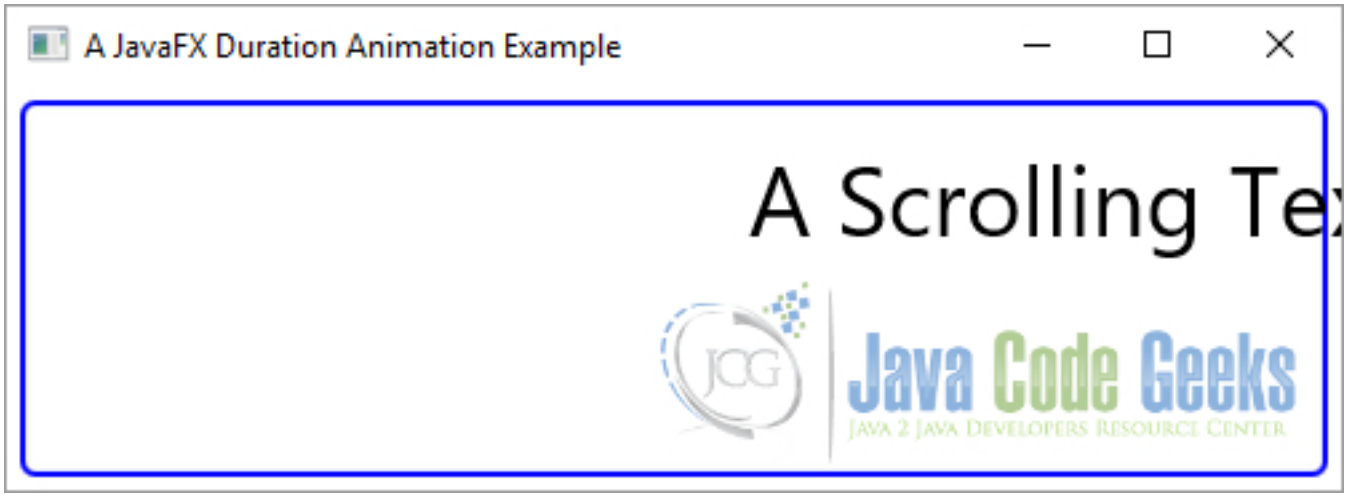

Figure 4.3: A JavaFX Animation Example with a Duration

4.3.3 Looping an Animation

4.3.3.1 The Code

FxAnimationExample4.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxAnimationExample4 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text
                Text text = new Text("A Scrolling Text!");
                // Set the Font of the Text
                text.setFont(Font.font(36));

                // Create the VBox
                VBox root = new VBox(text);
                // Set the Size of the VBox
                root.setPrefSize(500, 100);
                // Set the Style-properties of the VBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("A JavaFX Loop Animation Example");
                // Display the Stage
                stage.show();

                // Get the Width of the Scene and the Text
                double sceneWidth = scene.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the start and end Key Frames
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth ↩
                    );
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
```

```
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create a Timeline
                Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(5);
                // Run the animation
                timeline.play();
        }
}
```

An animation can cycle multiple times, even indefinitely. The `cycleCount` property specifies the number of cycles in an animation, which defaults to 1. If you want to run the animation in an infinite loop, specify `Animation.INDEFINITE` as the `cycleCount`. The `cycleCount` must be set to a value greater than zero. If the `cycleCount` is changed while the animation is running, the animation must be stopped and restarted to pick up the new value.

In the following example, the animation will run five times.

```
// Create a Timeline
Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
// Let the animation run forever
timeline.setCycleCount(5);
// Run the animation
timeline.play();
```

4.3.3.2 The GUI

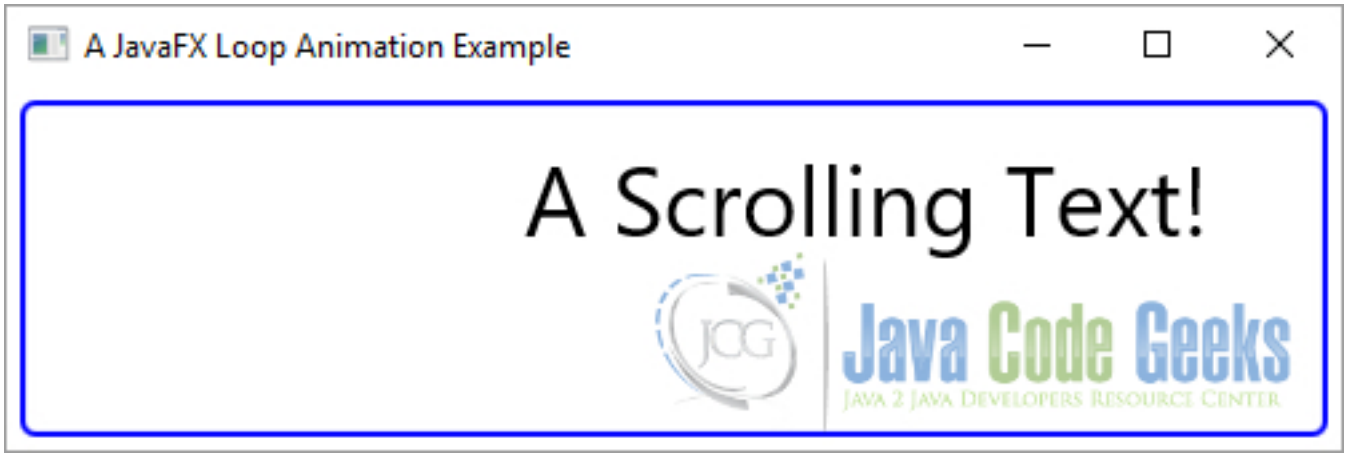

Figure 4.4: A JavaFX Animation Example with a Loop

4.3.4 Auto Reversing an Animation

4.3.4.1 The Code

FxAnimationExample5.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
```

```
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FxAnimationExample5 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text
                Text text = new Text("A Scrolling Text!");
                // Set the Font of te Text
                text.setFont(Font.font(36));

                // Create the VBox
                VBox root = new VBox(text);
                // Set the Size of the VBox
                root.setPrefSize(500, 100);
                // Set the Style-properties of the VBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("A JavaFX Auto Reverse Animation Example");
                // Display the Stage
                stage.show();

                // Get the Width of the Scene and the Text
                double sceneWidth = scene.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the start and end Key Frames
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth ↩
                    );
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create a Timeline
                Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(Timeline.INDEFINITE);
                // Reverse direction on alternating cycles
```

```
                timeline.setAutoReverse(true);
                // Run the animation
                timeline.play();
        }
}
```

By default, an animation runs only in the forward direction. For example, our scrolling text animation scrolled the text from right to left in one cycle. In the next cycle, the scrolling occurs again from right to left. Using the `autoReverse` property, you can define whether the animation is performed in the reverse direction for alternating cycles. By default, it is set to `false`. Set it to `true` to reverse the direction of the animation.

```
// Create a Timeline
Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
// Let the animation run forever
timeline.setCycleCount(Timeline.INDEFINITE);
// Reverse direction on alternating cycles
timeline.setAutoReverse(true);
// Run the animation
timeline.play();
```

If you change the `autoReverse`, you need to stop and restart the animation for the new value to take effect.

4.3.4.2 The GUI

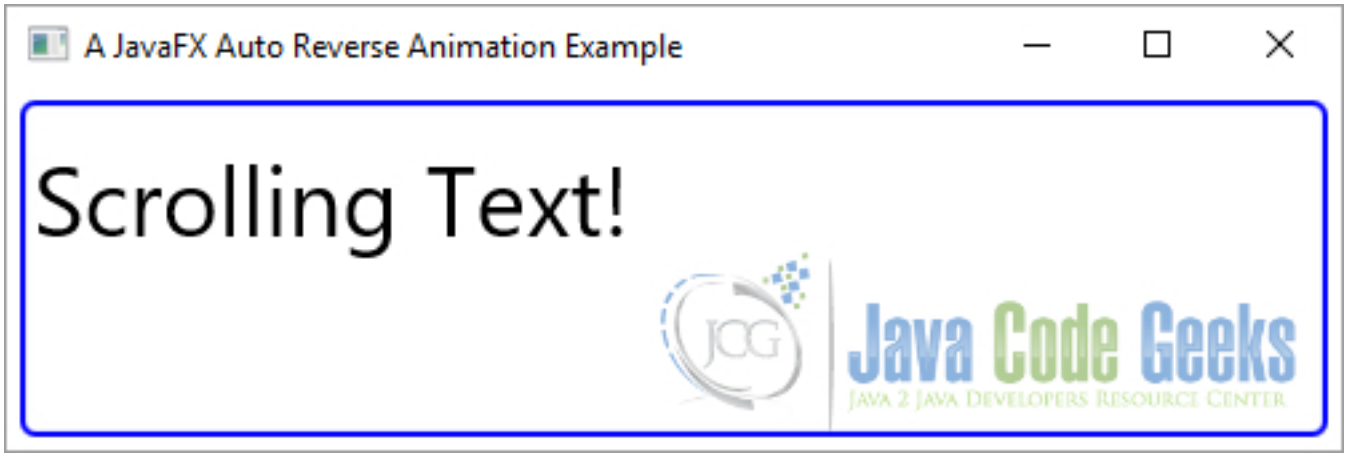

Figure 4.5: A JavaFX Animation Example with Auto Reverse

4.3.5 Adjusting the Speed of an Animation

4.3.5.1 The Code

FxAnimationExample6.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
```

```
import javafx.util.Duration;

public class FxAnimationExample6 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Text
                Text text = new Text("A Scrolling Text!");
                // Set the Font of the Text
                text.setFont(Font.font(36));

                // Create the VBox
                VBox root = new VBox(text);
                // Set the Size of the VBox
                root.setPrefSize(500, 100);
                // Set the Style-properties of the VBox
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("A JavaFX Rate Animation Example");
                // Display the Stage
                stage.show();

                // Get the Width of the Scene and the Text
                double sceneWidth = scene.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the start and end Key Frames
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), sceneWidth ↩
                    );
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create a Timeline
                Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(Timeline.INDEFINITE);
                // Play the animation at double the normal rate
                timeline.setRate(2.0);
                // Run the animation
                timeline.play();
```

```
        }
}
```

The `rate` property of the `Animation` class specifies the direction and the speed for the animation. The sign of its value indicates the direction. The magnitude of the value indicates the speed. A positive value indicates the play in the forward direction. A negative value indicates the play in the backward direction. A value of 1.0 is considered the normal rate of play, a value of 2.0 double the normal rate, 0.50 half the normal rate, and so on. A rate of 0.0 stops the play.

It is possible to invert the rate of a running animation. In that case, the animation is played in the reverse direction from the current position for the duration that has already elapsed. Note that you cannot start an animation using a negative rate. An animation with a negative rate will not start. You can change the rate to be negative only when the animation has played for a while.

```
// Create a Timeline
Timeline timeline = new Timeline(startKeyFrame, endKeyFrame);
// Let the animation run forever
timeline.setCycleCount(Timeline.INDEFINITE);
// Play the animation at double the normal rate
timeline.setRate(2.0);
// Run the animation
timeline.play();
```

4.3.5.2 The GUI

Figure 4.6: A JavaFX Animation Example with a Rate

4.4 Understanding Cue Points

4.4.1 The Code

FxAnimationExample7.java

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.geometry.VPos;
```

```
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;
import javafx.event.EventHandler;

public class FxAnimationExample7 extends Application
{
        // Create the Text
        Text text = new Text("A Scrolling Text!");
        // Create the Pane
        Pane pane = new Pane();
        // Create the Cue Points List View
        ListView<String> cuePointsListView = new ListView<String>();
        // Create the Timeline
        Timeline timeline = new Timeline();

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

        @Override
        public void start(Stage stage)
        {
                // Create the Text
                text.setTextOrigin(VPos.TOP);
                text.setFont(Font.font(24));

                // Set the Size of the ListView
                cuePointsListView.setPrefSize(100, 150);
                // Create the Pane
                pane = new Pane(text);

                // Create the BorderPane
                BorderPane root = new BorderPane();
                // Set the Size of the BorderPane
                root.setPrefSize(600, 250);

                // Set the Style-properties of the BorderPane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Add the Pane and ListView to the BorderPane
                root.setBottom(pane);
                root.setLeft(cuePointsListView);

                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Cue Points");
```

```
                // Display the Stage
                stage.show();

                // Setup the Animation
                this.setupAnimation();
                // Add the Cue Points to the List
                this.addCuePoints();
        }

        private void setupAnimation()
        {
                // Get width of Pane and Text
                double paneWidth = pane.getWidth();
                double textWidth = text.getLayoutBounds().getWidth();

                // Define the Durations
                Duration startDuration = Duration.ZERO;
                Duration endDuration = Duration.seconds(10);

                // Create the initial and final key frames
                KeyValue startKeyValue = new KeyValue(text.translateXProperty(), paneWidth) ↩
                    ;
                KeyFrame startKeyFrame = new KeyFrame(startDuration, startKeyValue);
                KeyValue endKeyValue = new KeyValue(text.translateXProperty(), -1.0 *  ↩
                    textWidth);
                KeyFrame endKeyFrame = new KeyFrame(endDuration, endKeyValue);

                // Create the Timeline
                timeline = new Timeline(startKeyFrame,endKeyFrame);
                // Let the animation run forever
                timeline.setCycleCount(Timeline.INDEFINITE);
                // Play the Animation
                timeline.play();
        }

        private void addCuePoints()
        {
                // Add two cue points directly to the map
                timeline.getCuePoints().put("4", Duration.seconds(4));
                timeline.getCuePoints().put("7", Duration.seconds(7));

                // Add all cue points from the map to the ListView in the order
                // of their durations
                cuePointsListView.getItems().add(0, "Start");
                cuePointsListView.getItems().addAll(1, timeline.getCuePoints().keySet());
                cuePointsListView.getItems().add("End");

                // Add Event Handler to the List
                cuePointsListView.setOnMousePressed(new EventHandler<MouseEvent>()
                {
                @Override
                public void handle(MouseEvent event)
                {
                    timeline.jumpTo(cuePointsListView.getSelectionModel().getSelectedItem() ↩
                        );
                }
            });
        }
}
```

You can set up cue points on a `Timeline`. Cue points are named instants on the `Timeline`. An animation can jump to a cue

point using the `jumpTo(String cuePoint)` method. An animation maintains an ObservableMap of cue points. The key in the map is the name of the cue points and the values are the corresponding duration on the timeline. Use the `getCuePoints()` method to get the reference of the cue points map.

There are two ways to add cue points to a `Timeline`.

- Giving a name to the KeyFrame you add to a timeline that adds a cue point in the cue point map
- Adding name-duration pairs to the map returned by the getCuePoints() method of the Animation class

The following snippet of code adds two cue points directly to the cue point map of a timeline:

```
// Add two cue points directly to the map
timeline.getCuePoints().put("4", Duration.seconds(4));
timeline.getCuePoints().put("7", Duration.seconds(7));

// Add all cue points from the map to the ListView in the order
// of their durations
cuePointsListView.getItems().add(0, "Start");
cuePointsListView.getItems().addAll(1, timeline.getCuePoints().keySet());
cuePointsListView.getItems().add("End");
```

The above example shows how to add and use cue points on a timeline. It adds two cue points, “4 seconds” and “7 seconds”, directly to the cue point map. The list of available cue points is shown in a ListView on the left side of the screen. A `Text` object scrolls with a cycle duration of 10 seconds.

4.4.2 The GUI

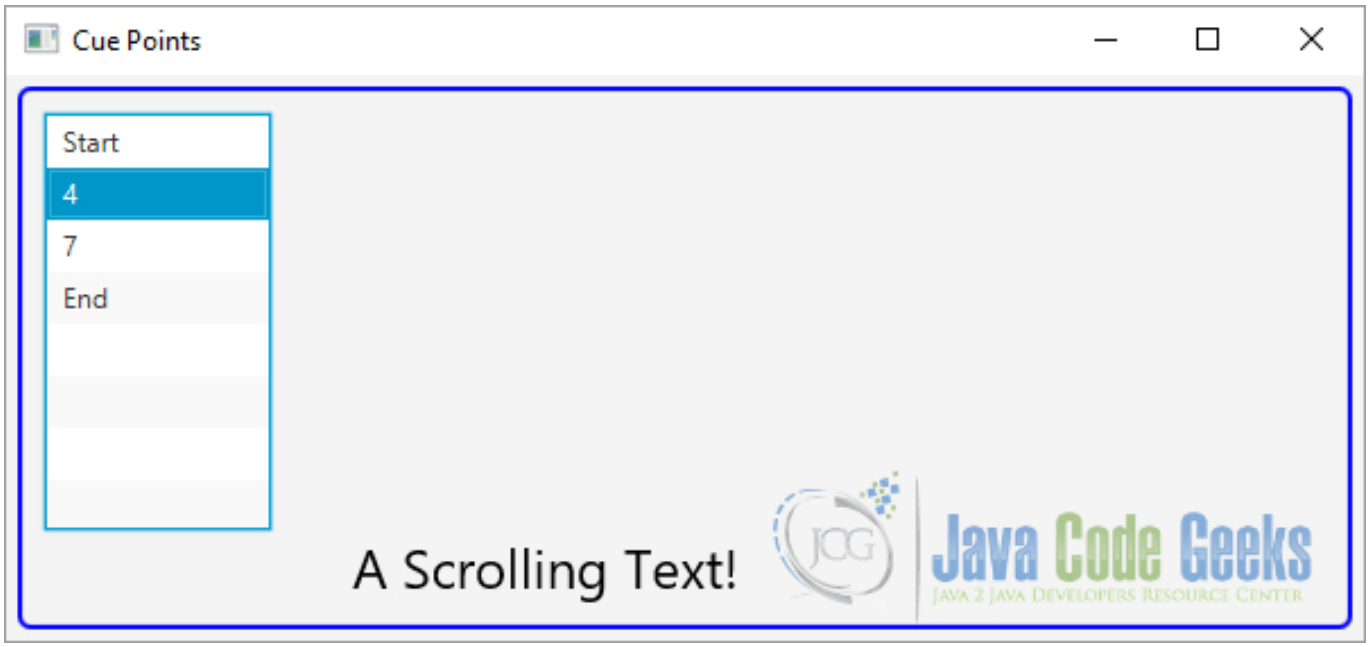

Figure 4.7: A JavaFX Animation Example with Cue Points

4.5 Download Java Source Code

This was an example of `javafx.animation`

Download

You can download the full source code of this example here: **JavaFxAnimationExample.zip**

Chapter 5

JavaFX Canvas Example

This is a JavaFX Canvas Example. Through the `javafx.scene.canvas` package, JavaFX provides the Canvas API that offers a drawing surface to draw shapes, images, and text using drawing commands. The API also gives pixel-level access to the drawing surface where you can write any pixels on the surface. The API consists of only two classes:

- Canvas
- GraphicsContext

A canvas is a bitmap image, which is used as a drawing surface. An instance of the Canvas class represents a canvas. It inherits from the Node class. Therefore, a `Canvas` is a `Node`.

It can be added to a `Scene Graph`, and effects and transformations can be applied to it. A `Canvas` has a graphics context associated with it that is used to issue drawing commands to the `Canvas`. An instance of the GraphicsContext class represents a graphics context.

The following examples uses Java SE 8 and JavaFX 2.2.

5.1 Creating a Canvas

5.1.1 The Code

FxCanvasExample1.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class FxCanvasExample1 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Canvas
```

```
                Canvas canvas = new Canvas(400, 200);
                // Set the width of the Canvas
                canvas.setWidth(400);
                // Set the height of the Canvas
                canvas.setHeight(200);

                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();

                // Draw a Text
                gc.strokeText("Hello Canvas", 150, 100);

                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Creation of a Canvas");
                // Display the Stage
                stage.show();
        }
}
```

The `Canvas` class has two constructors. The no-args constructor creates an empty canvas. Later, you can set the size of the canvas using its width and height properties. The other constructor takes the width and height of the canvas as parameters:

```
// Create a Canvas of zero width and height
Canvas canvas = new Canvas();

// Create a 400X200 canvas
Canvas canvas = new Canvas(400, 200);
```

5.1.2 The GUI

The following image shows the result of the above example:

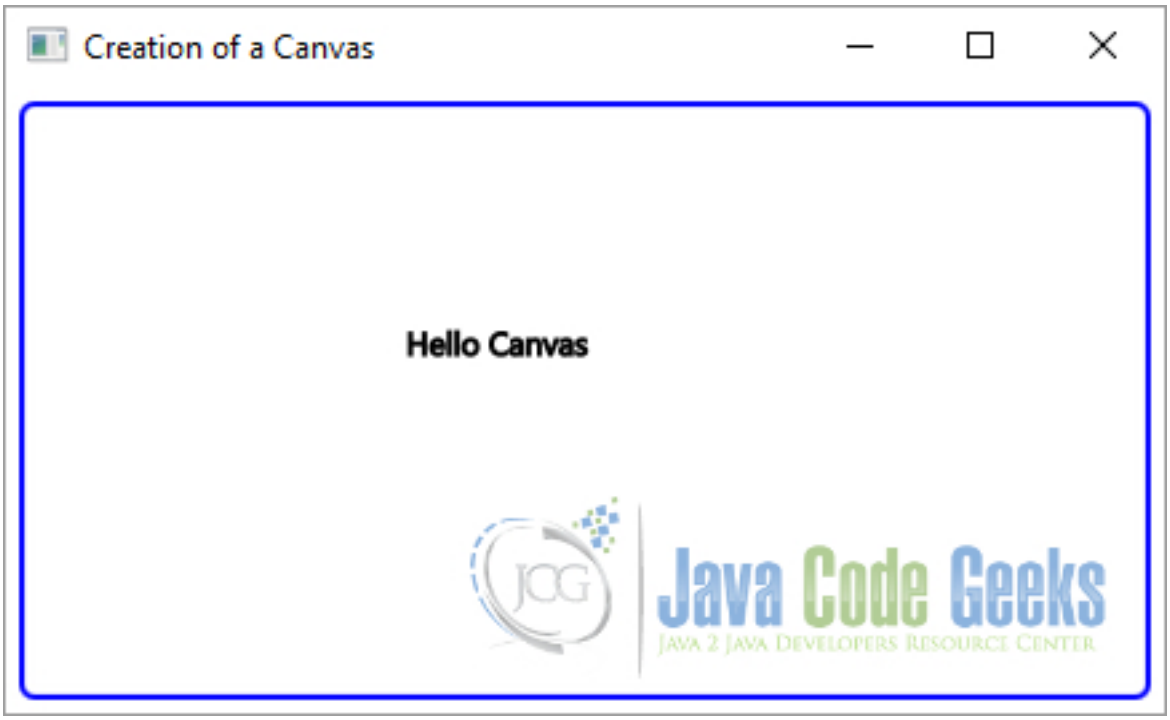

Figure 5.1: A simple JavaFX Canvas Example

5.2 Drawing on the Canvas

5.2.1 Introduction

Once you create a canvas, you need to get its graphics context using the `getGraphicsContext2D()` method, as in the following snippet of code:

```
// Get the graphics context of the canvas
GraphicsContext gc = canvas.getGraphicsContext2D();
```

All drawing commands are provided in the `GraphicsContext` class as methods. Drawings that fall outside the bounds of the `Canvas` are clipped. The canvas uses a buffer. The drawing commands push necessary parameters to the buffer. It is important to note that you should use the graphics context from any one thread before adding the `Canvas` to the Scene Graph.

Once the `Canvas` is added to the Scene Graph, the graphics context should be used only on the JavaFX Application Thread. The `GraphicsContext` class contains methods to draw the following types of objects:

- Basic shapes
- Text
- Paths
- Images
- Pixels

5.2.2 Drawing Basic Shapes

5.2.2.1 The Code

FxCanvasExample2.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.ArcType;
import javafx.stage.Stage;

public class FxCanvasExample2 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Canvas with a width of 400 px and a height of 200 px.
                Canvas canvas = new Canvas(400, 200);
                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();
                // Set line width
                gc.setLineWidth(2.0);
                // Set fill color
                gc.setFill(Color.RED);
                // Draw a rounded Rectangle
                gc.strokeRoundRect(10, 10, 50, 50, 10, 10);
                // Draw a filled rounded Rectangle
                gc.fillRoundRect(100, 10, 50, 50, 10, 10);
                // Change the fill color
                gc.setFill(Color.BLUE);
                // Draw an Oval
                gc.strokeOval(10, 70, 50, 30);
                // Draw a filled Oval
                gc.fillOval(100, 70, 50, 30);
                // Draw a Line
                gc.strokeLine(200, 50, 300, 50);
                // Draw an Arc
                gc.strokeArc(320, 10, 50, 50, 40, 80, ArcType.ROUND);
                // Draw a filled Arc
                gc.fillArc(320, 70, 50, 50, 00, 120, ArcType.OPEN);


                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
```

```
                // Set the Title of the Stage
                stage.setTitle("Drawing Basic Shapes on a Canvas");
                // Display the Stage
                stage.show();
        }
}
```

The `GraphicsContext` class provides two types of methods to draw the basic shapes. The method `fillXxx()` draws a shape `Xxx` and fills it with the current fill paint. The method `strokeXxx()` draws a shape `Xxx` with the current stroke. Use the following methods for drawing shapes:

- fillArc()
- fillOval()
- fillPolygon()
- fillRect()
- fillRoundRect()
- strokeArc()
- strokeLine()
- strokeOval()
- strokePolygon()
- strokePolyline()
- strokeRect()
- strokeRoundRect()

The following snippet of code draws a rounded rectangle. The stroke color is red and the stroke width is 2px. The upper-left corner of the rectangle is at (10, 10). The rectangle is 50px wide and 50px high. The `arcWidth` and the `arcHeight` are 10 px.

```
// Create the Canvas with a width of 400 px and a height of 200 px.
Canvas canvas = new Canvas(400, 200);
// Get the graphics context of the canvas
GraphicsContext gc = canvas.getGraphicsContext2D();
// Set line width
gc.setLineWidth(2.0);
// Set fill color
gc.setFill(Color.RED);
// Draw a rounded Rectangle
gc.strokeRoundRect(10, 10, 50, 50, 10, 10);
```

5.2.2.2 The GUI

The following image shows a canvas with a few basic shapes (rectangular, oval, etc.):

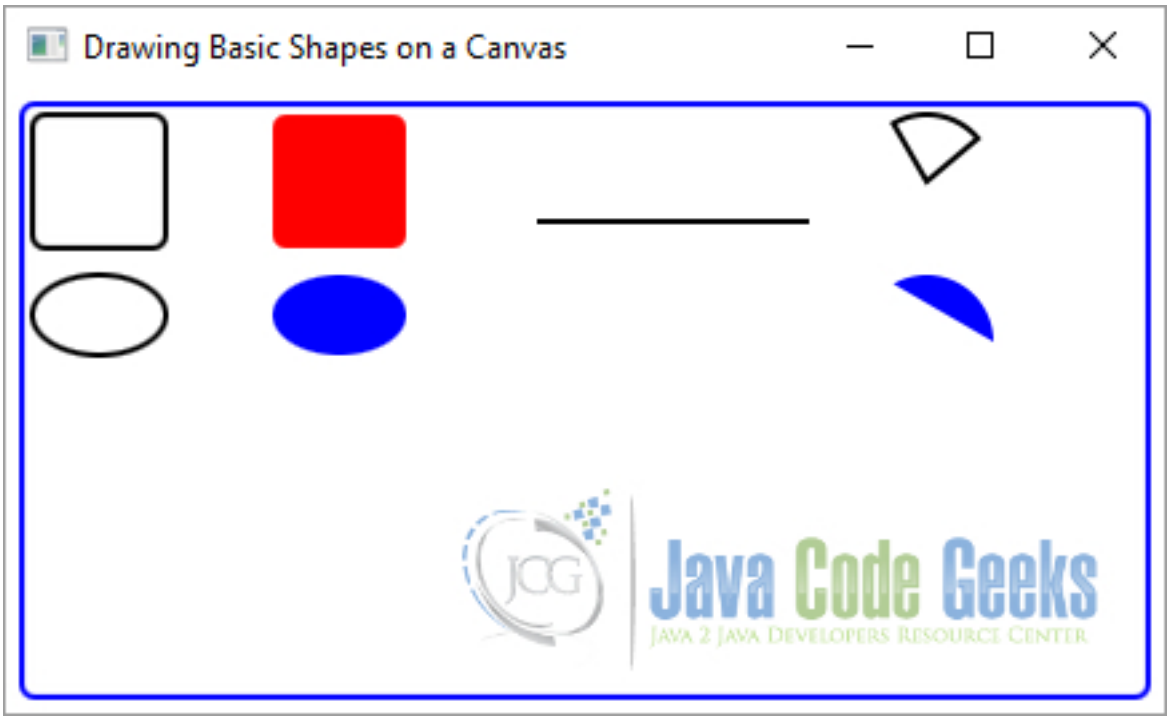

Figure 5.2: Drawing Basic Shapes on a JavaFX Canvas

5.2.3 Drawing Text

5.2.3.1 The Code

FxCanvasExample3.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxCanvasExample3 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Canvas
                Canvas canvas = new Canvas(400, 200);
                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();
                // Set line width
                gc.setLineWidth(1.0);
                // Set fill color
                gc.setFill(Color.BLUE);
```

```
                // Draw a Text
                gc.strokeText("This is a stroked Text", 10, 50);
                gc.strokeText("This is a stroked Text with Max Width 300 px", 10, 100, 300) ↩
                    ;
                // Draw a filled Text
                gc.fillText("This is a filled Text", 10, 150);
                gc.fillText("This is a filled Text with Max Width 400 px", 10, 200, 400);

                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                              "-fx-border-style: solid inside;" +
                              "-fx-border-width: 2;" +
                              "-fx-border-insets: 5;" +
                              "-fx-border-radius: 5;" +
                              "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Drawing a Text on a Canvas");
                // Display the Stage
                stage.show();
        }
}
```

You can draw text using the `fillText()` and `strokeText()` methods of the `GraphicsContext` using the following snippets of code:

- void strokeText(String text, double x, double y)
- void strokeText(String text, double x, double y, double maxWidth)
- void fillText(String text, double x, double y)
- void fillText(String text, double x, double y, double maxWidth)

Both methods are overloaded. One version lets you specify the text and its position. The other version lets you specify the maximum width of the text as well. If the actual text width exceeds the specified maximum width, the text is resized to fit the specified the maximum width.

The following snippet of code draws a blue filled `Text`.

```
// Create the Canvas
Canvas canvas = new Canvas(400, 200);
// Get the graphics context of the canvas
GraphicsContext gc = canvas.getGraphicsContext2D();
// Set line width
gc.setLineWidth(1.0);
// Set fill color
gc.setFill(Color.BLUE);

// Draw a filled Text
gc.fillText("This is a filled Text", 10, 150);
```

5.2.3.2 The GUI

The following GUI shows a few examples of stroked and filled texts:

Figure 5.3: Drawing a Text on a JavaFX Canvas

5.2.4 Drawing Paths

5.2.4.1 The Code

FxCanvasExample4.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class FxCanvasExample4 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
```

```
        public void start(Stage stage)
        {
                // Create the Canvas
                Canvas canvas = new Canvas(400, 200);
                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();
                // Set line width
                gc.setLineWidth(2.0);
                // Set the Color
                gc.setStroke(Color.GREEN);
                // Set fill color
                gc.setFill(Color.LIGHTCYAN);

                // Start the Path
                gc.beginPath();
                // Make different Paths
                gc.moveTo(50, 50);
                gc.quadraticCurveTo(30, 150, 300, 200);
                gc.fill();
                // End the Path
                gc.closePath();
                // Draw the Path
                gc.stroke();

                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Drawing Paths on a Canvas");
                // Display the Stage
                stage.show();
        }
}
```

Use can use path commands and SVG path strings to create a Shape of your choice. A path consists of multiple subpaths. The following methods are used to draw paths:

- beginPath()
- lineTo(double x1, double y1)
- moveTo(double x0, double y0)
- quadraticCurveTo(double xc, double yc, double x1, double y1)
- appendSVGPath(String svgpath)
- arc(double centerX, double centerY, double radiusX, double radiusY, double startAngle, double length)
- arcTo(double x1, double y1, double x2, double y2, double radius)

- bezierCurveTo(double xc1, double yc1, double xc2, double yc2, double x1, double y1)
- closePath()
- stroke()
- fill()

The `beginPath()` and `closePath()` methods start and close a path, respectively. Methods such as `arcTo()` and `lineTo()` are the path commands to draw a specific type of subpath. Do not forget to call the `stroke()` or `fill()` method at the end, which will draw an outline or fill the path.

The following snippet of code draws a quadratic curve. The color of the curve is green and the fill color is lightcyan.

```
// Start the Path
gc.beginPath();
// Make different Paths
gc.moveTo(50, 50);
gc.quadraticCurveTo(30, 150, 300, 200);
gc.fill();
// End the Path
gc.closePath();
```

5.2.4.2 The GUI

The following image shows a simple example how to draw a path on a canvas:

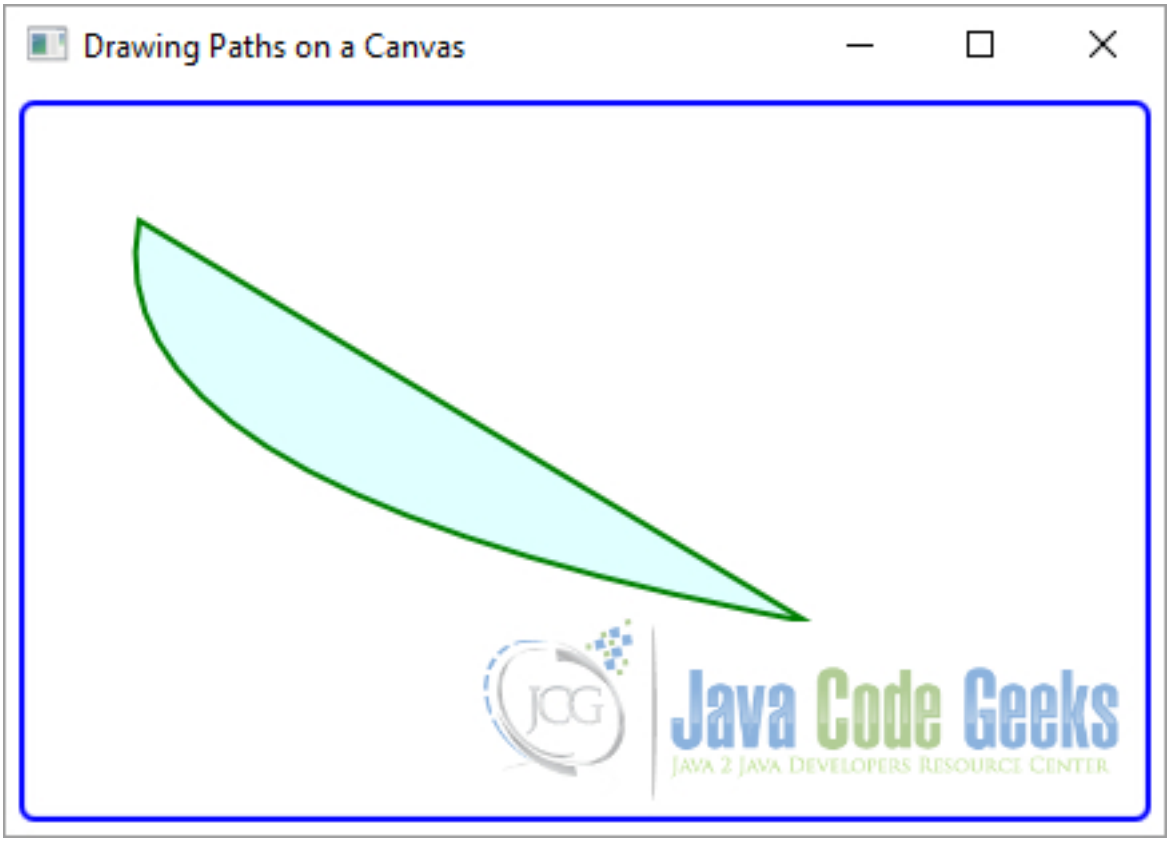

Figure 5.4: Drawing Paths on a JavaFX Canvas

5.2.5 Drawing Images

5.2.5.1 The Code

FxCanvasExample5.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class FxCanvasExample5 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Canvas
                Canvas canvas = new Canvas(400, 200);
                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();
                // Load the Image
                String imagePath = "file:\\Path-To-Your-Image\\java-logo.gif";
                Image image = new Image(imagePath);
                // Draw the Image
                gc.drawImage(image, 10, 10, 200, 200);
                gc.drawImage(image, 220, 50, 100, 70);

                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Drawing an Image on a Canvas");
                // Display the Stage
                stage.show();
        }
}
```

You can draw an Image on the `Canvas` using the `drawImage()` method. The method has three versions:

- void drawImage(Image img, double x, double y)

- void drawImage(Image img, double x, double y, double w, double h)
- void drawImage(Image img, double sx, double sy, double sw, double sh, double dx, double dy, double dw, double dh)

You can draw the whole or part of the `Image`. The drawn image can be stretched or shortened on the canvas.

The following snippet of code draws the first whole image with a size of 200 px x 200 px on the canvas at (10, 10). The second image will be drawn at (220, 50). The width is 100 px and the height is 70 px.

```
// Create the Canvas
Canvas canvas = new Canvas(400, 200);
// Get the graphics context of the canvas
GraphicsContext gc = canvas.getGraphicsContext2D();
// Load the Image
String imagePath = "file:\\Path-To-Your-Image\\java-logo.gif";
Image image = new Image(imagePath);
// Draw the Image
gc.drawImage(image, 10, 10, 200, 200);
gc.drawImage(image, 220, 50, 100, 70);
```

5.2.5.2 The GUI

The following GUI shows a canvas, which contains two images:

Figure 5.5: Drawing an Image on a JavaFX Canvas

5.2.6 Writing Pixels

5.2.6.1 The Code

FxCanvasExample6.java

```
import java.nio.ByteBuffer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.PixelFormat;
import javafx.scene.image.PixelWriter;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class FxCanvasExample6 extends Application
{
        private static final int RECT_WIDTH = 25;
        private static final int RECT_HEIGHT = 25;

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

        @Override
        public void start(Stage stage)
        {
                Canvas canvas = new Canvas(400, 200);
                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();
                // Set line width
                gc.setLineWidth(2.0);

                // Write custom pixels to create a pattern
                writePixels(gc);

                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Writing Pixels on a Canvas");
                // Display the Stage
                stage.show();
        }

        private void writePixels(GraphicsContext gc)
        {
```

```
                // Define properties of the Image
                int spacing = 5;
                int imageWidth = 300;
                int imageHeight = 100;
                int rows = imageHeight/(RECT_HEIGHT + spacing);
                int columns = imageWidth/(RECT_WIDTH + spacing);

                // Get the Pixels
                byte[] pixels = this.getPixelsData();

                // Create the PixelWriter
                PixelWriter pixelWriter = gc.getPixelWriter();

                // Define the PixelFormat
                PixelFormat<ByteBuffer> pixelFormat = PixelFormat.getByteRgbInstance();

                // Write the pixels to the canvas
                for (int y = 0; y < rows; y++)
                {
                        for (int x = 0; x < columns; x++)
                        {
                                int xPos = 50 + x * (RECT_WIDTH + spacing);
                                int yPos = 50 + y * (RECT_HEIGHT + spacing);
                                pixelWriter.setPixels(xPos, yPos, RECT_WIDTH, RECT_HEIGHT,
                                                pixelFormat, pixels, 0, RECT_WIDTH * 3);
                        }
                }
        }

        private byte[] getPixelsData()
        {
                // Create the Array
                byte[] pixels = new byte[RECT_WIDTH * RECT_HEIGHT * 3];
                // Set the ration
                double ratio = 1.0 * RECT_HEIGHT/RECT_WIDTH;
                // Generate pixel data
                for (int y = 0; y < RECT_HEIGHT; y++)
                {
                        for (int x = 0; x < RECT_WIDTH; x++)
                        {
                                int i = y * RECT_WIDTH * 3 + x * 3;
                                if (x <= y/ratio)
                                {
                                        pixels[i] = -1;
                                        pixels[i+1] = 1;
                                        pixels[i+2] = 0;
                                }
                                else
                                {
                                        pixels[i] = 1;
                                        pixels[i+1] = 1;
                                        pixels[i+2] = 0;
                                }
                        }
                }

                // Return the Pixels
                return pixels;
        }
}
```

You can also directly modify pixels on the `Canvas`. The `getPixelWriter()` method of the `GraphicsContext` object

returns a PixelWriter that can be used to write pixels to the associated canvas:

```
Canvas canvas = new Canvas(200, 100);
GraphicsContext gc = canvas.getGraphicsContext2D();
PixelWriter pw = gc.getPixelWriter();
```

Once you get a `PixelWriter`, you can write pixels to the canvas.

The following method creates an array of pixels with the corresponding with, height and RGB-Code:

```
private byte[] getPixelsData()
{
        // Create the Array
        byte[] pixels = new byte[RECT_WIDTH * RECT_HEIGHT * 3];
        // Set the ration
        double ratio = 1.0 * RECT_HEIGHT/RECT_WIDTH;
        // Generate pixel data
        for (int y = 0; y < RECT_HEIGHT; y++)
        {
                for (int x = 0; x < RECT_WIDTH; x++)
                {
                        int i = y * RECT_WIDTH * 3 + x * 3;
                        if (x <= y/ratio)
                        {
                                pixels[i] = -1;
                                pixels[i+1] = 1;
                                pixels[i+2] = 0;
                        }
                        else
                        {
                                pixels[i] = 1;
                                pixels[i+1] = 1;
                                pixels[i+2] = 0;
                        }
                }
        }

        // Return the Pixels
        return pixels;
}
```

The following method draws the pixels on the canvas.

```
private void writePixels(GraphicsContext gc)
{
        // Define properties of the Image
        int spacing = 5;
        int imageWidth = 300;
        int imageHeight = 100;
        int rows = imageHeight/(RECT_HEIGHT + spacing);
        int columns = imageWidth/(RECT_WIDTH + spacing);

        // Get the Pixels
        byte[] pixels = this.getPixelsData();

        // Create the PixelWriter
        PixelWriter pixelWriter = gc.getPixelWriter();

        // Define the PixelFormat
        PixelFormat<ByteBuffer> pixelFormat = PixelFormat.getByteRgbInstance();

        // Write the pixels to the canvas
        for (int y = 0; y < rows; y++)
```

```
        {
                for (int x = 0; x < columns; x++)
                {
                        int xPos = 50 + x * (RECT_WIDTH + spacing);
                        int yPos = 50 + y * (RECT_HEIGHT + spacing);
                        pixelWriter.setPixels(xPos, yPos, RECT_WIDTH, RECT_HEIGHT,
                                        pixelFormat, pixels, 0, RECT_WIDTH * 3);
                }
        }
}
```

5.2.6.2 The GUI

The following GUI shows the result of the written pixels on the canvas:

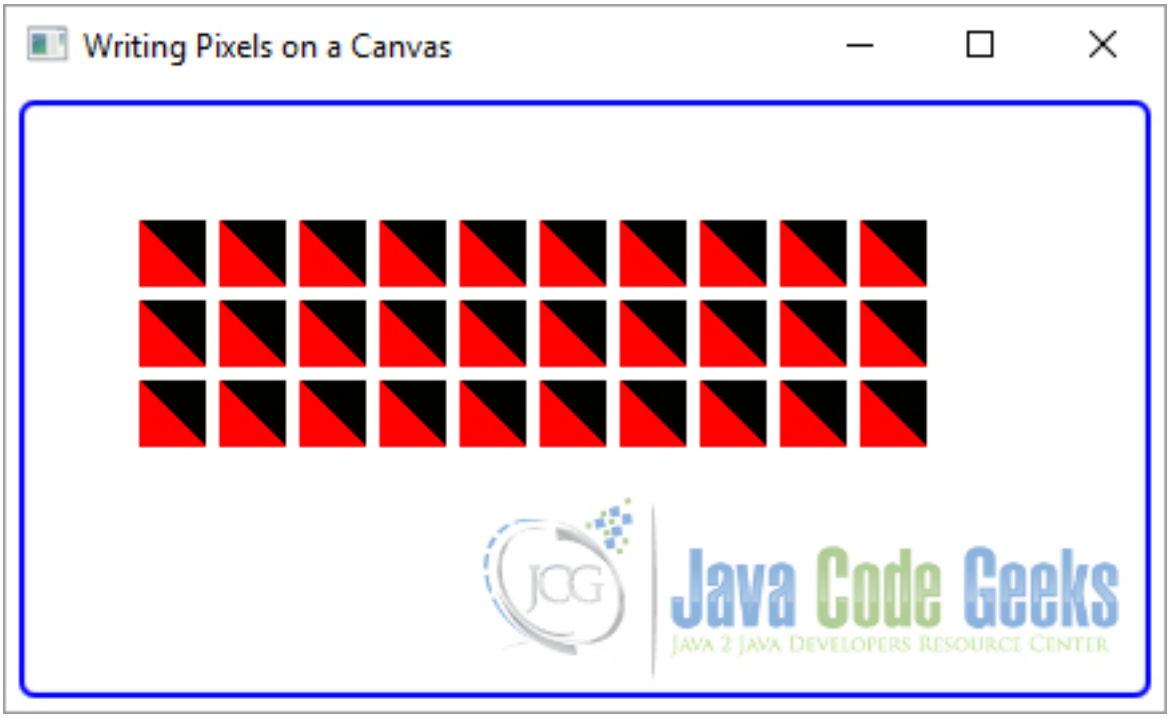

Figure 5.6: Drawing Pixels on a JavaFX Canvas

5.3 Clearing the Canvas Area

5.3.1 The Code

FxCanvasExample7.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
```

```
public class FxCanvasExample7 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage)
        {
                // Create the Canvas
                Canvas canvas = new Canvas(400, 200);
                // Get the graphics context of the canvas
                GraphicsContext gc = canvas.getGraphicsContext2D();
                // Set line width
                gc.setLineWidth(2.0);
                // Set fill color
                gc.setFill(Color.GREEN);
                // Draw a rounded Rectangle
                gc.fillRoundRect(50, 50, 300, 100, 10, 10);
                // Clear the rectangular area from the canvas
                gc.clearRect(80, 80, 130, 50);

                // Create the Pane
                Pane root = new Pane();
                // Set the Style-properties of the Pane
                root.setStyle("-fx-padding: 10;" +
                                "-fx-border-style: solid inside;" +
                                "-fx-border-width: 2;" +
                                "-fx-border-insets: 5;" +
                                "-fx-border-radius: 5;" +
                                "-fx-border-color: blue;");

                // Add the Canvas to the Pane
                root.getChildren().add(canvas);
                // Create the Scene
                Scene scene = new Scene(root);
                // Add the Scene to the Stage
                stage.setScene(scene);
                // Set the Title of the Stage
                stage.setTitle("Clearing the Area of a Canvas");
                // Display the Stage
                stage.show();
        }
}
```

The `Canvas` is a transparent area. Pixels will have colors and opacity depending on what is drawn at those pixels. Sometimes you may want to clear the whole or part of the canvas so the pixels are transparent again.

The `clearRect()` method of the `GraphicsContext` lets you clears a specified area on the `Canvas`:

The following code snippet clears a rectangular area inside of the drawn rectangular:

```
// Clear the rectangular area from the canvas
gc.clearRect(80, 80, 130, 50);
```

5.3.2 The GUI

The following GUI shows a simple example how you can delete a given area of a canvas:

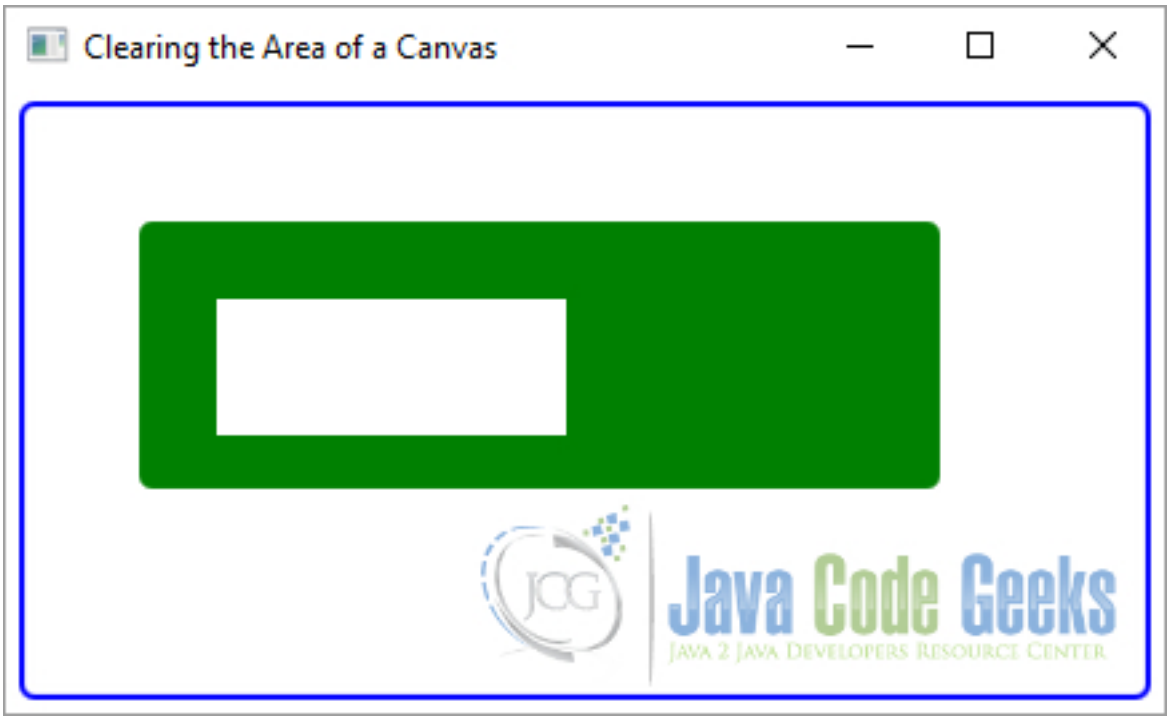

Figure 5.7: Clearing an Area on a JavaFX Canvas

5.4 Download Java Source Code

This was an example of `javafx.scene.canvas`

Download

You can download the full source code of this example here: **JavaFxCanvasExample.zip**

Chapter 6

JavaFX FXML Tutorial

This is a JavaFX FXML Tutorial. In this tutorial we will discuss how to use FXML for creating the GUI of an application. The first three chapters are also part of the article JavaFX FXML Controller Example. Given the fact, that this article represents a tutorial, it contains also the Controller Example.

FXML is an XML-based language designed to build the user interface for JavaFX applications. You can use FXML to build an entire Scene or part of a `Scene`. FXML allows application developers to separate the logic for building the UI from the business logic. If the UI part of the application changes, you do not need to recompile the JavaFX code. Instead you can change the FXML using a text editor and rerun the application. You still use JavaFX to write business logic using the Java language. An FXML document is an XML document.

A JavaFX scene graph is a hierarchical structure of Java objects. XML format is well suited for storing information representing some kind of hierarchy. Therefore, using FXML to store the scene-graph is very intuitive. It is common to use FXML to build a scene graph in a JavaFX application.

The following examples uses Java SE 8.

6.1 Introduction to FXML

6.1.1 The FXML Code

FxFXMLExample1.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2">
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
  <children>
    <Label fx:id="inputLbl" alignment="CENTER_LEFT" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Please insert Your Input here:"  ↩
        textAlignment="LEFT" />
```

```
    <TextField fx:id="inputText" prefWidth="100.0" />
    <Button fx:id="okBtn" alignment="CENTER_RIGHT" contentDisplay="CENTER" mnemonicParsing= ↩
        "false" text="OK" textAlignment="CENTER" />
    <Label fx:id="outputLbl" alignment="CENTER_LEFT" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Your Input:" textAlignment="LEFT" />
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="200.0" wrapText="true" />
  </children>
</VBox>
```

6.1.2 Adding UI Elements

The root element of the FXML document is the top-level object in the object-graph. The top-level object of the above example is a VBox. Therefore, the root element of your FXML would be:

```
<VBox>
</VBox>
```

How do you know that to represent a `VBox` in the object-graph, you need to use a tag in FXML? It is both difficult and easy. It is difficult because there is no documentation for FXML tags. It is easy because FXML has a few rules explaining what constitutes a tag name. For example, if a tag name is the simple or fullqualified name of a class, the tag will create an object of that class. The above element will create an object of the `VBox` class. The above FXML can be rewritten using the fully qualified class name:

```
<javafx.scene.layout.VBox>
</javafx.scene.layout.VBox>
```

In JavaFX, layout panes have children. In FXML, layout panes have children as their child elements. You can add a Label and a Button and other elements to the `VBox` as follows:

```
<children>
        <Label/>
        <TextField/>
        <Button/>
        <Label/>
        <TextArea/>
</children>
```

This defines the basic structure of the object-graph for our application. It will create a `VBox` with two labels, a TextField, a TextArea and a `Button`.

6.1.3 Importing Java Types in FXML

To use the simple names of Java classes in FXML, you must import the classes as you do in Java programs. There is one exception. In Java programs, you do not need to import classes from the `java.lang package`. However, in FXML, you need to import classes from all packages, including the `java.lang package`. An import processing instruction is used to import a class or all classes from a package. The following processing instructions import the `VBox`, `Label`, and `Button` classes:

```
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Button?>
```

The following import processing instructions import all classes from the `javafx.scene.control` and `java.lang` packages:

```
<?import javafx.scene.control.*?>
<?import java.lang.*?>
```

6.1.4 Setting Properties in FXML

You can set properties for Java objects in FXML. A property for an object can be set in FXML if the property declaration follows the JavaBean conventions. The attribute name or the property element name is the same as the name of the property being set. The following FXML creates a `TextField` and sets its `prefWidth` property using an attribute:

```
<TextField fx:id="inputText" prefWidth="100.0" />
```

6.1.5 Specifying FXML Namespace

FXML does not have an XML schema. It uses a namespace that needs to be specified using the namespace prefix "fx". For the most part, the FXML parser will figure out the tag names such as tag names that are classes, properties of the classes, and so on. FXML uses special elements and attribute names, which must be qualified with the "fx" namespace prefix. Optionally, you can append the version of the FXML in the namespace URI. The FXML parser will verify that it can parse the specified.

The following FXML declares the "fx" namespace prefix.

```
<VBox xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2">...</VBox>
```

6.1.6 Assigning an Identifier to an Object

An object created in FXML can be referred to somewhere else in the same document. It is common to get the reference of UI objects created in FXML inside the JavaFX code. You can achieve this by first identifying the objects in FXML with an `fx:id` attribute. The value of the `fx:id` attribute is the identifier for the object. If the object type has an id property, the value will be also set for the property. Note that each Node in JavaFX has an id property that can be used to refer to them in CSS. The following is an example of specifying the `fx:id` attribute for a `Label`.

```
<Label fx:id="inputLbl"/>
```

6.1.7 The Corresponding Java Class

FxFXMLExample1.java

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample1 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample1.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);
```

```
                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A simple FXML Example");
                // Display the Stage
                stage.show();
        }
}
```

An FXML document defines the view part (the GUI) of a JavaFX application. You need to load the FXML document to get the object-graph it represents. Loading an FXML is performed by an instance of the FXMLLoader class. The `FXMLLoader` class provides several constructors that let you specify the location, charset, resource bundle, and other elements to be used for loading the document. `FXMLLoader` supports loading a FXML document using an InputStream. The following snippet of code loads the same FXML document using an `InputStream`.

```
// Create the FXMLLoader
FXMLLoader loader = new FXMLLoader();
// Path to the FXML File
String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample1.fxml";
FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);
```

Internally, the `FXMLLoader` reads the document using streams, which may throw an IOException. All versions of the `load()` method in `FXMLLoader` class throw `IOException`. In your application, you will need to handle the exception. The `FXMLLoader` class contains several versions of the `load()` method. Some of them are instance methods and some static methods. You need to create an `FXMLLoader` instance and use the instance `load()` method, if you want to retrieve more information from the loader, such as the controller reference, resource bundle, the location, charset, and root object.

```
// Create the Pane and all Details
VBox root = (VBox) loader.load(fxmlStream);
```

What do you do next after loading an FXML document? The loader returns a `VBox`, which is set as the root for the `Scene`. The rest of the code is the same as you have been using except for one difference in the declaration of the `start()` method. The method declares that it may throw an `IOException`, which you had to add because you have called the `load()` method of the `FXMLLoader` inside the method.

```
// Create the Scene
Scene scene = new Scene(root);
// Set the Scene to the Stage
stage.setScene(scene);
// Set the Title to the Stage
stage.setTitle("A simple FXML Example");
// Display the Stage
stage.show();
```

6.1.8 The GUI

The following mage shows the application after starting. But at this time, a click on the OK-Button has no effect. The reason for this behaviour is the fact, that we have not defined an EventHandler at this time.

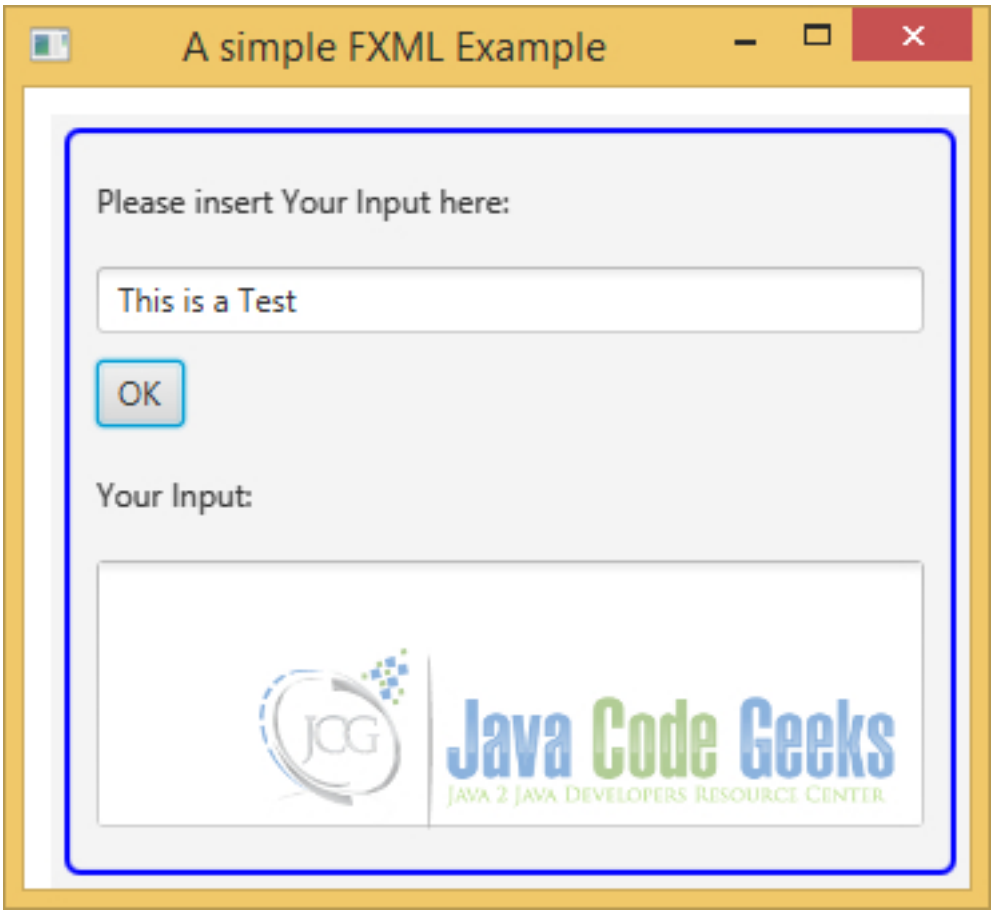

Figure 6.1: A simple JavaFX FXML Example

6.2 Using Script Event Handlers

6.2.1 The FXML Code

FxFXMLExample2.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2">
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
  <children>
```

```
    <Label fx:id="inputLbl" alignment="CENTER_LEFT" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Please insert Your Input here:"  ↩
        textAlignment="LEFT" />
    <TextField fx:id="inputText" prefWidth="100.0" />
    <Button fx:id="okBtn" alignment="CENTER_RIGHT" contentDisplay="CENTER" mnemonicParsing= ↩
        "false" onAction="printOutput();" text="OK" textAlignment="CENTER" />
    <Label fx:id="outputLbl" alignment="CENTER_LEFT" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Your Input:" textAlignment="LEFT" />
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="200.0" wrapText="true" />
    <fx:script>
                function printOutput()
                {
                        outputText.setText(inputText.getText());
                }
        </fx:script>
  </children>
</VBox>
```

You can set event handlers for nodes in FXML. Setting an event handler is similar to setting any other properties. In FXML, you can specify two types of event handlers:

- Script Event Handlers
- Controller Event Handlers

In this chapter we will discuss Script Event Handlers. The Controller Event Handlers will be discussed in the following chapter.

The script event handler is used when the event handler is defined in a scripting language. The value of the attribute is the script itself, such as a function call or one or more statements. The following snippet of FXML sets the ActionEvent handler for a `Button` that calls the `printOutput()` function defined using JavaScript.

```
<?language JavaScript?>

<fx:script>
        function printOutput()
        {
                outputText.setText(inputText.getText());
        }
</fx:script>
```

If you want to execute the function `printOutput()` when the `Button` is clicked, you can set the event handler as:

```
<Button fx:id="okBtn" onAction="printOutput();" text="OK" textAlignment="CENTER" />
```

6.2.2 The Corresponding Java Class

FxFXMLExample2.java

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample2 extends Application
{
        public static void main(String[] args)
```

```
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample2.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);

                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example with a Script Event Handler");
                // Display the Stage
                stage.show();

        }
}
```

6.2.3 The GUI

The following image shows the result of our program after inserting a Text in the `TextField` and pressing the `Button` "OK":

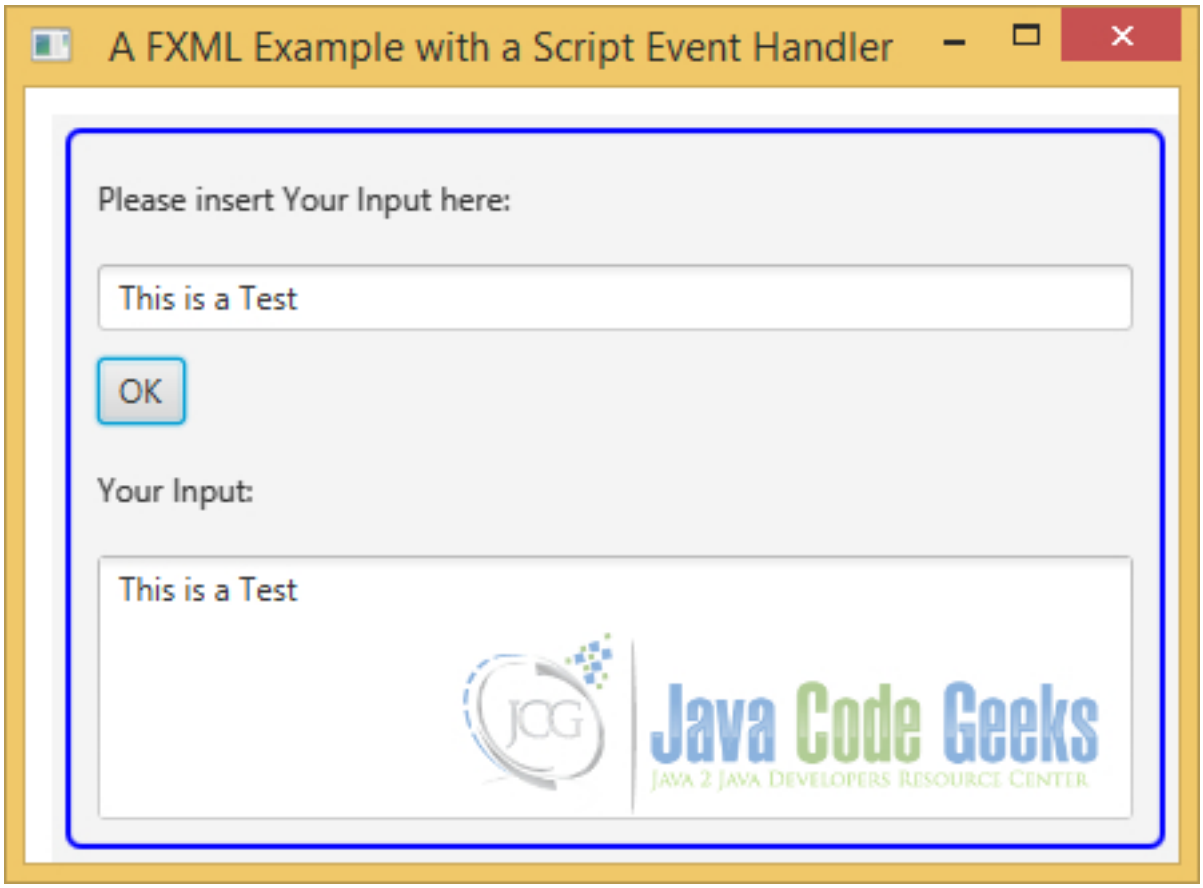

Figure 6.2: A JavaFX FXML Example with a JavaScript Event Handler

6.3 Using Controller Event Handlers

6.3.1 The FXML Code

FxFXMLExample3.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    fx:controller="FxFXMLController">
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
  <children>
    <Label fx:id="inputLbl" alignment="CENTER_LEFT" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Please insert Your Input here:"  ↩
        textAlignment="LEFT" />
```

```
    <TextField fx:id="inputText" prefWidth="100.0" />
    <Button fx:id="okBtn" alignment="CENTER_RIGHT" contentDisplay="CENTER" mnemonicParsing= ↩
        "false" onAction="#printOutput" text="OK" textAlignment="CENTER" />
    <Label fx:id="outputLbl" alignment="CENTER_LEFT" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Your Input:" textAlignment="LEFT" />
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="200.0" wrapText="true" />
  </children>
</VBox>
```

A controller is simply a class name whose object is created by FXML and used to initialize the UI elements. FXML lets you specify a controller on the root element using the `fx:controller` attribute. Note that only one controller is allowed per FXML document, and if specified, it must be specified on the root element. The following FXML specifies a controller for the `VBox` element.

```
<VBox fx:id="vbox" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx ↩
    /2.2" fx:controller="FxFXMLController">
```

A controller needs to conform to some rules and it can be used for different reasons:

- The controller is instantiated by the FXML loader.
- The controller must have a public no-args constructor. If it does not exist, the FXML loader will not be able to instantiate it, which will throw an exception at the load time.
- The controller can have accessible methods, which can be specified as event handlers in FXML.
- The FXML loader will automatically look for accessible instance variables of the controller. If the name of an accessible instance variable matches the fx:id attribute of an element, the object reference from FXML is automatically copied into the controller instance variable. This feature makes the references of UI elements in FXML available to the controller. The controller can use them later, such as binding them to model.
- The controller can have an accessible initialize() method, which should take no arguments and have a return type of void. The FXML loader will call the initialize() method after the loading of the FXML document is complete.

6.3.2 The Controller Class

FxFXMLController.java

```
import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;

public class FxFXMLController
{
        @FXML
        // The reference of inputText will be injected by the FXML loader
        private TextField inputText;

        // The reference of outputText will be injected by the FXML loader
        @FXML
        private TextArea outputText;

        // location and resources will be automatically injected by the FXML loader
        @FXML
        private URL location;

        @FXML
```

```
        private ResourceBundle resources;

        // Add a public no-args constructor
        public FxFXMLController()
        {
        }

        @FXML
        private void initialize()
        {
        }

        @FXML
        private void printOutput()
        {
                outputText.setText(inputText.getText());
        }
}
```

The controller class uses a `@FXML` annotation on some members. The `@FXML` annotation can be used on fields and methods. It cannot be used on classes and constructors. By using a `@FXML` annotation on a member, you are declaring that the FXML loader can access the member even if it is private. A public member used by the FXML loader does not need to be annotated with `@FXML`. However, annotating a public member with `@FXML` is not an error. It is better to annotate all members, public and private, used by the FXML loader with the `@FXML` annotation. This tells the reader of your code how the members are being used.

The following FXML sets the `printOutput()` method of the controller class as the event handler for the `Button`:

```
<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    fx:controller="FXFXML.FxFXMLController">

<Button fx:id="okBtn" alignment="CENTER_RIGHT" contentDisplay="CENTER" mnemonicParsing=" ↩
    false" onAction="#printOutput" text="OK" textAlignment="CENTER" />
```

There are two special instance variables that can be declared in the controller and they are automatically injected by the FXML loader:

- @FXML private URL location;
- @FXML private ResourceBundle resources;

The location is the location of the FXML document. The resources is the reference of the ResourceBundle. When the event handler attribute value starts with a hash symbol (#), it indicates to the FXML loader that `printOutput()` is the method in the controller, not in a script.

The event handler method in the controller should conform to some rules:

- The method may take no arguments or a single argument. If it takes an argument, the argument type must be a type assignment compatible with the event it is supposed to handle.
- Conventionally, the method return type should be void, because there is no taker of the returned value.
- The method must be accessible to the FXML loader: make it public or annotate it with @FXML.
- When the FXML loader is done loading the FXML document, it calls the initialize() method of the controller. The method should not take any argument. It should be accessible to the FXML loader. In the controller, you used the @FXML annotation to make it accessible to the FXML loader.

6.3.3 The Corresponding Java Class

FxFXMLExample3.java

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample3 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample3.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);

                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example with a Controller");
                // Display the Stage
                stage.show();

        }
}
```

6.3.4 The GUI

The following image shows the result of our program:

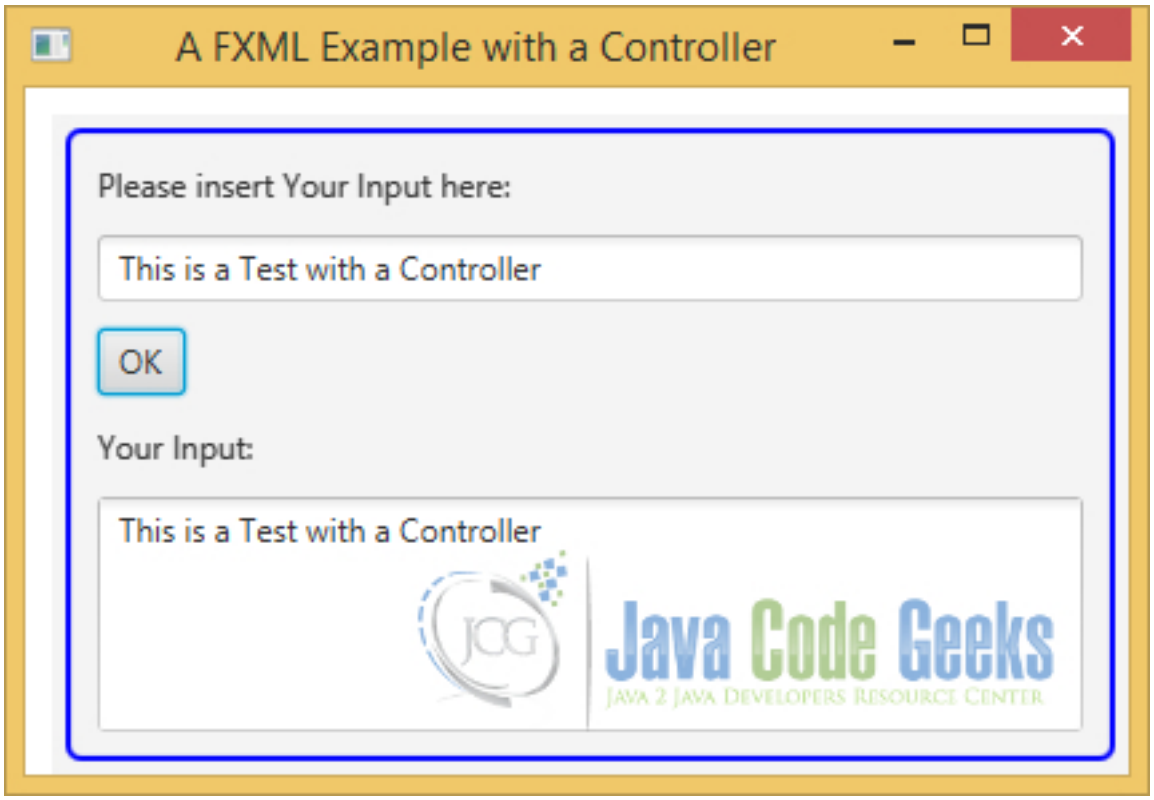

Figure 6.3: A JavaFX FXML Controller Example

6.4 Including FXML Files

6.4.1 The FXML Code

In this example the FXML Code is divided into three FXML Files.

A separate FXML File for creating a `Label`:

FxFXMLLabel.fxml

```
<?import javafx.scene.control.*?>
<?import java.lang.Integer?>

<Label alignment="CENTER_LEFT" cache="true" cacheHint="SCALE" prefHeight="30.0" prefWidth=" ↩
    200.0" textAlignment="$position" />
```

A separate FXML File for creating a `Button`:

FxFXMLButton.fxml

```
<?import javafx.scene.control.*?>

<Button alignment="CENTER_RIGHT" contentDisplay="CENTER" mnemonicParsing="false"  ↩
    textAlignment="CENTER" />
```

And at least, the main FXML File:

FxFXMLExample4.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    fx:controller="FXFXML.FxFXMLController">
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
  <children>
    <fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="inputLbl" text="Please insert Your ↩
         Input here:"/>
    <TextField fx:id="inputText" prefWidth="100.0" />
    <fx:include source="/FXFXML/FxFXMLButton.fxml" fx:id="okBtn" text="OK" onAction="# ↩
        printOutput" />
    <fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="outputLbl" text="Your Input:"/>
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="100.0" />
  </children>
</VBox>
```

An FXML document can include another FXML document using the `<fx:include>` element. The objectgraph generated by the nested document is included at the position where the nested document occurs in the containing document. The `<fx:include>` element takes a source attribute whose value is the path of the nested document.

```
<fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="inputLbl" text="Please insert Your  ↩
    Input here:"/>
```

If the nested document path starts with a leading forward slash, the path is resolved relative to the CLASSPATH. Otherwise, it is resolved related to the containing document path. The `<fx:include>` element can have the `fx:id` attribute and all attributes that are available for the included object. The attributes specified in the containing document override the corresponding attributes in the included document.

For example, if you include an FXML document, which creates a `Label`, you can specify the text property in the included document as well as the containing document. When the containing document is loaded, the text property from the containing document will be used.

```
<fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="inputLbl" text="Please insert Your  ↩
    Input here:"/>
```

For example, if you include an FXML document, which creates a `Button`, you can specify the text property in the included document as well as the containing document. When the containing document is loaded, the text property from the containing document will be used.

```
<fx:include source="/FXFXML/FxFXMLButton.fxml" fx:id="okBtn" text="OK" onAction="# ↩
    printOutput" />
```

An FXML document may optionally specify a controller using the fx:controller attribute for the root element. The rule is that you can have maximum of one controller per FXML document. When you nest documents, each document can have its own controller.

```
<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    fx:controller="FXFXML.FxFXMLController">
```

6.4.2 The Corresponding Java Class

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample4 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample4.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);

                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example which includes FXML Files");
                // Display the Stage
                stage.show();
        }
}
```

6.4.3 The GUI

The following image shows the GUI of the above example. It loads the FxFXMLExample4.fxml and adds the loaded `VBox` to the `Scene`. It displays a window with the OK `Button` from the FxFXMLButton.fxml file and the labels are loaded from the FxFXMLLabel.fxml file.

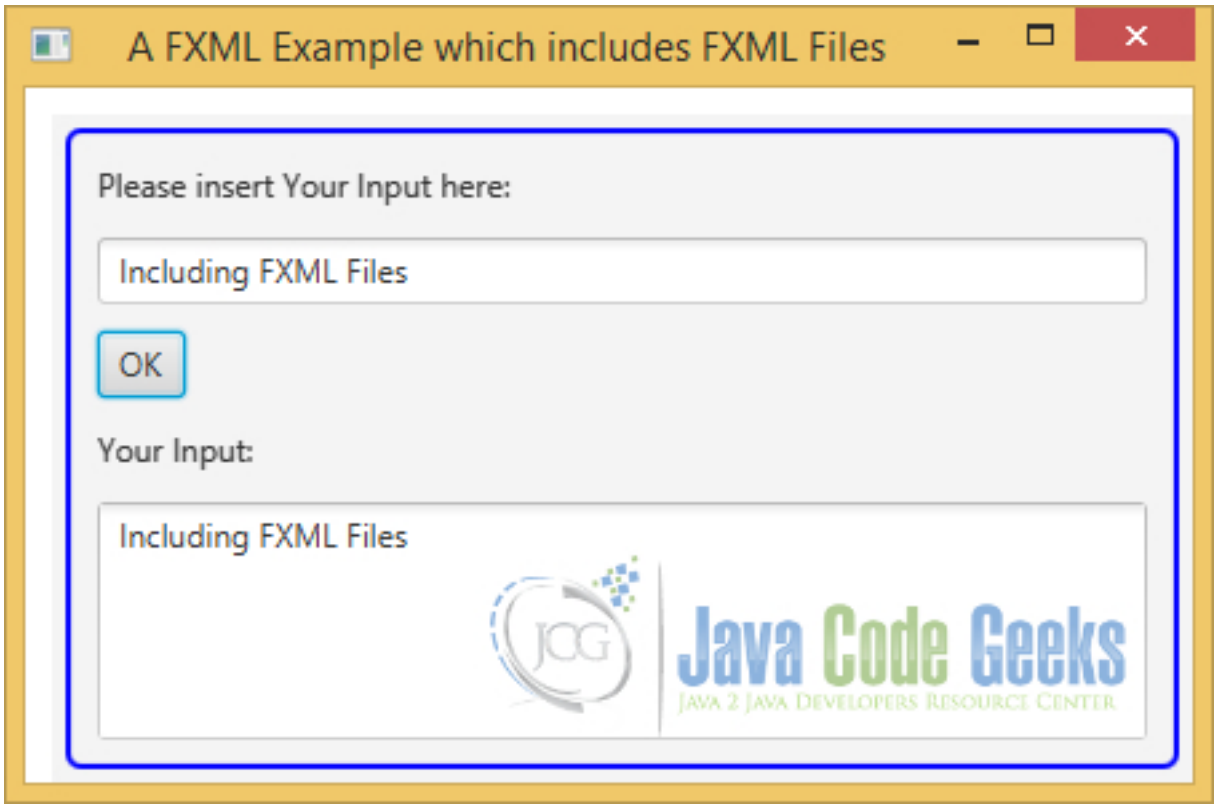

Figure 6.4: A JavaFX FXML Example with external FXML Files

6.5 Reusable Objects and Referencing Another Element

6.5.1 The FXML Code

FxFXMLExample5.fxml

```
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.image.Image?>
<?import javafx.scene.image.ImageView?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0" ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2" ↩
    >
        <fx:define>
                <Image url="/FXFXML/JavaFx.jpg" fx:id="javaImg"/>
        </fx:define>
        <children>
        <fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="javaLbl" text="The JavaFX ↩
            Image:"/>
                <ImageView fx:id="view">
                        <image>
                                <fx:reference source="javaImg"/>
                        </image>
                </ImageView>
        </children>
</VBox>
```

6.5.2 Creating Reusable Objects in FXML

Sometimes, you need to create objects that are not directly part of the object-graph. However, they may be used somewhere else in the FXML document. You can create an object in FXML without making it part of the object-group using the `<fx:define>` block. You can refer to the objects created in the block by their `fx:id` in the attribute value of other elements. The attribute value must be prefixed with a dollar symbol ($).

```
<fx:define>
        <Image url="/FXFXML/JavaFx.jpg" fx:id="javaImg"/>
</fx:define>
```

6.5.3 Referencing Another Element

You can reference another element in the document using the `<fx:reference>` element. The `fx:id` attribute specifies the `fx:id` of the referred element. `<fx:reference source="fx:id of the source element"/>`

The following FXML content uses an `<fx:reference>` element to refer to an Image.

```
<ImageView fx:id="view">
        <image>
                <fx:reference source="javaImg"/>
        </image>
</ImageView>
```

6.5.4 The Corresponding Java Class

FxFXMLExample5.java

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample5 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample5.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);

                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
```

```
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example with Reuseable Objects");
                // Display the Stage
                stage.show();
        }
}
```

6.5.5 The GUI

In the following GUI, you can see the effect of using a referenced element in FXML:

Figure 6.5: A JavaFX FXML Example with Reusable Objects

6.6 Using Constants

6.6.1 The FXML Code

FxFXMLExample6.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.Pos?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    fx:controller="FXFXML.FxFXMLController">
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
        <alignment><Pos fx:constant="CENTER" fx:id="alignCenter"/></alignment>
  <children>
    <Label fx:id="inputLbl" alignment="$alignCenter" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Please insert Your Input here:" />
    <TextField fx:id="inputText" prefWidth="100.0" />
    <Button fx:id="okBtn" alignment="$alignCenter" mnemonicParsing="false" onAction="# ↩
        printOutput" text="OK" />
    <Label fx:id="outputLbl" alignment="$alignCenter" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="Your Input:" />
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="200.0" wrapText="true" />
  </children>
</VBox>
```

Classes, interfaces, and enums may define constants, which are static, final variables. You can refer to those constants using the `fx:constant` attribute. The attribute value is the name of the constant. The name of the element is the name of the type that contains the constant.

Note that all enum constants belong to this category and they can be accessed using the `fx:constant` attribute. The following element accesses the `Pos.CENTER` enum constant.

```
<alignment><Pos fx:constant="CENTER" fx:id="alignCenter"/></alignment>
```

6.6.2 The Corresponding Java Class

FxFXMLExample6.java

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample6 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }
```

```
        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample6.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);

                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example with Constants");
                // Display the Stage
                stage.show();
        }
}
```

6.6.3 The GUI

The following GUI represents the effect of the usage of the constant `alignCenter`:

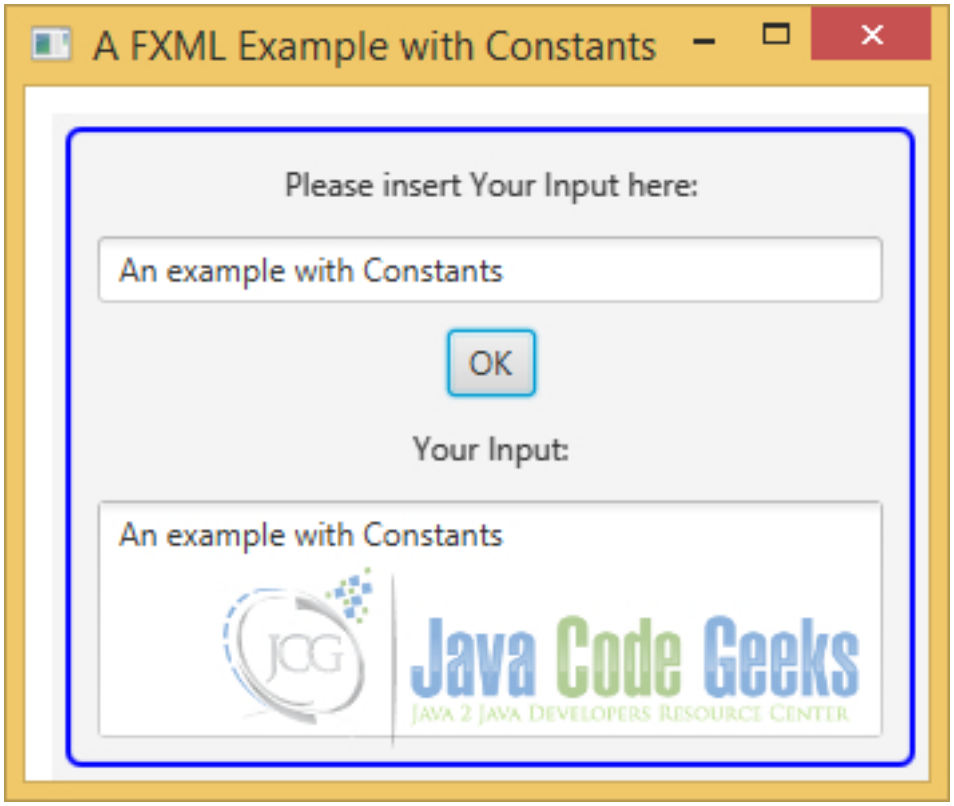

Figure 6.6: A JavaFX FXML Example with Constants

6.7 Binding Properties

6.7.1 The FXML Code

FxFXMLExample7.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    >
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
  <children>
    <fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="inputLbl" text="Please insert Your ↩
         Input here:"/>
    <TextField fx:id="inputText" prefWidth="100.0" />
    <fx:include source="/FXFXML/FxFXMLLabel.fxml" fx:id="outputLbl" text="Your Input:"/>
    <TextArea fx:id="outputText" text="${inputText.text}" prefHeight="100.0" prefWidth=" ↩
        100.0" />
  </children>
</VBox>
```

FXML supports simple property bindings. You need to use an attribute for the property to bind it to the property of another element or a document variable. The attribute value starts with a $ symbol, which is followed with a pair of curly braces.

The following FXML content creates a VBox with two TextFields. The text property of the outputText field is bound to the text property of the inputText field.

```
<TextArea fx:id="outputText" text="${inputText.text}" prefHeight="100.0" prefWidth="100.0"  ↩
    />
```

6.7.2 The Corresponding Java Class

FxFXMLExample7.java

```
import java.io.FileInputStream;
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample7 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }
```

```
        @Override
        public void start(Stage stage) throws IOException
        {
                // Create the FXMLLoader
                FXMLLoader loader = new FXMLLoader();
                // Path to the FXML File
                String fxmlDocPath = "Path-To-Your-FXML-Files/FxFXMLExample7.fxml";
                FileInputStream fxmlStream = new FileInputStream(fxmlDocPath);

                // Create the Pane and all Details
                VBox root = (VBox) loader.load(fxmlStream);

                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example with Binding Properties");
                // Display the Stage
                stage.show();
        }
}
```

6.7.3 The GUI

In the following GUI the text from the `TextField` will be direct copied to the `TextArea`:

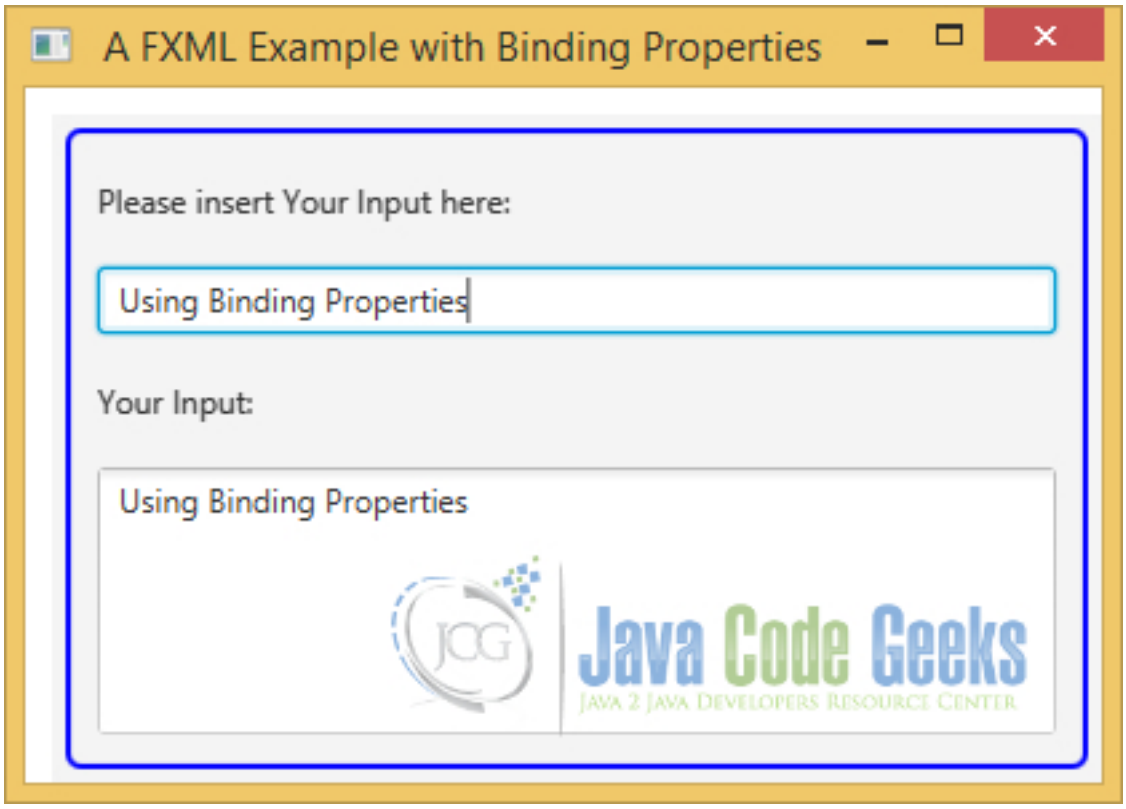

Figure 6.7: A JavaFX FXML Example with Binding Properties

6.8 Using Resource Bundles

6.8.1 The FXML Code

FxFXMLExample8.fxml

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.Pos?>

<VBox fx:id="vbox" layoutX="10.0" layoutY="10.0" prefHeight="250.0" prefWidth="300.0"  ↩
    spacing="10" xmlns:fx="https://javafx.com/fxml/1" xmlns="https://javafx.com/javafx/2.2"  ↩
    fx:controller="FXFXML.FxFXMLController">
  <style>
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
  </style>
        <alignment><Pos fx:constant="CENTER_LEFT" fx:id="alignCenter"/></alignment>
  <children>
    <Label fx:id="inputLbl" alignment="$alignCenter" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="%input" />
    <TextField fx:id="inputText" prefWidth="100.0" />
    <Button fx:id="okBtn" alignment="$alignCenter" mnemonicParsing="false" onAction="# ↩
        printOutput" text="OK" />
    <Label fx:id="outputLbl" alignment="$alignCenter" cache="true" cacheHint="SCALE"  ↩
        prefHeight="30.0" prefWidth="200.0" text="%output" />
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="200.0" wrapText="true" />
  </children>
</VBox>
```

Using a `ResourceBundle` in FXML is much easier than using it in Java code. Specifying the keys from a `ResourceBundle` in attribute values uses the corresponding values for the default Locale. If an attribute value starts with a % symbol, it is considered as the key name from the resource bundle.

At runtime, the attribute value will come from the specified `ResourceBundle` in the `FXMLLoader`. If you want to use a leading % symbol in an attribute value, escape it with a backward slash (e.g., "%key").

Our example uses "%input" and "%output" as the value for the text property of the `Label`. The attribute value starts with a % symbol. The `FXMLLoader` will look up the value of the "input" and "output" in the `ResourceBundle` and use it for the text property.

```
<Label fx:id="inputLbl" alignment="$alignCenter" cache="true" cacheHint="SCALE" prefHeight= ↩
    "30.0" prefWidth="200.0" text="%input" />
<Label fx:id="outputLbl" alignment="$alignCenter" cache="true" cacheHint="SCALE" prefHeight ↩
    ="30.0" prefWidth="200.0" text="%output" />
```

6.8.2 The Properties Files for the Resource Bundles

In our example, we are using three `ResourceBundle` files: one for default `Locale` named Labels.properties, one for the German `Locale` named Labels_de.properties and one for the English `Locale` named Labels_en.properties:

Labels.properties

```
input=Input:
output=Output:
```

Labels_de.properties

```
input=Bitte geben Sie her Ihren Text ein:
output=Ihre Eingabe:
```

Labels_en.properties

```
input=Please insert Your Input here:
output=Your Input:
```

6.8.3 The Corresponding Java Class

FxFXMLExample8.java

```
import java.io.IOException;
import java.util.Locale;
import java.util.ResourceBundle;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FxFXMLExample8 extends Application
{
        public static void main(String[] args)
        {
                Application.launch(args);
        }

        @Override
        public void start(Stage stage) throws IOException
        {
                FXMLLoader fxmlLoader = new FXMLLoader();
        fxmlLoader.setResources(ResourceBundle.getBundle("FXFXML.Labels", new Locale("de")) ↩
            );
        VBox root = (VBox) fxmlLoader.load(this.getClass().getResource("FxFXMLExample8.fxml ↩
            ").openStream());
        // replace the content
                // Create the Scene
                Scene scene = new Scene(root);
                // Set the Scene to the Stage
                stage.setScene(scene);
                // Set the Title to the Stage
                stage.setTitle("A FXML Example using Resource Bundles");
                // Display the Stage
                stage.show();
        }
}
```

6.8.4 The GUI

The following GUI shows the effect of using a `ResourceBundle` for the German `Locale`:

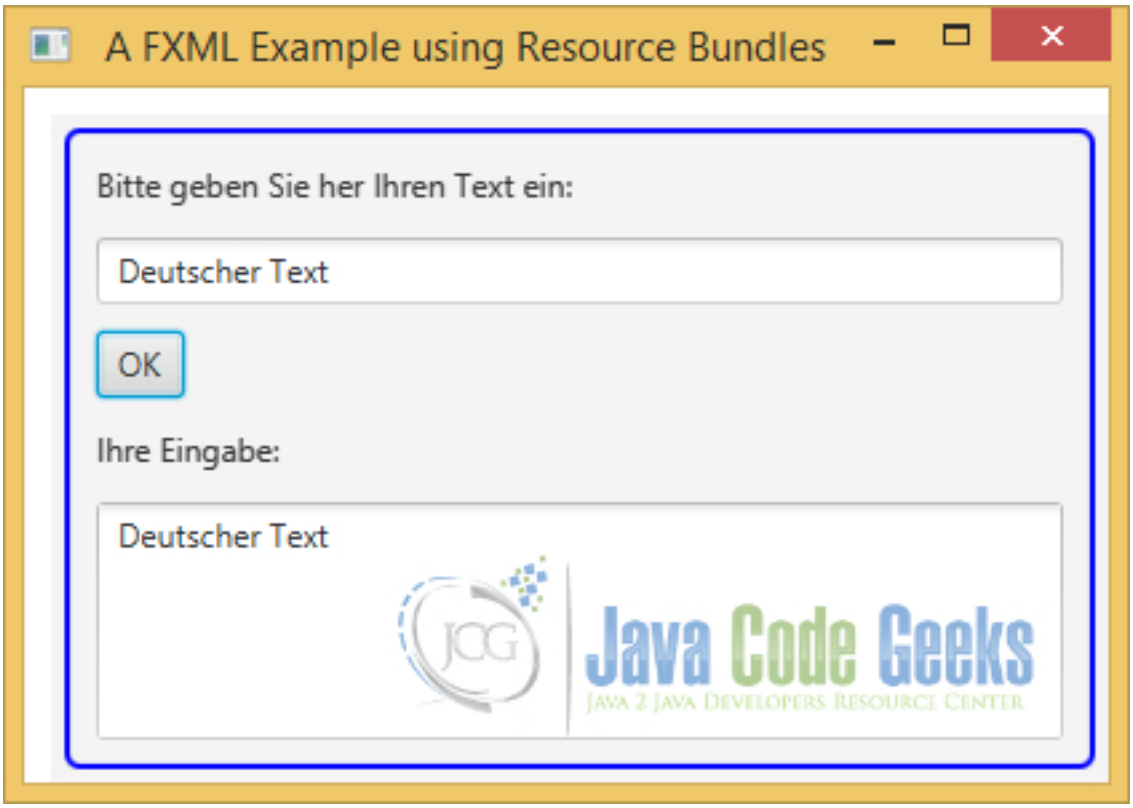

Figure 6.8: A JavaFX FXML Example with a ResourceBundle

6.9 Download Java Source Code

This was a JavaFX FXML Tutorial.

Download

You can download the full source code of this example here: **JavaFxFXMLTutorial.zip**

Chapter 7

JavaFX Applications with e(fx)clipse

This is an example how to use the e(fx)clipse IDE for creating JavaFX Projects and Applications.

The `e(fx)clipse` standard library provides some useful extensions for writing JavaFX code. The library offers, among other features, additional layout panels, using FXML, Eclipse databinding for JavaFX properties, and much more.

The following instructions were written with a clean install of the Eclipse Java EE IDE for Web Developers. The Eclipse Version was Mars.1 Release (4.5.1).The used Java Version was Java8 SE.

7.1 Installing the e(fx)clipse IDE

At first you have to start your Eclipse Software. Thereafter go to the Help Menu and select the "Install New Software..." option.

The following Dialog will appear:

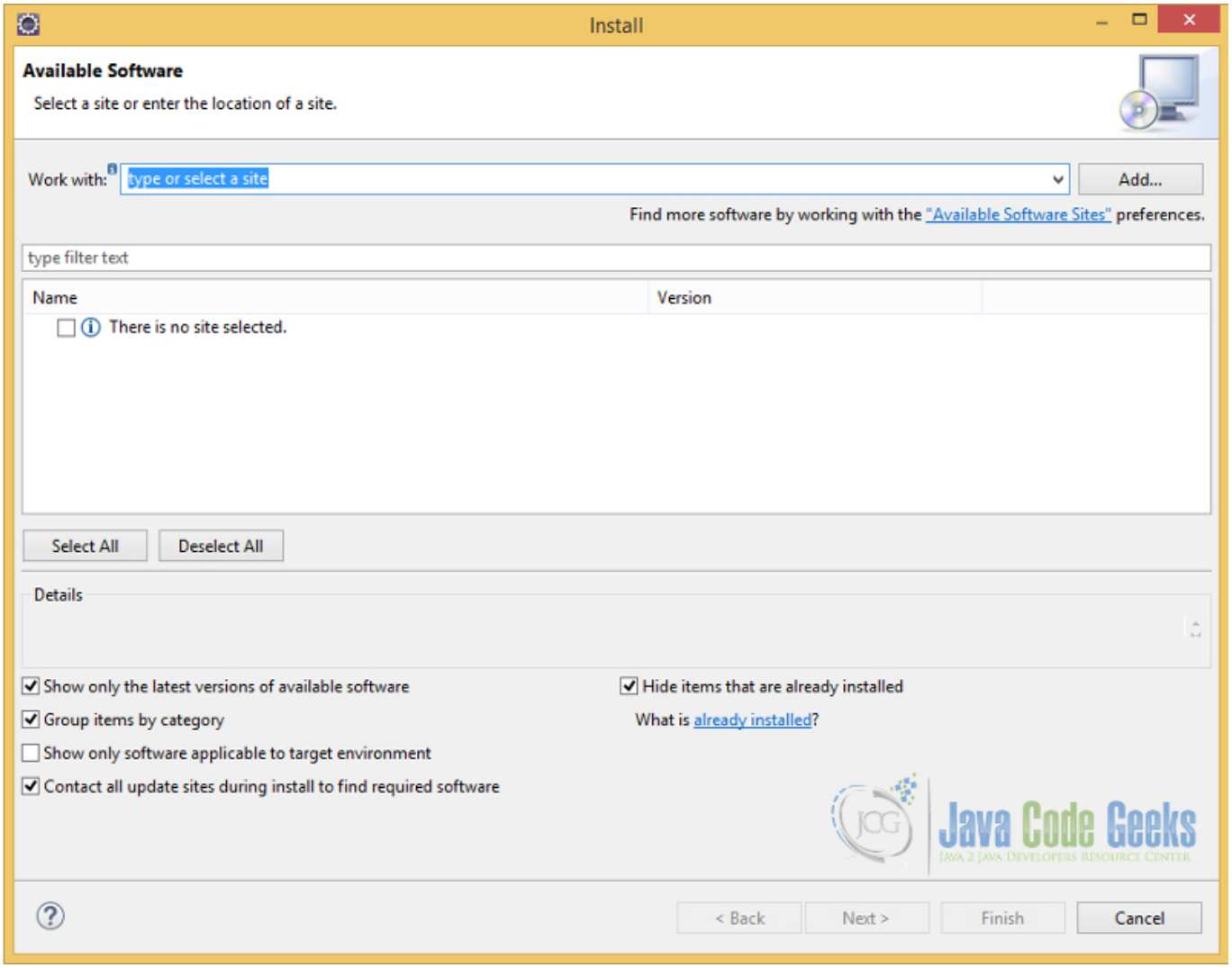

Figure 7.1: The Install new Software Dialog

Now you have to define a Repository for the `e(fx)clipse` Installation. After clicking the Add Button, the "Add Repository" Dialog will appear:

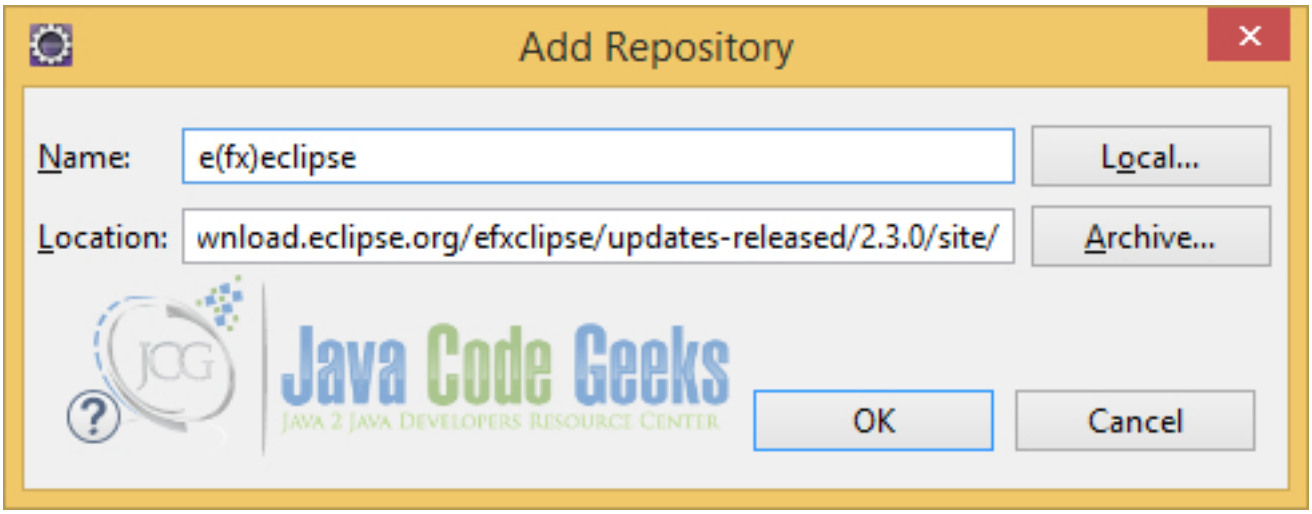

Figure 7.2: The Add Repository Dialog

You have to enter the name of the repository and the Location of the software. I have chosen `e(fx)clipse` as name and https://download.eclipse.org/efxclipse/updates-released/2.3.0/site as Location for the following examples.

After defining the Repository, all possible items of the Update-Site will appear. Now you can choose, which items should be installed:

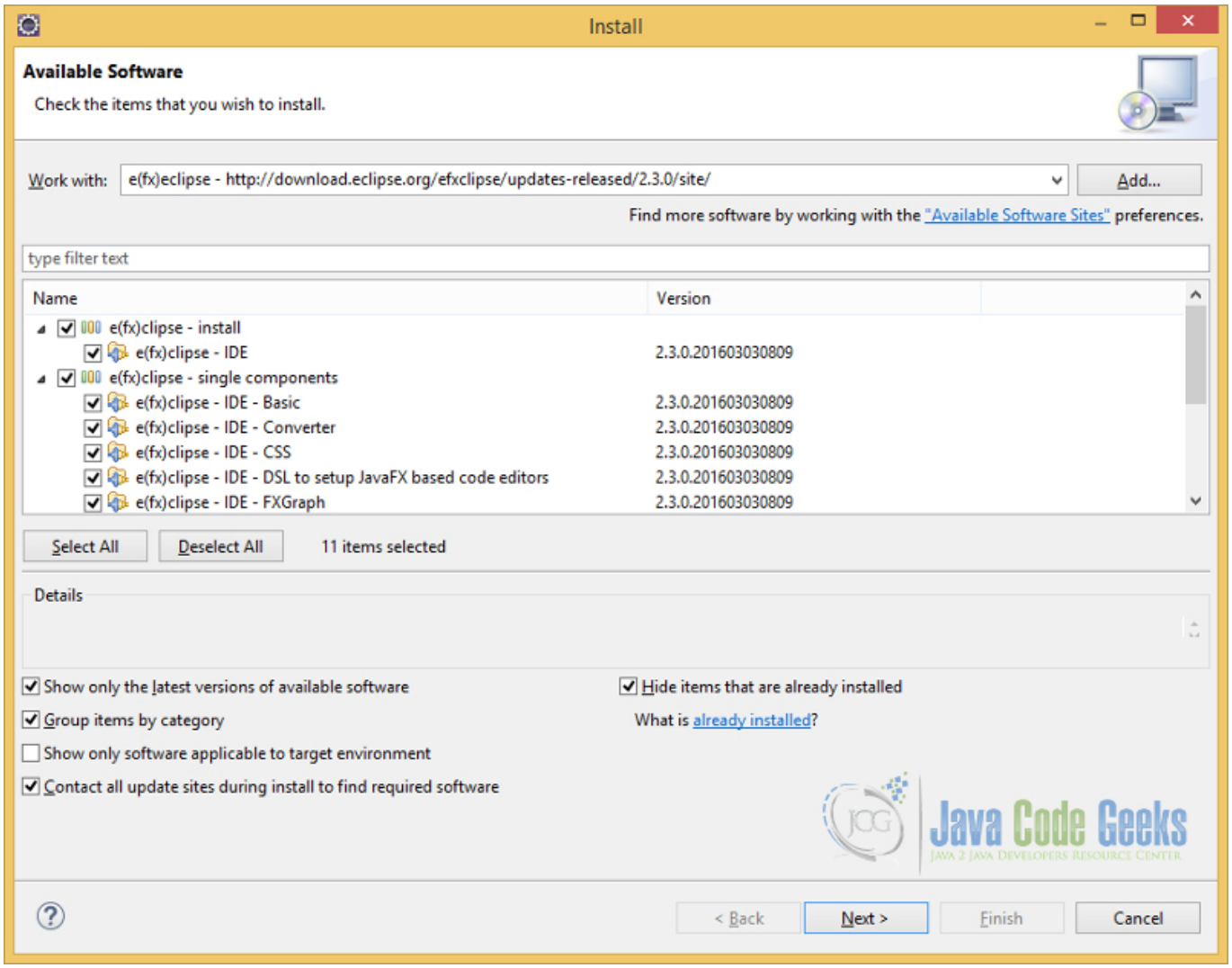

Figure 7.3: The Check Items Dialog

At the end of the Selection and pressing the Next Button, an Overview of the selected items will apppear:

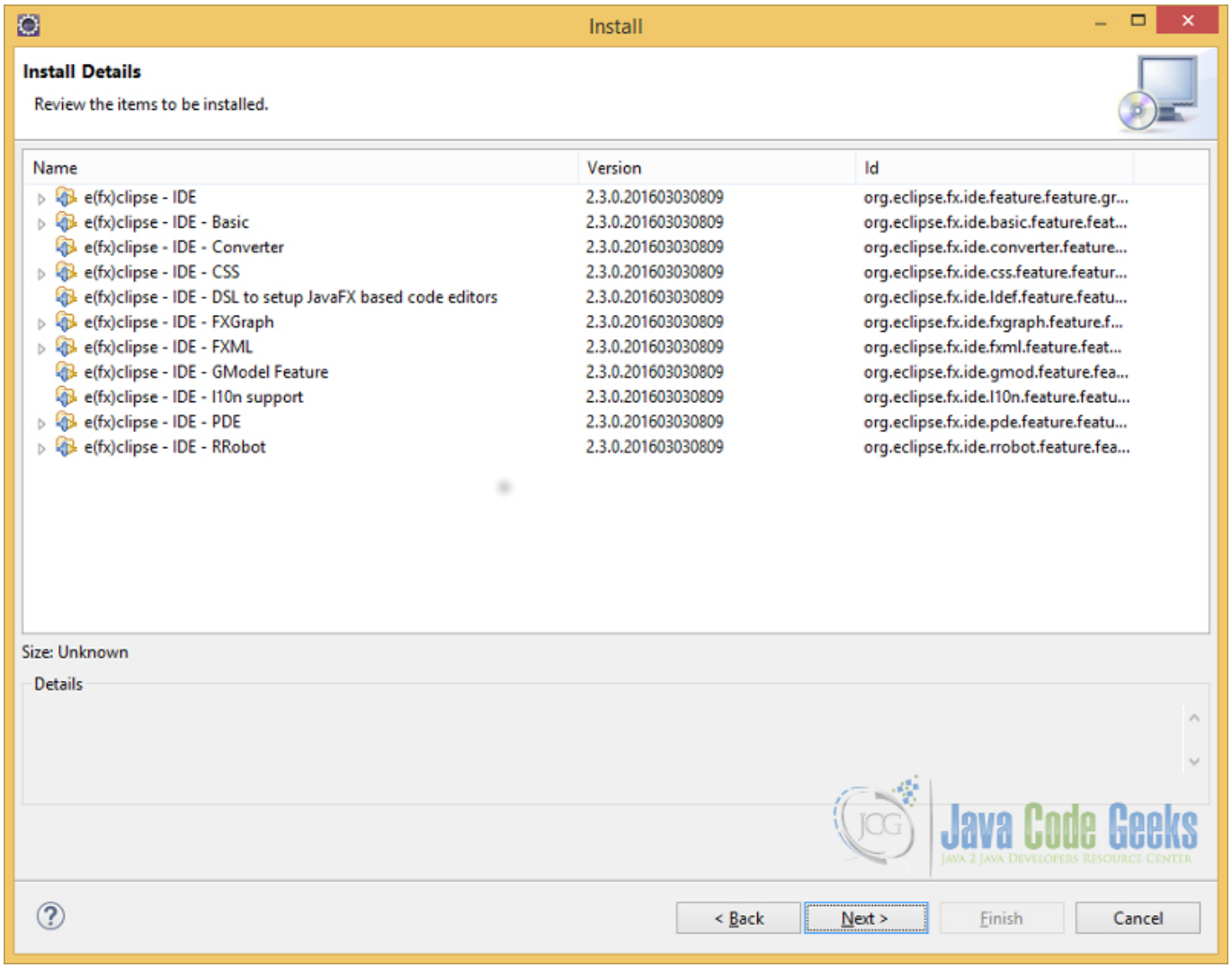

Figure 7.4: The Install Details Overview

At next, the terms of the License Agreement have to be accepted:

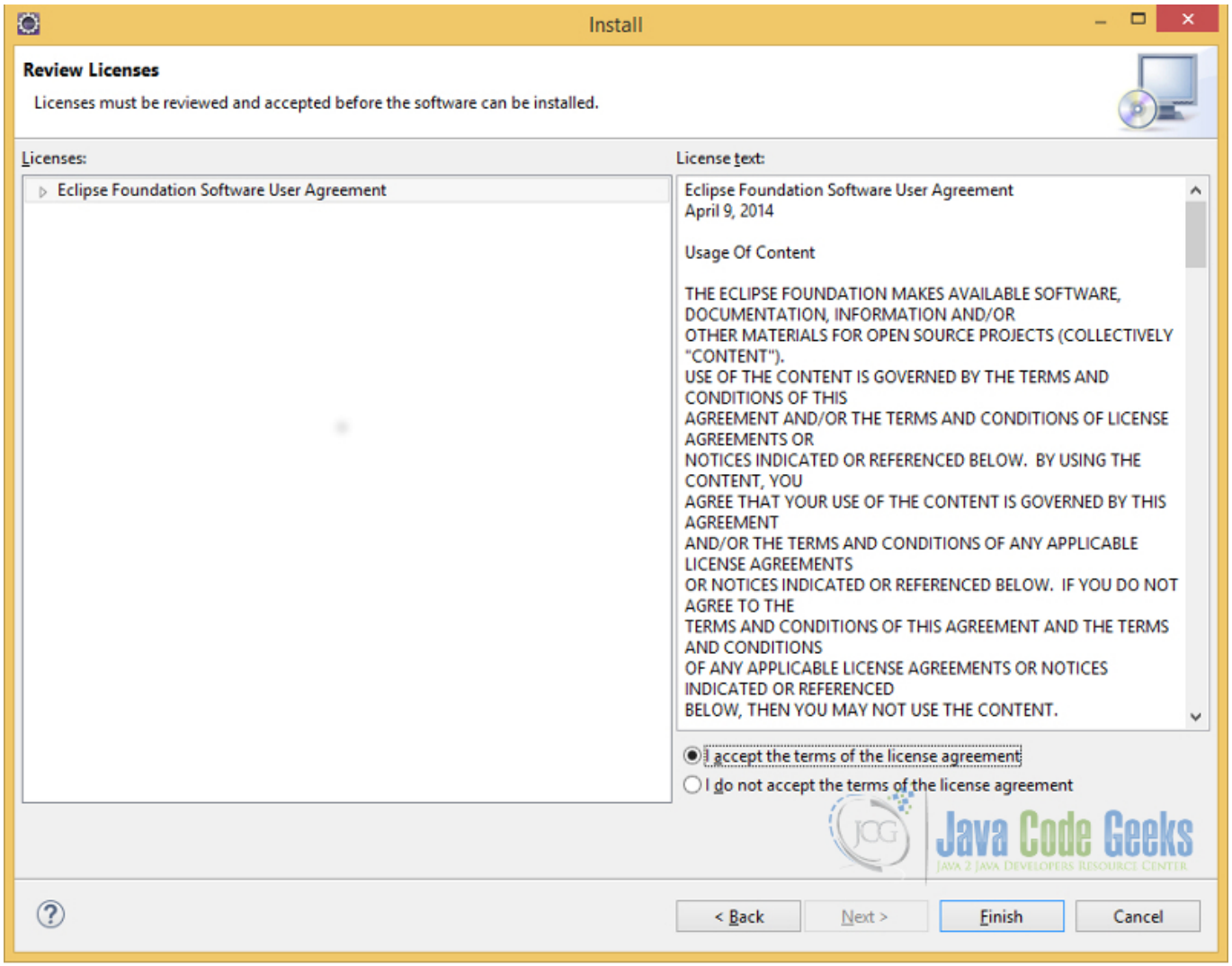

Figure 7.5: The Review License Dialog

After clicking the Finish Button, the installation will start:

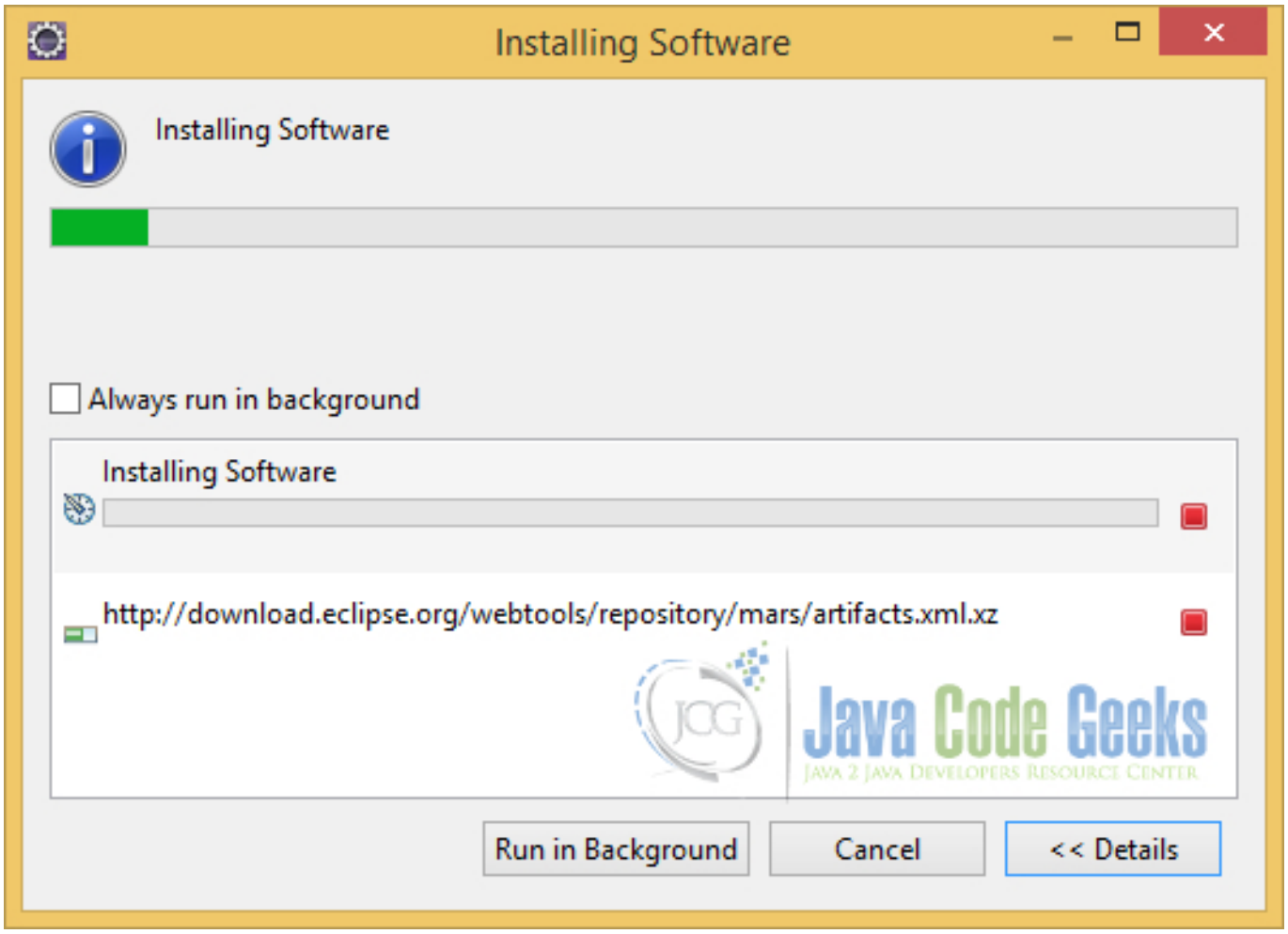

Figure 7.6: The Install Dialog

When the installation process is finished, you have to restart your Eclipse. Thereafter you can create JavaFX Projects with the `e(fx)clipse` IDE in your Eclipse Environment.

7.2 Your first JavaFX Example with e(fx)clipse

In this example, I only discuss how you can generate the Project and the necessary changes in the created files. If you want to learn more about JavaFX, please read my JavaFX Tutorial for Beginners.

7.2.1 Creation of the JavaFX Project

At first you have to create a JavaFx Project. Go to the File Menu and choose New Project. Select the "JavaFX Project" entry in the wizard:

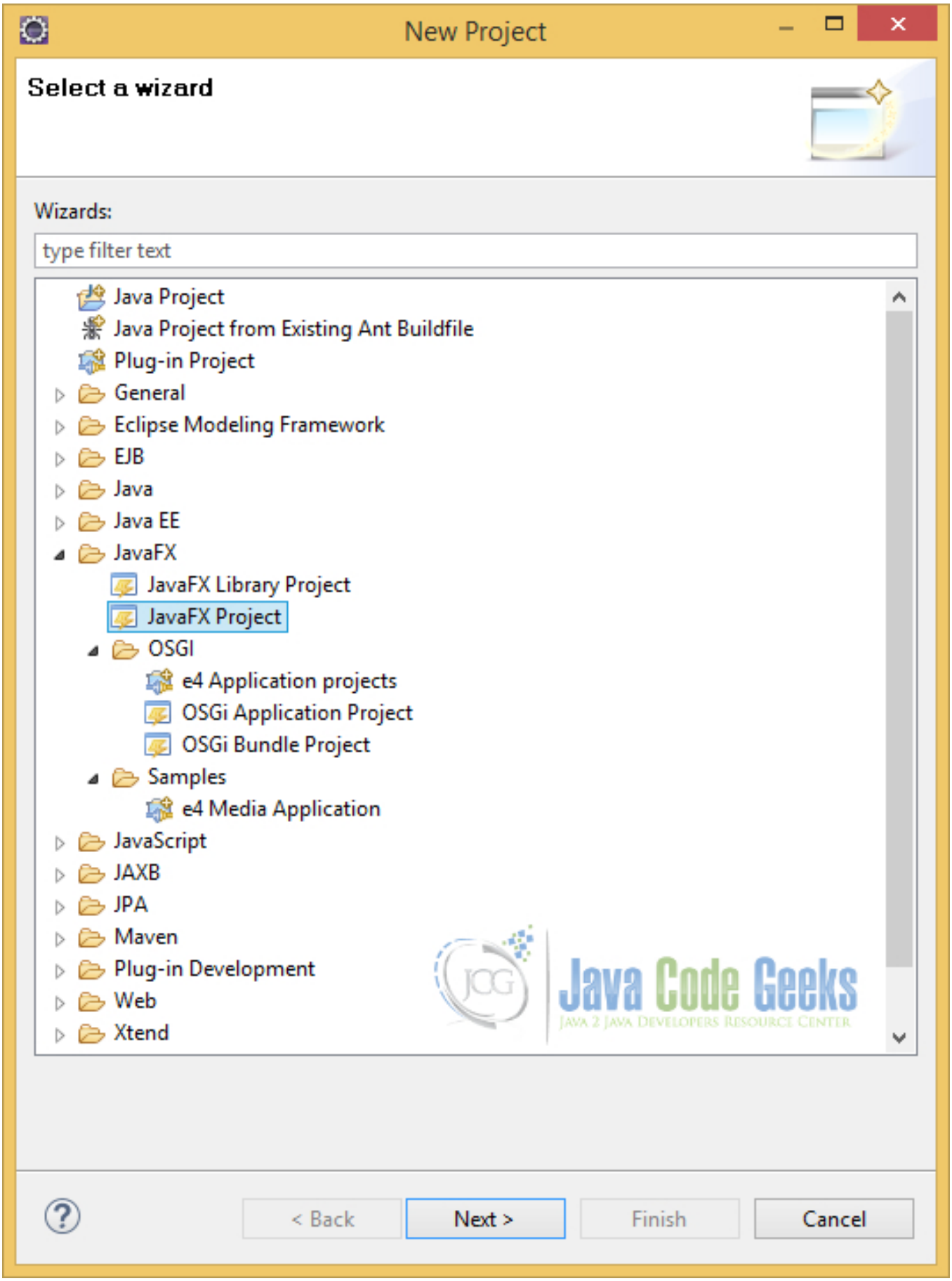

Figure 7.7: Choose the Type of the new Java Project

Enter a project name and click Next:

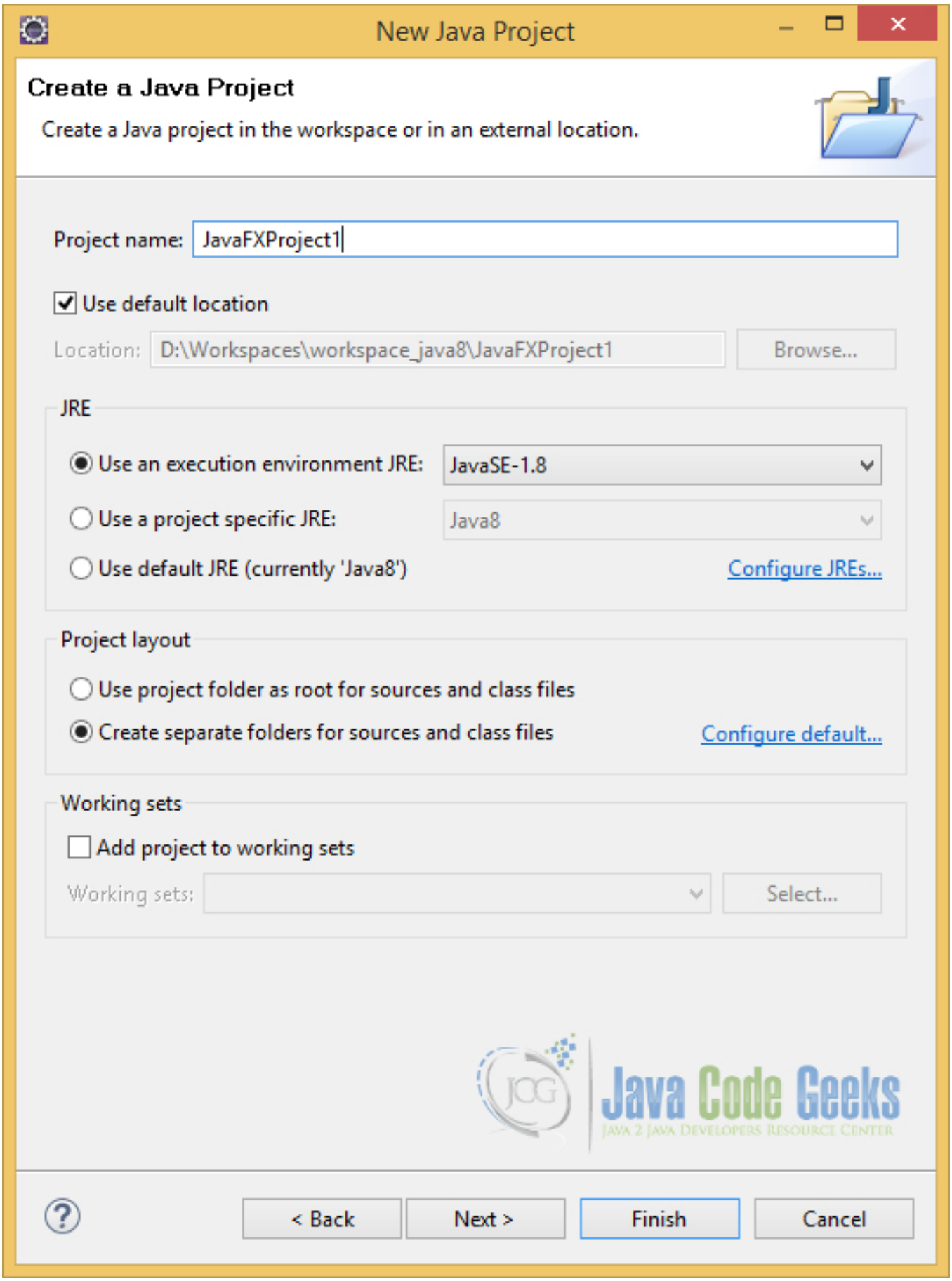

Figure 7.8: The Create Java Project Dialog in Eclipse

Now you can add other external libraries, if it is necessary:

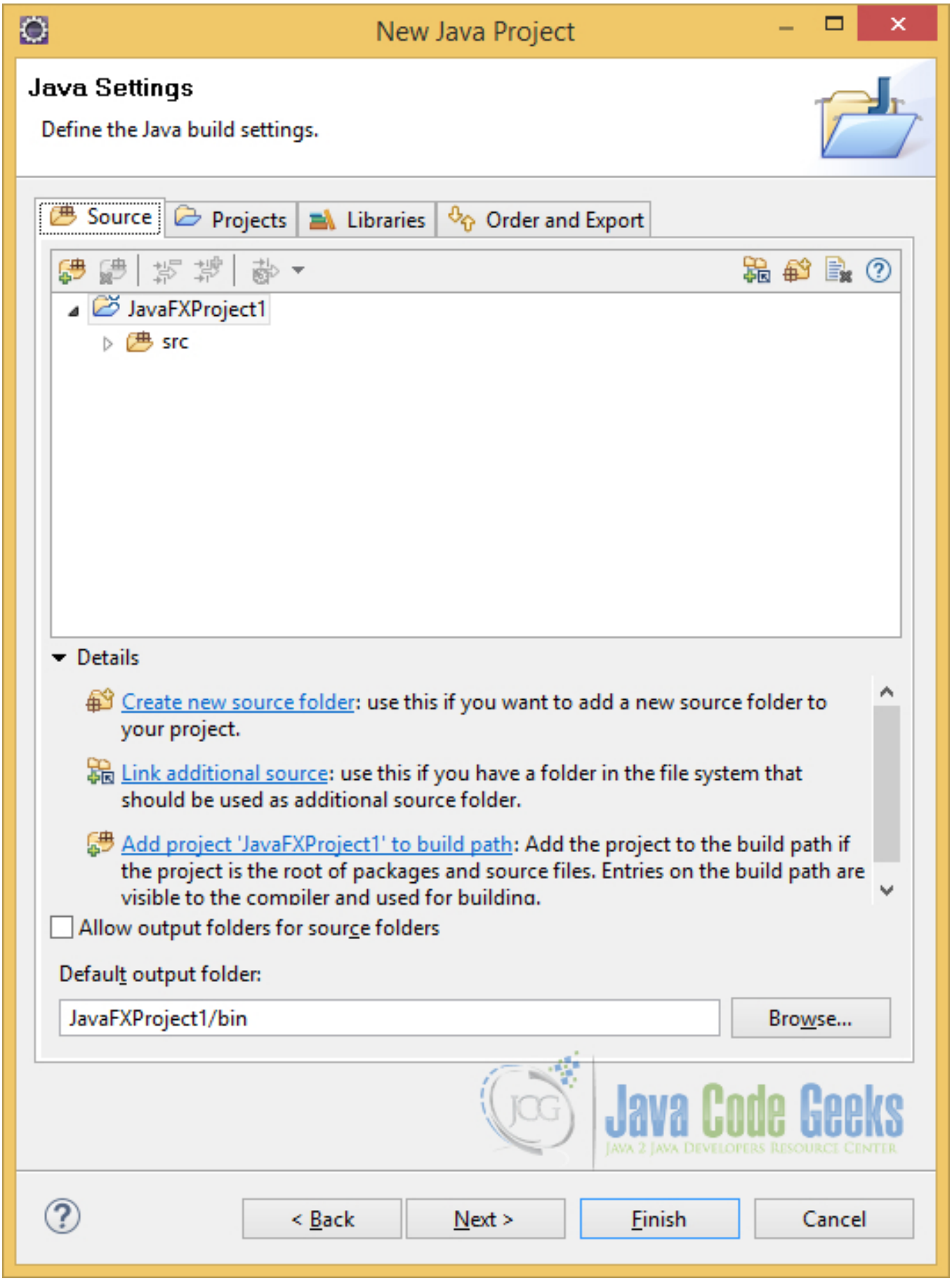

Figure 7.9: The Java Settings Dialog

The next step represents the Selection of the "Application Type". There exists Desktop and Mobile. For this article I have chosen Desktop for the creation of a Desktop Application.

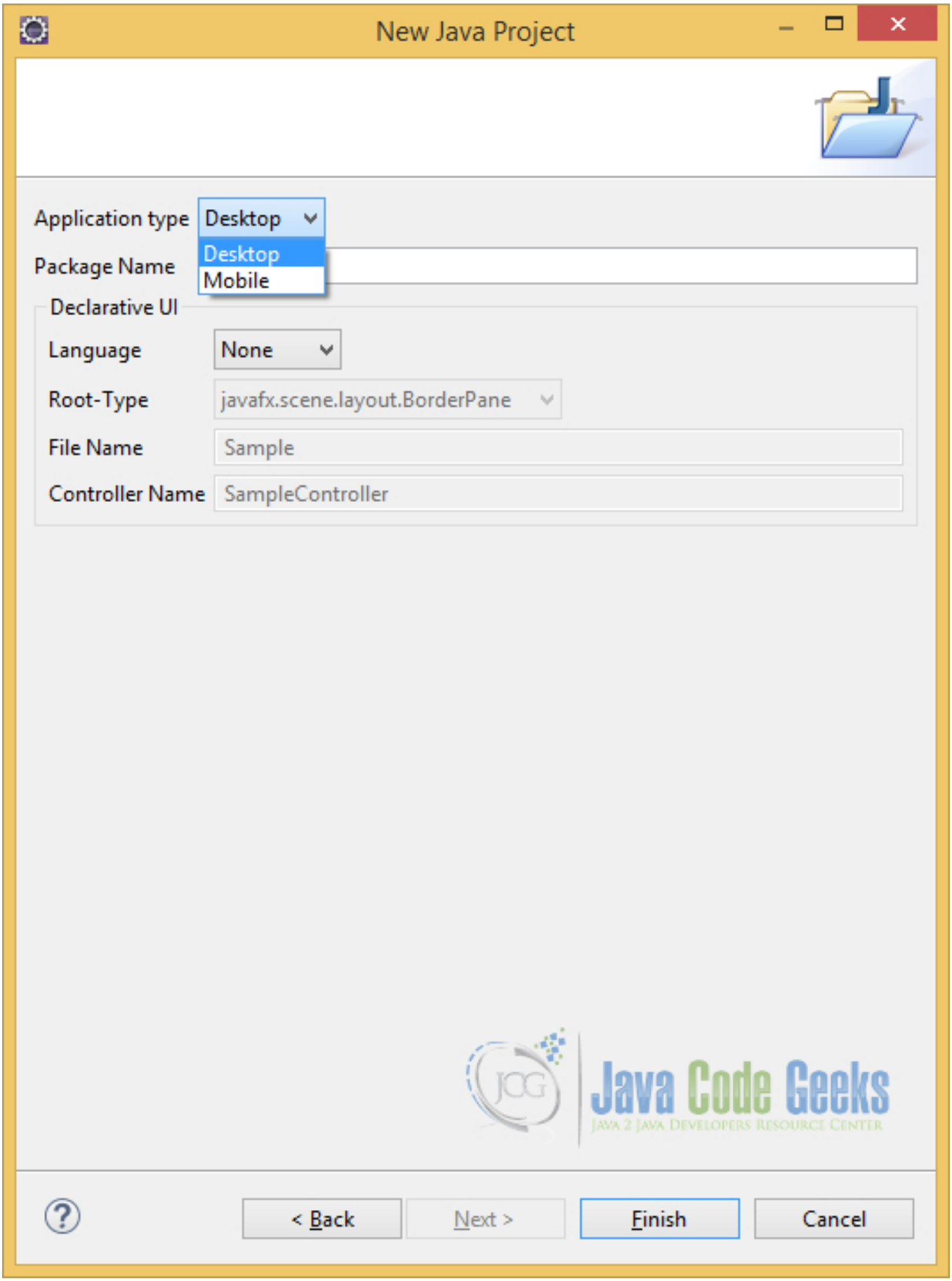

Figure 7.10: Selecting the Application Type of the new JavaFX Project

Now you have to choose the Language. You can select None, FXML and FxGraph. None means, that the Project contains only Java Files and StyleSheets. If you want to use FXML for developing your GUI, you have to select FXML. FXGraph is a simple DSL for the definition of a JavaFX 2.x object graph.

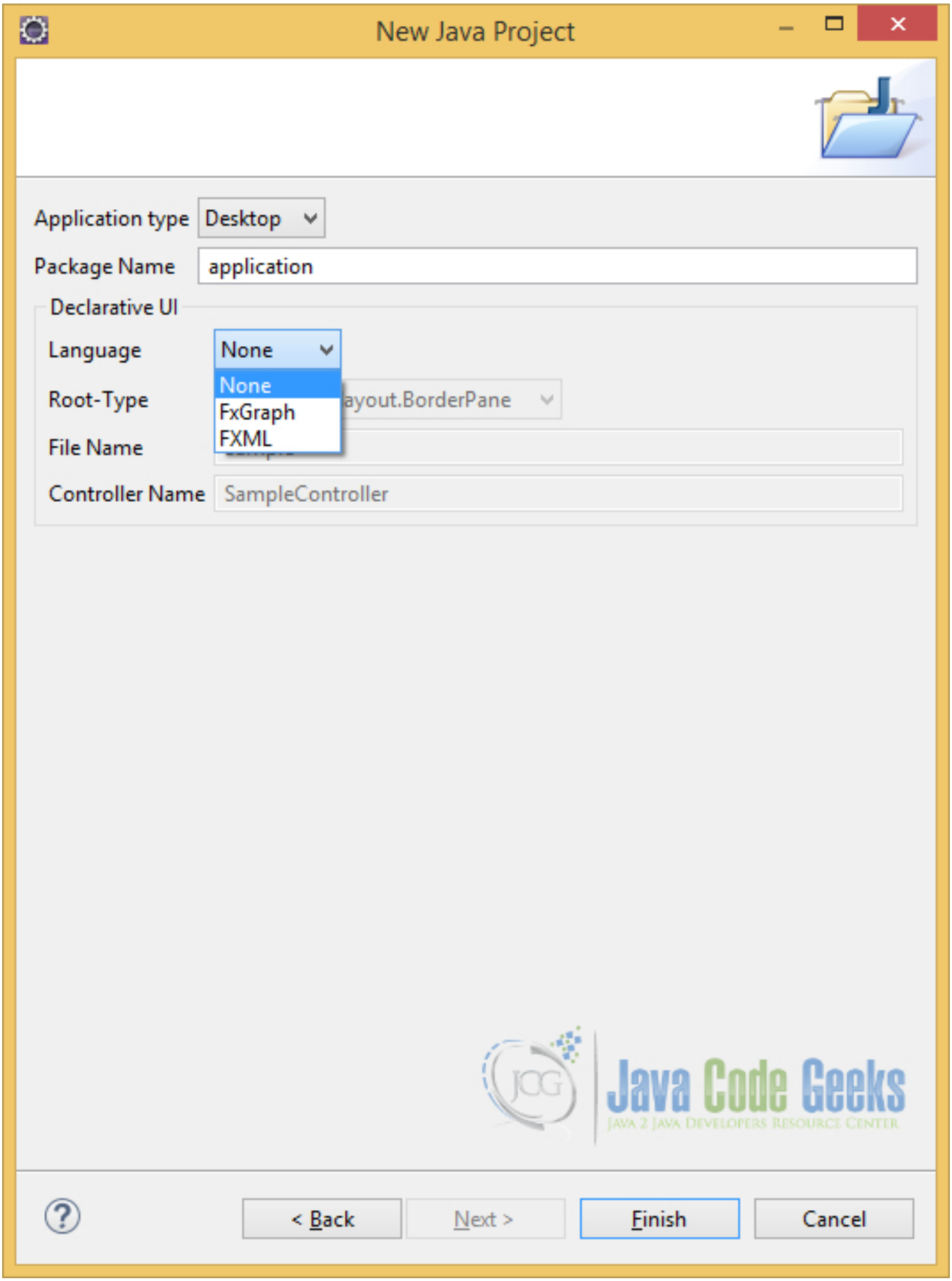

Figure 7.11: Selecting the Language of the new JavaFX Project

After a click on the Finish Button Eclipse will create the Project and some Classes and Stylesheets:

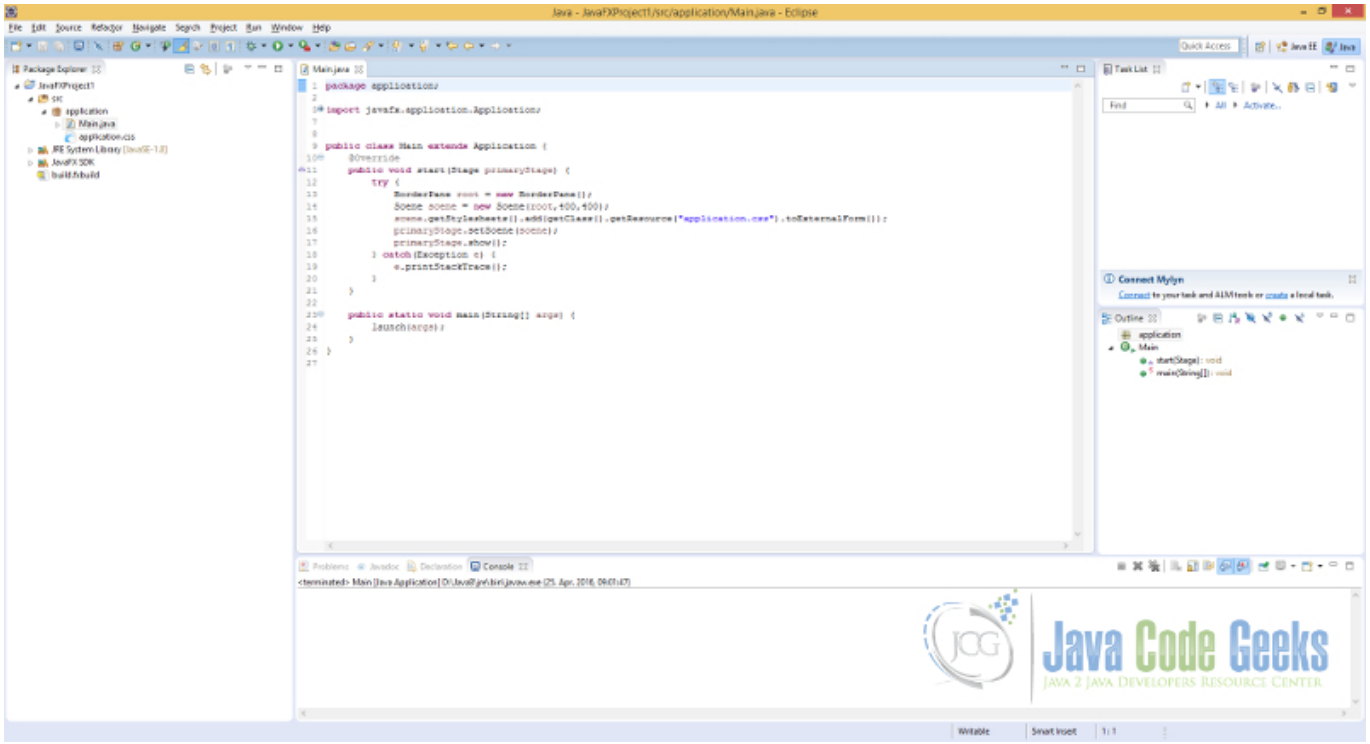

Figure 7.12: The genereted Main Class

Thereafter, the `application` Package contains the following Files:

- Main.java
- application.css

7.2.2 Changing the Main Class

The following code snippet shows the generated `Main` class:

Main.java

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;

public class Main extends Application {
        @Override
        public void start(Stage primaryStage) {
                try {
                        BorderPane root = new BorderPane();
                        Scene scene = new Scene(root,400,400);
                        scene.getStylesheets().add(getClass().getResource("application.css" ↩
                            ).toExternalForm());
                        primaryStage.setScene(scene);
                        primaryStage.show();
                } catch(Exception e) {
                        e.printStackTrace();
```

```
                }
        }

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

Now you are able to change the `Main` class, create other Java classes and so on. I only have written a small example. The following Java Code represents my modified `Main` class:

Main.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;

public class Main extends Application
{
        // Create the TextField for the input
        private TextField inputArea = new TextField();
        // Create the TextArea for the Output
        private TextArea outputArea = new TextArea();

        @Override
        public void start(Stage primaryStage)
        {
                try
                {
                        // Create the Label for the Header
                        Label headerLbl = new Label("Please insert your Message in the  ↩
                            TextArea!");
                        // Create the Label for the Input
                        Label inputLbl = new Label("Input: ");
                        // Create the OK-Button
                        Button okBtn = new Button("OK");

                        // add an EventHandler to the OK-Button
                        okBtn.setOnAction(new EventHandler<ActionEvent>()
                        {
                                @Override
                                public void handle(ActionEvent event)
                                {
                                        writeOutput(inputArea.getText());
                                }
                        });

                        // Create the BorderPane
                        BorderPane root = new BorderPane();
                        // Store the Header Label in the Top Region
                        root.setTop(headerLbl);
                        // Store the OK Button in the Top Region
                        root.setRight(okBtn);
                        // Store the Output Area in the Right Region
```

```
                        root.setBottom(outputArea);
                        // Store the Input Label in the Bottom Region
                        root.setLeft(inputLbl);
                        // Store the Input Area in the Center Region
                        root.setCenter(inputArea);

                        // Set the alignment of the Header Label to bottom center
                        BorderPane.setAlignment(headerLbl,Pos.BOTTOM_CENTER);
                        // Set the alignment of the Input Label to center left
                        BorderPane.setAlignment(inputLbl,Pos.CENTER_LEFT);
                        // Set the alignment of the OK Button to center right
                        BorderPane.setAlignment(okBtn,Pos.CENTER_RIGHT);

                        // Create the Scene
                        Scene scene = new Scene(root);
                        // Add the StyleSheets to the Scene
                        scene.getStylesheets().add(getClass().getResource("application.css" ↩
                            ).toExternalForm());
                        // Add the scene to the Stage
                        primaryStage.setScene(scene);
                        // Set the title of the Stage
                        primaryStage.setTitle("A JavaFx Example created with e(fx)clipse");
                        // Display the Stage
                        primaryStage.show();
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }

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

        // Method to log the Message to the Output-Area
        private void writeOutput(String msg)
        {
                this.outputArea.appendText("Your Input: " + msg + "\n");
        }
}
```

The following image shows the modified class in the Eclipse Workspace:

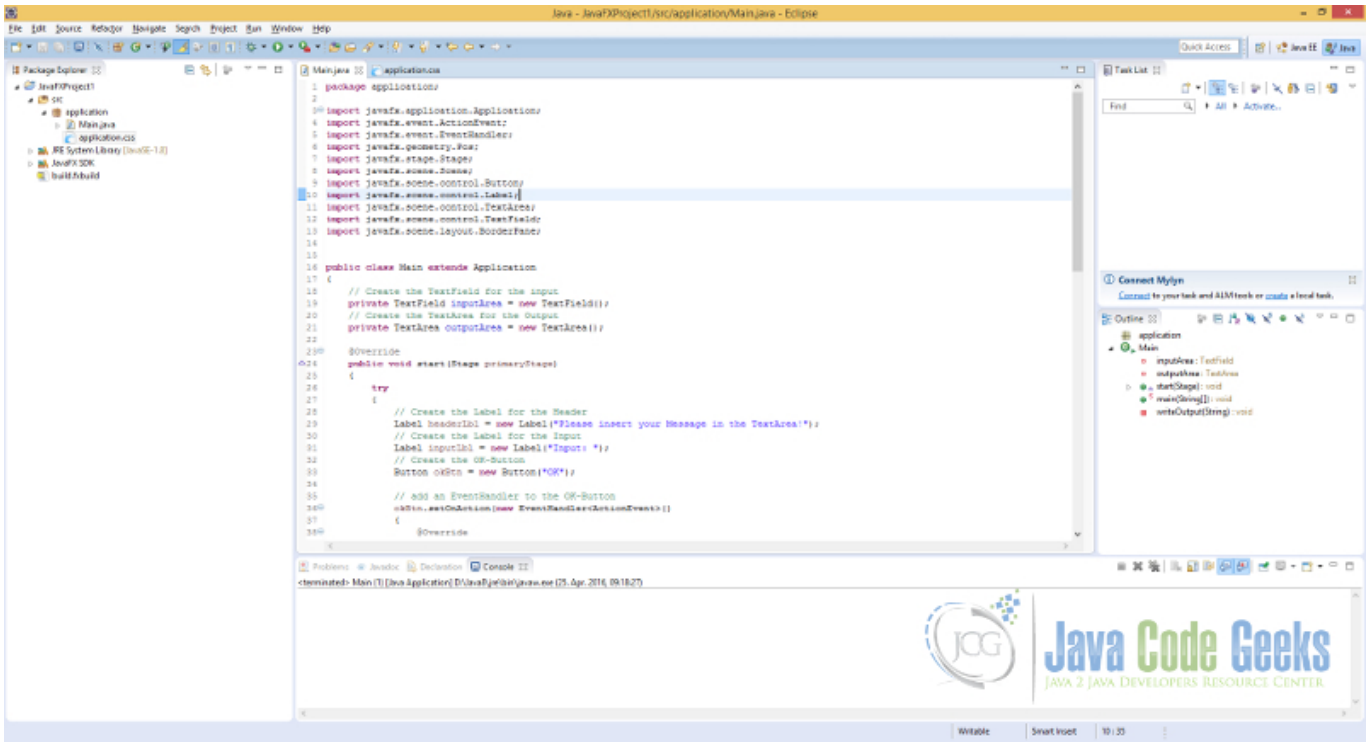

Figure 7.13: The changed Main Class

7.2.3 Changing the StyleSheet

The generated StyleSheet is empty at the beginning:

application.css

```
/* JavaFX CSS - Leave this comment until you have at least create one rule which uses -fx- ↩
    Property */
```

Now you can change the style of some GUI Elements like the Scene, the Button, etc. I have only made changes for the `Scene`:

application.css

```
/* JavaFX CSS - Leave this comment until you have at least create one rule which uses -fx- ↩
    Property */
.root
{
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
}
```

The following image shows the modified StyleSheet in the Eclipse Workspace:

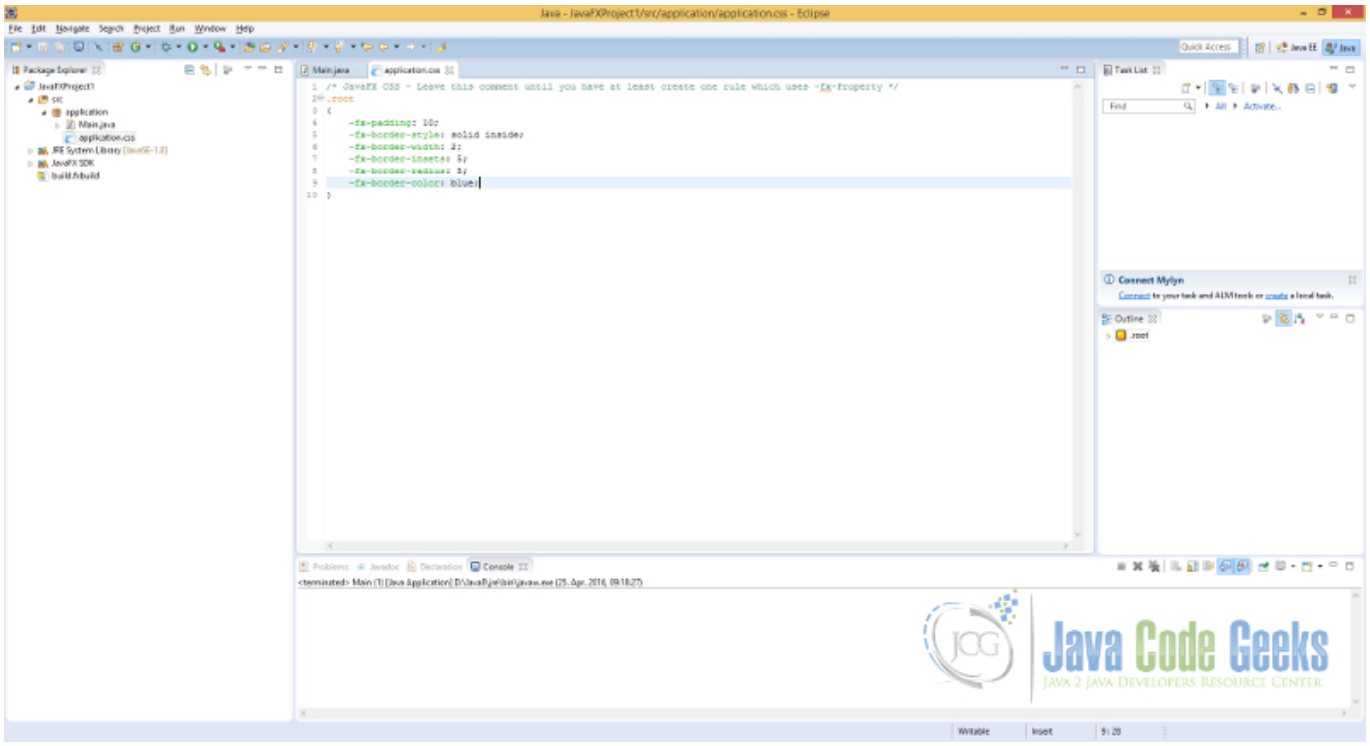

Figure 7.14: The changed Application StyleSheet

7.2.4 The GUI

The following GUI represents the result of all described changes:

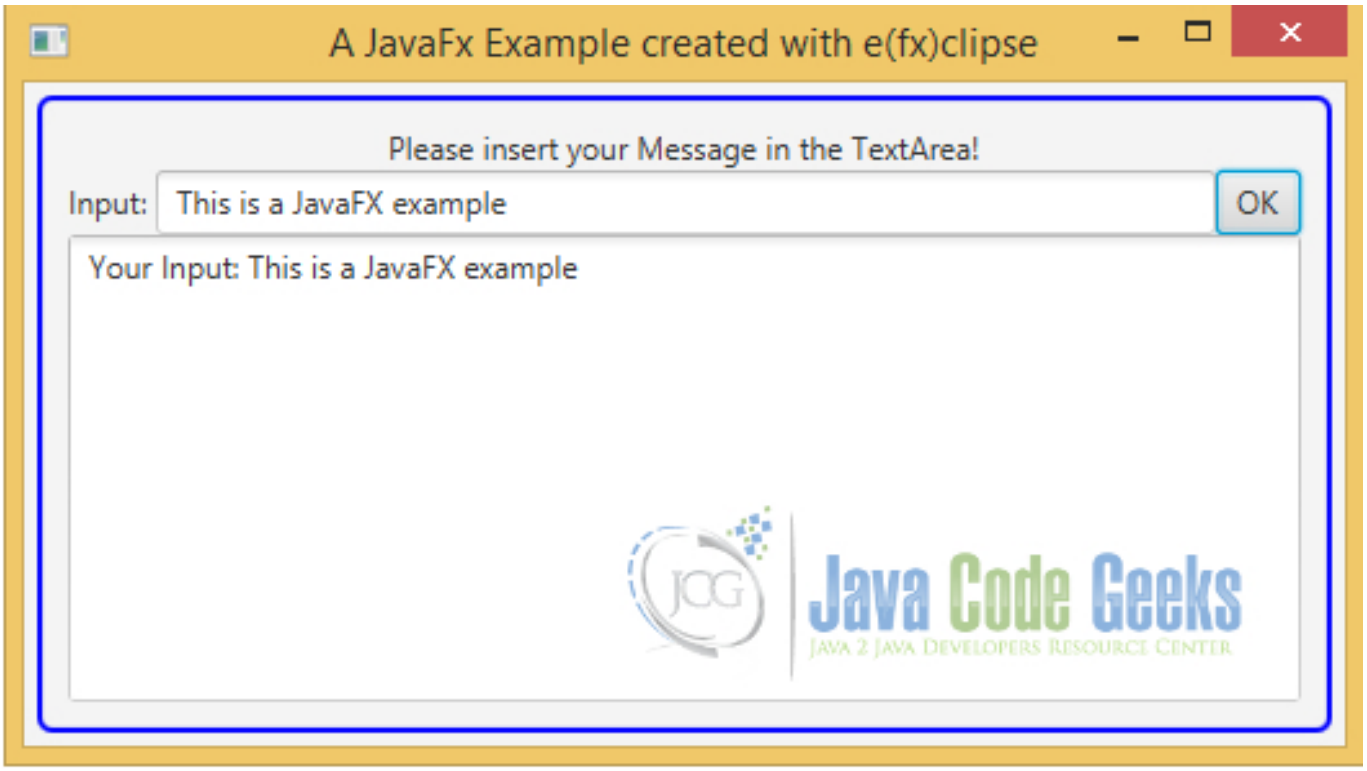

Figure 7.15: A simple JavaFX Example created with the e(fx)clipse IDE

7.3 A JavaFX FXML Example with e(fx)clipse

In this example, I only discuss how you can generate the Project and which files you have to change. If you want to read more about FXML, please read my JavaFX FXML Tutorial.

7.3.1 Creation of the JavaFX Project

At first you have to create a JavaFx Project. Go to the File Menu and choose New Project. Select the "JavaFX Project" entry in the wizard:

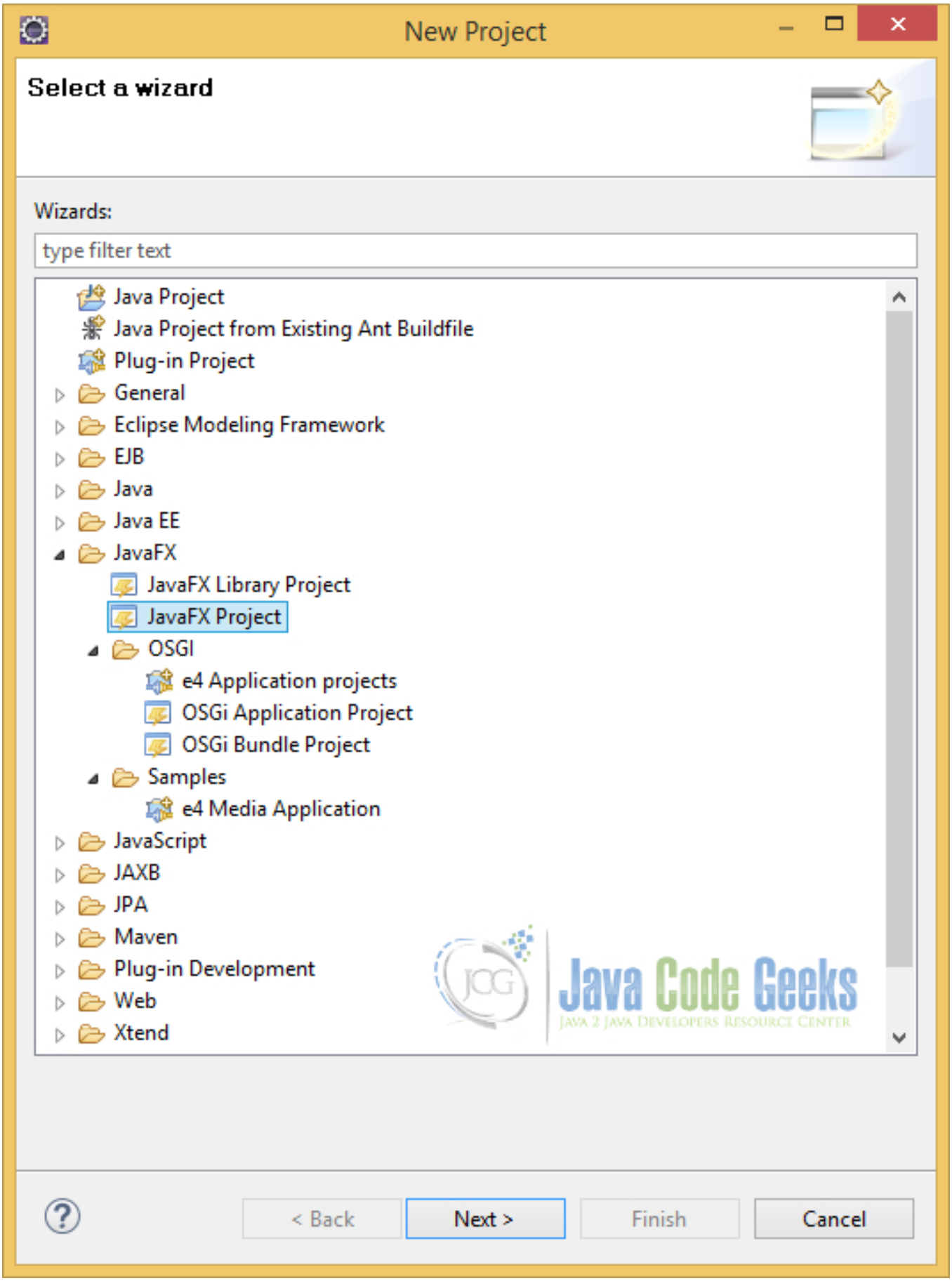

Figure 7.16: Choose the Type of the new Java Project

Like in the previous example, you must enter a project name and click Next:

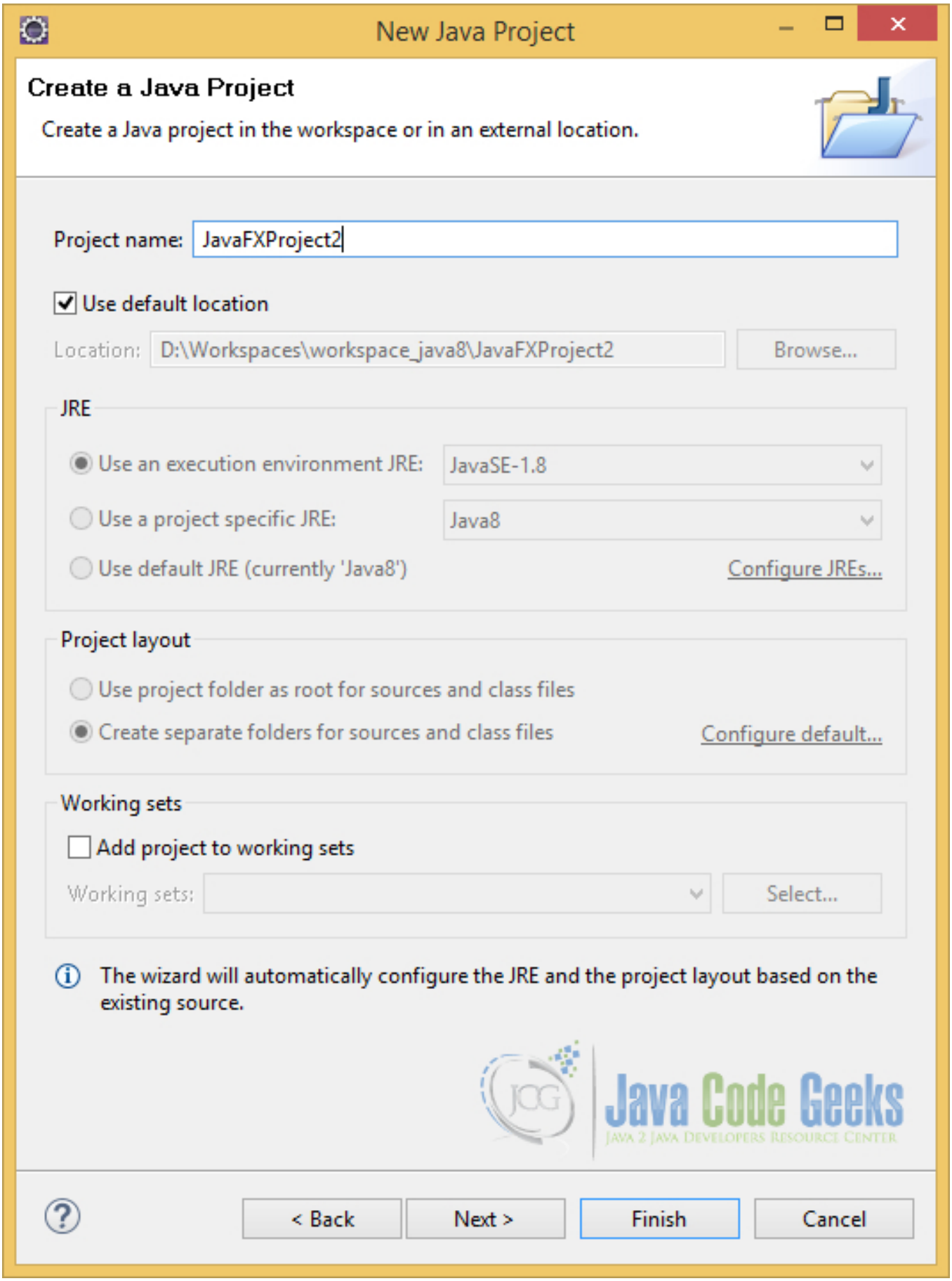

Figure 7.17: The Create Java Project Dialog in Eclipse

Now you can add other external libraries, if it is necessary:

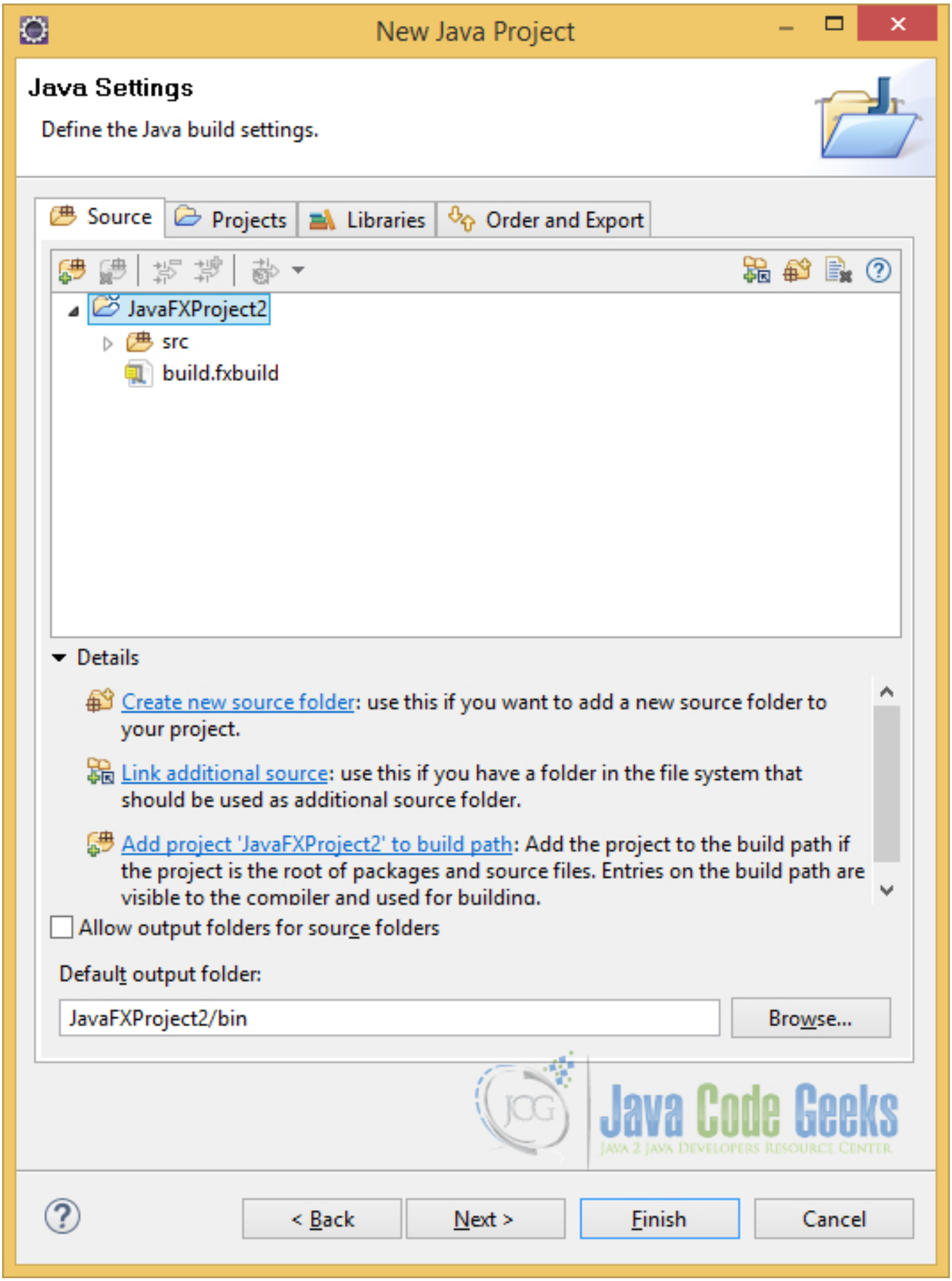

Figure 7.18: The Java Settings Dialog

Now you have to define the Application Type and the Language. The Application Type of this example is Desktop again. The Language is FXML, because we are creating a FXML Example. Given this fact, we have to define the Name of the FXML File in the "File Name" Field and the Name of the Controller class in the "Controller Name" Field.

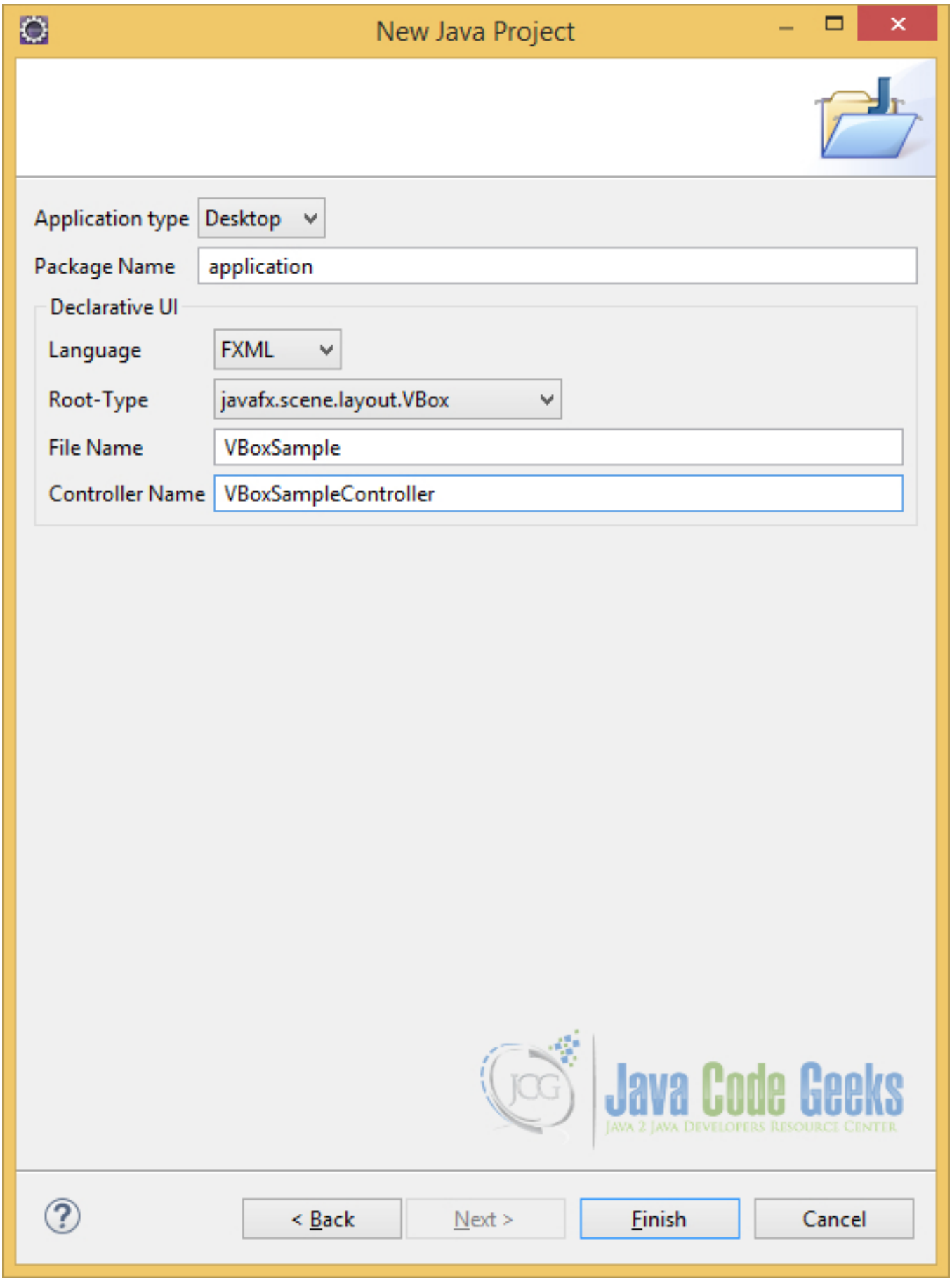

Figure 7.19: The Details of the new JavaFX Project

After a click on the Finish Button Eclipse creates the Project and its corresponding Java Classes, FXML Files and Stylesheets:

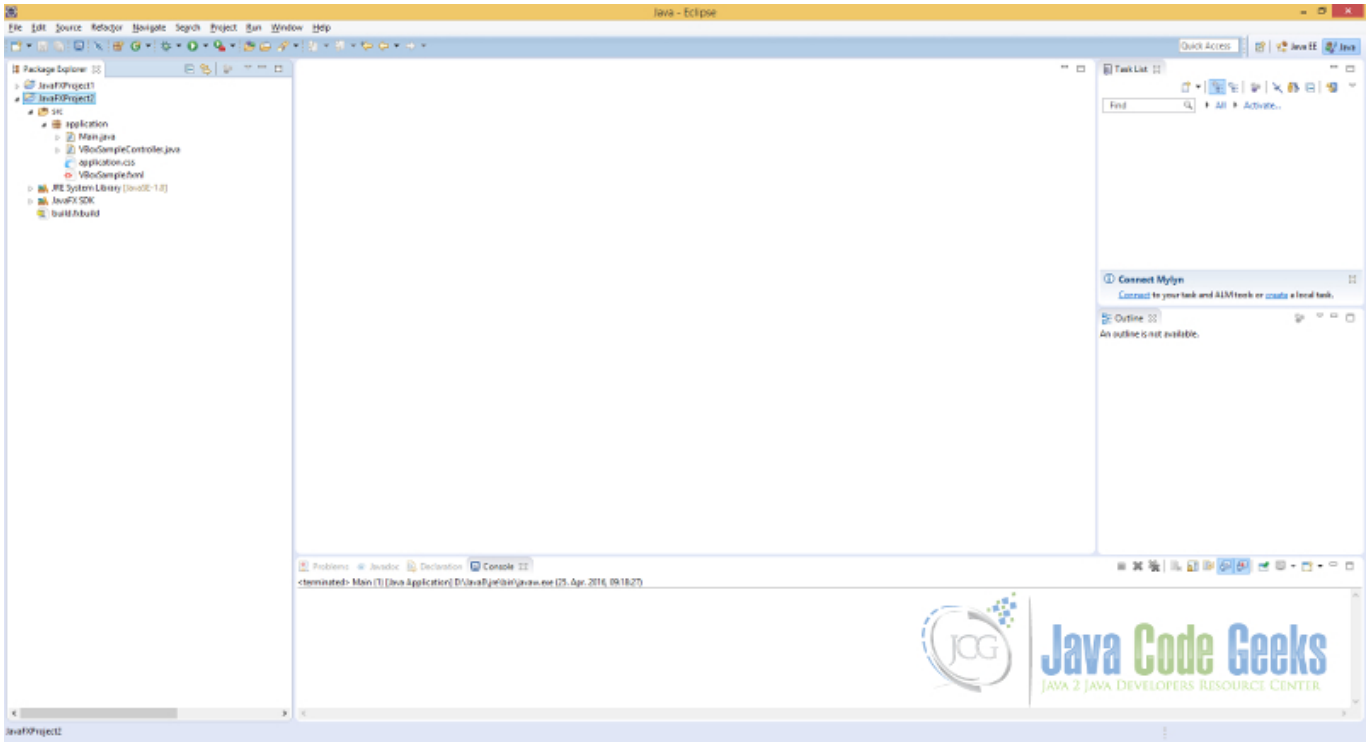

Figure 7.20: The Overview of the new JavaFX Project

Thereafter, the `application` Package contains the following Files:

- Main.java
- application.css
- VBoxSample.fxml
- VBoxSampleController.java

7.3.2 Changing the Main Class

The generated `Main` class contains the following Java Code:

Main.java

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.fxml.FXMLLoader;

public class Main extends Application {
        @Override
        public void start(Stage primaryStage) {
                try {
                        VBox root = (VBox)FXMLLoader.load(getClass().getResource(" ↩
                            VBoxSample.fxml"));
                        Scene scene = new Scene(root,400,400);
```

```
                        scene.getStylesheets().add(getClass().getResource("application.css" ↩
                            ).toExternalForm());
                        primaryStage.setScene(scene);
                        primaryStage.show();
                } catch(Exception e) {
                        e.printStackTrace();
                }
        }

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

Given the fact, that we want to create a FXML Example, there are only small changes in the `Main` class necessary:

Main.java

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.fxml.FXMLLoader;


public class Main extends Application
{
        @Override
        public void start(Stage primaryStage)
        {
                try
                {
                        // Load the FXML File
                        VBox root = (VBox)FXMLLoader.load(getClass().getResource(" ↩
                            VBoxSample.fxml"));

                        // Create the Scene
                        Scene scene = new Scene(root,400,400);
                        // Add the StyleSheet to the Scene
                        scene.getStylesheets().add(getClass().getResource("application.css" ↩
                            ).toExternalForm());
                        // Set the Title to the Stage
                        primaryStage.setTitle("A FXML Example created with e(fx)clipse");
                        // Add the Scene to the Stage
                        primaryStage.setScene(scene);
                        // Show the Stage
                        primaryStage.show();
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }

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

7.3.3 Changing the StyleSheet

The generated StyleSheet is empty at the beginning:

application.css

```
/* JavaFX CSS - Leave this comment until you have at least create one rule which uses -fx- ↩
    Property */
```

The modified StyleSheet is the same like in the previous example:

application.css

```
/* JavaFX CSS - Leave this comment until you have at least create one rule which uses -fx- ↩
    Property */

.root
{
        -fx-padding: 10;
        -fx-border-style: solid inside;
        -fx-border-width: 2;
        -fx-border-insets: 5;
        -fx-border-radius: 5;
        -fx-border-color: blue;
}
```

7.3.4 Changing the FXML File

If you open the generetad FXML File, you will see that only the root node is defined at the beginning:

VBoxSample.fxml

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

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

<VBox xmlns:fx="https://javafx.com/fxml/1" fx:controller="application.VBoxSampleController" ↩
    >
        <!-- TODO Add Nodes -->
</VBox>
```

Now you have to define the GUI in the FXML File. This can be done with the Editor in Eclipse. Another option represents the JavaFX Scene Builder. If you want learn more about this tool, you can read my JavaFX Scene Builder Tutorial.

After designing the GUI, the File contains the following FXML Code:

VBoxSample.fxml

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

<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextArea?>

<VBox xmlns:fx="https://javafx.com/fxml/1" fx:controller="application.VBoxSampleController" ↩
    >
  <children>
    <Label alignment="CENTER_LEFT" cache="true" cacheHint="SCALE" prefHeight="30.0"  ↩
        prefWidth="200.0" text="Please insert Your Input here:" textAlignment="LEFT" />
    <TextField fx:id="inputText" prefWidth="100.0" />
```

```
    <Button alignment="CENTER_RIGHT" contentDisplay="CENTER" mnemonicParsing="false"  ↩
        onAction="#printOutput" text="OK" textAlignment="CENTER" />
    <Label alignment="CENTER_LEFT" cache="true" cacheHint="SCALE" prefHeight="30.0"  ↩
        prefWidth="200.0" text="Your Input:" textAlignment="LEFT" />
    <TextArea fx:id="outputText" prefHeight="100.0" prefWidth="200.0" wrapText="true" />
  </children>
</VBox>
```

7.3.5 Changing the Controller Class

The created `Controller` class is also empty at the beginning:

VBoxSampleController.java

```
public class VBoxSampleController {

}
```

So, it is also necessary that you make the necessary changes in the `Controller` class. Otherwise, a Click on the OK `Button` has no effect, because the Method `printOutput()` is not defined.

VBoxSampleController.java

```
package application;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;

public class VBoxSampleController
{
        @FXML
        // The reference of inputText will be injected by the FXML loader
        private TextField inputText;

        // The reference of outputText will be injected by the FXML loader
        @FXML
        private TextArea outputText;

        // location and resources will be automatically injected by the FXML loader
        @FXML
        private URL location;

        @FXML
        private ResourceBundle resources;

        // Add a public no-args constructor
        public VBoxSampleController()
        {
        }

        @FXML
        private void initialize()
        {
        }

        @FXML
        private void printOutput()
```

```
        {
                outputText.setText(inputText.getText());
        }
}
```

7.3.6 The GUI

The following image shows the GUI of this example after inserting a text into the TextField and a click on the OK `Button`. The `printOutput()` Method will copy the text into the TextArea.

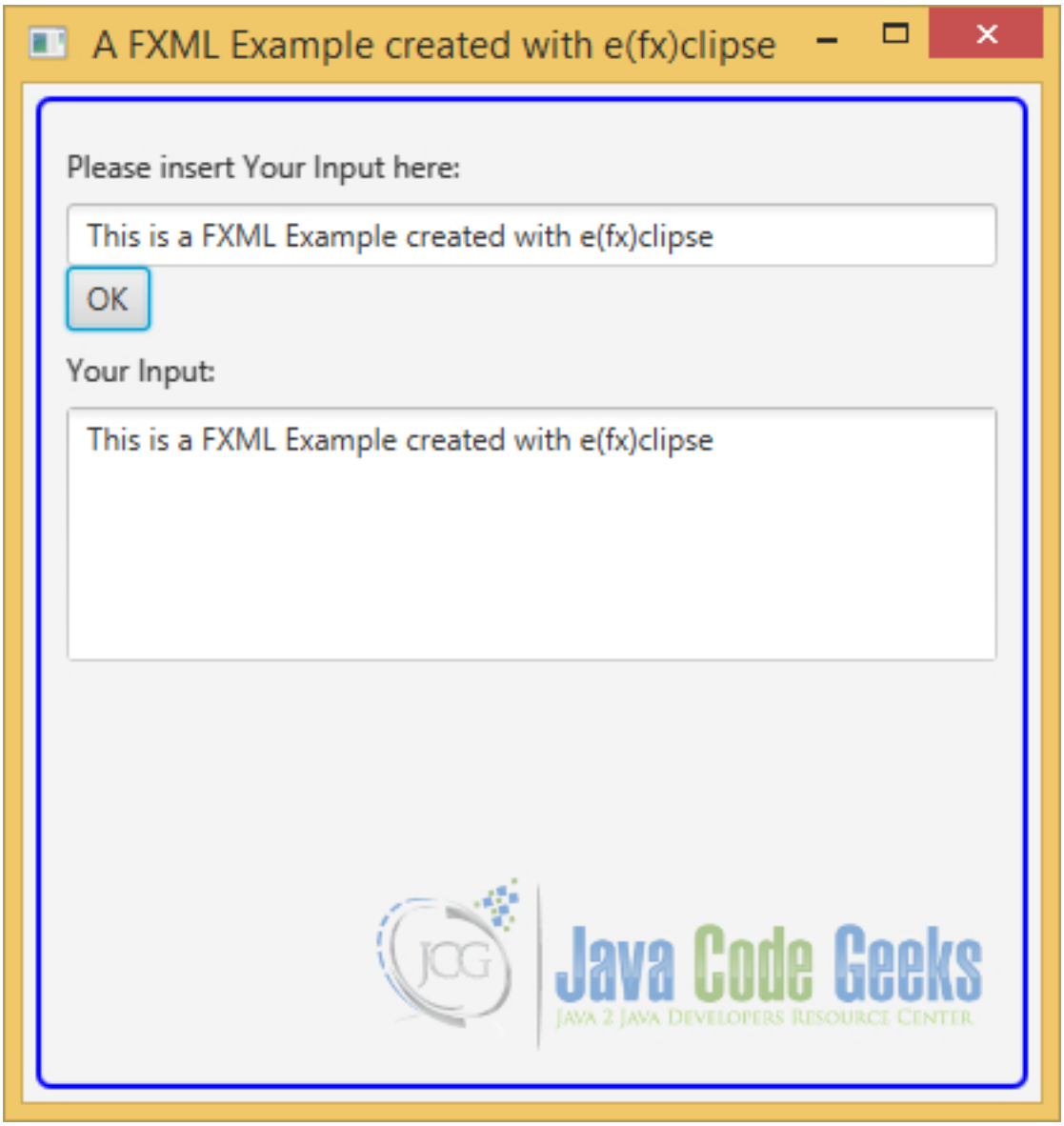

Figure 7.21: A JavaFX FXML Example created with the e(fx)clipse IDE

7.4 Download Java Source Code

This was an example of `e(fx)clipse`

Download

You can download the full source code of this example here: **JavaFXefxclipse.zip**