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: