5 Ein Layout erstellen

In diesem Kapitel wollen wir uns genauer ansehen, wie in JavaFX das Layout von Benutzeroberflächen konzipiert ist. Dabei sehen wir uns zuerst die eingebauten Layoutmanager an und erstellen anschließend eine eigene LayoutPane.

5.1 Die eingebauten Layouts verwenden

In JavaFX ist es möglich, für alle UI-Komponenten die Größe und Position festzulegen und so das Layout zu bestimmen. So ein absolutes Layout ist aber nur in wenigen Fällen wirklich sinnvoll einzusetzen. Sobald eine Anwendung auf Größenänderungen reagieren soll, wird ein solches Layout extrem komplex. Deshalb gibt es das Konzept des Layoutmanagers, der den vorgegebenen Platz sinnvoll auf die Komponenten verteilt. Die Position und Größe der einzelnen Komponenten wird dabei nicht mehr absolut angegeben, sondern lediglich durch Constraints beschränkt. So kann man zum Beispiel festlegen, dass ein Button immer einen definierten Abstand zur rechten unteren Ecke beibehält, oder definieren, welche Komponente beim Vergrößern des Fensters den überschüssigen Platz beansprucht.

JavaFX kommt mit einer Sammlung von eingebauten Layoutcontainern. Das Konzept ist ein wenig anders als bei Swing, wo dem Container ein dedizierter Layoutmanager übergeben wird. Bei JavaFX übernimmt der Container selbst die Aufgabe des Layoutens. Ansonsten ist das Vorgehen allerdings recht ähnlich. Die einzelnen Komponenten werden dem Container hinzugefügt und dann mit Constraints versehen. Es entfällt lediglich das Setzen des Layoutmanagers.

5.1.1 VBox und HBox

Die einfachsten Layout-Panes sind `VBox` und HBox. VBox legt die Kindkomponenten einfach vertikal nebeneinander, die HBox macht dasselbe in horizontaler Richtung.

```
HBox hbox = new HBox();
hbox.setPadding(new Insets(10,20,20,20));
hbox.setSpacing(15);
Button left = new Button("I'm left");
Button right = new Button("I'm right");
hbox.getChildren().addAll(left, right);
```

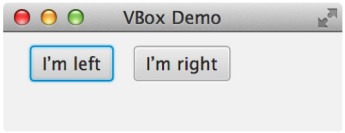

Abb. 5–1 *Layout mit der HBox*

Der Code ist eigentlich selbsterklärend: Wir erzeugen den Layoutcontainer und setzen den Innenabstand (Padding) der einzelnen Seiten mittels eines `Insets`-Objekts. Dann bestimmen wir den Abstand der Kindelemente (Spacing) und fügen die Child-Nodes mittels `getChildren().addAll(…)` hinzu. Für die `VBox` funktioniert das analog.

Wie setze ich Constraints?

Je nach Layoutmanager lassen sich für die vom Layout verwalteten Komponenten auch Constraints setzen. Bei einer Größenänderung der `VBox` kann man zum Beispiel festlegen, welche Komponente den zusätzlich verfügbaren Raum einnimmt:

```
VBox vbox = new VBox(8); // spacing ist 8
Button claimExcessSpace = new Button("lass mich wachsen!");
claimExcessSpace.setMaxHeight(1000);
VBox.setVgrow(claimExcessSpace, Priority.ALWAYS);
Button staySmall = new Button("lass mich in Ruhe!");
staySmall.setMaxHeight(1000);
vbox.getChildren().addAll(claimExcessSpace, staySmall);
```

Beide Buttons haben in diesem Beispiel eine maximale Größe von 1000, aber nur einem wird der überschüssige Platz zugeteilt.

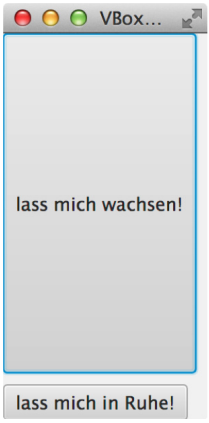

Abb. 5–2 *Mittels Constraints lässt sich zum Beispiel die Verteilung von freiem Platz regeln.*

Das Festlegen eines Constraints über eine statische Methode ist etwas ungewöhnlich. Der Grund dafür ist, dass die Constraints für die einzelnen Nodes nicht im Layoutcontainer selbst (hier der `VBox`), sondern vom Node gespeichert werden. Die statische Methode setzt im Node eine `LayoutProperty`, die dann vom Layoutcontainer gelesen werden kann. Ich persönlich hätte es für den Nutzer der API als intuitiver empfunden, wenn man alle `LayoutConstraints` auf der Instanz setzen würde und auch gleich beim Hinzufügen zum Container setzen könnte.

5.2 Die BorderPane verwenden

In Swing ist das `BorderLayout` einer der beliebtesten Layoutmanager. Dieses Layout ist insbesondere deshalb so populär, weil es sehr einfach ist und trotzdem bereits relativ viele Anforderungen abdeckt: Man kann vor allem ganz einfach ein Menü oder einen Toolbar und eine Taskleiste oberhalb und unterhalb von einem Zentralbereich positionieren und bei Bedarf links und rechts noch Navigationsfenster oder Detailansichten anordnen, ohne sich mit Constraints herumzuschlagen. Bei JavaFX verwendet man dazu die fünf Bereiche der `BorderPane`:

```
BorderPane borderPane = new BorderPane();
borderPane.setTop(toolbar);
borderPane.setBottom(taskbar);
borderPane.setCenter(document);
borderPane.setLeft(navigator);
borderPane.setRight(properties);
```

Vergrößert man das Fenster, so wird überschüssiger Platz der Komponente im Zentrum zugewiesen. Anders als in anderen Layoutmanagern funktioniert das hier nicht:

```
borderPane.getChildren.addAll(toolbar, taskbar,document, ...);
```

Komponenten, die so hinzugefügt wurden, werden nicht dargestellt. Das ist ein wenig inkonsistent und sollte meiner Meinung nach im Sinne einer einheitlichen API behoben werden.

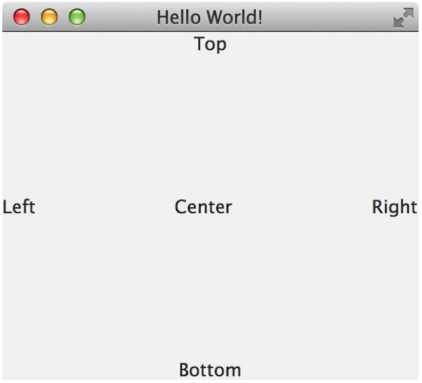

Abb. 5–3 *Die BorderPane ist gut als Basislayout für Anwendungen geeignet.*

5.3 Layouts mit der AnchorPane erstellen

Die `AnchorPane` wird verwendet, wenn man Komponenten in einem bestimmten Abstand vom Fensterrand positionieren möchte. Ändert man die Fenstergröße, dann behalten diese Komponenten ihre Position relativ zum Fensterrand bei. Das ist hilfreich, wenn man zum Beispiel in einem typischen Dialog eine Button-Leiste in der rechten unteren Ecke platzieren möchte:

```
AnchorPane anchorePane = new AnchorPane();
Button save = new Button(„save");
Button help = new Button(„help");
Button cancel = new Button(„cancel");
HBox buttons = new HBox();
buttons.setSpacing(12);
buttons.getChildren.addAll(cancel, save, help);
anchorPane.getChildren().add(buttons);
AnchorPane.setRightAnchor(buttons, 10);
AnchorPane.setBottomAnchor(buttons, 20);
```

5.4 Die FlowPane verwenden

Die `FlowPane` verhält sich wie das `FlowLayout` in Swing. Sie wird typischerweise verwendet, um eine größere Anzahl Komponenten möglichst platzsparend anzuzeigen. Der Dateisystem-Explorer des Betriebssystems hat meist eine solche Ansicht, um Dateien als Icons anzuzeigen. Die `FlowPane` füllt – je nach Orientierung – eine Zeile oder Spalte mit den Kindkomponenten auf, bis der Komponentenrand erreicht ist, danach wird umbrochen:

```
FlowPane iconView = new FlowPane();
iconView.setVgap(10);
iconView .setHgap(20);
for (int i = 0; i < images.length; i++) {
          iconView.getChildren().add(new ImageView(image[i]);
}
```

5.5 Layout mit der StackPane

Die `StackPane` platziert alle Kindelemente in der Reihenfolge ihres Hinzufügens übereinander. Das ist sehr hilfreich, um Layers einzusetzen. So lässt sich zum Beispiel ein Overlay realisieren. Im folgenden Beispiel verwenden wir so ein Overlay anstatt eines Pop-up-Dialogs, um eine Eingabe zu bestätigen:

```
Button ok = new Button("OK");
Button cancel = new Button("Cancel");

HBox hBox = new HBox(cancel, ok);
hBox.setBackground(new Background(new
BackgroundFill(Color.WHITE.deriveColor(1, 1, 1, .7), CornerRadii.EMPTY,
Insets.EMPTY)));
hBox.setSpacing(10);
hBox.setVisible(false);
hBox.setAlignment(Pos.BOTTOM_CENTER);
hBox.setPadding(new Insets(0, 0, 10, 0));
EventHandler<ActionEvent> h = e-> {
   hBox.setVisible(!hBox.isVisible());
   };
ok.setOnAction(h);
cancel.setOnAction(h);
Button button = new Button("Do Something!");
button.setOnAction( h);
StackPane root = new StackPane(button, hBox);
```

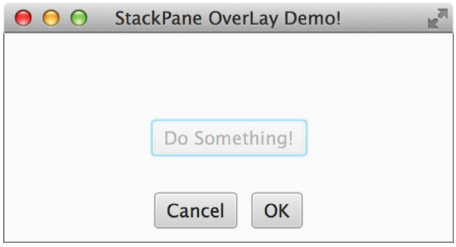

Abb. 5–4 *Eine StackPane kann verwendet werden , um die Anwendung in einzelne Layers aufzuteilen*

Ein weiterer Einsatzbereich ist das Erstellen komplexerer Komponenten durch das Übereinanderstapeln von simplen Nodes:

```
Circle circle = new Circle(50);
circle.setStroke(Color.BLACK);
circle.setFill(Color.WHITE);
Text number = new Text("53");
number.setFont(Font.font("Arial",FontWeight.BOLD ,64));
StackPane herbie = new StackPane(circle, number);
```

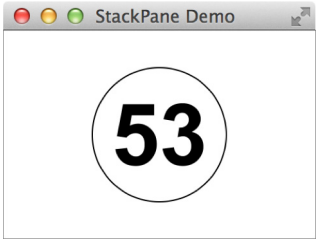

Abb. 5–5 *Mit der StackPane lassen sich Nodes einfach stapeln.*

Wie positioniere ich Elemente in der StackPane?

Standardmäßig werden Komponenten in der `StackPane` zentriert. Das macht es einfach, die Komponenten zu positionieren. Will man eine andere Anordnung, so kann man diese mit `StackPane.setAlignment` für jeden Node separat festlegen. Der Rand um jede Komponente kann mit `StackPane.setMargin` bestimmt werden:

```
Rectangle rect = new Rectangle(100, 100);
ArrayList<Stop> stops = new ArrayList<Stop>();
stops.add(new Stop(0, Color.LIGHTGREEN));
stops.add(new Stop(1, Color.FORESTGREEN));
rect.setFill(new LinearGradient(0, 0, 1, 1, true, CycleMethod.NO_CYCLE,
stops));
rect.setStroke(Color.WHITE);
Text center = new Text("Ba");
center.setFont(Font.font("Arial", FontWeight.BOLD, 64));
center.setFill(Color.WHITE);
Text topRight = new Text("+2");
topRight.setFont(Font.font("Arial", 8));
topRight.setFill(Color.WHITE);
Text topLeft = new Text("137.33");
topLeft.setFont(Font.font("Arial", 8));
topLeft.setFill(Color.WHITE);
Text bottomLeftSmall = new Text("2-8-18-7");
bottomLeftSmall.setFont(Font.font("Arial", 8));
bottomLeftSmall.setFill(Color.WHITE);
Text bottomLeft = new Text("56");
bottomLeft.setFont(Font.font("Arial", 10));
bottomLeft.setFill(Color.WHITE);

StackPane.setAlignment(topRight, Pos.TOP_RIGHT);
StackPane.setAlignment(topLeft, Pos.TOP_LEFT);
StackPane.setAlignment(bottomLeft, Pos.BOTTOM_LEFT);
```

```
StackPane.setAlignment(bottomLeftSmall, Pos.BOTTOM_LEFT);
StackPane.setMargin(bottomLeftSmall, new Insets(0, 0, 2, 4));
StackPane.setMargin(bottomLeft, new Insets(0, 0, 14, 4));
StackPane.setMargin(topRight, new Insets(5));
StackPane.setMargin(topLeft, new Insets(5));

StackPane barium = new StackPane(rect, center, topRight, topLeft, bottomLeft,
bottomLeftSmall);
barium.setMaxSize(Region.USE_PREF_SIZE, Region.USE_PREF_SIZE);
```

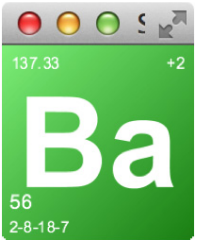

Abb. 5–6 *Nodes können bei der StackPane nicht nur zentriert gestapelt werden.*

5.6 Layout mit der TilePane

Die `TilePane` stellt ihre Kindelemente als gleich große »Kacheln« dar. Die Kachelhöhe wird dabei von dem Kindknoten mit der größten `prefHeight`, die Kachelbreite von dem Kindknoten mit der größten `prefWidth` übernommen. Wenn man dieses Verhalten überschreiben möchte, kann man die `prefTileWidth` und `prefTileHeight` auch direkt setzen. Die `TilePane` wird dann versuchen, die Kindelemente an die angegebene Größe anzupassen. Wenn das nicht klappt, weil der Child-Node diese Größe nicht erlaubt, so wird sie innerhalb der Tile mithilfe des angegebenen `TileAlignment` platziert.

Das Alignment kann für jeden einzelnen Child-Node wie üblich mittels einer statischen Methode (`TilePane.SetAlignment`) eingestellt werden. Ebenso lässt sich auch der Rand um den Node innerhalb seiner Kachel bestimmen:

```
TilePane tilePane = new TilePane();
for (int i = 0; i < 10; i++) {
   Rectangle rectangle = new Rectangle(i*3, i*3);
   rectangle.setFill(new Color( ((double)i*10)/250,
     ((double)i*10)/250,
     ((double)i*10)/250, 1));
   tilePane.getChildren().add(rectangle);
   TilePane.setAlignment(rectangle, Pos.BOTTOM_RIGHT);
   TilePane.setMargin(rectangle, new Insets(i));
}
```

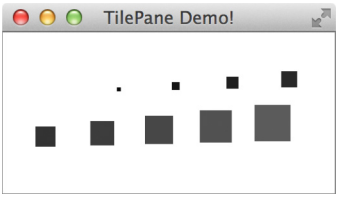

Abb. 5–7 *Die Größe der Tiles in der TilePane wird durch das höchste und das breiteste Element bestimmt.*

Ist ein Kindknoten größer als die Kachelgröße, so kann er auch über die Kachelgrenzen hinausragen, denn die `TilePane` unternimmt kein »Clipping«, d.h., sie schneidet überstehende Ränder nicht ab.

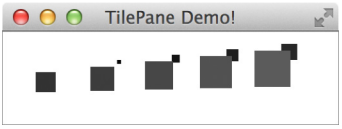

Abb. 5–8 *Sind einzelne Tiles größer als die prefTileHeight oder prefTileWidth, so überlappen sie mit anderen Tiles.*

Diesen Effekt können wir testen, indem wir in unserem Beispiel die `prefTileHeight` kleiner als unsere Rechtecke wählen:

```
tilePane.setPrefTileHeight(25);
```

5.7 Layout mit der GridPane

Unter den Layout-Panes der JavaFX-API ist die `GridPane` die mächtigste Komponente. Mit dieser Pane lassen sich ähnlich wie mit dem berüchtigten GridbagLayout in Swing auch komplexere Abhängigkeiten zwischen Kindkomponenten darstellen.

5.7.1 Wie füge ich Nodes hinzu?

Dieses Layout basiert auf einem Gitter. Die Abstände zwischen Spalten (`vGap`) und Reihen (`hGap`) lassen sich unabhängig festlegen, und mit `setPadding` können wir einen Rand um die Komponente definieren:

```
GridPane gridPane = new GridPane();
gridPane.setVgap(5);
gridPane.setHgap(8);
gridPane.setPadding(new Insets(10));
```

Child-Nodes werden mit der Methode `add` hinzugefügt. Dabei übergibt man neben dem Node zumindest auch den Zeilen- und Spaltenindex:

```
Image image = new Image(getClass()
    .getResource("duke-wave.png")
    .toExternalForm(), 100, 100, true, true);
final ImageView logo = new ImageView(image);
gridPane.add(logo, 3, 0);
```

Soll der Node mehrere Zeilen oder Spalten überspannen, muss man auch hierfür jeweils einen Wert angeben. Das folgende Label überspannt drei Spalten:

```
Label heading = new Label("This is a GridPane");
gridPane.add(heading, 0, 0, 3, 1);
```

Als Nächstes wollen wir einige Anpassungen an den Reihen und Spalten vornehmen. Fügen wir dazu noch ein paar Komponenten hinzu:

```
gridPane.add(new Label("Name:"), 1, 1);
gridPane.add(new TextField("Name eingeben"), 2, 1);
gridPane.add(new Label("Beruf:"), 1, 2);
gridPane.add(new TextField("Beruf eingeben"), 2, 2);
gridPane.add(new Button("speichern"), 3, 4);
```

Wenn man so ein Layout manuell erstellt, vertut man sich leicht einmal bei einem einzelnen Wert, und es ist dann gar nicht so einfach, den Fehler zu finden. Deshalb gibt es eine Art Debug-Modus. Mit `setGridLinesVisible` kann man sich das Gitter anzeigen lassen. Dabei werden auch `vGap` und `hGap` durch Linien angezeigt. So lassen sich Fehler leichter entdecken:

```
gridPane.setGridLinesVisible(true);
```

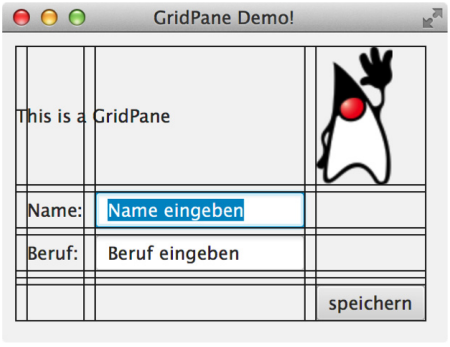

Abb. 5–9 *Die GridPane mit eingeblendeten Gitterlinien*

In Abbildung 5–9 sehen wir das Ergebnis: Die Spalten nehmen standardmäßig die Breite des breitesten enthaltenen Elements an, die Zeilen übernehmen die Höhe des höchsten enthaltenen Nodes. Leere Zeilen haben daher die Höhe 0. Das sieht man anhand der Zeile 3 oberhalb des »speichern«-Buttons, die lediglich als Linie zwischen oberem und unterem `vGap` angezeigt wird.

5.7.2 Wie passt man Höhe und Breite der Columns und Rows an?

Wir können das Standardverhalten verändern, indem wir `ColumnConstraints` und `RowConstraints` für einzelne Spalten oder Zeilen festlegen. Nehmen wir an, der Abstand zwischen Titelzeile und Formular ist uns zu groß. Dann können wir zum Beispiel für eine Zeile eine feste Höhe angeben:

```
gridPane.getRowConstraints().add(0, new RowConstraints(40));
```

Die Zeile hat nun eine feste Höhe. Wir könnten auch noch eine minimale, bevorzugte und maximale Breite setzen. Alternativ können Höhe oder Breite aber auch prozentual angegeben werden. Hier setzen wir zum Beispiel die Breite der 2. Spalte auf 25% der `GridPane`-Breite:

```
ColumnConstraints cc = new ColumnConstraints();
cc.setPercentWidth(25);
gridPane.getColumnConstraints().addAll(new ColumnConstraints(), cc);
```

Das »leere« `ColumnConstraint`, das ich hier mit einfüge, ist notwendig, denn die Liste, die ich von `getColumnConstraints` zurück erhalte, ist anfangs leer, und ein `add(1, cc)` würde zu einer `IndexOutOfBoundsException` führen. Mithilfe der `Row-` und `ColumnConstraints` lassen sich noch viele weitere Eigenschaften der Zeilen und Spalten festlegen. Am wichtigsten ist das Wachstumsverhalten, wenn mehr als genug Platz verfügbar ist. So können wir zum Beispiel festsetzen, dass die Spalte mit den Eingabefeldern den übrigen Platz erhält:

```
ColumnConstraints cc2 = new ColumnConstraints();
cc2.setHgrow(Priority.ALWAYS);
gridPane.getColumnConstraints().addAll(new ColumnConstraints(), cc, cc2);
```

5.7.3 Wie werden einzelne Elemente ausgerichtet?

Noch sind die einzelnen Nodes innerhalb der `GridPane` nicht ausgerichtet. Legen wir nun für die Überschrift und das Icon ein Alignment fest, sodass sie jeweils links und rechts oben verankert sind:

```
GridPane.setHalignment(logo, HPos.RIGHT);
GridPane.setValignment(logo, VPos.TOP);
GridPane.setHalignment(heading, HPos.LEFT);
GridPane.setValignment(heading, VPos.TOP);
```

In Abbildung 5–10 sehen Sie nun das Endergebnis mit allen vorgenommenen Änderungen.

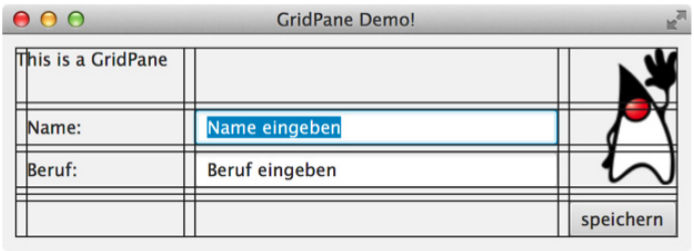

Abb. 5–10 *Die fertige Komponente*

5.8 Unmanaged Nodes

Wenn man in einer Komponente einzelne Nodes absolut positionieren möchte, ruft man dazu auf dem entsprechenden Node die Methode `setmanaged(false)` auf. Der Layoutmanager kümmert sich dann nicht um die Komponente und sie wird entsprechend ihren `layoutX`- und `layoutY`-Properties positioniert. Im folgenden Beispiel haben wir die `init`-Methode der Demo-Applikation »StockLineChart-App«[1] verändert und so die Anwendung mit einer lästigen Message versehen:

```
Group root = new Group();
final Scene scene = new Scene(root);
primaryStage.setScene(scene);
root.getChildren().add(createChart());
Label text = new Label("Demo Version\nBuy a License!");
text.setFont(Font.font("Verdana", 56));
text.resizeRelocate(60,60, 450, 200);
root.getChildren().add(text);
text.setManaged(false);
```

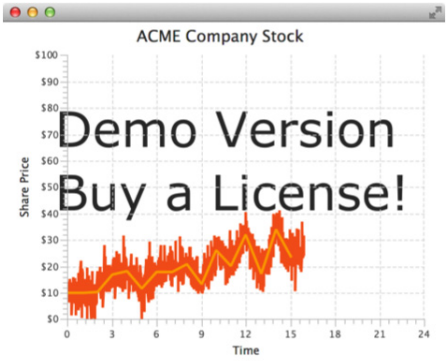

Abb. 5–11 *Unmanaged Nodes werden beim Layout vom Layoutcontainer ignoriert.*

1. Diese Applikation ist Teil der in Ensemble enthaltenen Demos und kann unter *http://download.oracle.com/otn-pub/java/jdk/8u25-b17-demos/jdk-8u25-macosx-x86_64-demos.zip* heruntergeladen werden.

5.9 Eigene Layoutcontainer erstellen

Mithilfe der eingebauten JavaFX-Layouts lassen sich bereits viele Anwendungsfälle abdecken. Sie sind auch sehr gut mit dem SceneBuilder zu verwenden. Das werden wir in Abschnitt 6.3 im Detail behandeln. Es gibt jedoch Dinge, die mit den bestehenden Containern nicht einfach umzusetzen sind. In diesem Fall kann man recht einfach einen eigenen Layoutcontainer erstellen. Alle Layoutcontainer in JavaFX leiten von `Region` ab. Um das gewünschte Layout zu realisieren, wird die Methode `layoutChildren` der Superklasse `Parent` überschrieben. Die Klasse `Parent` selbst weist den Nodes lediglich ihre bevorzugte Größe zu und lässt sie ansonsten in Ruhe. Wir können das Verhalten ändern, indem wir den Nodes zum Beispiel in der Methode `layoutChildren` eine Größe und eine Position zuweisen. Dazu können wir die Methoden `resize`, `relocate` oder `resizeRelocate` verwenden:

```
class CardStackLayout extends Region{

   @Override
   public ObservableList<Node> getChildren() {
     return super.getChildren();
   }

   @Override
   protected void layoutChildren() {
     super.layoutChildren();
     ObservableList<Node> children = getChildren();
     int i = 0;
     for (Node child : children) {
       child.relocate(i, 0);
       i+= 10;
     }
   }
}
```

Wir verwenden die Methode `relocate` und verschieben den Child-Node jeweils um 10 Pixel nach rechts. Dieses Layout können wir zum Beispiel verwenden, um einen Stapel Karten[2] darzustellen:

```
String[] values = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "ace",
          "jack", "king", "queen"};
String[] colors = {"clubs", "diamonds", "hearts", "spades"};
EventHandler<MouseEvent> clickHandler;
CardStackLayout cardStackLayout = new CardStackLayout();
for (int i = 0; i < colors.length; i++) {
   String color = colors[i];
   for (int j = 0; j < values.length; j++) {
     String value = values[j];
     String rn = "cards/" + value + "_of_" + color + ".png";
```

2. Bilder sind lizenzfrei verfügbar und wurden erstellt von Byron Knoll: *http://code.google.com/p/vector-playing-cards/*.

```
      Image image = new Image(getClass().getResource(rn)
              .toExternalForm(), 200, 200, true, false);
      ImageView card = new ImageView(image);
      cardStackLayout.getChildren().add(card);
    }
  }
```

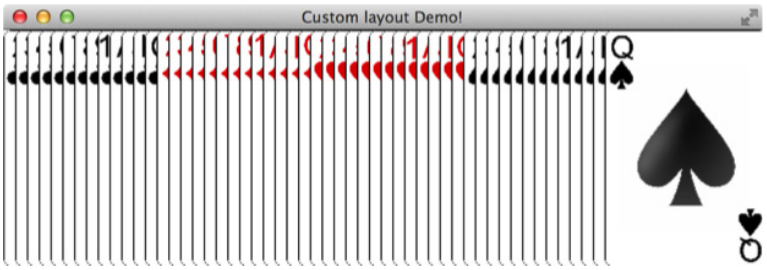

Abb. 5–12 *CardStackLayout, ein eigener Layoutcontainer*

Natürlich hätten wir die `ImageViews` auch einfach absolut positionieren können. Der Vorteil eines eigenen Layouts liegt jedoch darin, dass es dynamisch auf Änderungen reagiert und das Update automatisch von der Scene angestoßen wird. Das wird am besten deutlich, wenn wir zwei Kartenstapel kombinieren. Per Klick auf eine Karte soll diese in den anderen Stapel wandern:

```
String[] values = {"2", "3", "4", "5", "6", "7", "8", "9", "10",
          "ace", "jack", "king", "queen"};
String[] colors = {"clubs", "diamonds", "hearts", "spades"};
EventHandler<MouseEvent> clickHandler;
CardStackLayout cardStackLayout = new CardStackLayout();
CardStackLayout cardStackLayout2 = new CardStackLayout();
clickHandler = new EventHandler<MouseEvent>() {

  @Override
  public void handle(javafx.scene.input.MouseEvent event) {
    Node source = (Node) event.getSource();
    Parent parent = source.getParent();
    if (parent == cardStackLayout) {
      cardStackLayout.getChildren().remove(source);
      cardStackLayout2.getChildren().add(source);
    }
    if (parent == cardStackLayout2) {
      cardStackLayout2.getChildren().remove(source);
      cardStackLayout.getChildren().add(source);
    }

  }
};
for (int i = 0; i < colors.length; i++) {
  String color = colors[i];
  for (int j = 0; j < values.length; j++) {
    String value = values[j];
```

```
        String rn = "cards/" + value + "_of_" + color + ".png";
        Image image = new Image(getClass().getResource(rn)
                .toExternalForm(), 200, 200, true, false);
        ImageView card = new ImageView(image);
        card.setOnMouseClicked(clickHandler);
        cardStackLayout.getChildren().add(card);
      }
    }
    VBox vBox = new VBox(cardStackLayout, cardStackLayout2);
```

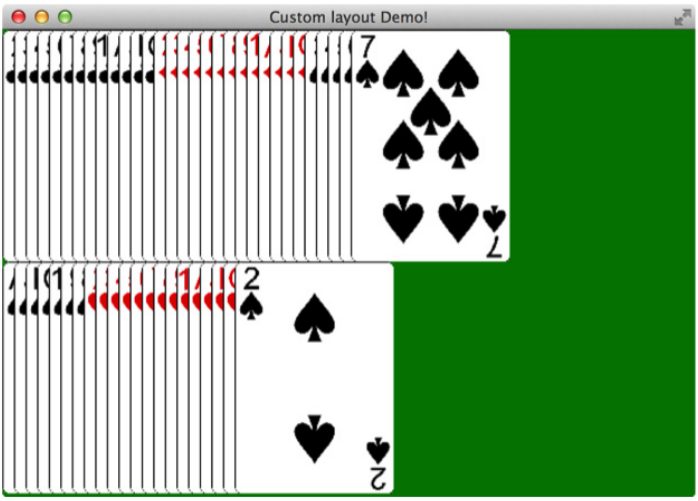

Abb. 5–13 *Layoutcontainer werden automatisch aktualisiert. So können mit wenig Aufwand dynamisch Nodes positioniert werden.*

5.10 Workshop: Ein bestimmtes Layout umsetzen

Nachdem Sie jetzt einen guten Überblick über die verfügbaren Layouts haben, sehen wir uns an, wie man diese kombiniert, um ein vorgegebenes Layout umzusetzen. Vom Designer kommt folgende Vorgabe für das Layout unseres Twitter-Clients: