Good bye Swing – hello JavaFX

Christoph Rein und Stefan Kühnlein, OPITZ CONSULTING GmbH

Im Zeitalter von iOS und Android fordern Benutzer verstärkt optisch moderne und ansehnliche Anwendungen. Die Erwartungen gehen dabei von komplexen Farbverläufen, Transformationen (Drehungen und Zoom), animierten Übergängen bis hin zur flüssigen Bedienbarkeit von Applikationen per Touchscreen sowie Responsive Design. Solche Anforderungen können ältere Technologien nicht mehr erfüllen. Insbesondere bei der Entwicklung von Desktop-Anwendungen waren die Möglichkeiten mit Java zuletzt eher beschränkt. Das Abstract Window Toolkit (AWT) gilt bereits seit Längerem als veraltet und wird daher kaum noch verwendet. Alternativ wurden noch Swing und das Standard Widget Toolkit (SWT) eingesetzt, die mit ihrem Oberflächen-Design jedoch etwas verstaubt wirken.

Mit Einführung des Release 8 zur Entwicklung von Benutzeroberflächen soll JavaFX zum neuen Technologie-Standard erhoben werden. Da die Weiterentwicklung von Swing zur gleichen Zeit eingestellt wurde, löst die neue Framework-Version den Standard damit formal ab. Im Gegensatz zu Swing beherrscht JavaFX den Umgang mit modernen Features sehr gut. Damit ist diese Technologie aktuell die erste Wahl für die Entwicklung von neuen, modernen Desktop-Anwendungen. In vielen Fällen ist jedoch auch die Erweiterung bestehender Swing-Applikationen oder die Wiederverwendung von Swing-Komponenten in JavaFX eine bevorzugte Alternative. Mit einer Gegenüberstellung von Swing und JavaFX zeigt dieser Artikel die wichtigsten Vorteile des neuen Standards und Möglichkeiten zur Migration von Swing-Komponenten in JavaFX oder in die andere Richtung auf.

Grundgerüst und Aufbau

In Swing stellt der Frame das Standard-Fenster für eine GUI dar. Es wird durch die Klasse JFrame repräsentiert und erzeugt *(siehe Abbildung 1)*. Alle anderen Container und Komponenten sind darin aufgenommen. Den Einstiegspunkt eines Frames und der entsprechenden Container-Hierarchie bildet ein „RootPane", der in Form eines „JPanel" definiert wird und das Layout bestimmt. Wenn die Anwendung zusätzliche Layouts benötigt, müssen dafür weitere JPanels erstellt und dem Top-Level-Panel angefügt werden. Der RootPane enthält dann den sogenannten „LayeredPane", über den auf den „ContentPane" und die eigentlichen Elemente einer Swing-Anwendung zu-

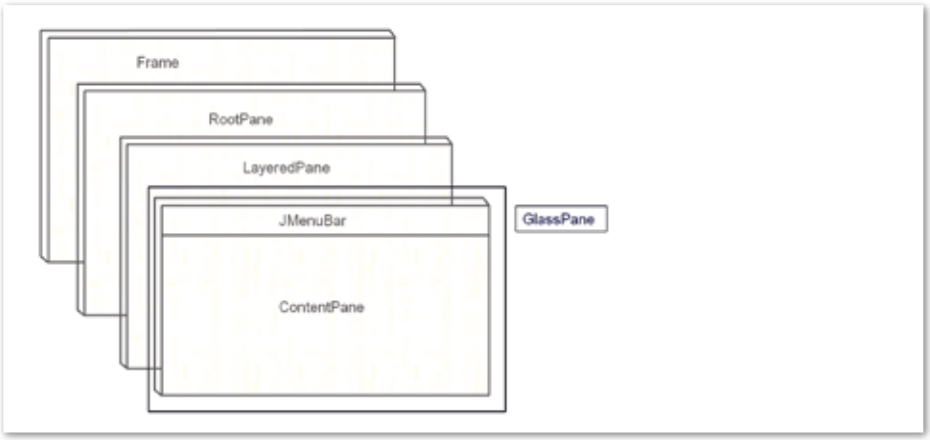

Abbildung 1: Swing Frame

gegriffen werden kann. Die oberste Schicht bildet der „GlassPane", mit dem eine unsichtbare Struktur über das GUI gelegt und so bestimmte Events abgefangen werden können. Der Zugriff erfolgt normalerweise nur auf den ContentPane und dessen Inhalt.

Im Gegensatz zu Swing ist die Grundstruktur einer JavaFX-Applikation an die eines Theaterstücks angelehnt. Als Rahmen steht eine „Stage" zur Verfügung, auf der die Anwendung gezeigt wird. Zu Beginn gibt es immer eine Stage in Form von Fenstern, aus der sich bei Bedarf weitere Stages öffnen lassen. Daneben existieren verschiedene „Scenes", die auf die Stage geladen werden können. Jede Scene enthält einen „Scene Graph" *(siehe Abbildung 2)*. Dieser bildet das Grundgerüst einer JavaFX-Anwendung und besteht aus genau einem Hauptknoten („Root Node") sowie mehreren Unterknoten („Leaf Node" oder „Branch Node"). Der Root Node kann einen oder mehrere Unterknoten enthalten, die entweder selbst Knoten aufnehmen („Branch Node") oder einen Endknoten darstellen können („Leaf Node").

Der „Scene Graph" ist also eine Sammlung aller Elemente, die die Benutzeroberfläche bilden. In JavaFX sind Layouts immer Unterklassen der Node-Klasse, sie enthalten also eine Reihe von Nodes. Die Art der enthaltenen Nodes ist dabei ebenfalls beliebig. Das Ergänzen von neuen Layouts gestaltet sich so um einiges flexibler als mit dem Hinzufügen von JPanels in Swing. *Listing 1* zeigt ein einfaches Beispiel einer Swing-Klasse, *Listing 2* dieselbe Klasse in JavaFX; die *Abbildung 3* und 4 stellen die entsprechenden Ergebnisse dar.

```
public class HelloWorldSwing
{
    public static void
main(String[] args)
    {
        JFrame frame = new
JFrame();
        frame.setTitle("Hello
Swing");
        JPanel panel = new JPan-
el();
        JButton button = new
JButton("Hello Swing");
        panel.add(button);
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Listing 1

Abbildung 3: „Hello World" mit JavaFX

Abbildung 4: „Hello World" mit Swing

Entwicklung von Oberflächen

Die GUI-Programmierung in Swing gestaltet sich gerade bei größeren Projekten sehr aufwändig und mühsam: Jede Komponente muss einzeln erzeugt und explizit angeordnet werden. Auf diese Weise entstehen sehr schnell komplexe, unübersichtliche und extrem schwer zu wartende Applikationen. Alternativen zu dieser Art der Oberflächen-Programmierung bieten deklarative Oberflächen. Verschiedene Open-Source-Lösungen wie SwiXml *[3]* bieten für diese Fälle entsprechende Ansätze an. Oracle hingegen hat keinen Standard entwickelt, der die deklarative Entwicklung von Benutzeroberflächen unterstützt.

Mit FXML bietet Java erstmals die Möglichkeit, grafische Komponenten schnell deklarativ und unabhängig von der Programmlogik zu definieren. Dabei handelt es sich um eine XML-basierte Definition zur Implementierung von deklarativen JavaFX-Oberflächen. Die Grundidee ist, dass eine FXML-Datei im Sinne des MVC-Patterns oder auch des MVVM-Patterns nur die UI beschreibt. Die Logik wird dann im Controller implementiert, der die entsprechenden UI-Elemente über die Annotation „@FXML" injiziert. Damit bietet Java eine Alternative zum klassischen API, in dem Knoten in Bäume angehängt werden.

Mit der XML-basierten Definition der Benutzeroberfläche ermöglicht FXML eine hierarchische Gliederung. Damit kann die Aufteilung einer GUI in Container und Kom-

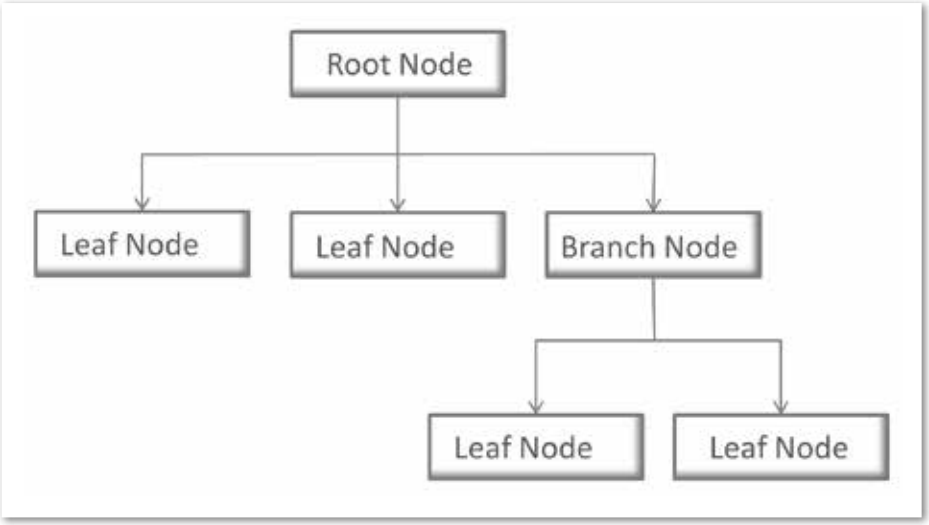

Abbildung 2: JavaFX-Scene-Graph

```
public class HelloWorldFx extends
Application {

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

   @Override
    public void start(Stage pri-
maryStage) throws IOException {
        primaryStage.
setTitle("JavaFX");

        Button button = new But-
ton();
        button.prefHeight(200);
        button.setText("Hello
Fx");
        StackPane root = new
StackPane();
        root.getChildren().
addAll(button);
        Scene scene = new
Scene(root, 300, 150);
        primaryStage.
setScene(scene);
        primaryStage.show();
    }
}
```

Listing 2

ponenten gut dargestellt werden. Für den Entwurf von Oberflächen existieren bereits Tools, mit denen die GUI-Container grafisch erstellt und im XML-Format abgespeichert sind. Das bekannteste Tool ist der „Scene Builder" von Gloun *[1]*. Für die Erstellung von GUI-Containern mit dem Scene Builder sind keine Java-Kenntnisse notwendig, sodass dieses Tool bereits während der Analyse-Phase eines Projekts zur Spezifikation von Desktop-Anwendungen verwendet werden kann. *Listing 3* zeigt eine einfache FXML-Datei zur Validierung von Benutzernamen und Passwort, die mithilfe des Scene Builder definiert wurde.

Ein Layout in JavaFX gestalten

Die individuelle Gestaltung von Oberflächen in Swing ist unter Umständen mit sehr viel Aufwand verbunden. Will man beispielsweise eine Hintergrund-Grafik einbinden, muss das JPanel dafür erst erweitert werden. Für die Definition einer Schriftart ist es notwendig, erst ein Font-Objekt anzulegen, das erst nach Beenden der Anwendung oder des JFrame aus dem Speicher gelöscht wird. Selbst wenn diese Hürden überwunden sind, ist man mit den Möglichkeiten von Swing recht eingeschränkt. Komplexe Farbverläufe oder einfache Effekte wie Schatten oder Reflektionen können entweder nur mit viel Aufwand oder gar nicht dargestellt werden. Anderen modernen Oberflächen-Frameworks hinkt Swing daher auch mit den neuesten Versionen hinterher.

Mit der Möglichkeit zur Einbindung von Cascading Style Sheets (CSS) bietet die JavaFX-Bibliothek ein weiteres sehr nützliches Feature. Das Aussehen der JavaFX-Anwendung kann damit flexibel und dynamisch verändert werden. Das Design kann entweder direkt im Code oder vollkommen unabhängig von der Logik in einer eigenen CSS-Datei gekapselt werden. Praktisch jedes Element einer Oberfläche ist mit CSS individuell definierbar; Styles für Elemente wie zum Beispiel eine Schriftart können jedoch auch einfach global definiert werden und sind so allgemein gültig. Ein unternehmensspezifisches Corporate Design kann damit beispielsweise einmal definiert und einfach in die jeweilige Scene integriert werden. Der Nutzer kann selbst entscheiden, welche CSS-Datei er der Scene hinzufügt, es ist aber ebenfalls möglich, eine Datei global für alle Scenes einzubinden.

```
<BorderPane xmlns="http://javafx.com/javafx/8.0.40" xmlns:fx="http://
javafx.com/fxml/1" fx:controller="application.ListDemoController">
   <center>
      <GridPane BorderPane.alignment="CENTER">
         <children>
            <Label text="Username:"></Label>
            <Label text="Password:" GridPane.rowIndex="1"></Label>
            <TextField GridPane.columnIndex="1"></TextField>
            <TextField GridPane.columnIndex="1" GridPane.rowIndex="1"></
TextField>
         </children>
      </GridPane>
   </center>
</BorderPane>
```

Listing 3

```
public class HelloSwingLayout extends JFrame {

  public HelloSwingLayout() {
     this.setTitle("Swing Layout");
     JPanel bgPanel = new JPanel() {

        public void paintComponent(Graphics g) {
           super.paintComponent(g);
           Image bgImage = null;

           try {
bgImage = ImageIO.read(new File("images/background.jpg"));
           } catch (IOException e) {
              e.printStackTrace();
           }
              g.drawImage(bgImage, 50, 50, this);
        }
     };
     bgPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
     Font font = new Font("Serif", Font.PLAIN, 24);
     JButton btn = new JButton("Hello Swing");
     btn.setFont(font);
     bgPanel.add(btn);
     this.add(bgPanel);
     this.setBounds(0, 0, 300, 150);
     this.setVisible(true);
  }
}
```

Listing 4

```
.root {
    -fx-font-size: 14pt;
    -fx-font-family: "Tahoma";
    -fx-background-image: url("background.jpg");
}

.button {
    -fx-text-fill: #0000ff;
    -fx-border-radius: 20;
    -fx-padding: 5;

}
```

Listing 5

Listing 4 zeigt eine einfache CSS-Datei, die ein Hintergrundbild einbindet sowie das Style einer Komponente anpasst. Mit „scene.getStylesheets().add ()" kann die Datei der jeweiligen Scene zugefügt werden. Um in Swing dasselbe Ergebnis zu bekommen, ist dagegen etwas mehr Aufwand nötig. *Listing 5* zeigt dazu, wie eine Grafik mithilfe von „paintComponent ()" und eines Image-Objekts eingebunden werden kann. Im Vergleich zu JavaFX ist dafür wesentlich mehr Code nötig. Für die Änderung der Schriftart

des Buttons muss, wie bereits erwähnt, erst ein Objekt erzeugt werden.

Properties und Bindings

Die Methoden und Konzepte in Swing sind für eine statische Kontrolle und Modifikation der Oberfläche ausgelegt. Moderne Benutzeranforderungen gehen jedoch dahin, Daten unmittelbar bei der Eingabe zu validieren und Änderungen sofort zu registrieren. In JavaFX wurden dafür die Konzepte der Properties und Bindings entwickelt und mit einigen nützlichen Features ausgestattet. JavaFX-Properties basieren dabei auf dem bekannten JavaBean Model, durch das bisher Properties eines Objekts repräsentiert wurden. Mit diesem Model wurden Namenskonventionen eingeführt, die für die Konsistenz über verschiedene Projekte hinweg sorgen.

Das JavaFX-API bietet für bestimmte Klassen bereits implementierte Properties. Es bietet eine Reihe von Property-Basisklassen für primitive Datentypen.

Bestimmte Klassen wie „javafx.scene.shape.Rectangle" verfügen bereits über Properties für „arcHeight", „arcWidth", „height" oder „width". Jede Property besitzt dabei standardmäßig eine „get()"/„set()"-Methode. Um über die Änderung einer Property informiert zu werden, kann zusätzlich ein „ChangeListener()" implementiert sein, der dazu sowohl den alten als auch den neuen Wert der Property ausgeben kann. Der Listener funktioniert dabei ähnlich wie der „PropertyChange Listener()" aus dem JavaBean Model.

Mit Bindings bringt JavaFX ein weiteres enorm mächtiges und einfaches Konzept mit, mit dem verschiedene Properties aneinander gebunden werden können. Eine „Property a" kann damit an eine „Property b" gebunden werden, sodass bei der Änderung von „a" die Property „b" mit dem neuen Wert von „a" aktualisiert wird. Werte können dabei mit „bind()" und „bindBidirectional()" sowohl in eine als auch in beide Richtungen gebunden werden. „unbind()"entfernt ein Binding zudem ganz einfach aus dem Code. *Listing 6* zeigt ein einfaches bidirektionales Binding zweier Slider. Dabei werden die Eigenschaften „valueProperty()" der beiden Slider aneinander gebunden, sodass bei Veränderung eines Werts der jeweils andere Wert ebenfalls verändert wird.

Neben solchen einfacheren Beispielen kann es auch vorkommen, dass das Standard-API nicht ausreichend ist. Für diesen Fall steht dem Entwickler das sogenannte „Low-Level-Binding" zur Verfügung. Damit wird eine Klasse erweitert und die Methode „computeValue()" überschrieben, um so den aktuellen Wert des Binding zurückzugeben. *Listing 7* zeigt dazu ein Beispiel mit einer angepassten Subklasse von „DoubleBinding()". Durch „super.bind()" werden die Abhängigkeiten an „DoubleBinding" übergeben. *[2]*

Migration von Swing- und JavaFX-Komponenten

Mit Einführung und wachsender Beliebtheit von JavaFX stellt sich die Frage, ob man bereits bestehende Swing-Applikationen mit JavaFX kombiniert. Dabei kann sowohl die Integration alter Swing-Anwendungen in ein neues JavaFX-Projekt gewünscht sein als auch eine Erweiterung von Swing durch neue Komponenten.

Die Gründe für eine Kombination aus beiden Technologien können unterschiedlich sein. Manchmal decken bestehende Swing-Anwendungen neuere Features nicht ab, in anderen Fällen kann es sinnvoll sein, komplexe Swing-Komponenten in eine neue JavaFX-Anwendung zu übertragen, statt diese neu zu implementieren. Auf diese Weise sinkt der Entwicklungsaufwand enorm.

Um Swing-Komponenten auch weiterhin nutzen zu können oder um alte Anwendungen noch zu erweitern, wurden zwei Erweiterungen eingeführt, die einerseits ermöglichen, JavaFX in Swing zu migrieren, und andererseits erlauben, Swing-Komponenten in JavaFX einzubetten.

JFXPanel

Für die Einbettung von JavaFX-Komponenten in Swing-Anwendungen wurde das JFXPanel entwickelt. Es verhält sich dabei grundsätzlich wie ein JPanel, kann aber eine JavaFX-Scene darstellen. Als Teil der Swing-Hierarchie kann das JFXPanel jedem JFrame hinzugefügt werden. Statt der Stage mit „Stage.setScene()" ein Scene-Objekt anzuhängen, lässt sich dem Panel mit „setScene()" eine JavaFX-Komponente hinzufügen; die Methode akzeptiert eine Instanz von JavaFX-Scene. Mit „getScene()" erhält man die Scene, die in dem JFXPanel enthalten ist. So können alle JavaFX-Komponenten in Swing dargestellt werden, inklusive ihrer Features.

Listing 8 zeigt ein einfaches Beispiel zur Einbindung von JavaFX-Code. Das entsprechende Ergebnis ist in der *Abbildung 5* dargestellt. Im oberen Teil wird dabei das JFXPanel

```
Slider slider1 = SliderBuilder.create().layoutX(50).layoutY(50).build();
Slider slider2 = SliderBuilder.create().layoutX(50).layoutY(100).build();

slider1.valueProperty().bindBidirectional(slider2.valueProperty());
```

Listing 6

```
public class Main {

    public static void main(String[] args) {

        final DoubleProperty a = new SimpleDoubleProperty(1);
        final DoubleProperty b = new SimpleDoubleProperty(2);
        final DoubleProperty c = new SimpleDoubleProperty(3);
        final DoubleProperty d = new SimpleDoubleProperty(4);

        DoubleBinding db = new DoubleBinding() {
             {
                super.bind(a, b, c, d);
            }
             @Override
            protected double computeValue() {
                return (a.get() * b.get()) + (c.get() * d.get());
            }
        };
        System.out.println(db.get());
        b.set(3);
        System.out.println(db.get());
    }
}
```

Listing 7

Abbildung 5: Swing-Fenster mit einem JavaFX-Control

erzeugt und dem JFrame hinzugefügt. In der Methode „createScene()" können sämtliche JavaFX-Komponenten wie gewohnt eingebunden werden. So ist auch die Nutzung von Bindings innerhalb eines JFrame ohne Weiteres möglich. Auch können die Komponenten mithilfe von CSS verändert sowie Bilder oder Media-Content eingebunden werden. Dafür wird in „run()" mit „getScene()" auf die aktuelle Scene zugegriffen und so eine CSS-Datei genutzt. Natürlich lässt sich auf diese Weise auch eine FXML-Datei einbinden, um so deklarative Oberflächen und damit verbunden auch Tools wie Scene Builder zu nutzen.

Ein „InputEvent" wird automatisch von Swing an JavaFX weitergegeben. Bei einer Änderung von JavaFX-Content übernimmt Swing diese ebenfalls. Beachten muss man dabei, dass beide Technologien auf unabhängigen Threads arbeiten. JavaFX arbeitet mit dem „FXApplicationThread", Swing mit dem „Event Dispatch Thread" (EDT).

Zwar beinhaltet die JavaFX-Bibliothek verschiedene Threads, der Application-Thread sticht jedoch dadurch hervor, dass auf ihm alle Events verarbeitet, Animationen ausgeführt sowie alle Knoten erstellt und verändert werden. Um bei der Integration eine Runtime-Exception zu vermeiden, müssen die beiden Threads synchronisiert werden.

Abbildung 6: JavaFX-Fenster mit einem Swing-Control

```
public class JFXPanelExample {
  private static void initAndShowGUI() {

      JFrame frame = new JFrame("JavaFX in Swing");
      final JFXPanel fxPanel = new JFXPanel();
      frame.add(fxPanel);
      frame.setSize(300, 200);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      Platform.runLater(new Runnable() {
         @Override
         public void run() {
            initFX(fxPanel);
             fxPanel.getScene().getStylesheets().add(getClass()
.getResource("application.css").toExternalForm());
         }
      });
  }

  private static void initFX(JFXPanel fxPanel) {
      // This method is invoked on the JavaFX thread
      Scene scene = createScene();

      fxPanel.setScene(scene);
  }
  private static Scene createScene() {
      Group root = new Group();
      Scene scene = new Scene(root, Color.ALICEBLUE);

      Button btn = new Button();
      btn.setText("JavaFX in Swing");
      btn.setLayoutX(25);
      btn.setLayoutY(10);

Slider slider1 = SliderBuilder.create().layoutX(25).layoutY(75).build();
Slider slider2 = SliderBuilder.create().layoutX(25).layoutY(125).build();

          slider1.valueProperty().bindBidirectional(slider2.valueProperty());

          root.getChildren().addAll(btn, slider1,e slider2);

      return (scene);
  }

  public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            initAndShowGUI();
         }
      });
  }
}
```

Listing 8

Durch die Erstellung einer JFXPanel-Instanz wird implizit die JavaFX-Runtime gestartet. Die Methode „initFX()" erstellt eine JavaFX-Scene auf dem JavaFX-Thread. Das Konstrukt „Platform.runLater (){ }" – eingebettet in die „initFX"-Methode – sorgt dafür, dass die Methode auf dem FX-Thread aufgerufen wird und nicht auf dem Standard-EDT-Thread.

SwingNode

Der vorherige Abschnitt hat gezeigt, wie JavaFX in eine bestehende Swing-Anwendung integriert werden kann. Die umgekehrte Integration ist ebenfalls möglich. Um Swing-Komponenten in JavaFX zu integrieren, gibt es die Klasse „SwingNode" mit den Methoden „setContent()" und „getContent()". Diese fügen dem SwingNode JComponents hinzu. Beide Methoden können dabei sowohl vom EDT als auch vom FX-Thread aufgerufen werden. Im Gegensatz dazu muss beim Zugriff auf eine Swing-Komponente auf die Synchronisierung der Threads geachtet werden. In diesem Fall muss das Setzen auf dem Swing EDT erfolgen, standardmäßig erfolgt der Aufruf in einer JavaFX-Anwendung auf dem FXApplication-Thread. *Listing 9*

```
public class SwingNodeExample extends Application {

         @Override
         public void start(Stage stage) {

          final SwingNode swingNode = new SwingNode();
              createAndSetSwingContent(swingNode);
              StackPane pane = new StackPane();
              pane.getChildren().add(swingNode);
              stage.setScene(new Scene(pane, 100, 50));
              stage.show();
         }

         private void createAndSetSwingContent(final SwingNode swingNode) {
              SwingUtilities.invokeLater(new Runnable() {
                  @Override
                  public void run() {

                    JButton button = new JButton("Hello JavaFX");
                    swingNode.setContent(button);

                  }
              });
         }

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

Listing 9

zeigt beispielhaft, wie ein „swing.Jbutton" in eine JavaFX-Anwendung integriert werden kann. Das zugehörige Ergebnis ist in der *Abbildung 6* dargestellt. Dabei wird eine SwingNode-Instanz erstellt, der dann ein Swing JButton zugefügt wird. Der Aufruf „SwingUtilities.invokeLater()" startet den EDT und erstellt einen neuen JButton. Um auf JavaFX-Elemente zugreifen zu können, muss wiederum „Platform.runLater(Runnable)" gestartet werden.

Eine Komponente wie der JButton oder ein komplexer JTable kann so in JavaFX weiterverwendet werden. Die ursprünglichen Eigenschaften eines Swing-Elements bleiben dabei erhalten. Auch das standardmäßige Swing-Design wird mit übernommen, wodurch optisch eine deutliche Abgrenzung zur moderneren JavaFX-Oberfläche entsteht

Fazit

Mit JavaFX steht Java-Entwicklern eine moderne Technik zur Verfügung, um grafisch ansprechende Applikationen zu erstellen. Eine Vielzahl weiterer Komponenten und Konzepte wie JavaFX-MediaSupport oder die Möglichkeit, HTML-Content in Java-Applikationen zu rendern, lassen es zu, moderne und stabile Anwendungen zu entwickeln. Durch deklarative Oberflächen können UIs einfach erstellt, verändert oder angepasst werden. Mit der Einbindung von CSS wird das Aussehen einer grafischen Darstellung zudem noch weiter individualisiert und optimiert. Mithilfe von Bindings lassen sich außerdem Werte auf einfache Weise voneinander abhängig machen.

Die Verbindung zu Swing schafft weiteren Spielraum im Hinblick auf bereits bestehende Anwendungen. Mit dem Einsatz von JavaFX in bestehende Swing-Applikationen ergeben sich Vorteile, wie die einfache und schnelle Umsetzung neuer Features und die zeitnahe Nutzbarkeit der Komponenten durch den Anwender. Jede Komponente kann dabei Schritt für Schritt ausgetauscht werden. Die Weiterverwendung bestehender komplexer Swing-Komponenten und der dadurch vermiedene Nachbau in JavaFX können den Entwicklungsaufwand senken.

Referenzen

[1] http://gluonhq.com/open-source/scene-builder
[2] https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm
[3] http://www.swixml.org

Christoph Rein
christoph.rein@opitz-consulting.com

Christoph Rein hat an der Universität Regensburg Wirtschaftsinformatik studiert und ist seit dem Jahr 2015 als Consultant für die OPITZ CONSULTING GmbH tätig. Dort beschäftigt er sich im Java-Umfeld speziell mit der Modernisierung von Benutzeroberflächen in JavaFX, unter anderem für einen großen Kunden aus Deutschland.

Stefan Kühnlein
stefan.kühnlein@opitz-consulting.com

Stefan Kühnlein ist seit Oktober 2013 als Solution Architect für die OPITZ CONSULTING GmbH tätig. Sein Schwerpunkt liegt im Entwurf von SOA-basierten Architekturen sowie in der Auswahl moderner Software-Architekturen und entsprechender technischer Frameworks. Aktuell unterstützt er gerade ein größeres Modernisierungsprojekt, dessen Frontend mit JavaFX entwickelt wird.