5 Einstieg JavaFX 8

In diesem Kapitel stelle ich Ihnen mit JavaFX das aktuellste und modernste GUI-Framework von Java vor, das Swing als Oberflächentechnologie ablösen soll. Dieser Schritt ist notwendig, weil Java in den letzten Jahren im Desktop-Bereich deutlich an Einfluss an die aufstrebende Konkurrenz aus dem Microsoft-Lager (.NET/WPF) aber auch an immer populärer werdende Webapplikationen verloren hat. JavaFX tritt nun an, die GUI-Programmierung zu erleichtern und attraktive GUIs entwickeln zu können.

Weil derzeit die Literatur zu JavaFX doch noch recht überschaubar ist, ganz besonders bei deutschsprachigen Titeln, beschreibe ich Ihnen zunächst die Grundlagen von JavaFX in Version 2.2. Das ist die Version, in der JavaFX im JDK 7 ausgeliefert wird. Abschnitt 5.1 beginnt mit einer Einführung in JavaFX. Danach schauen wir uns in Abschnitt 5.2 die deklarative Gestaltung von GUIs an, wodurch eine gute Trennung von Design und Funktionalität möglich wird. In Abschnitt 5.3 lernen wir die Vorzüge von JavaFX in Form von Effekten und Animation u. v. m. kennen. Anhand meiner Ausführungen erhalten Sie einen fundierten Überblick und eine gute Basis sowohl für eigene Experimente als auch zum Nachvollziehen der Beschreibungen zu den Neuerungen von JavaFX 8 in Abschnitt 5.4.

Weil ich nicht jedes Detail beschreiben kann, gehe ich davon aus, dass Sie sich schon etwas intensiver mit GUIs und Swing beschäftigt haben, um dem Text inhaltlich gut folgen zu können.[1]

5.1 Einführung – JavaFX im Überblick

In diesem Abschnitt gebe ich Ihnen einen kurzen Überblick und Einstieg in JavaFX. Wir lernen zunächst einige Grundbegriffe und dann eine einfache Form von Action Handling kennen. Anschließend schauen wir uns das Layoutmanagement an.

5.1.1 Motivation für JavaFX und Historisches

Neben .NET-Anwendungen bieten mittlerweile auch viele Webanwendungen häufig moderne, ansprechende GUIs mit hohem Bedienkomfort. Diesbezüglich wurden Webanwendungen früher aufgrund ihrer rudimentären Interaktivität und insbesondere wegen des fehlenden Komforts von Desktop-Entwicklern oftmals belächelt. Mittlerweile

[1] Literaturempfehlungen finden Sie in Abschnitt 7.3.

hat sich die Situation aber geändert: Webanwendungen haben beträchtliche Fortschritte gemacht und einige klassische Desktop-Applikationen vom Bedienkomfort her mitunter sogar überholt. Erschwerend kommt hinzu, dass in Webanwendungen grafische Effekte und Animationen bereits zum guten Ton gehören, wodurch bei vielen Entwicklern und vor allem Nutzern der Wunsch entsteht, Derartiges auch zur Verbesserung der Benutzbarkeit in Desktop-Anwendungen einsetzen zu können.

Aus dem Gesagten wird klar, dass sich eine ernsthafte Konkurrenz zu Java-Desktop-GUIs entwickelt hat. Der schwindende Einfluss von Java im GUI-Bereich war wohl ein wichtiger Grund, weshalb JavaFX entwickelt wurde. Es wurde zunächst als Skriptsprache entworfen, die die grafische Gestaltung ansprechender GUIs z. B. durch Effekte sowie Animationen unterstützt. Die Programmierung mithilfe von Skriptcode stellte jedoch für viele Swing-Entwickler eine gewisse Hürde dar, weil zuerst die Skriptsprache erlernt werden musste und außerdem keine gute Integration in das Java-API stattfand. In Form der Skriptsprache hat sich JavaFX nie wirklich durchgesetzt. Nach diesem erfolglosen Versuch wurde es zunächst ruhig um JavaFX, bis dann Ende 2011 auf der Java One, einer bedeutenden Java-Konferenz in San Francisco, die Version 2 von JavaFX vorgestellt wurde. Mit JavaFX 2 findet eine Abkehr von der Skriptsprache statt und es wird eine Integration in das Java-API vorgenommen. Nachdem JavaFX längere Zeit eher ein Schattendasein geführt hat, wird es nun von Oracle stark gefördert sowie aktiv weiterentwickelt. Aktuell ist JavaFX in Version 8. Diese enthält diverse Neuerungen und ist Bestandteil von JDK 8.

5.1.2 Grundsätzliche Konzepte

In diesem Abschnitt lernen wir wichtige Grundlagen von JavaFX anhand einer Hello-World-Applikation kennen. Dieses Programm gibt einen Text in einem Fenster aus. Bevor wir jedoch mit der Implementierung der Anwendung beginnen, möchte ich auf folgende zentrale Hauptbestandteile einer JavaFX-Applikation eingehen:

- **Stage** — Die sogenannte Stage vom Typ `javafx.Stage` bildet die »Bühne« oder den Rahmen für eine JavaFX-Applikation und stellt die Verbindung zum genutzten Betriebssystem dar – vergleichbar mit einem `JFrame` in Swing.
- **Scene** — Eine sogenannte Scene ist vom Typ `javafx.scene.Scene`. Sie entspricht grob der Containerkomponente `ContentPane` in Swing, und ist dasjenige Element eines JavaFX-GUIs, in dem alle Bedienelemente platziert werden (eventuell indirekt durch Verschachtelungen ähnlich zu den Containern in Swing).
- **Scenegraph** und **Nodes** — Ähnlich wie bei AWT, SWT oder Swing ist auch bei JavaFX die Benutzeroberfläche hierarchisch organisiert: Der Inhalt einer `Scene` ist ein Baum, bestehend aus Knoten mit dem Basistyp `javafx.scene.Node`. Man spricht bei dem Baum auch vom ***Scenegraph***. Die Anordnung der Bedienelemente (`Node`s) wird durch spezielle Container ähnlich zu den `LayoutManager`n in Swing bestimmt. Diese Container besitzen selbst wieder den Basistyp `Node` und sind Bestandteil des Scenegraphs.

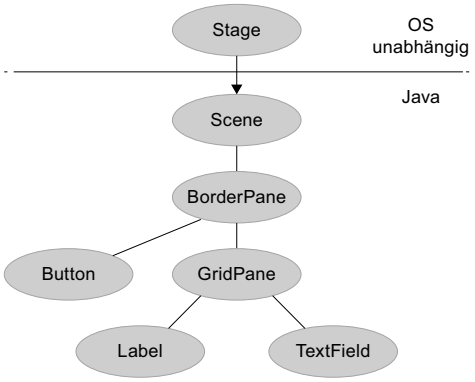

Abbildung 5-1 *Verbindung von Stage, Scene und Node*

Um einen ersten Eindruck davon zu gewinnen, wie man JavaFX-Applikationen erstellt, realisieren wir nun die Hello-World-Applikation. Praktischerweise gibt es im JDK die Basisklasse `javafx.application.Application`, die diverse Funktionalitäten bereitstellt, sodass lediglich noch die abstrakte Methode `start(Stage)` implementiert werden muss. Dort wird das GUI konstruiert. Als Container nutzen wir den Typ `javafx.scene.layout.StackPane`, in der die Bedienelemente wie in einem Kartenstapel übereinander angeordnet werden. Zur Darstellung eines Textes fügen wir dort ein Label vom Typ `javafx.scene.control.Label` folgendermaßen ein:

```
import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class FirstJavaFxExample extends Application
{
    @Override
    public void start(final Stage stage) throws Exception
    {
        final StackPane stackPane = new StackPane();

        // Label erzeugen und hinzufügen
        final Node labelNode = new Label("Hello JavaFX World!");
        stackPane.getChildren().add(labelNode);

        // Stage, Scene und Stackpane verbinden
        stage.setScene(new Scene(stackPane, 250, 75));
        // Titel und Resizable-Eigenschaft setzen
        stage.setTitle("FirstJavaFxExample");
        stage.setResizable(true);
        // Positionierung und Sichtbarkeit
        stage.centerOnScreen();
        stage.show();
    }
    // ...
}
```

Weil wir im Listing diverse unbekannte Klassen sehen und die Aktionen zum Aufbau des GUIs in JavaFX-Applikationen in etwa immer einem ähnlichen Muster folgen, werden hier einmalig die `import`-Anweisungen gezeigt und zudem einführend die obigen Programmzeilen detaillierter beschrieben. Das `Label` ist eine Spezialisierung einer `Node`. Eine Instanz davon wird der `StackPane` hinzugefügt, indem über `getChildren()` die `Node`s im Container ermittelt und dann über `add(Node)` erweitert werden. Anschließend dient die `StackPane` als Eingabe für die Konstruktion einer `Scene`, die dann der `Stage` per Methodenaufruf von `setScene(Scene)` zugewiesen wird. Für die `Stage` werden noch verschiedene Eigenschaften wie Titel und Größenveränderlichkeit gesetzt. Abschließend wird durch Aufrufe von `centerOnScreen()` sowie `show()` ein Fenster mit dem zuvor konstruierten Inhalt zentriert auf dem Bildschirm dargestellt. Führen wir die Applikation FIRSTJAVAFXEXAMPLE aus, so erscheint ein Fenster mit einem Text, etwa wie in Abbildung 5-2.

Abbildung 5-2 *Erste JavaFX-Applikation*

Zwei Dinge möchte ich noch explizit erwähnen:

1. Im Gegensatz zu Swing-Applikationen muss man als Anwender das Fenster bzw. die Stage nicht selbst erstellen, etwa per Konstruktoraufruf `new JFrame()`, sondern die `Stage` wird vom JavaFX automatisch erzeugt und an die Methode `start(Stage)` übergeben. Dadurch erreicht man eine gute Plattformunabängigkeit, weil der Inhalt der `Scene` und alle dort dargestellten Elemente den Basistyp `Node` besitzen und so eine Abstraktion von den tatsächlichen Bedienelementen des Betriebssystems ermöglichen.
2. Wiederum im Gegensatz zu Swing nimmt JavaFX viele Schritte beim Starten der Applikation ab und sorgt auf diese Weise für einen Thread-sicheren Start. Dabei wird die Methode `start(Stage)` automatisch aufgerufen, wenn wir die Methode `launch()` ausführen. Dies geschieht in der `main()`-Methode wie folgt:

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

Erweiterung des Hello-World-Beispiels um Action Handling

Das vorherige Beispiel hat einen einführenden Überblick über die grundlegenden Zusammenhänge beim Entwurf einer JavaFX-Applikation gegeben. Nun wollen wir

Benutzerinteraktionen unterstützen und einen ersten Eindruck von der Verarbeitung bekommen. Dazu ändern wir das Beispiel folgendermaßen leicht ab: Statt eines `Label`s nutzen wir einen `javafx.scene.control.Button`, den wir in einer `javafx.scene.layout.FlowPane` platzieren. Zur Interaktion registrieren wir einen `javafx.event.EventHandler<javafx.event.ActionEvent>`. Als Reaktion auf das Drücken des Buttons erzeugen wir neue `Label`s, die wir der `FlowPane` dynamisch hinzufügen. Das Beschriebene realisieren wir wie folgt:

```
@Override
public void start(final Stage stage)
{
    final Button button = new Button();
    button.setText("Add 'Hello World' Label");

    final FlowPane pane = new FlowPane();
    pane.getChildren().add(button);

    // EventHandler registrieren
    button.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(final ActionEvent event)
        {
            pane.getChildren().add(new Label("- Hello World! -"));
        }
    });

    stage.setScene(new Scene(pane, 400, 100));
    stage.setTitle("JavaFxActionHandlingExample");
    stage.setResizable(true);
    stage.centerOnScreen();
    stage.show();
}
```

An diesem Beispiel wird neben dem Action Handling ein dynamisch veränderliches Layout gezeigt. Das Programm JAVAFXACTIONHANDLINGEXAMPLE produziert nach ein paar Klicks auf den Button eine Ausgabe ähnlich zu der in Abbildung 5-3.

Abbildung 5-3 *JavaFX und Action Handling*

Relativ schnell wird diese Applikation langweilig und wir drücken das Schließkreuz. Ganz natürlich wird die Applikation dadurch beendet. Tatsächlich könnte uns das als praktisches Feature auffallen, denn in Swing wurde die Applikation nur dann beendet, wenn man eine passende Default-On-Close-Operation oder einen speziellen `java.awt.event.WindowListener` registriert hatte.

Wenn Sie bereits Erfahrung mit Swing haben, ist Ihnen bestimmt positiv aufgefallen, dass JavaFX das Layout der Applikation automatisch aktualisiert und dies nicht wie bei Swing einige Aktionen erfordert. Das führt uns zum Thema Layoutmanagement.

5.1.3 Layoutmanagement

In den bisherigen Beispielen haben wir bereits zwei verschiedene Layout-Containerkomponenten von JavaFX kennengelernt, die folgende Ausrichtungen bereitstellen:

- `StackPane` – Die `StackPane` platziert die einzelnen `Node`s wie in einem Stapel Karten übereinander. Damit ist es auf einfache Weise möglich, `Node`s miteinander zu kombinieren. Sofern die einzelnen `Node`s unterschiedlich groß oder teilweise transparent sind, kann man einen Überlagerungseffekt erzielen.
- `FlowPane` – Die `FlowPane` erinnert an das `FlowLayout` aus Swing. Hier werden `Node`s sukzessive dargestellt und je nach Verlaufsrichtung horizontal oder vertikal umbrochen. Dies geschieht auf Grundlage der Breite bzw. Höhe der `FlowPane`.

In JavaFX ist eine Vielzahl weiterer Layouts definiert, unter anderem folgende:

- `BorderPane` – Die `javafx.scene.layout.BorderPane` ist sehr ähnlich zum `java.awt.BorderLayout` in Swing. Analog dazu bietet auch die `BorderPane` die bekannten fünf Bereiche, in denen `Node`s platziert werden können (oder die auch unbelegt bleiben können). Diese Anordnung eignet sich für gebräuchliche Layouts mit einer Toolbar oder einer Menüzeile oben sowie einer Statuszeile unten. Auf der linken Seite kann eine Navigation dargestellt werden sowie zusätzliche Informationen auf der rechten Seite. Die Hauptinformation ist mittig platziert.
- `GridPane` – Mithilfe der `javafx.scene.layout.GridPane` lassen sich Anordnungen an einem Raster realisieren, ähnlich wie mit dem aus Swing bekannten `java.awt.GridLayout` oder dem komplexeren `java.awt.GridBagLayout` respektive dem `FormLayout` aus der GUI-Bibliothek JGoodies (verfügbar unter `http://www.jgoodies.com/`). In einer `GridPane` können `Node`s beliebigen Zellen im Raster zugeordnet werden und sich auch über den Bereich mehrerer Zellen erstrecken. Mithilfe von `GridPane`s lassen sich sehr gut Eingabemasken realisieren, die eine in Spalten und Zeilen ausgerichtete Darstellung von Bedienelementen erfordern.
- `HBox` und `VBox` – Die `javafx.scene.layout.HBox` ist ein einfaches Layout: Die `Node`s werden horizontal in einer Zeile ausgerichtet. Für die `javafx.scene.layout.VBox` erfolgt die Ausrichtung in der Vertikalen, ähnlich einer Spalte.

Layouts am Beispiel

Im folgenden Beispiel verwende ich exemplarisch die zuletzt genannten und zuvor noch unbenutzten Layouts, um eine recht typische Oberfläche zu gestalten, die eine Toolbar oben, eine Navigationsleiste links und verschiedene Textfelder zentral anbietet.

Bereits bei nur etwas komplexeren Programmen lohnt es sich, dem Design und der Strukturierung im Vorfeld einige Gedanken zu widmen. Würde man stattdessen direkt loslegen und das GUI vollständig innerhalb der `start(Stage)`-Methode aufbauen, so wäre diese selbst für das Beispiel schon recht lang und etwas unübersichtlich. Das gilt in zunehmendem Maße, wenn das zu konstruierende GUI komplexer wird. Dann bietet sich oftmals an, die einzelnen Bestandteile des GUIs mithilfe von eigenen Methoden zu konstruieren, wie dies nachfolgend durch die drei `createXYZ()`-Methoden und die entsprechende Platzierung der erzeugten Container in der `BorderPane` gezeigt ist:

```
@Override
public void start(final Stage primaryStage)
{
    final BorderPane borderPane = new BorderPane();
    borderPane.setTop(createToolbarPane());
    borderPane.setCenter(createInputPane());
    borderPane.setLeft(createNavigationPane());

    primaryStage.setTitle(LayoutCombinationExample.class.getSimpleName());
    primaryStage.setScene(new Scene(borderPane, 350, 200));
    primaryStage.show();
}

private Pane createToolbarPane()
{
    final HBox hbox = new HBox(5);
    hbox.getChildren().addAll(new Text("TOP"), new Button("HBox1"),
                                               new Button("HBox2"));
    return hbox;
}

private Pane createInputPane()
{
    final GridPane gridPane = new GridPane();
    final Label label1 = new Label("Label1");
    final TextField textfield1 = new TextField();
    final Label label2 = new Label("Label2");
    final TextField textfield2 = new TextField();
    final Button button = new Button("Button");

    gridPane.add(label1, 0, 0);
    gridPane.add(textfield1, 1, 0);
    gridPane.add(label2, 0, 1);
    gridPane.add(textfield2, 1, 1);
    gridPane.add(button, 1, 2);
    return gridPane;
}

private Pane createNavigationPane()
{
    final VBox vbox = new VBox(5);
    vbox.getChildren().addAll(new Text("LEFT"), new Button("VBox1"),
                                                new Button("VBox2"));
    return vbox;
}
```

Im Listing sehen wir verschiedene Besonderheiten und Methodenaufrufe, die wir noch nicht kennen. Da hier lediglich die Erstellung des Layouts mithilfe von Methoden von größerem Interesse ist, gehe ich auf die anderen Details später ein.

Starten Sie das Programm LAYOUTCOMBINATIONEXAMPLE, so kommt es zu folgender Ausgabe, die die Arbeitsweise der genannten Layouts verdeutlicht.

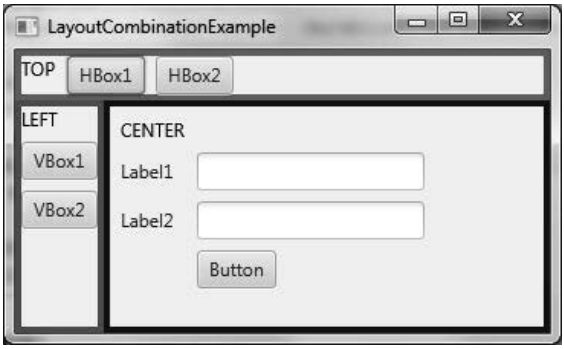

Abbildung 5-4 *Kombination von Layouts*

Hinweis: Komponentenbildung – Gestaltung komplexerer Layouts

Um grafische Elemente zu verschachteln und komplexere Strukturen aus Basisbausteinen zusammenzusetzen, kann man die einzelnen Bestandteile des GUIs mithilfe von Methoden erzeugen. Spielt der Aspekt Komponentenbildung und Wiederverwendbarkeit eine Rolle, so kann man statt Methoden besser eigenständige Klassen nutzen.

Man kann die beiden Ansätze sogar gewinnbringend kombinieren. Nachfolgend verdeutliche ich dies für die Toolbar. Diese ist nun in Form einer Klasse `ToolbarPane` implementiert und wird mithilfe der Fabrikmethode `createToolbarPane()` erzeugt. Somit ändert sich für die nutzende (obige) Applikation nichts an ihrem strukturellen Aufbau. Das Beschriebene kann man wie folgt umsetzen:

```
private Pane createToolbarPane()
{
    return new ToolbarPane();
}

static class ToolbarPane extends Pane
{
    public ToolbarPane()
    {
        final HBox hbox = new HBox(5);
        hbox.getChildren().addAll(new Text("TOP"),
                                  new Button("HBox1"), new Button("HBox2"));

        this.getChildren().add(hbox);
    }
}
```

Die `HBox` am Beispiel

Weil die `HBox` ein einfach zu verstehendes Layout realisiert, kann ich bei dessen Nutzung auf ein paar praktische Besonderheiten von JavaFX aufmerksam machen.

Dazu schauen wir auf folgendes Beispiel, in dem in einer `HBox` drei Bedienelemente, nämlich ein `Label`, ein `TextField` und ein `Button` mit auf 24pt vergrößerter Schriftart wie folgt hinzugefügt werden:

```
@Override
public void start(final Stage primaryStage)
{
    final Label label = new Label("Label");
    final TextField textfield = new TextField();
    final Button button = new Button("Button");
    button.setFont(Font.font(24));

    final HBox root = new HBox();
    root.getChildren().addAll(label, textfield, button);

    primaryStage.setTitle(FirstHBoxExample.class.getSimpleName());
    primaryStage.setScene(new Scene(root, 250, 70));
    primaryStage.show();
}
```

Führen wir das Programm FIRSTHBOXEXAMPLE aus, so erhalten wir eine Ausgabe wie in Abbildung 5-5.

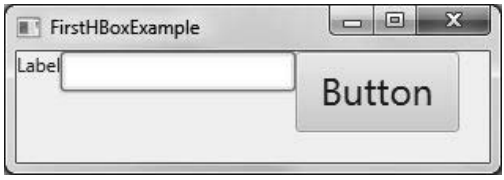

Abbildung 5-5 *Erstes Beispiel einer* `HBox`

Man erkennt sehr schön, dass die Bedienelemente horizontal, also innerhalb einer Zeile, angeordnet werden, wie wir dies von der `HBox` erwarten. Allerdings fallen gleich mehrere Dinge negativ auf:

- Die Bedienelemente werden ohne jeglichen Abstand direkt aneinander gezeichnet.
- Die Bedienelemente sind an ihrer oberen Kante ausgerichtet, was unruhig wirkt.

Besonderheiten von Layouts

Die beiden zuvor aufgelisteten kleineren Probleme im Layout inklusive möglicher Abhilfen wollen wir nachfolgend ein wenig genauer betrachten. Mit den gewonnenen Erkenntnissen bauen wir das obige Beispiel aus.

Besonderheit 1: Abstände Häufig ist es sinnvoll, zwischen den Bedienelementen einen gewissen Abstand vorzugeben, anstatt sie direkt aneinander zu zeichnen. Dazu kann man für die meisten Layouts die Methoden `setHgap(double)` und `setVgap(double)` nutzen. Für die Klassen `HBox` und `VBox` werden diese Methoden jedoch nicht angeboten – nur jeweils eine davon wäre für die `HBox` bzw. die `VBox` passend. Daher lassen sich die Abstände für diese beiden Layouts spezifisch über Konstruktorparameter festlegen. Außerdem existiert alternativ dazu die Methode `setSpacing(double)`.

Neben Abständen zwischen Bedienelementen bietet es sich an, auch insgesamt um alle Bedienelemente einen Abstand zum Fensterrand vorzugeben. Dies kann man mithilfe der Klasse `javafx.geometry.Insets` und einem Aufruf von `setPadding(Insets)` erreichen:

```
// Abstand 10 Pixel zwischen Bedienelementen
final FlowPane pane = new FlowPane();
pane.setHgap(10);
pane.setVgap(10);

// Spezialbehandlung HBox und VBox
final HBox hbox = new HBox(10);
final VBox vbox = new VBox(10);

// 7 Pixel Abstand vom Rand
pane.setPadding(new Insets(7,7,7,7));
hbox.setPadding(new Insets(7,7,7,7));
vbox.setPadding(new Insets(7,7,7,7));
```

Besonderheit 2: Ausrichtung an der Basislinie Die Darstellung ungleich hoher Bedienelemente direkt hintereinander ist optisch recht unharmonisch. Deutlich besser ist es, die Bedienelemente an einer virtuellen Linie auszurichten, die sich durch die in den Bedienelementen dargestellten Texte ergibt. Das spezifiziert man in JavaFX ganz einfach folgendermaßen:

```
// Linksbündige Ausrichtung an der Basislinie
hbox.setAlignment(Pos.BASELINE_LEFT);
```

Hier kommt die Aufzählung `javafx.geometry.Pos` zum Einsatz, in der verschiedene Positionierungen für X- und Y-Ausrichtung definiert sind, unter anderem etwa `TOP_LEFT`, `CENTER`, `BOTTOM_RIGHT` oder aber das oben verwendete `BASELINE_LEFT`, das eine linksbündige Ausrichtung auf der Höhe der Basislinie vornimmt.

Abhilfen im Einsatz Mit den gerade gewonnenen Erkenntnissen wollen wir die angesprochenen Probleme lösen. Zunächst geben wir bei der Konstruktion der `HBox` einen Abstand vor, der zwischen den einzelnen Bedienelementen eingehalten werden soll. Zudem setzen wir einen Rand über `setPadding(Insets)`. Darüber hinaus sorgen wir durch Angabe von `Pos.BASELINE_LEFT` im Aufruf von `setAlignment(Pos)` dafür, dass die Bedienelemente an der Basislinie ausgerichtet sind:

```
@Override
public void start(final Stage primaryStage)
{
    final Label label = new Label("Label");
    final TextField textfield = new TextField();
    final Button button = new Button("Button");
    button.setFont(Font.font(24));

    // Besonderheit 1a: Abstand 10 Pixel zwischen Bedienelementen
    final HBox root = new HBox(10);
    // Besonderheit 1b: 7 Pixel Abstand vom Rand
    root.setPadding(new Insets(7,7,7,7));
    // Besonderheit 2: Ausrichtug an der Basislinie
    root.setAlignment(Pos.BASELINE_LEFT);

    root.getChildren().addAll(label, textfield, button);

    primaryStage.setTitle(HBoxWithAlignmentsExample.class.getSimpleName());
    primaryStage.setScene(new Scene(root, 300, 70));
    primaryStage.show();
}
```

Wenn wir das Programm HBoxWithAlignmentsExample starten, dann sehen wir die Verbesserung im Layout: Die Bedienelemente sind an einer Basislinie ausgerichtet. Beim Betrachten von Abbildung 5-6 erkennen wir auch, dass die Größe des Fensters nicht mehr ausreicht, um die Texte in den Bedienelementen vollständig darzustellen. Dann sorgt JavaFX dafür, dass die Texte durch Auslassungszeichen (eine sogenannte ***Ellipsis***, meistens als drei Punkte (...) dargestellt) abgekürzt werden. Insbesondere bei einem größenveränderlichen Fenster ist dies eine wünschenswerte Eigenschaft, die man nun out-of-the-Box mitgeliefert bekommt und besser noch, die sich zudem noch vielfältig konfigurieren lässt, wie wir dies im nachfolgenden Absatz kennenlernen werden.

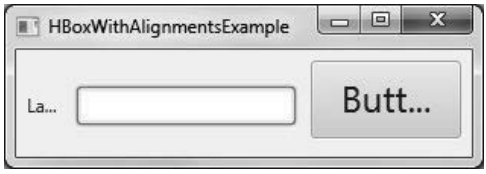

Abbildung 5-6 *Beispiel eines Layouts mit* `HBox` *mit Ausrichtung*

Besonderheit bei Größenveränderungen

Wir haben zuvor erkannt, dass JavaFX bereits diverse kleinere Verbesserungen und Bequemlichkeitsfunktionalitäten (Convenience) bereitstellt. Diese helfen unter anderem dabei, auf Größenveränderungen eines Fensters zu reagieren.

Nachfolgend möchte ich auf weitere Convenience-Funktionalitäten von JavaFX eingehen. Zunächst einmal auf die Konfigurierbarkeit der automatischen Verkürzung eines Textes durch Darstellung einer Ellipsis. Dabei lässt sich festlegen, an welcher Position die Ellipsis dargestellt werden soll. Zudem kann man auch die zur Abkürzung

genutzte Zeichenfolge abändern. Des Weiteren kann man bestimmen, wie und welche Bedienelemente bei Größenanpassungen des Containers in ihrer Größe verändert werden sollen und welche nicht. Das wird über die Konstanten `ALWAYS`, `SOMETIMES` und `NEVER` aus der Aufzählung `javafx.scene.layout.Priority` gesteuert.

Mit diesem Wissen wollen wir das vorherige Beispiel anpassen: Wir legen für das `Label` und den `Button` durch Aufruf von `setTextOverrun(OverrunStyle)` das Verhalten beim Kürzen sowie mit `setEllipsisString(String)` die Zeichenfolge der Ellipsis fest. Für das `TextField` bestimmen wir durch Aufruf von `setHgrow(Priority)` mit der Priorität `ALWAYS`, dass Größenänderungen der `HBox`-Containerkomponente immer auch zu Größenänderungen des Textfelds führen. Zunächst sorgt die `HBox` aber dafür, dass alle enthaltenen Elemente ihre gewünschte Größe erhalten. Der darüber hinaus zur Verfügung stehende Platz wird dann an das `TextField` vergeben. All dies implementieren wir folgendermaßen:

```
@Override
public void start(final Stage primaryStage)
{
    final Label label = new Label("Label");
    label.setTextOverrun(OverrunStyle.ELLIPSIS);  // Standard
    final TextField textfield = new TextField();
    final Button button = new Button("This is a button");
    button.setFont(Font.font(24));

    // Setzen des Strings "##~##" als Ellipsis-Abkürzung
    button.setEllipsisString("##~##");
    button.setTextOverrun(OverrunStyle.CENTER_ELLIPSIS);

    final HBox root = new HBox(10);
    root.setPadding(new Insets(7,7,7,7));
    root.setAlignment(Pos.BASELINE_LEFT);
    root.getChildren().addAll(label, textfield, button);

    // Größenveränderung
    HBox.setHgrow(textfield, Priority.ALWAYS);

    primaryStage.setTitle(ResizableHBoxExample.class.getSimpleName());
    primaryStage.setScene(new Scene(root, 390, 120));
    primaryStage.show();
}
```

Nach dem Start des Programms RESIZABLEHBOXEXAMPLE kommt es in etwa zu einer Darstellung wie in Abbildung 5-7. Verändern Sie ein wenig die Größe des Fensters und beobachten Sie die Auswirkungen der zuletzt durchgeführten Erweiterungen.

Abbildung 5-7 *Größenveränderliches Layout mit einer* `HBox`

Die GridPane am Beispiel

Für professionelle Anwendungen benötigt man recht häufig eine Platzierung von Bedienelementen, die anhand eines Rasters erfolgt. Innerhalb der einzelnen Rasterzellen sollen Bedienelemente individuell ausgerichtet werden können (links-, rechtsbündig oder zentriert). Während in Swing das `GridLayout` nicht so viel Flexibilität bietet und man mit dem `GridBagLayout` häufig bezüglich der Konfigurationsangaben kämpfen musste, gestaltet sich die Arbeit mit dem JavaFX-Layoutcontainer `GridPane` deutlich angenehmer, weil die Zuordnung zum Raster auf einfache Weise erfolgt. Außerdem kann man bei Bedarf, etwa zu Debugging-Zwecken, Rasterlinien einblenden.

Mithilfe einer `GridPane` wollen wir nun einen recht einfachen Login-Dialog gestalten. Dort werden in zwei Zeilen jeweils in einer eigenen Spalte ein `Label` gefolgt von einem `TextField` platziert. In einer dritten Zeile wird in der zweiten Spalte ein Login-Button angeordnet. Zur Demonstration unterschiedlicher Ausrichtungen wählen wir für die `Label`s einmal links- und einmal rechtsbündig. Der `Button` wird auch rechtsbündig ausgerichtet. Um das praktische Feature der Gitterlinien demonstrieren zu können, nutzen wir eine `Checkbox` und einen `EventHandler<ActionEvent>`, über den diese Hilfslinien ein- bzw. ausgeschaltet werden können. Die beschriebenen Funktionalitäten realisieren wir folgendermaßen:

```
@Override
public void start(final Stage primaryStage) throws Exception
{
    final GridPane gridPane = new GridPane();
    gridPane.setPadding(new Insets(10, 10, 10, 10));
    gridPane.setHgap(7);
    gridPane.setVgap(7);

    final Label lblName = new Label("Name:");
    final TextField tfName = new TextField();

    final Label lblPassword = new Label("Password:");
    final PasswordField pfPassword = new PasswordField();

    final Button btnLogin = new Button("Login");

    // Bereitstellung von Gitterlinien
    final CheckBox checkBoxShowGridLines = new CheckBox("Show Gridlines");
    checkBoxShowGridLines.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(final ActionEvent event)
        {
           gridPane.setGridLinesVisible(checkBoxShowGridLines.isSelected());
        }
    });

    // Zuordnung zum Grid (Node, X-Position, Y-Position)
    gridPane.add(lblName, 0, 0);
    gridPane.add(tfName, 1, 0);
    gridPane.add(lblPassword, 0, 1);
    gridPane.add(pfPassword, 1, 1);
    gridPane.add(btnLogin, 1, 2);
    gridPane.add(checkBoxShowGridLines, 0, 5);
```

```
    // Layoutbesonderheiten
    GridPane.setHalignment(lblName, HPos.LEFT);
    GridPane.setHalignment(lblPassword, HPos.RIGHT);
    GridPane.setHalignment(btnLogin, HPos.RIGHT);

    primaryStage.setScene(new Scene(gridPane, 300, 150));
    primaryStage.setTitle("GridPaneExample");
    primaryStage.show();
}
```

Abbildung 5-8 zeigt die Ausgabe des Programms GRIDPANEEXAMPLE mit und ohne aktivierte Gitterlinien.

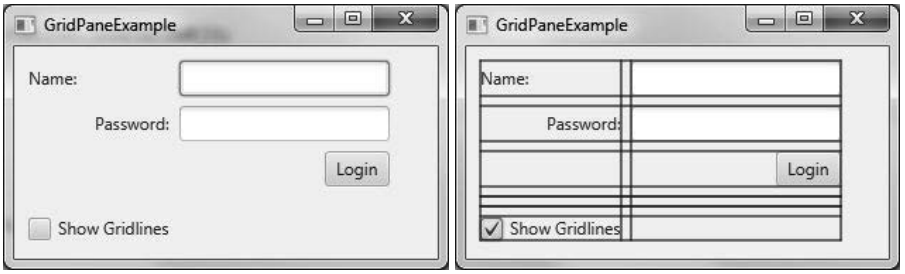

Abbildung 5-8 *Beispiel eines Layouts mit einer* `GridPane`

Durch die in der Abbildung gezeigten Gitterlinien erkennt man die Positionierung durch spezielle Abstandsspalten/-zeilen. Das ist eine mögliche Form der Nutzung. Alternativ dazu kann man die Abstände auch über die bereits kennengelernten Vorgaben über `Insets` erzielen. Bei komplexeren Eingabemasken kann das durchaus der bessere Weg sein, da das Gitter möglicherweise nicht die benötigte Flexibilität erlaubt.

Abschließend möchte ich noch kurz auf einige Dinge eingehen, die Ihnen vielleicht schon beim Betrachten des Listings als Fragen in den Sinn gekommen sind.

Frage: Sollte man Präfixe für Bedienelemente nutzen? In diesem Beispiel werden Präfixe für Bedienelemente verwendet. Wann sollte man diese verwenden? Eine allgemeingültige Antwort auf diese Frage gibt es sicher nicht. Zum Teil bietet es sich an, für Bedienelemente verschiedene Kürzel wie `lbl` für `Label` oder `tf` für `TextField` zu nutzen, etwa um das Label `lblName` deutlich von dem korrespondierenden Textfeld `tfName` unterscheiden zu können. Es gibt noch eine andere Form der Namensgebung: Man kann eine Suffix-Notation nutzen. Damit ergeben sich teilweise lesbare Namen wie `loginButton`. Aber für `nameLabel` und `nameTextfield` stößt auch diese Notation an ihre Grenzen.

Manchmal empfinde ich Präfixe bzw. Suffixe als hilfreich, manchmal als störend. Wichtig ist vor allem, dass der Rest des Namens aussagekräftig ist.