Einführung in die Programmierung mit Java

Teil 11: Grafische Benutzerschnittstellen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

12. Dezember 2017

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 11: Grafische Benutzerschnittstellen

Motivation

- Eine grafische Benutzerschnittstelle (GUI) gestattet es, Eingaben durch Schaltknöpfe, Menüs, Schiebeschalter etc. mausgesteuert zu tätigen und Ausgaben in Fenstern zu präsentieren.
- Java stellt große Bibliotheken zur Gestaltung grafischer Benutzeroberflächen (GUIs) zur Verfügung. Name der Bibliothek: **JavaFX**.
- Javafx und alle anderen solchen Bibliotheken bauen sehr stark auf Vererbung auf.
- Eine Alternative zu Javafx, die auch standardmäßig zu Java gehört, heißt Swing.

Anwendungsbeispiel: Bankkonten

Wir möchten ein klickbares Fenster der folgenden Art.

Javafx Anwendungen

Will man in einer Anwendung Javafx verwenden, dann muss eine Javafx Anwendung geschrieben werden; man kann nicht Javafx von einer normalen Anwendung aus benutzen.

Javafx Anwendungen haben immer folgende Struktur:

```
import javafx.application.Application;
import javafx.stage.Stage;
public class BankkontoGUI extends Application {
    public void start(Stage primaryStage) {
      /* hier stehen alle Befehle */
      primaryStage.show();
    }
    public static void main(String[] args) {
      launch(args);
    }
}
```

Die Szene

Man kann nicht direkt ins Hauptfenster schreiben, sondern muss eine "Szene" definieren, welche den Inhalt des Hauptfensters ausschließlich seines Titels repräsentiert.

```
...
import javafx.scene.Scene;
import javafx.scene.control.TextField;
...
   public void start(Stage primaryStage) {
      TextField betragFeld = new TextField("Betrag");
      Scene scene = new Scene(betragFeld);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
```

Szenegraph

- Die Szene enthält nur ein anzeigbares Objekt, welches per Konstruktor übergeben wird.
- Um mehrere Objekte in die Szene zu stellen, verwendet man "Glasscheiben", `Pane`, welche mehrere Objekte, die selber wieder Glasscheiben sein können, enthalten.
- So entsteht ein Szenengraph, bzw. ein Szenenbaum, dessen Knoten, d.h. Verzweigungen, Glasscheiben sind und an dessen Blättern sich die eigentlichen grafischen Objekte befinden.
- Neben Textfeldern sind diese geometrische Objekte (Linien, Rechtecke, Ellipsen, etc), Knöpfe, Auswahlmenüs, Fortschrittsbalken, etc.

Verwendung der Glasscheiben (Panes)

```
...
import javafx.scene.layout.Pane;
        ...
TextField betragFeld = new TextField("Betrag");
TextField kontostandFeld = new TextField("Kontostand");
Pane scheibe = new Pane();
scheibe.getChildren().addAll(betragFeld,kontostandFeld);
       kontostandFeld.relocate(50,50);
Scene scene = new Scene(scheibe);
primaryStage.setScene(scene);
```

- Beachte die Indirektion über `getChildren()` beim Einstellen der Textfelder in die Glasscheibe.
- Leider liegen jetzt alle Textfelder übereinander und man sieht nur das zuletzt eingehängte `kontostandFeld`.

Manuelles Positionieren

Mit der `relocate()` Methode können einmal eingehängte Objekte nachträglich verschoben werden.

```
...
kontostandFeld.relocate(0,50);
...
```

Durch `addAll` wurde das Textfeld bei der Glasscheibe "angemeldet". Nachträgliches Verschieben wird an die Scheibe und von dort an Szene und Stage weitergemeldet. Mehr zu diesem "Weitermelden" später.

Layouts

- Das manuelle Positionieren ist zum einen mühsam und verträgt sich zum anderen schlecht mit Fenstern variabler Größe und unterschiedlichen Endgeräten.
- Günstiger ist die Verwendung von Panes mit Layout-Funktion. Es gibt u.a.
  - `FlowPane`, Fluss-Layout. Komponenten werden nacheinander eingesetzt.
  - `GridPane`, Gitter-Layout. Komponenten werden in ein tabellenartiges Gitter eingesetzt. Die Höhe und Breite der Zellen richtet sich nach den Inhalten.
  - `BorderPane`, Rand-Layout. Kann bis zu fünf Komponenten an den Positionen N,S,O,W und Mitte enthalten.
- Es gibt noch weitere solche Layout-behafteten Panes. Die einfache Pane ist nützlich für die Platzierung komplizierter geometrischer Objekte, etwa eines Diagramms.
- Natürlich können die einzelnen Komponenten einer solchen Pane selbst wieder Panes, ggf. mit Layout, sein.

Szenegraph unseres Fensters, I

Die “aktiven” Komponenten, hier Textfelder und Knöpfe, führen wir als Instanzvariablen:

```
import javafx.scene.control.Button;
...
    private TextField kontostandFeld;
    private Button einzahlKnopf;
    private Button abhebeKnopf;
    private TextField betragFeld;
...
    public void start(Stage primaryStage) {
...
```

Man beachte die Verwendung von Knöpfen (Buttons).

Szenegraph unseres Fensters, II

Die anderen können lokale Variablen der start-Methode sein, bzw. bei größeren Anwendungen in Hilfsmethoden geführt werden.

```
import javafx.scene.control.Label;
...
    public void start(Stage primaryStage) {
        FlowPane kontostandScheibe = new FlowPane();
        Label kontostandEtikett = new Label("Kontostand: ");
        kontostandScheibe.getChildren().
                addAll(kontostandEtikett,kontostandFeld);
        FlowPane betragScheibe = new FlowPane();
        Label betragEtikett = new Label("Betrag: ");
        betragScheibe.getChildren().
                addAll(betragEtikett,betragFeld);
```

Wir verwenden Etiketten (Labels) um die Textfelder zu beschriften und fassen Textfeld mit entsprechendem Etikett in einer FlowPane zusammen. Ebenso kommen die beiden Knöpfe in eine gemeinsame FlowPane (`knopfScheibe` (Übung)).

Szenegraph unseres Fensters, III

Schließlich werden alle drei FlowPanes untereinander in eine GridPane gestellt. Der `setVgap` Befehl erhöht den Zeilenabstand.

```
GridPane fensterInhalt = new GridPane();
fensterInhalt.setVgap(10);
fensterInhalt.add(kontostandScheibe,0,0);
fensterInhalt.add(betragScheibe,0,1);
fensterInhalt.add(knopfScheibe,0,2);
Scene scene = new Scene(fensterInhalt);
primaryStage.setScene(scene);
primaryStage.show();
```

Die GridPane bildet dann den Inhalt unserer Szene und damit des Fensters.

Aktionen und Ereignisse

Nun müssen wir unser GUI mit Leben füllen.
Den Inhalt der Textfelder kann man mit `getText` und `setText` auslesen und verändern.
Wie reagieren wir auf das Drücken der Knöpfe?

Früher. . .

. . . gab es zu jedem Knopf eine Methode (oder Funktion) mit Boole'schem Rückgabewert.
War gerade ein Knopf gedrückt, so war der Rückgabewert `true`; ansonsten `false`.
Im Hauptprogramm musste man dann ständig diese Methode aufrufen um ja keinen Knopfdruck zu verpassen.
Diese Methode, bezeichnet als *polling*, gilt inzwischen als überholt.
Stattdessen benutzt man **ereignisgesteuerte Eingabenbehandlung** (*event-driven*):

Aktionen und Ereignisse

Mit der Methode `setOnAction` kann man an einen Knopf einen **Behandler** (*action handler*) anheften.
Das ist ein Objekt, welches die Schnittstelle (Interface)

```
EventHandler<ActionEvent>
```

implementiert. Diese Schnittstelle enthält nur eine einzige Methode (Functional Interface):

```
public void handle(ActionEvent e)
```

Der *action handler* muss also so eine Methode implementieren.
Ist der *action handler* an einen Knopf geheftet, so wird diese Methode immer dann aufgerufen, wenn der Knopf gedrückt wird.
Und zwar mit einem Parameter der Klasse `ActionEvent`, aus dem man im Rumpf von `handle` z.B. den Knopf, der gedrückt wurde, ablesen kann (die "Quelle des Ereignisses").

Beispiel

```
public class Behandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent ereignis) {
        System.out.println("Es wurde " + ereignis.getSource()
                           + " gedrueckt.");
    }
}
```

und in `start()`:

```
EventHandler<ActionEvent> behandler = new Behandler();
einzahlKnopf.setOnAction(behandler);
abhebeKnopf.setOnAction(behandler);
```

Ausgabe:

```
Es wurde Button@527ba09f[styleClass=button]'einzahlen' gedrueckt
Es wurde Button@7b3313b8[styleClass=button]'abheben' gedrueckt.
```

Sinvollere Behandler

Für sinnvollere Aktionen brauchen wir

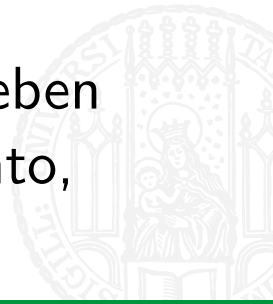
- Eine Instanzvariable `konto` der Klasse `Bankkonto`
- Innerhalb von `handle` Zugriff auf `konto`, sowie auf die Textfelder und Knöpfe.

Übergabe der GUI-Komponenten per Konstruktor

Wir können alle diese Komponenten auch als Instanzvariablen des Behandlers führen und über den Konstruktor übergeben.
Z.B.: in `Behandler.java`

```
private Bankkonto konto;
...
public Behandler(Bankkonto konto, Button abhebeKnopf,...){
   this.konto = konto;
   ...
}
```

Vorteil Anschaulich und im Einklang mit bisherigem Stoff
Nachteil Sehr umständlich, wenn viele Komponenten übergeben werden müssen. Schon in diesem Beispiel sind es ja fünf! (Konto, zwei Textfelder, zwei Knöpfe)



Innere Klassen

Alternativ hat man die Möglichkeit, die Definition der Klasse `Behandler` innerhalb von `BankkontoGUI {...}` vorzunehmen.
Hierbei verwendet man ein neues Java Sprachkonstrukt: **Innere Klassen**
Eine innere Klasse hat auch Zugriff auf die privaten Instanzvariablen ihrer umgebenden Klasse und verhält sich ansonsten wie eine normale Klasse.
Im Computer realisiert werden innere Klassen wie Möglichkeit 1, also explizite Übergabe der Instanzvariablen bei Konstruktion.

Vorteil Wird oft benutzt, steht so im Buch.

Nachteil Theorie etwas unklar (Sichtbarkeitsbereiche, etc.), Aufblähung der Sprache.

Beispiel

```
public class BankkontoGUI extends Application {
    private Bankkonto konto;
    ...
    public void start(Stage primaryStage) {
        this.konto = new Bankkonto("Steffen", 1000.);
        ...
        EventHandler behandler = new Behandler();
        einzahlKnopf.setOnAction(behandler);
        abhebeKnopf.setOnAction(behandler);
        kontostandFeld.setText("" + konto.getKontostand());
        ...
    }
    class Behandler implements EventHandler<ActionEvent> {
        public void handle(ActionEvent ereignis) {
     /*Fortsetzung folgt*/
```

Beispiel

```
    Object gedrueckt = ereignis.getSource();
    double betrag = Double.parseDouble(betragFeld.getText());
    if (gedrueckt == einzahlKnopf) {
       konto.einzahlen(betrag);
       kontostandFeld.setText("" + konto.getKontostand());
    } else if (gedrueckt == abhebeKnopf) {
       konto.abheben(betrag);
       kontostandFeld.setText("" + konto.getKontostand());
    }
  } /* Ende der Methode handle */
 } /* Ende der inneren Klasse Behandler */
} /* Ende der Klasse BankkontoGUI */
```

Achtung: == ist hier die physikalische Gleichheit von Objekt(referenz)en.

Anonyme innere Klassen

Eine weitere Möglichkeit besteht darin, eine innere Klasse ohne eigenen Namen an Ort und Stelle zu definieren.

Dafür muss nur der Name eines implementierten Interfaces vorhanden sein. Man schreibt also z.B. in `start`:

```
EventHandler<ActionEvent> behandler =
  new EventHandler<ActionEvent>(){
      public void handle(ActionEvent ereignis) {
        Object gedrueckt = ereignis.getSource();
          ...
            }};
einzahlKnopf.setOnAction(behandler);
    ...
```

Allgemein erzeugt also die Syntax
`new` *Interface*`(){`*code*`};`
ein neues Objekt einer namenlosen Klasse, die die Schnittstelle *Interface* implementiert.

Jedem Knopf seinen eigenen Behandler

Nachdem jetzt der Aufwand für die Bereitstellung der Behandler-Klasse weitgehend entfällt, kann man auch jedem Knopf seine eigenes anonymes Behandler-Objekt geben:

```
einzahlKnopf.setOnAction(new EventHandler<ActionEvent>(){
  public void handle(ActionEvent ereignis){
    konto.einzahlen(Double.parseDouble(betragFeld.getText()));
    kontostandFeld.setText("" + konto.getKontostand());
  }});
abhebeKnopf.setOnAction(new EventHandler<ActionEvent>(){
      ...
  });
```

Lambda-Ausdrücke

Im besonderen Falle, in dem das zu implementierende Interface genau eine Methode enthält und die anonyme Klasse keine Instanzvariablen benötigt, gibt es seit Java 8 eine weitere elegante Möglichkeit: “Lambdas”.

Beispiel

```
abhebeKnopf.setOnAction(ereignis -> {
  konto.abheben(Double.parseDouble(betragFeld.getText()));
  kontostandFeld.setText("" + konto.getKontostand())});
```

Lambdas allgemein

Allgemein kann ein Methodenparameter, dessen Typ ein Interface mit nur einer Methode ist, durch einen Lambda-Ausdruck gegeben werden.

Dieser hat die Form
$(x_1, \ldots, x_n)$ `->` *body*
wobei n die Zahl der formalen Parameter der einzigen Methode des verlangten Interfaces ist.
Der Rumpf *body* ist entweder ein Java Ausdruck, der die Parameter $x_1, \ldots, x_n$ benutzen darf, oder ein in {}-Klammern stehender Block, ggf. mit `return` Statement.

Der Lambda-Ausdruck steht dann für ein frisch erzeugtes Objekt zu dem geforderten Interface, dessen einzige Methode wie im Rumpf beschrieben implementiert ist.

Beispiel

```
interface IntTester {public boolean test(int x);}

class IstGerade implements IntTester {
    public boolean test(int x) { return x % 2 == 0;}
}
...
    public static boolean  fueralle(int[] a, IntTester p) {
        boolean result = true;
        for (int x:a) result &= p.test(x);
        return result;
    }

    int a[] = {2,4,6,8,10,122};
    int b[] = {4,25,289}
    System.out.println(fueralle(a,new IstGerade()));
    System.out.println(fueralle(b,new IstGerade()));
```


Ausgabe: `true` und `false`.

Verwendung von Lambdas im Beispiel

```
int a[] = {2,4,6,8,10,122};
int b[] = {4,25,289}
System.out.println(fueralle(a,x -> x<1000));
System.out.println(fueralle(b,x -> {
    int q = (int)Math.sqrt(x);
    return x == q*q;}));
```

Ausgabe: `true` und `true`.

Anstelle eines Lambdas kann man mit doppeltem Doppelpunkt auch eine Funktion geeigneten Typs angeben, z.B.:

```
IstGerade obj = new IstGerade();
// dynamische Methode test1 mit Objekt obj aufrufen
System.out.println(fueralle(b, obj::test1));
// statische Methode test2 aus Klasse IstGerade aufrufen
System.out.println(fueralle(a, IstGerade::test2));
```

Beobachter

Wir haben das Problem, die Kontostandanzeige mit dem tatsächlichen Kontostand konsistent zu halten.
Jede Änderung des Kontostandes (auch durch andere GUIs, das andere Programmteile, o.ä.) sollen sofort wiedergegeben werden.
Die Lösung besteht im Entwurfsmuster **Beobachter** (**Observer**).
Jedes Bankkonto verwaltet eine Liste von Beobachtern, die sich bei ihm registriert haben. Ändert sich der Kontostand, so ruft es bei jedem registrierten Beobachter eine bestimmte Methode auf.

Konkretes Beispiel

Wir verwenden die Schnittstelle

```
public interface BankkontoBeobachter {
    public void anzeigen(Bankkonto b);
}
```

Die Klasse Bankkonto muss nun so erweitert werden, dass eine Liste von BankkontoBeobachtern verwaltet wird und die Methode `anzeigen` entsprechend aufgerufen wird:

Konkretes Beispiel

```
public class Bankkonto {
    private double kontostand;
    private ArrayList<BankkontoBeobachter> beobachter;
    public void benachrichtigen() {
        for (BankkontoBeobachter beob : beobachter)
            beob.anzeigen(this);
    }
    public void einzahlen(double betrag) {
        kontostand = kontostand + betrag;
        benachrichtigen();
    }
    public void abheben(double betrag) {...}
    public void anheften(BankkontoBeobachter beob) {
        beobachter.add(beob);
    }
}
```

Konkretes Beispiel

Die Klasse `BankkontoGUI` implementiert nun `BankkontoBeobachter`.

```
public class BankkontoGUI extends Application
                    implements BankkontoBeobachter {
    ...

    public void anzeigen(Bankkonto k) {
        kontostandFeld.setText("" +
                        konto.getKontostand());
    }
```

Jetzt brauchen wir uns um die Aktualisierung des Textfeldes nicht mehr zu kümmern. Sie erfolgt ganz automatisch.

Observer und Observable

Die Beobachterverwaltung ist in Java auch vorgefertigt in Form einer Klasse `Observable` von der beobachtbare Klassen, z.B. `Bankkonto` dann erben können ...
und einer Schnittstelle `Observer`, die die Beobachter, z.B. `BankkontoGUI` implementieren müssen. Diese Schnittstelle enthält eine Methode

```
void update(Observable o,
            Object arg)
```

Die Klasse `Observable` stellt Methoden `addObserver` (entspricht unserem `anheften`) und `notifyObservers` (entspricht unserem `benachrichtigen`) bereit. Zusätzlich muss bei jeder Änderung die Methode `setChanged` aufgerufen werden.

Im Beispiel Bankkonto

```
class Bankkonto extends Observable {
    private double kontostand;

    void einzahlen(double betrag) {
        kontostand = kontostand + betrag;
        setChanged();
        notifyObservers();
    }
...
```

Achtung: Seit JDK 9 sind `Observer` und `Observable` "deprecated", also nicht mehr empfohlen, da das Beobachter-Muster inzwischen besser (aber komplizierter) umgesetzt werden kann, siehe etwa letzte Folie.

Anmelden von Beobachtern: entweder so:

```
public class BankkontoGUI extends Application
                          implements Observer {
    public void update(Observable o, Object arg) {
        kontostandFeld.setText("" +
                                   konto.getKontostand());
    }
...
    public void start(Stage primaryStage) {
        konto = new Bankkonto();
        konto.addObserver(this);
    ...
```

Oder unter Verwendung von Lambdas

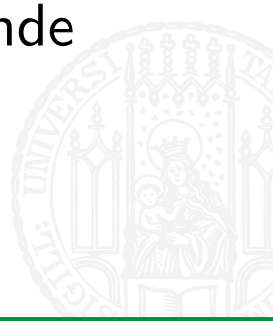
In `start`:

```
konto.addObserver((o,arg) ->
    kontostandFeld.setText("" + konto.getKontostand()));
```

Ebenso können wir weitere Beobachter hinzufügen:

```
Slider ktoAnzeige = new Slider();
...
konto.addObserver((o,arg)->
    ktoAnzeige.setValue((int)konto.getKontostand()));
```

Wie gewohnt erzeugt der Lambda Ausdruck automatisch ein Objekt der verlangten Schnittstelle, ohne dass eine entsprechende Klasse eingeführt oder verwendet werden müsste.



Model-View-Controller

- Oft kann eine Anwendung mit GUI in die folgenden drei Teile zerlegt werden:
  - Ein "Modell" (model): die Daten, mit denen über die GUI interagiert wird, bzw. die darzustellen sind. Beispiele: das Bankkonto, der aktuelle Spielzustand, eine Simulation.
  - Eine "Ansicht" (view): die konkrete grafische Darstellung des Modells und der zugehörigen Bedienelemente. Beispiele: das Bankkonto-Fenster mit den Knöpfen, die Darstellung eines Spiels, die verschiedenen Bildschirme bei einer Android-App,
  - Die "Steuerung" (controller): die Verwaltung der einzelnen Teile der Ansicht. In unserem Beispiel die Listener und Observer-Methoden, ausserdem, die Umschaltung zwischen Bildschirmen, etc.
- Es wird empfohlen, diese drei Teile voneinander möglichst getrennt zu halten, z.B. nicht:

Typische Verletzungen von "Model-View-Controller"

- In der Klasse Bankkonto und ihren Methoden direkt in Textfelder schreiben
- Den Kontostand ausschliesslich in Form des Inhalts des Textfeldes zu speichern
- Das Fenster über Methoden der Klasse Bankkonto zu erzeugen.

Bei der Verwendung von Frameworks wie JavaFX ist die Trennung von View und Controller nicht so leicht durchzusetzen und auch nicht mehr so wichtig, da der grösste Teil der Steuerung vom Framework bereitgestellt wird.

In unserem Beispiel befinden sich View und Controller auch in derselben Klasse.

Motivation für FXML

- Die Festlegung des Aussehens der Benutzeroberfläche durch Java Befehle ist umständlich und unübersichtlich.
- Als Abhilfe wurde (und das ist auch bei anderen Bibliotheken und Frameworks so) ein XML Standard eingeführt, hier FXML, mit dem GUI Layouts formal beschrieben werden können.
- Mit speziellen Ladefunktionen können dann solche XML Spezifikationen einer GUI eingelesen werden und die entsprechenden Objekte automatisch erzeugt werden.
- Solche XML Spezifikationen können dann auch interaktiv mit einem entsprechenden grafischen Editor, hier z.B. dem "SceneBuilder" erzeugt werden.
  http://gluonhq.com/open-source/scene-builder/
- Die grafische Ansicht der GUI befindet sich dann an separater Stelle, vgl. Model-View-Controller Prinzip.

FXML Definition für unser Beispiel

Im Beispiel könnte der FXML-Code in einer Datei BankkontoGUI.fxml des folgenden Inhalts stehen:

```
 <?xml
version="1.0" encoding="UTF-8"?>

<?import javafx.scene.paint.*?>
<?import javafx.scene.effect.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>

<GridPane xmlns="http://javafx.com/javafx/8.0.66"
          xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <FlowPane>
         <children>
```

FXML-Datei Fortsetzung

```
            <Label text="Kontostand: "/>
            <TextField fx:id="kontostandFeld" editable="false"/>
        </children>
    </FlowPane>
    <FlowPane GridPane.rowIndex="1">
        <children>
            <Label text="Betrag: " />
            <TextField fx:id="betragFeld" text="0.0"/>
        </children>
    </FlowPane>
    <FlowPane GridPane.rowIndex="2">
        <children>
            <Button fx:id="einzahlKnopf" text="einzahlen"/>
            <Button fx:id="abhebeKnopf" text="abheben"/>
        </children>
    </FlowPane>
  </children>
</GridPane>
```

Erklärung der XML-Datei

- Die FXML Datei ist ähnlich einer HTML-Datei strukturiert.
- Die einzelnen Komponenten können entweder einzelne Tags mit Attributen sein, wie das `Label` oder das `TextField`
- oder geschachtelte Komponenten mit Unterkomponenten wie `GridPane` oder `FlowPane` sein
- Das `xmlns` Attribut zu Beginn definiert den entsprechenden XML-Namensraum, in ihm sind die verwendeten Komponenten festgelegt
- Die `import` Direktiven werden vom FXML-Lader benötigt, siehe folgende Folien
- Man kann die FXML-Datei auch interaktiv mit dem SceneBuilder erzeugen und in der Regel wird das auch so gemacht. Siehe Demo.

Verwendung der FXML Spezifikation

Man kann jetzt in `start` die FXML-Datei wie folgt laden:

```
import javafx.fxml.FXMLLoader;
import java.net.URL;
import javafx.fxml.FXML;
...
import javafx.collections.ObservableMap;

FXMLLoader fxmlloader = new FXMLLoader(new URL(
 "file:///home/mhofmann/EIP15/Vorlesung/BankkontoGUI.fxml"));
  GridPane fensterInhalt=(GridPane)fxmlloader.load();
  ObservableMap<String,Object> namespace = fxmlloader.getNamespa
  einzahlKnopf = (Button)namespace.get("einzahlKnopf");
  abhebeKnopf = (Button)namespace.get("abhebeKnopf");
  kontostandFeld = (TextField)namespace.get("kontostandFeld");
  betragFeld = (TextField)namespace.get("betragFeld");
```

Erklärung

- Der Konstruktor `FXMLLoader` erzeugt einen neuen FXML-Lader für unsere FXML-Datei.
- Die `load`-Methode verarbeitet dann die FXML Datei und liefert das entsprechende Wurzelobjekt, hier die GridPane zurück.
- Mit `getNamespace` erhält man die Gesamtheit der erzeugten aktiven Objekte über ihren fx-Bezeichner (der war als entsprechendes Attribut in der FXML-Datei dazugegeben)
- Die aktiven Objekte werden wie gewohnt vorher als private Instanzvariablen deklariert.

Laden mit Reflexion

Man kann die Komponenten auch so laden:

```
FXMLLoader fxmlloader = new FXMLLoader(new URL(
"file:///home/mhofmann/EIP15/Vorlesung/BankkontoGUI.fxml"));
fxmlloader.setController(this);
GridPane fensterInhalt=(GridPane)fxmlloader.load();
```

Die oben deklarierten privaten Instanzvariablen werden dann mit den entsprechenden erzeugten Objekten automatisch verbunden.

Dies verwendet die sogenannte Reflexion, ein zusätzliches Java-Feature, von dem bisher nicht die Rede war. Reflexion erlaubt es, auf die syntaktische Struktur von Objekten zur Laufzeit zuzugreifen.
Vorsicht: die Typprüfung wird durch Reflexion weitgehend außer Kraft gesetzt; Typfehler zeigen sich oft erst zur Laufzeit.

Für das Verständnis von Frameworks kann eine oberflächliche Kenntnis des Konzeptes nützlich sein.

Beispiel für Reflexion

```
import java.lang.reflect.Field;
...
public static void set50(Object x,String vname) throws Exception
   Field v = x.getClass().getDeclaredField(vname);
   v.setAccessible(true);
   v.set(x,50);
}
```

Diese Methode `set50` kann ein beliebiges `int`-Feld eines beliebigen Objektes auf 50 setzen. Das Objekt und der Name des Feldes (als String!) werden übergeben.

Weiterführende Themen zum Selbststudium

- Verändern des Aussehens der GUI-Komponenten durch Style-Sheets (CSS)
- Weitere GUI-Komponenten: Accordeon, Pulldown Menu, Charts, ...
- DoubleProperty, BooleanProperty, etc.: Observer-Pattern für elementare Datentypen
- Properties und Bindings: Verallgemeinerung des Observer-Patterns
- Geometrische Objekte in Panes zeichnen
- Affine Transformationen (Drehen, Verschieben, etc) auf GUI-Komponenten anwenden
- 3D Grafik: Kameras, Lampen

Zusammenfassung GUI

- Die Bibliothek JavaFX stellt GUI Komponenten bereit.
- Mit Ereignissen (Event) und Zuhörern (Listener) wird auf Benutzereingaben reagiert. Letztendlich wird bei Benutzereingaben eine vom Programmierer für diesen Zweck bereitgestellte Methode aufgerufen.
- Innere Klassen erlauben die komfortable Implementierung der erforderlichen Behandler-Klassen.
- Lambda-Ausdrücke ermöglichen das Übergeben von Code an andere Methoden mit reduziertem notationellen Umstand.
- Die GUI kann durch Verwendung des Beobachter-Musters mit den dargestellten Daten konsistent gehalten werden.
- Model-View-Controller: Empfehlung, das zugrundeliegende Modell von der grafischen Darstellung und Bedienoberfläche getrennt zu halten
- FXML ermöglicht die kompakte Definition des Layouts von GUI Komponenten.