3 Properties und Bindings

Eine Aufgabe einer Benutzeroberfläche ist es, den Zustand von Datenobjekten darzustellen und dem Benutzer des Programms die Möglichkeit zu geben, diesen Zustand zu verändern. Der Benutzer bedient zum Beispiel einen Schieberegler, der die Breite eines Rechtecks regelt, und in Abhängigkeit vom eingestellten Wert muss der »width«-Wert des Datenmodells aktualisiert und die Berechnung der Rechtecksfläche neu angestoßen werden. Das kann eine Menge Code erfordern.

In einigen Programmiersprachen gibt es deshalb das Konzept von *Properties*, um Eigenschaften eines Datenobjekts zu repräsentieren, und *Bindings*, um Abhängigkeiten dieser Properties zu deklarieren. Die Synchronisierung funktioniert dann automatisch. In Java fehlen bislang echte Properties und Bindings, sodass man dafür entweder auf Programmbibliotheken von Drittparteien zurückgreifen oder sehr viel Code schreiben muss.

JavaFX führt nun endlich Properties und Bindings ein, die diese Grundaufgabe einer Benutzeroberfläche erheblich erleichtern. Diese Properties werden Ihnen überall in JavaFX begegnen. Jeder Node hat praktischerweise eine Vielzahl dieser Properties, was die Programmierung sehr erleichtert. Properties und Bindings sind ein neues Feature, mit dem JavaFX die Standard-APIs von Java erweitert. Die Anwendung ist dabei nicht auf Programme beschränkt, die ein JavaFX-GUI haben.

3.1 Beans und Properties

3.1.1 Klassische JavaBean-Properties

Sehen wir uns zunächst einmal an, wie Properties bislang modelliert wurden und wie sich JavaFX-Properties davon unterscheiden.

```
public class MyBean implements Serializable {
   private String name;
   public static final String PROP_NAME = "name";
```

```
    public String getName() {
      return name;
    }

    public void setName(String name) {
      String oldName = this.name;
      this.name = name;
      propertyChangeSupport.firePropertyChange(PROP_NAME, oldName, name);
  }

    private transient final PropertyChangeSupport propertyChangeSupport = new
  PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
      propertyChangeSupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
      propertyChangeSupport.removePropertyChangeListener(listener);
    }
  }
```

Listing 3–1 *Eine klassische JavaBean mit einer Property*

Bei einer klassischen JavaBean sind die Properties mithilfe von privaten Feldern realisiert. Auf den Wert der Felder greift man über Getter- und Setter-Methoden zu, die einer einfachen Namenskonvention folgen. Auf das Präfix `get` oder `set` folgt der Feldname beginnend mit einem Großbuchstaben. Soll auf Änderungen reagiert werden, muss die Bean eine Methode zur Verfügung stellen, um PropertyChangeListener hinzuzufügen. Bei Änderungen des Wertes über die Setter-Methode, ist diese dann dafür verantwortlich, die Listener zu informieren. Dabei wird der Listener nicht für ein bestimmtes Property registriert, sondern für alle Properties einer Bean. Er muss dann selbst anhand des Namens der Property prüfen, ob das entsprechende Event tatsächlich interessant ist oder ob er es ignorieren kann. Das ist nicht gerade elegant gelöst und zudem fehleranfällig.

3.1.2 Die neuen JavaFX-Properties

In JavaFX werden Properties etwas anders umgesetzt.

```
  public class MyBean implements Serializable {
    private final StringProperty sample = new SimpleStringProperty();

    public String getSample() {
      return sample.get();
    }

    public void setSample(String value) {
      sample.set(value);
    }
```

```
    public StringProperty sampleProperty() {
      return sample;
    }
}
```

Listing 3–2 *Eine JavaFX-Bean mit einer Property*

Der Code ist deutlich übersichtlicher geworden. Setter und Getter behalten dieselbe Methodensignatur. Aber intern verwenden sie ein `StringProperty`-Objekt, um den Wert zu lesen und zu schreiben. Der Boilerplate-Code zum Aktualisieren der Listener ist weggefallen. Das übernimmt die StringProperty nämlich selbst. Alles, was wir gegenüber dem Beans-Modell lernen müssen, ist eine neue Namenskonvention: Der Name des Getters für die Property beginnt mit deren Namen gefolgt von »Property«. Der Getter für die Property mit dem Namen »age« hieße also zum Beispiel »ageProperty«. Für unsere sample-Property heißt die Methode also `sampleProperty`. Wenn Sie diese Bean in die Hände bekommen, dann können Sie darauf einen Listener registrieren:

```
myBean.sampleProperty().addListener(someListener);
```

Der Listener wird also für ein bestimmtes Property registriert und nicht für alle Properties der Bean. Dadurch brauchen wir keine Fallunterscheidung mehr. JavaFX-Properties sind also ein deutlicher Fortschritt gegenüber dem alten Programmiermodell.

3.1.3 Was sind die wichtigsten Klassen und Interfaces?

Bevor wir ins Detail gehen, sehen wir uns nun kurz an, welche Interfaces und APIs man kennen sollte und wie sie organisiert sind. Die wichtigsten Klassen und Interfaces, um mit Properties und Bindings zu arbeiten, stecken in den Packages `javafx.beans`, `javafx.beans.value`, `javafx.collections` und `javafx.beans.property`.

javafx.beans

In `javafx.beans` ist das Basis-Interface `Observable` definiert. Dieses Interface ist als Wrapper für einen Inhalt gedacht und erlaubt es, `InvalidationListener` zu registrieren. Wird der Inhalt durch eine Änderung ungültig, sodass sich abhängige Werte aktualisieren sollten, wird im registrierten Listener die Methode `invalidated` aufgerufen:

```
longProperty.addListener(new InvalidationListener() {

    @Override
    public void invalidated(Observable o) {
      // Reaktion auf das Ereignis
    }
});
```

Listing 3–3 *Hinzufügen eines Listeners*

Da aber das Interface keinen Zugriff auf den Wert selbst bietet, arbeiten wir stattdessen meist mit dem Subinterface `ObservableValue` oder davon abgeleiteten Interfaces und Implementierungen.

javafx.beans.value

Ein `ObservableValue` fungiert als Wrapper für einen Wert, der mit `getValue` abgefragt werden kann. Im Package `javafx.beans.value` gibt es jeweils typspezifische Varianten dazu. Damit lässt sich der Wert des »ungültigen« Objekts nun auch ermitteln:

```
longProperty.addListener(new InvalidationListener() {

  @Override
  public void invalidated(Observable o) {
    Long value = ((ObservableLongValue)o).get();
    System.out.println("value "+value);
  }
});
```

Zusätzlich erweitert das Interface die API um Methoden zum Registrieren von `ChangeListenern`. Die Unterschiede zwischen den Listenern, die auf den ersten Blick sehr ähnlich sind, sehen wir uns später noch genauer an. Wenn man einen `ObservableValue` überschreibbar machen möchte, kann man dazu das Interface `WriteableValue` implementieren, das dafür die Methode `setValue anbietet`.

javafx.beans.property

Jetzt kommen wir zu den Properties selbst. Diese sind in dem Package `javafx.beans.property` definiert. Hier gibt es zwei neue Interfaces `ReadOnlyProperty` und `Property`. Die `ReadOnlyProperty` fügt dem `ObservableValue` die Methoden `getBean` und `getName` hinzu und erlaubt, wenig überraschend, nur lesenden Zugriff auf den verwalteten Wert. `Property` hingegen leitet von `ReadOnlyProperty` ab und implementiert zusätzlich das Interface `WriteableValue`. Damit ist schreibender Zugriff möglich. So kann der Wert nun auch an andere `ObservableValues` gebunden werden, und ein automatisiertes Aktualisieren in Abhängigkeit von anderen Werten über *Binding* wird möglich. Zudem enthält dieses Package typspezifische Implementierungen dieser Interfaces.

javafx.collections

Bislang haben wir uns die Klassen für Einzelwerte angesehen. In JavaFX sind aber auch die `Collections` observierbar. `ObservableList`, `ObservableMap` und `ObservableSet` leiten jeweils vom `Observable` Interface ab und fügen jeweils Methoden hinzu, um die passenden `ChangeListener` zu registrieren. Das sind `ListChangeListener`, `MapChangeListener` und `SetChangeListener`. Die entsprechenden Collections

legt man mit der Helfer-Klasse `FXCollections` an, die viele Methoden bietet, um observierbare Collections zu erzeugen und zu manipulieren.

3.1.4 Wie legt man Properties an?

Einfache Properties

Im Package `javafx.beans.property` gibt es die Properties, die Primitive und Strings repräsentieren: `BooleanProperty`, `DoubleProperty`, `FloatProperty`, `IntegerProperty`, `LongProperty und StringProperty`. Das sind jeweils abstrakte Klassen. Um Instanzen zu erzeugen, gibt es jeweils eine Implementierung, deren Name mit »Simple« beginnt, also zum Beispiel `SimpleBooleanProperty`:

```
BooleanProperty booleanProperty = new SimpleBooleanProperty(true, „b", this);
DoubleProperty doubleProperty = new SimpleDoubleProperty(1.5, „d", this);
FloatProperty floatProperty = new SimpleFloatProperty(1.5f, „f", this);
IntegerProperty integerProperty = new SimpleIntegerProperty(123, „i", this);
LongProperty longProperty = new SimpleLongProperty(12345678991, „l", this);
StringProperty stringProperty = new SimpleStringProperty("hallo", „s", this);
```

Im Konstruktor übergeben wir hier drei optionale Werte: den initialen Wert, den Namen der Property und die zugehörige Bean. Es gibt jeweils Varianten des Konstruktors, die es erlauben, die optionalen Werte nicht zu setzen:

```
BooleanProperty booleanProperty = new SimpleBooleanProperty();
BooleanProperty booleanProperty = new SimpleBooleanProperty(true);
BooleanProperty booleanProperty = new SimpleBooleanProperty(true, „b");
BooleanProperty booleanProperty = new SimpleBooleanProperty(true, „b", this);
```

Der Wert der Property lässt sich über die `setValue`-Methode auch noch nachträglich setzen. Name und Bean sind jedoch finale Werte, die nur im Konstruktor übergeben werden können.

ObjectProperty

In einer ObjectProperty lassen sich beliebige Objekte speichern. Das wird zum Beispiel in der `ImageView` verwendet, um das dargestellte `Image` zu verwalten. Wir werden gleich im Abschnitt über Bindings ein Beispiel damit sehen. Normalerweise reicht uns auch hier die Standardimplementierung für das Erzeugen der Properties:

```
ObjectProperty<Image> objectProperty = new SimpleObjectProperty<>(img,"img",
this);
```

3.1.5 Wie findet man die Bean zu einer Property?

Manchmal ist es notwendig, zu einer Property die zugehörige Bean zu ermitteln. Im Quellcode von JavaFX finden sich zahlreiche Beispiele dafür. So muss etwa bei

einem Accordion-Control überwacht werden, welche TitledPane den Eingabefokus hat. Das wird gemacht, indem man denselben ChangeListener auf die focusedProperty jeder TitledPane setzt. Wenn eine TitledPane nun den Fokus erhält, feuert der ChangeListener, sucht sich die passende TitledPane und rückt sie in den Fokus:

```
private final ChangeListener<Boolean> paneFocusListener = new
ChangeListener<Boolean>() {
   @Override public void changed(ObservableValue<? extends Boolean>
observable, Boolean oldValue, Boolean newValue) {
     if (newValue) {
       final ReadOnlyBooleanProperty focusedProperty =
(ReadOnlyBooleanProperty) observable;
       final TitledPane tp = (TitledPane) focusedProperty.getBean();
       focus(accordion.getPanes().indexOf(tp));
     }
   }
};
```

Die Methode getBean(), die hier verwendet wird, ist im Interface ReadOnlyProperty definiert. Deshalb ist hier das Casten vom ObservableValue zur ReadOnlyProperty notwendig. Das funktioniert natürlich nur auf Properties, bei denen die Bean gesetzt ist. Hat der Erzeuger der Property den falschen Konstruktor verwendet und keine Bean übergeben, haben wir Pech gehabt.

3.1.6 Wie werden Properties schreibgeschützt?

Die Properties, die wir bislang verwendet haben, sind alle les- und schreibbar. Wenn wir aber nicht wollen, dass eine Eigenschaft der Bean von außen geändert werden kann, reicht es nicht, die Setter-Methode wegzulassen. Der Benutzer unserer Bean könnte sich einfach die Property holen und den Wert mit setValue überschreiben. In diesem Falle verwenden wir das Superinterface ReadOnlyProperty:

```
public class Person {
   private final ReadOnlyStringProperty name;

   public final String getName(){
     return name.getValue();
   }

   public final ReadOnlyStringProperty nameProperty(){
     return name;
   }
}
```

Jetzt müssen wir die ReadOnlyProperty nur noch erzeugen. Das geht ganz einfach, denn Property leitet von ReadOnlyProperty ab. Machen wir das also im Konstruktor:

```
public Person(String name) {
   this.name = new SimpleStringProperty(name);
}
```

Fällt Ihnen dabei etwas auf? Das ist zwar gültiger Code, aber unser eigentliches Problem wird dadurch nicht wirklich gelöst. Jeder, der eine Instanz unserer Bean bekommt, kann sich nun die Property holen, zu einer `StringProperty` casten und fröhlich den Wert ändern. Sie sollten stattdessen eine der eigens dafür gemachten Wrapper-Klassen nutzen:

```
public class Person {
   private ReadOnlyStringWrapper name;

   public Person(String name) {
     this.name = new ReadOnlyStringWrapper(name);
   }

   public final String getName(){
     return name.getValue();
   }

   public final ReadOnlyStringProperty nameProperty(){
     return name.getReadOnlyProperty();
   }
}
```

Ganz wichtig dabei ist, dass Sie den Wrapper nicht direkt verfügbar machen, denn sonst haben Sie nichts gewonnen. Die Wrapper-Klassen selbst sind ebenfalls Properties. Ein Benutzer könnte wie im vorigen Beispiel nach einem Typecast den Wert der Property ändern. Alle `ReadOnlyWrapper` haben aber die Methode `getReadOnlyProperty`, die eine sichere `ReadOnlyProperty` zurückgibt. So können Sie selbst innerhalb der Bean den Wert ändern, der Benutzer hat jedoch keine Chance mehr, durch einen Typecast den Wert zu verändern.

3.2 Wie verwendet man Bindings?

Wir haben gesehen, dass man auf einem Property Listener registrieren kann. Das ist weiter nicht besonders überraschend, denn das konnte man mit dem alten Bean-Modell ja auch. Wirklich interessant wird das Ganze, wenn man stattdessen die neuen Bindings verwendet, um Werte direkt aneinanderzubinden. In den meisten Fällen reagiert man auf die Änderung einer Property, indem man in Abhängigkeit von der Wertänderung die Werte einer oder mehrerer anderer Properties anpasst. In einem Listener ist das relativ umständlich. Viel direkter und mit weniger Code geht das mithilfe von Bindings. Betrachten wir dazu ein einfaches Beispiel mit unserer MyBean. Wenn die `sampleProperty` sich ändert, wollen wir den Text des Labels *nameLabel* im UI ändern. So würde das mit einem Listener aussehen:

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);
```

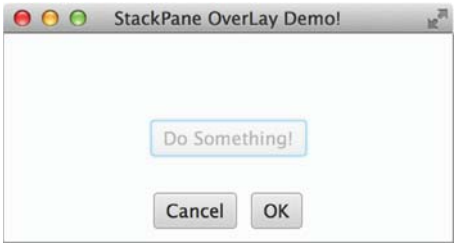

Abb. 5–4 *Eine StackPane kann verwendet werden , um die Anwendung in einzelne Layers aufzuteilen*

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);
```

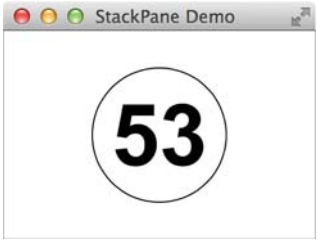

Abb. 5–5 *Mit der StackPane lassen sich Nodes einfach stapeln.*

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);
```

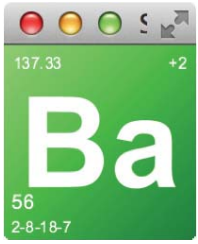

Abb. 5–6 *Nodes können bei der StackPane nicht nur zentriert gestapelt werden.*

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));
}
```

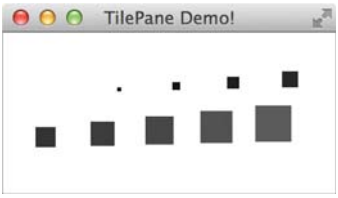

Abb. 5–7 *Die Größe der Tiles in der TilePane wird durch das höchste und das breiteste Element bestimmt.*

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);
```

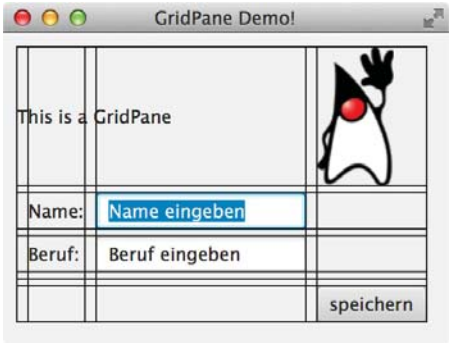

Abb. 5–9 *Die GridPane mit eingeblendeten Gitterlinien*

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);
```

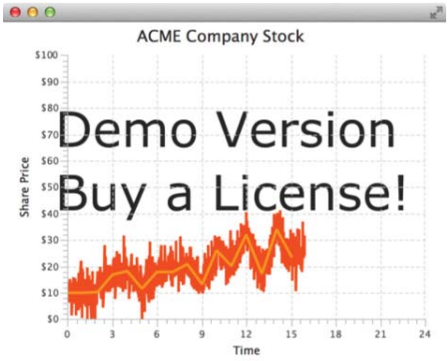

Abb. 5–11 *Unmanaged Nodes werden beim Layout vom Layoutcontainer ignoriert.*

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);
    }
  }
```

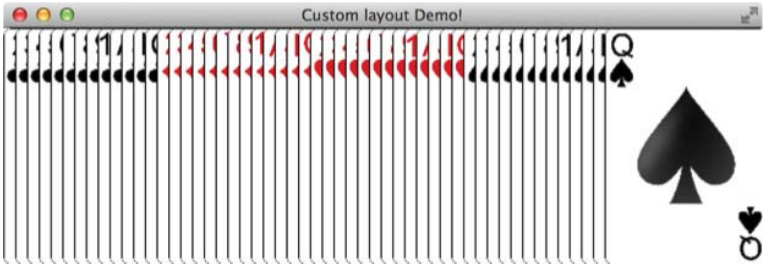

Abb. 5–12 *CardStackLayout, ein eigener Layoutcontainer*

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);
```

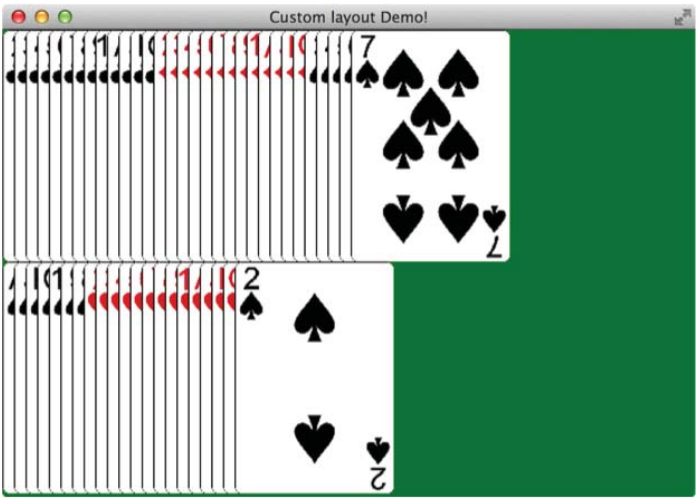

Abb. 5–13 *Layoutcontainer werden automatisch aktualisiert. So können mit wenig Aufwand dynamisch Nodes positioniert werden.*

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:

8 Charts erstellen

In diesem Kapitel werden wir Charts erstellen. Während die Standard-Swing-API noch über keine Chart-Komponente verfügt, hat das Entwicklerteam JavaFX eine sehr reichhaltige Standardbibliothek spendiert. Charts sind dabei nichts anderes als spezialisierte Nodes, die dem Layout hinzugefügt werden. Momentan kann man aus neun verschiedenen Charts auswählen.

Sie werden in diesem Kapitel lernen, wie man einfache Charts erzeugt, welche verschiedenen Chart-Typen zur Verfügung stehen, wie Sie das Erscheinungsbild anpassen können, wie Charts kontinuierlich dynamische Daten darstellen können und sogar wie man Charts erweitern kann, um eigene Animationen einzufügen.

8.1 Ein Diagramm anzeigen

Zeigen wir zunächst einmal ein einfaches `LineChart` an. Anschließend werden wir uns den Code etwas genauer ansehen:

```
NumberAxis xAxis = new NumberAxis("Zeit", 0, 40, 5);
NumberAxis yAxis = new NumberAxis("Preis", 0, 80, 10);
ObservableList<XYChart.Series<Double,Double>> lineChartData =
FXCollections.observableArrayList(
   new LineChart.Series<Double,Double>("Orakel",
            FXCollections.observableArrayList(
     new XYChart.Data<Double,Double>(0.0, 10.0),
     new XYChart.Data<Double,Double>(10.0, 34.4),
     new XYChart.Data<Double,Double>(20.0, 31.9),
     new XYChart.Data<Double,Double>(30.0, 42.3),
     new XYChart.Data<Double,Double>(40.0, 57.7)
   )),
   new LineChart.Series<Double,Double>("Essape",
             FXCollections.observableArrayList(
     new XYChart.Data<Double,Double>(0.0, 57.7),
     new XYChart.Data<Double,Double>(10.0, 42.3),
     new XYChart.Data<Double,Double>(20.0, 33.9),
```

```
        new XYChart.Data<Double,Double>(30.0, 21.7),
        new XYChart.Data<Double,Double>(40.0, 17.3)
      ))
);
LineChart chart = new LineChart(xAxis, yAxis, lineChartData);
```

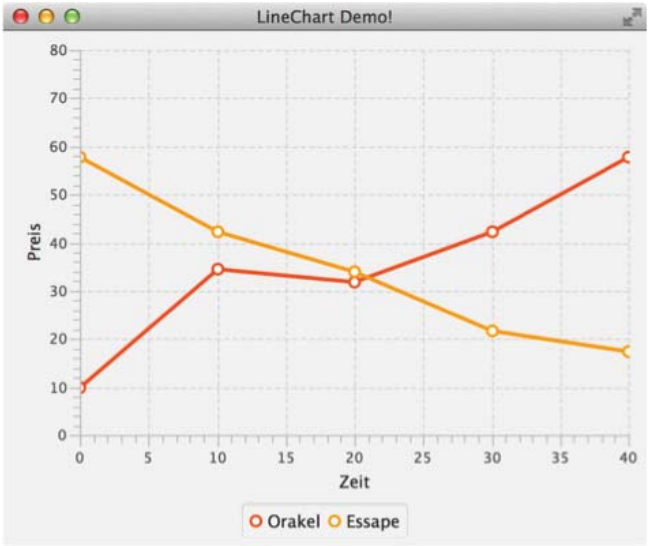

Abb. 8–1 *Ein einfaches LineChart*

Am Anfang definieren wir die beiden Achsen jeweils als `NumberAxis`, wobei wir dem Konstruktor Beschriftung, Ober- und Untergrenze sowie den Abstand der Markierungsbezeichner übergeben. Die einzelnen Datenpunkte einer Datenreihe werden dann jeweils in ein `XYChart.Series` gepackt. Anschließend werden diese beiden Reihen in eine `ObservableList` gesteckt. Diese dient dann als Datenmodell für das `LineChart`.

8.2 Aufbau der API

Alle Charts leiten von der Basisklasse `Chart` ab, die vor allem für das Zeichnen von Hintergrund, Legende und Titel verantwortlich ist. `XYCharts` erweitern diese Klasse, um die Fähigkeit, zwei Achsen darzustellen. Mit Ausnahme des `PieChart`, das keine Achsen benötigt und nur eindimensionale Daten darstellt, leiten alle anderen Klassen davon ab.

8.2.1 Achsen

Für die `XYCharts` werden eine x- und eine y-Achse definiert. Wir werden dabei vor allem mit der `NumberAxis` zu tun haben, deren Daten als Zahlen definiert sind, und der `CategoryAxis`, deren Wertzuordnung anhand von Strings erfolgt. Jeder String,

den wir der `CategoryAxis` übergeben, definiert dabei einen Markierungspunkt auf der Achse. In unserem ersten Beispiel verwenden wir nur `NumberAxis`. Wir haben diesen im Konstruktor einen Titel, Minimalwert, Maximalwert und den Abstand der Markierungsstriche (Tickmarken) übergeben. Verwenden wir stattdessen den parameterlosen Konstruktor und setzen keine Minimal- und Maximalwerte, so ermittelt die Achse diese selbst per »Autoranging« anhand der Daten. In unserem Beispiel wäre das:

```
NumberAxis xAxis = new NumberAxis();
NumberAxis yAxis = new NumberAxis();
```

Die einzelnen Properties der Achse lassen sich dann einfach über Setter-Methoden konfigurieren:

```
xAxis.setLabel("Zeit");
xAxis.setTickUnit(5);
```

Zusätzlich können wir auch noch angeben, dass trotz Autoranging die Null auf jeden Fall angezeigt werden soll:

```
xAxis.setForceZeroInRange(true);
```

Wen später neue Werte hinzukommen, die außerhalb des durch Autoranging bestimmten Bereichs liegen, passen sich die Achsen automatisch an. Dabei wird der Übergang animiert dargestellt, sodass der Benutzer die Veränderung leichter nachvollziehen kann. Diese Animation lässt sich jedoch auch abstellen. Um das zu testen, können Sie Ihrem Design einen Button hinzufügen, der einen neuen Wert außerhalb des bisherigen Wertebereichs einfügt:

```
Button button = new Button("Wert hinzufügen");
button.setOnAction(new EventHandler<ActionEvent>() {

   @Override
   public void handle(ActionEvent event) {
     lineChartData.get(0).getData().add(new XYChart.Data<Double,
       Double>(100.0, 17.3));
   }
});
```

Testen Sie nun das Verhalten bei Knopfdruck und deaktivieren Sie dann die Animation:

```
chart.setAnimated(false);
```

Vor allem wenn wir Livedaten betrachten, die sich häufig ändern, wie etwa einen Aktienkurs, bietet es sich an, diese Animation abzuschalten, da sie störend wirken kann.

Manche Chart-Typen, wie zum Beispiel die `BarCharts`, erwarten eine `ValueAxis` kombiniert mit einer `CategoryAxis`. Das ist erst mal verwirrend, weil egal ist, welche der Achsen von welchem Typ ist. Dadurch kann der Compiler einen Feh-

ler nicht erkennen, und eine falsche Kombination, wie zum Beispiel zwei numerische Achsen, wird erst zur Laufzeit in Form einer Exception gemeldet.

8.2.2 Daten

Die Daten eines `XYChart` sind in `XYChart.Series` organisiert. Das `XYChart` verwaltet die Series in einer `ObservableList`. Die Daten einer Serie oder Datenreihe sind in der Darstellung meist einheitlich gekennzeichnet, sodass die Zusammengehörigkeit erkennbar ist. In der Serie stecken dann die einzelnen Datenpunkte in Form von `XYChart.Data`. Wie der Name nahelegt, können wir einen x-Wert und einen y-Wert definieren. Ein zusätzlicher »extraValue« wird in `BubbleChart` verwendet, um die Größe der Blase anzugeben.

Neben den Daten kann dem Datenpunkt auch ein Node mitgegeben werden. Dieser Node wird dann in der Darstellung anstatt des diagrammspezifischen Standard-Nodes benutzt.

8.3 Welche Diagrammtypen gibt es?

Momentan gibt es sieben verschiedene Diagrammtypen, die jedoch um eigene Typen erweitert werden können: `LineChart`, `PieChart`, `AreaChart`, `StackedAreaChart`, `BarChart`, `BubbleChart` und `ScatterChart`.

8.3.1 LineChart und AreaChart

Das `LineChart` haben wir gerade in unserem ersten Beispiel verwendet. Es verbindet Daten einer Serie mit einer Linie. So können Trends leicht nachvollzogen werden. Eine klassische Anwendung für diesen Diagrammtyp ist das Aktien-Chart. `LineCharts` haben eine x- und eine y-Achse. In unserem Beispiel haben wir eine `NumberAxis` verwendet. Mit einer `CategoryAxis` können Sie statt der Zahlen andere Werte als »Tickmarken« anzeigen. Tauschen Sie dazu die x-Achse im Beispiel aus:

```
CategoryAxis xAxis = new CategoryAxis();
NumberAxis yAxis = new NumberAxis("Preis", 0, 80, 10);
ObservableList<XYChart.Series<String,Double>> lineChartData =
FXCollections.observableArrayList(
   new LineChart.Series<String,Double>("Orakel",
      FXCollections.observableArrayList(
     new XYChart.Data<String,Double>("Jan", 10.0),
     new XYChart.Data<String,Double>("Feb", 34.4),
     new XYChart.Data<String,Double>("Mär", 31.9),
     new XYChart.Data<String,Double>("Apr", 42.3),
     new XYChart.Data<String,Double>("Jun", 57.7)
   )),//... Analog für die zweite Datenreihe
);
```

Wie Sie sehen, müssen wir noch die Datentypen im Datenmodell anpassen, dann übernimmt das Chart die Bezeichnungen für die Tickmarken aus dem Datenmodell.

Das `AreaChart` unterscheidet sich von dem `LineChart` dadurch, dass die Flächen zwischen x-Achse und den Verbindungslinien zwischen den Datenpunkten einer Zeitreihe farbig ausgefüllt sind. Man unterscheidet zwischen gestapelten und überlappenden Flächendiagrammen. Das überlappende Flächendiagramm dient demselben Zweck wie das Liniendiagramm und macht es einfacher, Trends zu verfolgen, da zusammengehörige Datenpunkte durch eine gleichmäßig eingefärbte Fläche dargestellt werden. Löschen wir in unserem Beispiel die Zeile, in der das LineChart aufgebaut wird:

```
LineChart chart = new LineChart(xAxis, yAxis, lineChartData);
```

Danach ersetzen wir diese durch ein `AreaChart`:

```
AreaChart chart = new AreaChart(xAxis, yAxis, lineChartData);
```

So erhalten wir ein überlappendes Flächendiagramm.

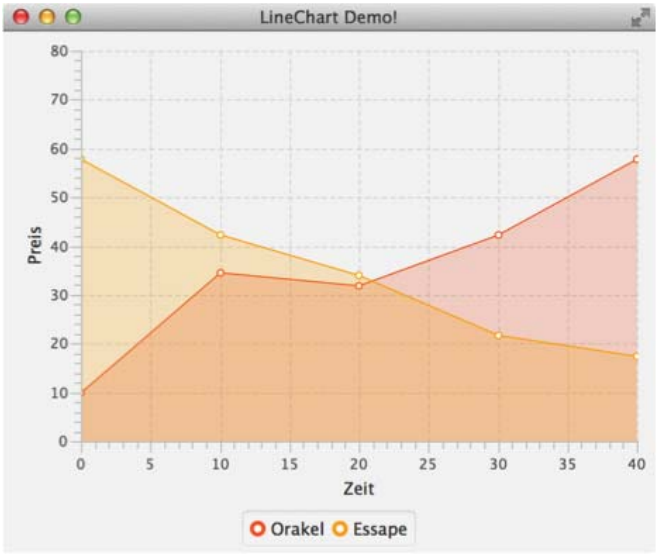

Abb. 8–2 *Die gleichen Daten wie in Abbildung 8–1, jetzt als AreaChart dargestellt*

Der Verlauf der Linien ist gleich, lediglich die Flächen darunter sind nun mit transparenter Farbe ausgefüllt. Wo sich die Flächen überlappen, entsteht ein Mischton. Um ein gestapeltes Diagramm zu erhalten, verwenden wir stattdessen folgenden Code:

```
StackedAreaChart chart = new StackedAreaChart(xAxis, yAxis, lineChartData);
```

Abb. 8–3 *Das StackedAreaChart addiert die Werte.*

Die Datenkurve für unsere fiktive Firma »Orakel« ist gleich geblieben. Die Datenwerte für die Firma »Essape« haben sich jedoch verändert. Der y-Wert der einzelnen Datenpunkte ist nun nicht mehr der übergebene Wert. Stattdessen wurde der übergebene Wert zum Wert der zuvor hinzugefügten Kurve hinzuaddiert. Zum Vergleich unserer »Aktienkurse« ist dieser Diagrammtyp damit ungeeignet. Man kann ihn hingegen sehr gut dafür verwenden, um die zeitliche Entwicklung der Zusammensetzung einer Gesamtmenge zu verdeutlichen. Die oberste Linie repräsentiert die kumulierten Werte der einzelnen Datenpunkte, wie in folgendem Beispiel, das fiktive Verkaufszahlen darstellt:

```
NumberAxis xAxis = new NumberAxis("Zeit", 1981, 1985, 1);
NumberAxis yAxis = new NumberAxis("Verkaufte Einheiten", 0, 160, 10);
yAxis.setTickLabelFormatter(new
NumberAxis.DefaultFormatter(yAxis,null,"k")));
ObservableList<XYChart.Series<Integer, Double>> lineChartData =
      FXCollections.observableArrayList(
    new LineChart.Series<Integer, Double>("C64",
         FXCollections.observableArrayList(
      new XYChart.Data<Integer, Double>(1981, 0.0),
      new XYChart.Data<Integer, Double>(1982, 10.0),
      new XYChart.Data<Integer, Double>(1983, 34.4),
      new XYChart.Data<Integer, Double>(1984, 31.9),
      new XYChart.Data<Integer, Double>(1985, 42.3)
    )),
    new LineChart.Series<Integer, Double>("ZX Spectrum",
         FXCollections.observableArrayList(
      new XYChart.Data<Integer, Double>(1981, 0.0),
      new XYChart.Data<Integer, Double>(1982, 57.7),
      new XYChart.Data<Integer, Double>(1983, 42.3),
```

```
    new XYChart.Data<Integer, Double>(1984, 33.9),
    new XYChart.Data<Integer, Double>(1985, 21.7)
  ))
  // weitere Daten...
  }
);
StackedAreaChart chart = new StackedAreaChart(xAxis, yAxis,
                    lineChartData);
chart.setTitle("Heimcomputer-Verkaufszahlen in Deutschland");
```

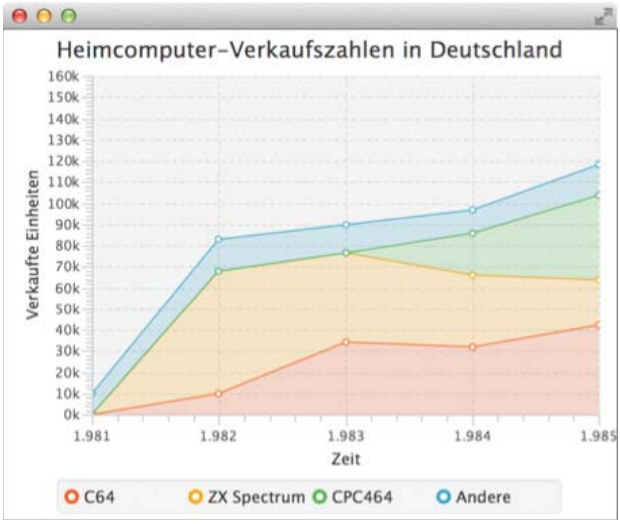

Abb. 8–4 *Das StackedAreaChart verdeutlicht die Zusammensetzung einer Gesamtmenge.*

In diesem Diagrammtyp lassen sich also gleichzeitig die Gesamtentwicklung der Verkaufszahlen aller Geräte und der Beitrag der einzelnen Modelle darstellen. Mit `setTitle` haben wir dem Chart zusätzlich eine Überschrift gegeben, und mittels eines `StringConverter` wird dem rohen y-Wert ein »k« angehängt, um zu verdeutlichen, dass es sich um Tausender handeln soll. Dazu bringt die Klasse `NumberAxis` praktischerweise einen DefaultFormatter mit, dem wir ein Präfix und ein Suffix übergeben können.

8.3.2 PieChart

`PieCharts` oder Tortendiagramme werden meist verwendet, um den prozentualen Anteil einer Kategorie an einer Gesamtmenge zu verdeutlichen. Sie haben im Unterschied zu `LineChart` und `AreaChart` nur eine Dimension und keine Achsen. Daher sieht hier der Code ein wenig anders aus:

```
PieChart pieChart = new PieChart(FXCollections.observableArrayList(
   new PieChart.Data("Java", 19),
   new PieChart.Data("C", 16),
   new PieChart.Data("C++", 9),
   new PieChart.Data("PHP", 7),
   new PieChart.Data("C#", 6),
   new PieChart.Data("Visual Basic", 5),
   new PieChart.Data("Python", 5),
   new PieChart.Data("Other", 33)
));
pieChart.setId("BasicPie");
pieChart.setTitle("TIOBE Index");
```

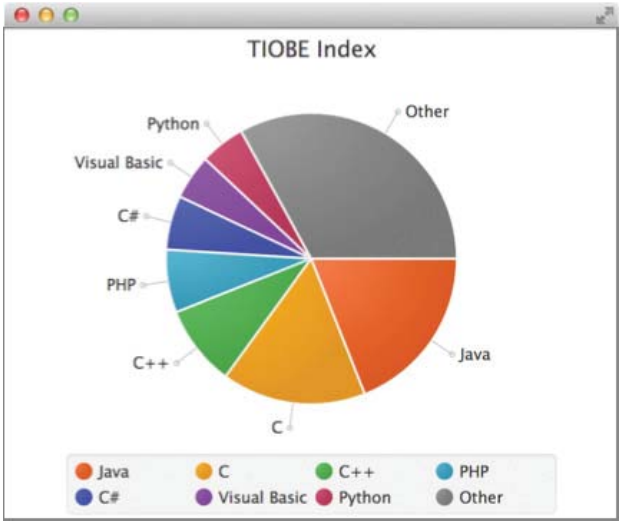

Abb. 8–5 *Tortendiagramme werden zur Anzeige eindimensionaler Daten verwendet.*

Statt der `XYChart.Data` für zweidimensionale Daten werden `PieChart.Data` verwendet, um jeweils Name und Wert eines Datensatzes festzulegen.

8.3.3 BarChart

Balken- und Säulendiagramme werden mithilfe der Klasse `BarChart` erstellt. Sie werden verwendet, um Daten verschiedener Kategorien zu vergleichen. Die Klasse erwartet eine `Number` oder `ValueAxis` kombiniert mit einer `CategoryAxis`. Übergibt man im Konstruktor zum Beispiel zwei `NumberAxis`, so bekommt man zur Laufzeit eine Exception. `BarCharts` eignen sich am besten dann, wenn nur wenige Datenpunkte von eindimensionalen oder zweidimensionalen Werten dargestellt werden sollen.

Für die Erstellung eines eindimensionalen `BarChart` übergeben wir einfach der `CategoryAxis` nur einen einzigen Wert:

```
CategoryAxis xAxis = new CategoryAxis();
NumberAxis yAxis = new NumberAxis();
BarChart<String,Number> bc = new BarChart<String,Number>(xAxis,yAxis);
xAxis.setCategories(FXCollections.<String>observableArrayList(""));
xAxis.setTickMarkVisible(false);
yAxis.setLabel("Performance");
XYChart.Series<String,Number> series1 = new XYChart
   .Series<String,Number>("JavaFX",FXCollections.
    observableArrayList(new XYChart.Data<String,Number>("",
      956)));
XYChart.Series<String,Number> series2 = new XYChart
   .Series<String,Number>("Swing",
    FXCollections.observableArrayList(new XYChart
      .Data<String,Number>("", 789)));
XYChart.Series<String,Number> series3 = new
XYChart.Series<String,Number>("JavaScript",
   FXCollections.observableArrayList(new XYChart
    .Data<String,Number>("", 312)));
```

Die Darstellung der Tickmarke ergibt an dieser Stelle wenig Sinn, deshalb stellen wir sie mit `setTickMarksVisible(false)` ab.

Zweidimensionale `BarCharts` sind vor allem dann gefragt, wenn die Daten der einzelnen Erhebungspunkte keine Zeitreihe darstellen und die Verbindung der Datenpunkte wie etwa in einem `LineChart` keinen Sinn ergeben würde. So werden sie zum Beispiel gerne bei Performance-Vergleichen verwendet, wenn die Ergebnisse mehrerer unabhängiger Tests in einem einzelnen Diagramm dargestellt werden sollen. Wir können unsere fiktiven Performance-Testdaten ganz einfach erweitern, indem wir der `CategoryAxis` mehrere Kategorien übergeben und dafür jeweils Datenpunkte erzeugen:

```
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
final BarChart<String, Number> bc = new BarChart<String, Number>
    (xAxis, yAxis);
yAxis.setTickLabelFormatter(new NumberAxis.DefaultFormatter(yAxis,
    null, " parsecs"));
xAxis.setCategories(FXCollections.<String>observableArrayList(
    "BubbleTrouble", "Kessel-Run"));
xAxis.setTickMarkVisible(false);
yAxis.setLabel("Performance (less is better)");
ObservableList<XYChart.Series<String, Number>> data =
   FXCollections.observableArrayList(
    new XYChart.Series<String, Number>("JavaFX",
      FXCollections.observableArrayList(
         new XYChart.Data<String, Number>("BubbleTrouble", 17, 9),
         new XYChart.Data<String, Number>("Kessel-Run", 11, 87))),
    new XYChart.Series<String, Number>("Swing",
      FXCollections.observableArrayList(
         new XYChart.Data<String, Number>("BubbleTrouble", 29, 1),
         new XYChart.Data<String, Number>("Kessel-Run", 33, 2))),
```

```
    new XYChart.Series<String, Number>("JavaScript",
      FXCollections.observableArrayList(
          new XYChart.Data<String, Number>("BubbleTrouble", 100),
          new XYChart.Data<String, Number>("Kessel-Run", 112))));
bc.setData(data);
```

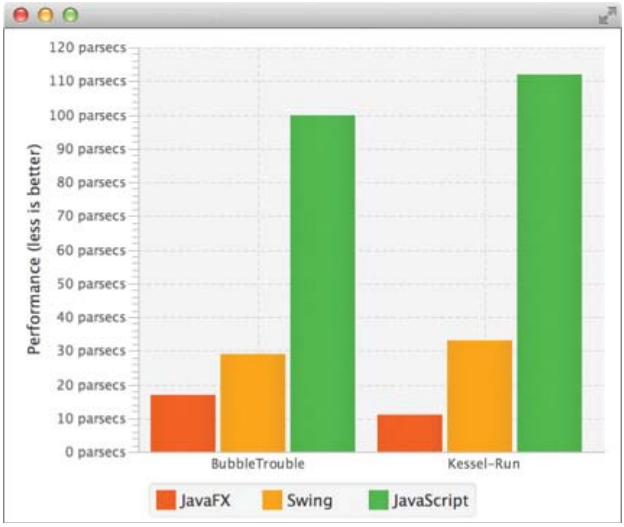

Abb. 8–6 *Ein BarChart mit zwei Kategorien*

8.3.4 ScatterChart

Streudiagramme oder `ScatterCharts` stellen Datenpunkte einer `XYChart.Series` in einer Punktwolke an der angegebenen xy-Position dar. Jeder Serie wird dabei zusätzlich ein Symbol zugewiesen, sodass die Zugehörigkeit eines Datenpunkts zu einer Serie leichter zu erkennen ist. Dieser Diagrammtyp lässt sich zur Korrelationsanalyse nutzen, um zum Beispiel Cluster zu erkennen.

```
NumberAxis xAxis = new NumberAxis();
NumberAxis yAxis = new NumberAxis();
ScatterChart<Number, Number> sc =
    new ScatterChart<Number, Number>(xAxis, yAxis);
xAxis.setLabel("X Axis");
yAxis.setLabel("Y Axis");
for (int s = 1; s < 5; s++) {
   XYChart.Series<Number, Number> series =
     new XYChart.Series<Number, Number>();
   series.setName("Data Series " + s);
   for (int i = 0; i < 30; i++) {
     series.getData().add(
       new XYChart.Data<Number, Number>(
          s % 2 == 0 ? 50 +(Math.random() * 98) / s :
             (Math.random() * 98)/ s,
```

```
            s % 2 == 0 ? 50 +(Math.random() * 98) / s :
                (Math.random() * 98)/ s));
    }
    sc.getData().add(series);
```

Abb. 8–7 *In ScatterCharts lassen sich gut Cluster erkennen.*

8.3.5 BubbleChart

Mithilfe von Blasendiagrammen lassen sich in einem zweidimensionalen Diagramm drei abhängige Merkmale darstellen. Die dritte Dimension wird dabei durch die Größe des Datenpunktes abgebildet. Der Wert wird als »extraValue« einfach dem `XYData`-Objekt im Konstruktor übergeben:

```
double SCALING_FACTOR = 2.5;
final NumberAxis xAxis = new NumberAxis("Entfernug zur Sonne",0, 5000, 500);
final NumberAxis yAxis = new NumberAxis("Einwohnerzahl (Millionen)",
   -500, 7000, 1000);

final BubbleChart<Number, Number> blc = new BubbleChart<Number, Number>(xAxis,
yAxis);

XYChart.Series merkur = new XYChart.Series("Merkur",
   FXCollections.observableArrayList(new XYChart.Data(58, 0,
     4.8*SCALING_FACTOR)));
XYChart.Series venus = new XYChart.Series("Venus",
   FXCollections.observableArrayList(new XYChart.Data(108 , 0,
     12.4*SCALING_FACTOR)));
XYChart.Series erde = new XYChart.Series("Erde",
   FXCollections.observableArrayList(new XYChart.Data(150, 6800,
     12.7*SCALING_FACTOR)));
XYChart.Series mars = new XYChart.Series("Mars",
```

```
    FXCollections.observableArrayList(new XYChart.Data(280, 0,
      6.8*SCALING_FACTOR)));
XYChart.Series jupiter = new XYChart.Series("Jupiter",
    FXCollections.observableArrayList(new XYChart.Data(775, 0,
      142.8*SCALING_FACTOR)));
XYChart.Series saturn = new XYChart.Series("Saturn",
    FXCollections.observableArrayList(new XYChart.Data(1440, 0,
      120.8*SCALING_FACTOR)));
XYChart.Series uranus = new XYChart.Series("Uranus",
    FXCollections.observableArrayList(new XYChart.Data(2870, 0,
      47.6*SCALING_FACTOR)));
XYChart.Series neptun = new XYChart.Series("Neptun",
    FXCollections.observableArrayList(new XYChart.Data(4500, 0,
      44.6*SCALING_FACTOR)));
```

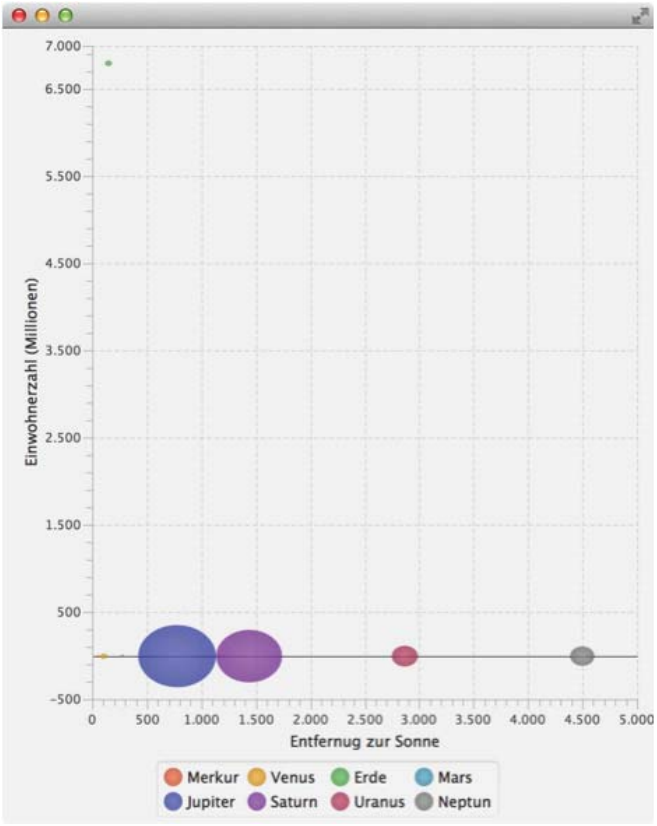

Abb. 8–8 *BubbleCharts haben die Blasengröße als dritte Dimension.*

8.4 Dynamische Charts

So wie wir die Charts bis jetzt verwendet haben, können wir sie gut für das Erstellen von Reports einsetzen. Sie sind aber genauso gut für die Darstellung veränderlicher Livedaten geeignet. Nehmen wir zum Beispiel einen fiktiven Aktienkurs:

```
NumberAxis xAxis = new NumberAxis();
final NumberAxis yAxis = new NumberAxis();
final LineChart<Number, Number> lineChart = new LineChart<Number,
Number>(xAxis, yAxis);
lineChart.setCreateSymbols(false);
lineChart.setAnimated(false);
xAxis.setForceZeroInRange(false);
XYChart.Series<Number, Number> dataSeries = new XYChart.Series<Number,
Number>();
lineChart.getData().add(dataSeries);
Timeline animation = new Timeline();
animation.getKeyFrames()
   .add(new KeyFrame(Duration.millis(100),
    new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent actionEvent) {
          dataSeries.getData().add(new XYChart
             .Data<Number, Number>(i++, Math.random() * 100));
          if (dataSeries.getData().size() > 30) {
             dataSeries.getData().remove(0);
          }
      }

    }
   ));
animation.setCycleCount(Animation.INDEFINITE);
animation.play();
```

Als Datenquelle verwenden wir einen Zufallsgenerator, der über eine `Timeline`-Animation jede Zehntelsekunde einen neuen Wert einfügt. Es werden jeweils maximal gleichzeitig 30 Werte angezeigt. Sobald diese Zahl erreicht ist, werden vom Anfang der Liste in gleicher Frequenz Werte gelöscht, sodass wir einen durchlaufenden Kurs erhalten, wie wir das von Aktienkursdarstellungen kennen. Da sich das Werteintervall der x-Achse kontinuierlich ändern soll, müssen wir mit `xAxis.setForceZeroInRange(false)` dafür sorgen, dass der Nullwert nicht mit angezeigt wird. Bei Aktienkursen ist es auch unüblich, dass ein Symbol am Datenpunkt dargestellt wird. Das schalten wir im Beispiel mit `lineChart.setCreateSymbols(false);` ab. Es ist vielleicht nicht sofort verständlich, weshalb wir die Animation abschalten mussten (`lineChart.setAnimated(false)`). Probieren Sie einfach mal aus, wie sich die Darstellung ändert, wenn Sie diese Zeile auskommentieren.

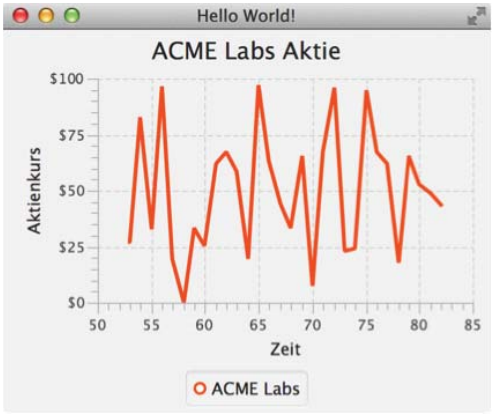

Abb. 8–9 *Ein Chart mit ständig aktualisierten Daten*