LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung

Wintersemester 2018/19

Kapitel 8:
Grafische Benutzerschnittstellen

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 5. Dezember 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Ziele

- Kennenlernen von Grafischen Benutzerschnittstellen (Graphical User Interfaces) als Anwendungsbeispiel für die objektorientierte Programmierung
- Erstellung individueller GUI-Klassen durch Erweiterung existierender Klassen der Java Bibliotheken AWT und Swing.
- Die Vorgehensweise zur Erstellung einer GUI verstehen und durchführen können

Ziele (2)

Verständnis aller Schritte der Erstellung einer GUI:

1. Erstellung des strukturellen Aufbaus der GUI
2. Verbindung der Ansicht (GUI) mit den inhaltlichen Objekten der Anwendung (Modell),
3. Ereignisgesteuerte Behandlung von Benutzereingaben (z.B. Knopfdruck).

AWT und Swing

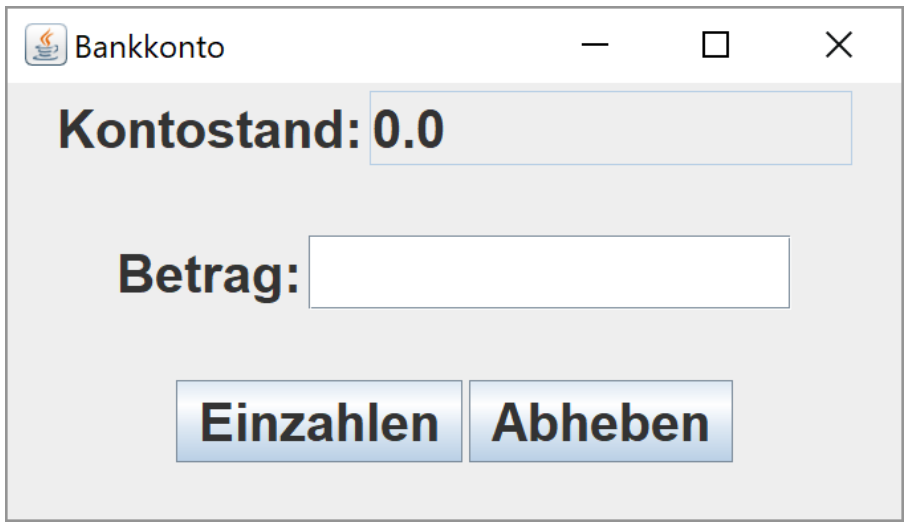
- AWT und Swing bieten eine Klassenbibliothek zur Programmierung grafischer Benutzerschnittstellen (GUIs) für Java-Programme.
- Mit Java 1.0 wurde 1996 das Abstract Window Toolkit (AWT) veröffentlicht
- Mit Java 1.2 wurde 1998 eine verbesserte Bibliothek namens Swing eingeführt
- Swing baut auf AWT auf
  (es verwendet Klassen aus dem AWT)
- Typische Import-Deklarationen in ein Programm, das AWT/Swing benutzt:

```
import java.awt.*;
import javax.swing.*;
```

Beispiel

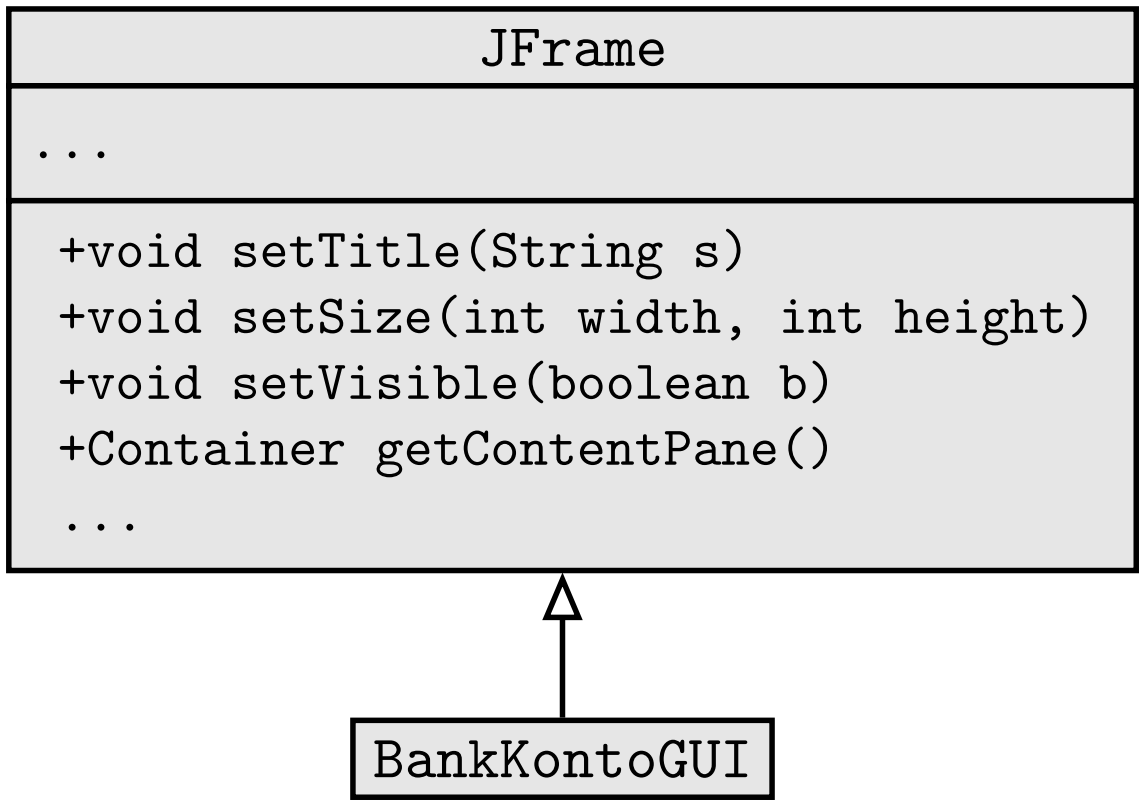
Unser Ziel:

Erstellung einer einfachen grafischen Anwendung für Bankkonten



Das Fenster

- Die Klasse `JFrame` stellt ein leeres Fenster zur Verfügung.
- Wir erweitern die Klasse (bilden eine Unterklasse) für unsere Anwendung:



Fenster erzeugen

```
import java.awt.*;
import javax.swing.*;
public class BankKontoGUI extends JFrame {
 public BankKontoGUI(){
  this.setTitle("Bankkonto");
  this.setSize(700,400);
 }
```

BankKontoGUI.java

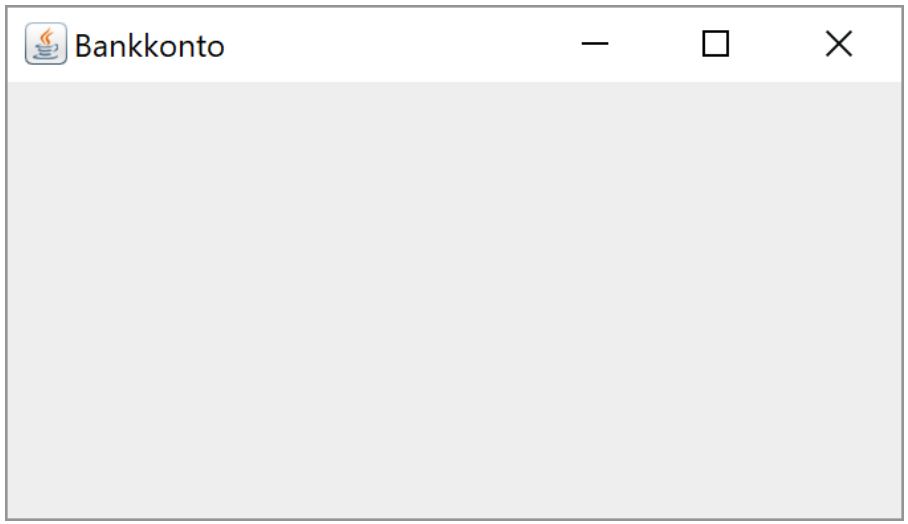
```
public class Main {
 public static void main(String[] args) {
  BankKontoGUI gui = new BankKontoGUI();
  gui.setVisible(true);
 }
}
```

Main.java

Fenster erzeugen (2)

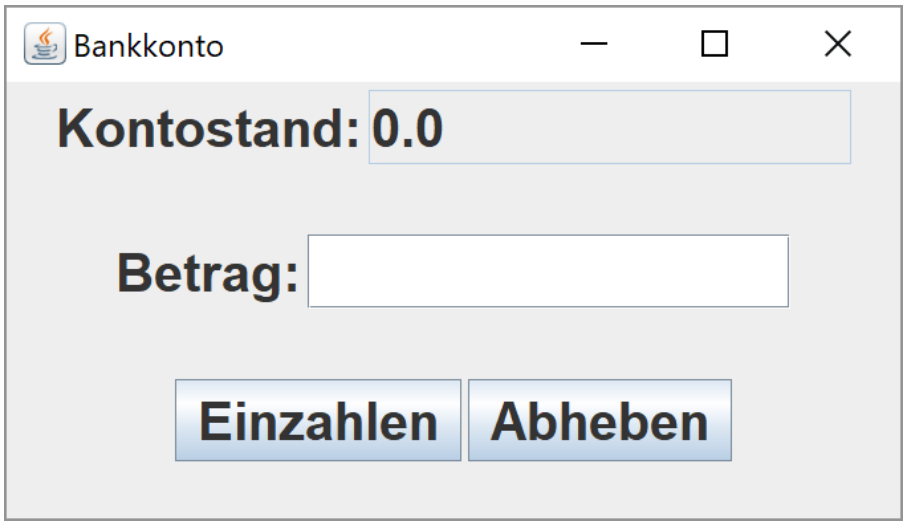
```
> javac Main.java
> java Main
```

erzeugt ein leeres Fenster:



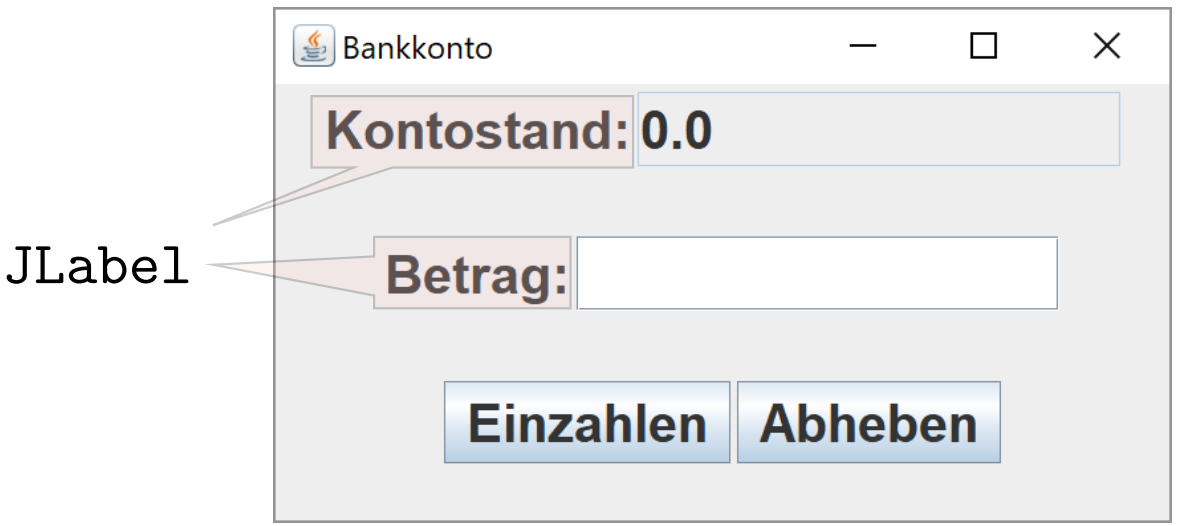
GUI-Elemente

- In Swing gibt es viele Klassen für die verschiedenen GUI-Elemente
- Textaufschriften, Knöpfe, Textfelder für Ein-/Ausgabbe usw. werden durch Objekte der Klassen `JLabel`, `JButton`, `JTextField` usw. repräsentiert



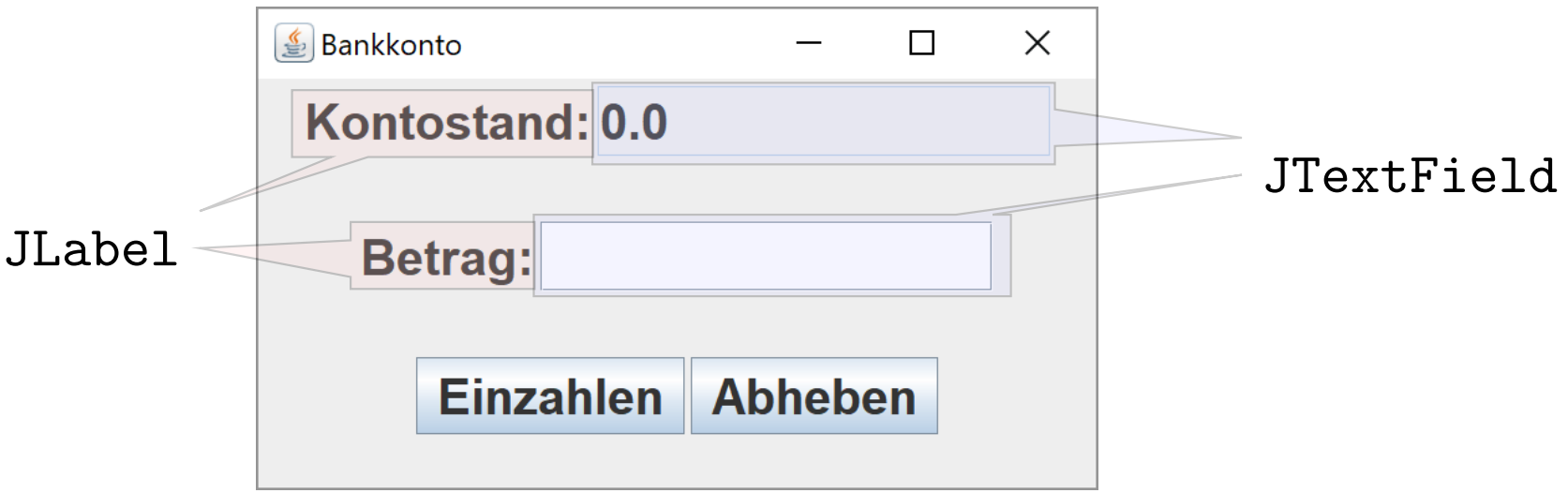
GUI-Elemente

- In Swing gibt es viele Klassen für die verschiedenen GUI-Elemente
- Textaufschriften, Knöpfe, Textfelder für Ein-/Ausgabbe usw. werden durch Objekte der Klassen `JLabel`, `JButton`, `JTextField` usw. repräsentiert



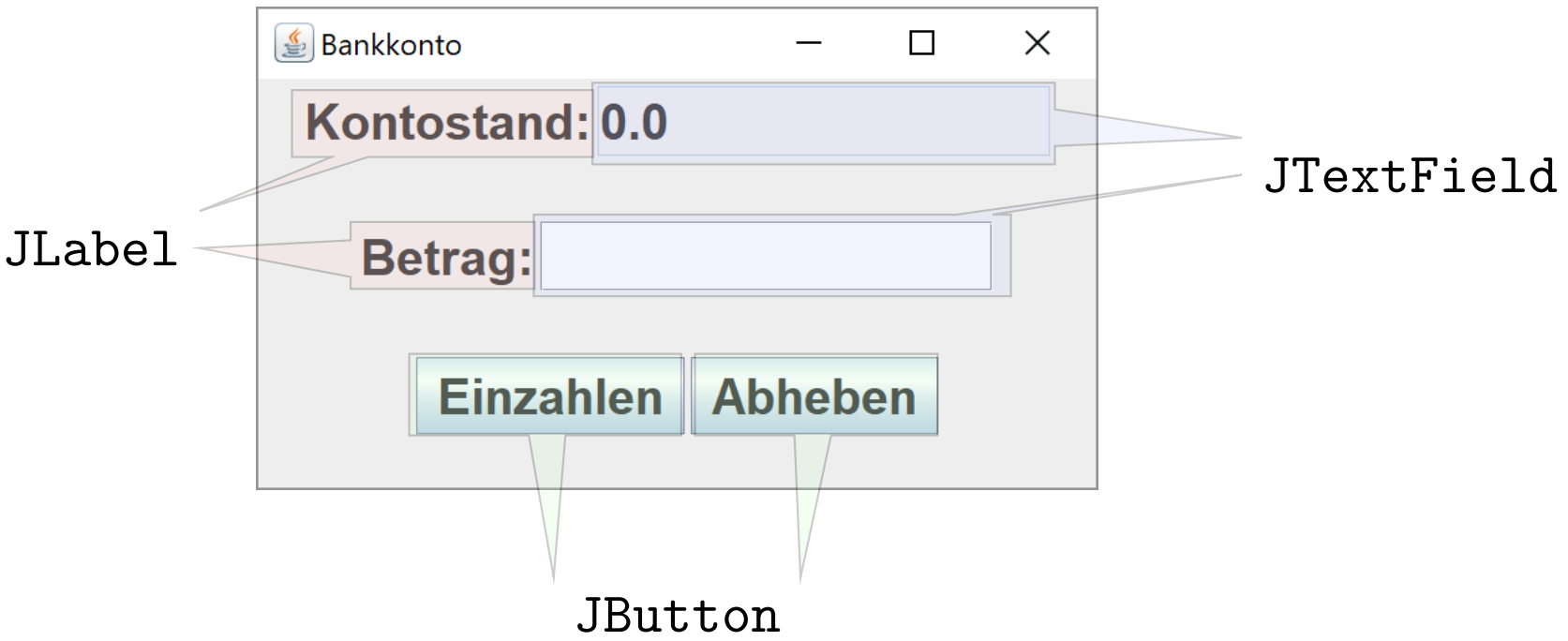
GUI-Elemente

- In Swing gibt es viele Klassen für die verschiedenen GUI-Elemente
- Textaufschriften, Knöpfe, Textfelder für Ein-/Ausgabbe usw. werden durch Objekte der Klassen `JLabel`, `JButton`, `JTextField` usw. repräsentiert



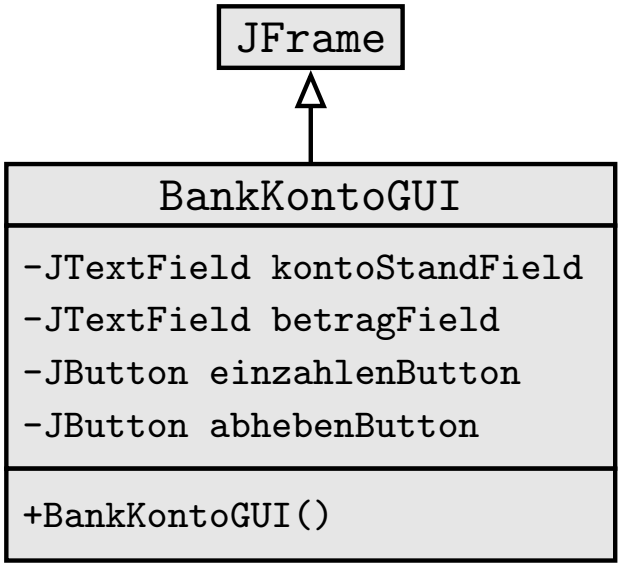
GUI-Elemente

- In Swing gibt es viele Klassen für die verschiedenen GUI-Elemente
- Textaufschriften, Knöpfe, Textfelder für Ein-/Ausgabbe usw. werden durch Objekte der Klassen `JLabel`, `JButton`, `JTextField` usw. repräsentiert



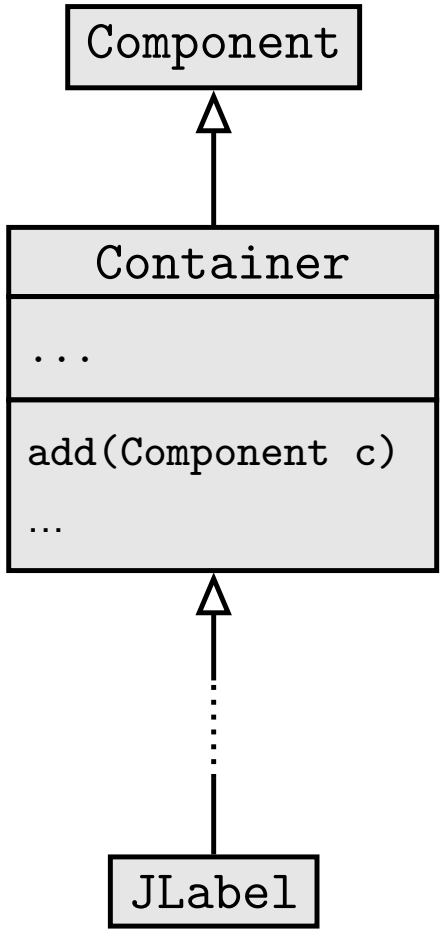
Anlegen von Instanzvariablen für die GUI-Elemente

```
...
public class BankKontoGUI extends JFrame {
 /* Attribute fuer GUI-Elemente */
 private JTextField kontoStandField;
 private JTextField betragField;
 private JButton einzahlenButton;
 private JButton abhebenButton;
 public BankKontoGUI() {
  /* Titel und Groesse der GUI */
  this.setTitle("Bankkonto");
  this.setSize(700, 400);
  /* lokale Variable fuer Schriftart*/
  Font font1 = new Font("SansSerif", Font.BOLD, 40);
  /* Initialisierung der Attribute */
  this.kontoStandField = new JTextField("123.0",10);
  this.kontoStandField.setEditable(false);
  this.kontoStandField.setFont(font1);
  this.betragField = new JTextField(10);
  this.betragField.setFont(font1);
  this.einzahlenButton = new JButton("Einzahlen");
  this.abhebenButton = new JButton("Abheben");
  this.einzahlenButton.setFont(font1);
  this.abhebenButton.setFont(font1);
  ...
```



Komponenten und Container

- Die einzelnen Teile einer GUI (`JLabel`,`JButton`, ...) sind Komponenten und von der Klasse `Component` abgeleitet.
- Viele Komponenten sind (abgeleitet von) `Container`.
- Ein Container ist eine Komponente, die selbst andere Komponenten enthalten kann. Beispiele:
  - Der Inhalt eines JFrame-Fensters ist ein Container.
  - Die Klasse JPanel ist ein Container, mit dem man mehrere Komponenten zu einer einzigen zusammenfassen kann.
- Die Klasse `Container` hat eine Methode `void add(Component c)`, mit der man neue Komponenten zu einem Container hinzufügen kann.
- Durch Container und Komponenten werden Bäume (Hierarchien) repräsentiert.



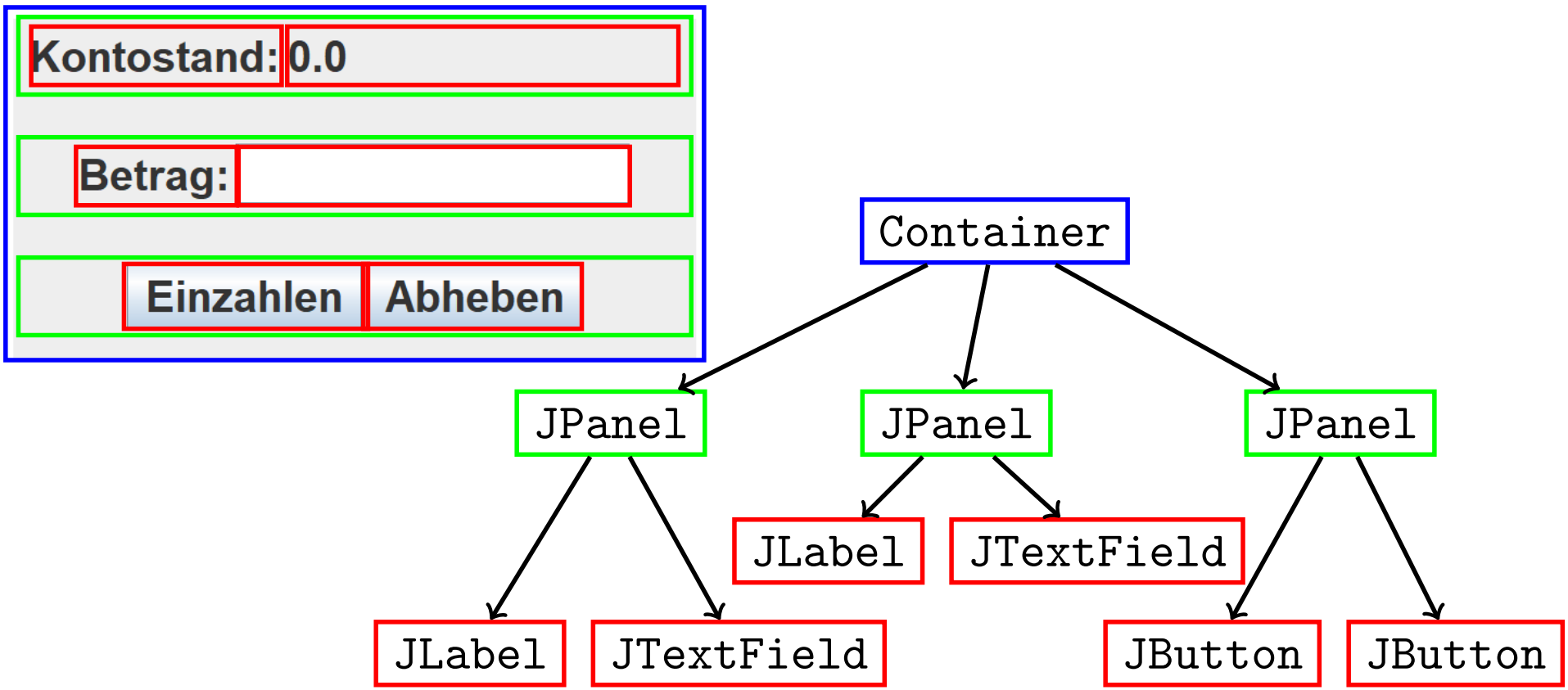
Strukturierung von Fenster-Inhalten

- Der Hintergrund eines `JFrame`-Fensters ist ein `Container`-Objekt („content pane“). Der Inhalt des Fensters ist darin als ein Baum von Objekten organisiert.

`Container`

Strukturierung von Fenster-Inhalten

- Der Hintergrund eines `JFrame`-Fensters ist ein `Container`-Objekt („content pane“). Der Inhalt des Fensters ist darin als ein Baum von Objekten organisiert.
- Für die Gruppierung verwenden wir Objekte der Klasse `JPanel`

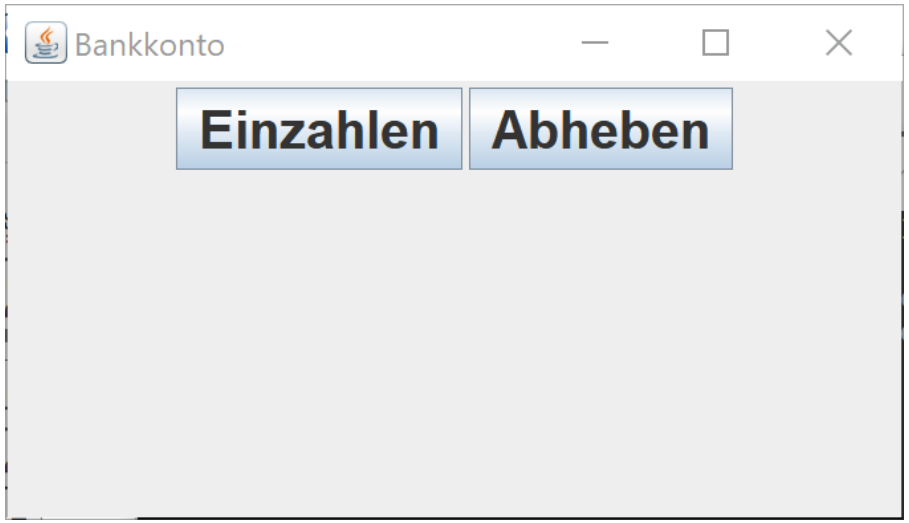


Aufbau der Baumstruktur und Setzen des Layouts

```
public BankKontoGUI() {
...
/* Baumstruktur anlegen und Layout setzen */
JPanel kontoStandPanel = new JPanel(); // lokale Variable
JLabel labelKontoStand = new JLabel("Kontostand:");
labelKontoStand.setFont(font1);
kontoStandPanel.add(labelKontoStand);
kontoStandPanel.add(this.kontoStandField);
JPanel betragPanel = new JPanel(); // lokale Variable
JLabel labelBetrag = new JLabel("Betrag:");
labelBetrag.setFont(font1);
betragPanel.add(labelBetrag);
betragPanel.add(this.betragField);
JPanel buttonPanel = new JPanel(); // lokale Variable
buttonPanel.add(this.einzahlenButton);
buttonPanel.add(this.abhebenButton);
// content pane
Container contentPane = this.getContentPane();
contentPane.add(kontoStandPanel);
contentPane.add(betragPanel);
contentPane.add(buttonPanel);
}
```

Layout setzen

Ausführen ergibt nicht das erwünschte Resultat:



- Problem: Wir haben nicht gesetzt, **wie** die Elemente angeordnet werden sollen
- In der Klasse `Container` gibt es die Methode

```
void setLayout(LayoutManager mgr)
```

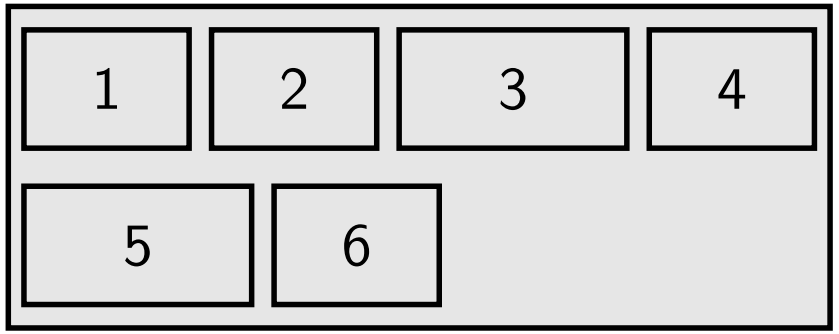
  mit der man einen LayoutManager einrichten kann, der die Komponenten anordnet
- Es gibt eine Reihe von LayoutManagern in Swing

Einige LayoutManager

FlowLayout

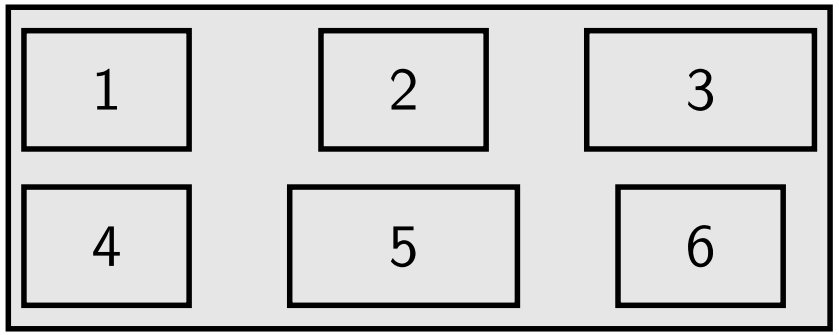
```
setLayout(new FlowLayout())
```

Standard bei JPanel



GridLayout

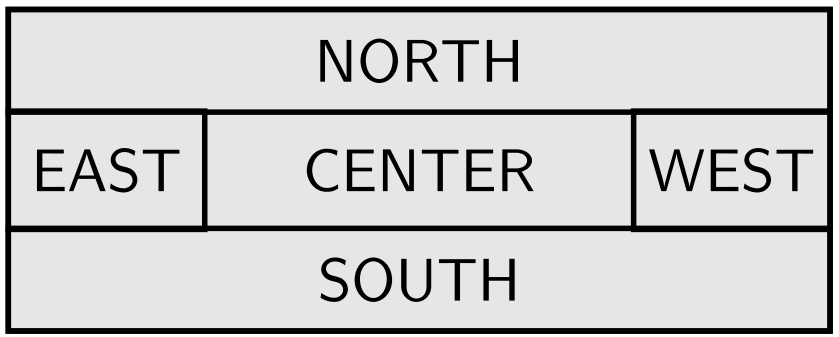
```
setLayout(new GridLayout(2,3))
```



BorderLayout

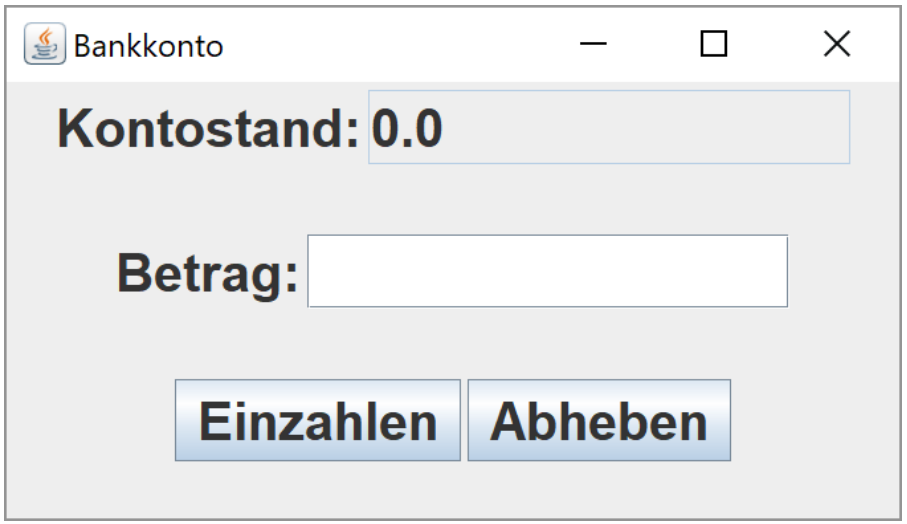
```
setLayout(new BorderLayout())
```

Standard bei Content-Pane in JFrame
Ohne Angabe der Position wird CENTER verwendet. Mit Angabe z.B.
`add(component,BorderLayout.SOUTH)`



Mit Grid-Layout

```
public BankKontoGUI() {
...
// content pane
Container contentPane = this.getContentPane();
// Grid mit 3 Zeilen und 1 Spalte
contentPane.setLayout(new GridLayout(3, 1));
contentPane.add(kontoStandPanel);
contentPane.add(betragPanel);
contentPane.add(buttonPanel);
}
```



Ansicht und Modell

- Bisher haben wir nur die **Ansicht** der Kontodaten programmiert
- Wir brauchen noch die eigentlichen **Daten**, in unserem Beispiel ein Bankkonto. Man spricht von dem **Modell**. Wir benutzen die Klasse `BankKonto`

Ansicht und Modell (2)

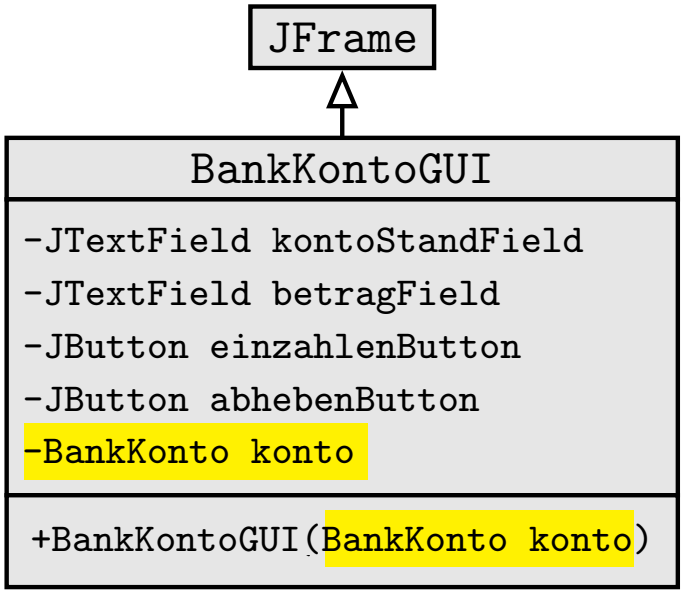
Ansicht und Modell sollten entkoppelt sein, denn:

- Modell exisitiert unabhängig von der Ansicht
- Verschiedene Ansichten für dasselbe Modell (GUI, Web-Interface, ...)
- Unabhängige Entwicklung von Ansicht und Modell unterstützt die Wartbarkeit

Ansicht und Modell im Beispiel (1)

- **Ansicht:** BankKontoGUI
- **Modell:** BankKonto

Anzeige-Objekte erhalten eine Referenz auf das anzuzeigende Modell




```
public class BankKontoGUI extends JFrame {
 ...
 /* Referenz auf das Modell */
 private BankKonto konto;
 ...
 /* Konstruktor bekommt Parameter zur Uebergabe des Modells */
 public BankKontoGUI(BankKonto konto){
  this.konto = konto;
  ...
  /* Setzen der Kontostand-Anzeige auf  Kontostand des Modells */
  this.kontoStandField.setText(
   Double.toString(this.konto.getKontoStand()));
  ...
 }
}
```

Ansicht und Modell im Beispiel (2)

Erzeugung von Ansicht und Modell:

```
public class Main {
 public static void main(String[] args) {
  /* erzeuge Modell */
  BankKonto konto = new BankKonto(123.0);
  /* erzeuge Ansicht */
  BankKontoGUI gui = new BankKontoGUI(konto);
  gui.setVisible(true);
 }
}
```

Schließen des Fensters

In unserem Beispielprogramm verwenden wir

```
/* Fuer ordnungsgemaesse Beendigung der Anwendung
   bei Schliessen (X) des Fensters */
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Dies führt dazu, dass das Schließen des Fensters, auch zum Schließen der gesamten Anwendung führt.

Ereignisse und Ereignisbehandlung

- In grafischen Anwendungen kann eine Vielzahl verschiedener Ereignisse auftreten, z.B.
  - Tastatur betätigen,
  - Maus klicken,
  - Fenster verschieben, vergrößern, schließen,
  - ...
- In AWT/Swing werden verschiedene **Ereignisklassen** unterschieden:
  - `ActionEvent`,
  - `WindowEvent`,
  - `KeyEvent`,
  - `MouseEvent`,
  - ...

Ereignisgesteuerte Eingabebehandlung

- **Registriere Methoden** bei der entsprechenden Komponente (z.B. ein `JButton`), die bei bestimmten Ereignissen (Knopf gedrückt, Text eingegeben,...) ausgeführt werden sollen.
- Die **Swing-Bibliothek übernimmt die Verwaltung** von Ereignissen, d.h. sie ruft die entsprechende Methoden automatisch auf.
- Der **Programmfluss** wird nur durch Ereignisse bestimmt.

Ereignisgesteuerte Eingabebehandlung(2)

- Damit eine Komponente (z.B. ein `JButton`) auf ein Ereignis reagieren kann, muss ein „Listener“ (Zuhörer) registriert werden, der bei Eintreten des Ereignisses benachrichtigt wird.

  Z.B. gibt es in der Klasse `JButton` die Methode

  ```
  void addActionListener(ActionListener listener)
  ```

- Ein `ActionListener` ist eine Klasse, die das Interface `ActionListener` implementiert:

  ```
  public interface ActionListener{
      void actionPerformed(ActionEvent e)}
  ```

- Für einen `JButton button` registriert man einen solchen Listener `l` mit `button.addActionListener(l)`
- Tritt nun ein Ereignis ein (z.B. Knopf gedrückt), so wird die Methode `l.actionPerformed(e)` aufgerufen. Das `ActionEvent e` enthält dabei Informationen zum / über das Ereignis.

Ereignisgesteuerte Eingabebehandlung (3)

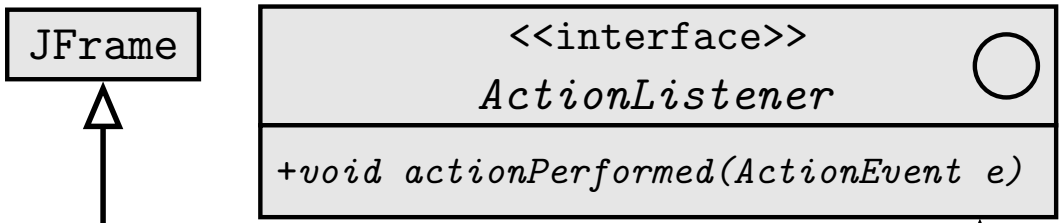
- Für unser Programm müssen wir daher mit `einzahlenButton.addActionListener(o)` und `abhebenButton.addActionListener(o)` Action-Listener registrieren.
- Dabei muss o ein Objekt einer Klasse sein, die das Interface `ActionListener` implementiert.
- Welche Klasse nehmen wir hierfür?
  - Die Klasse BankKontoGUI
  - Neue Klassen

  Wir gehen beide Möglichkeiten durch für unser Beispiel.

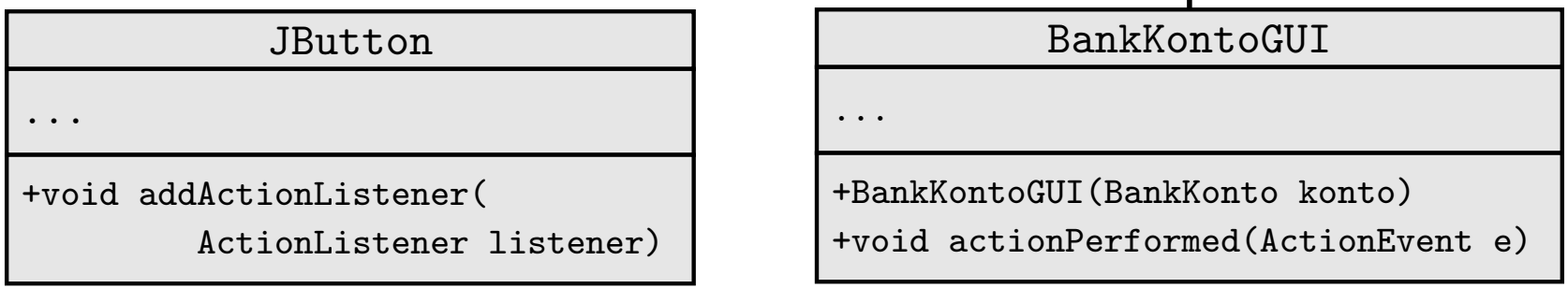
Variante 1: BankKontoGUI als ActionListener (1)

```
...
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class BankKontoGUI extends JFrame implements ActionListener {
  ...
  // Konstruktor
  public BankKontoGUI(BankKonto konto) {
    ...
    // Listener registrieren
    this.einzahlenButton.addActionListener(this);
    this.abhebenButton.addActionListener(this);
    ...
  }
  ...
```



Zur Erinnerung:



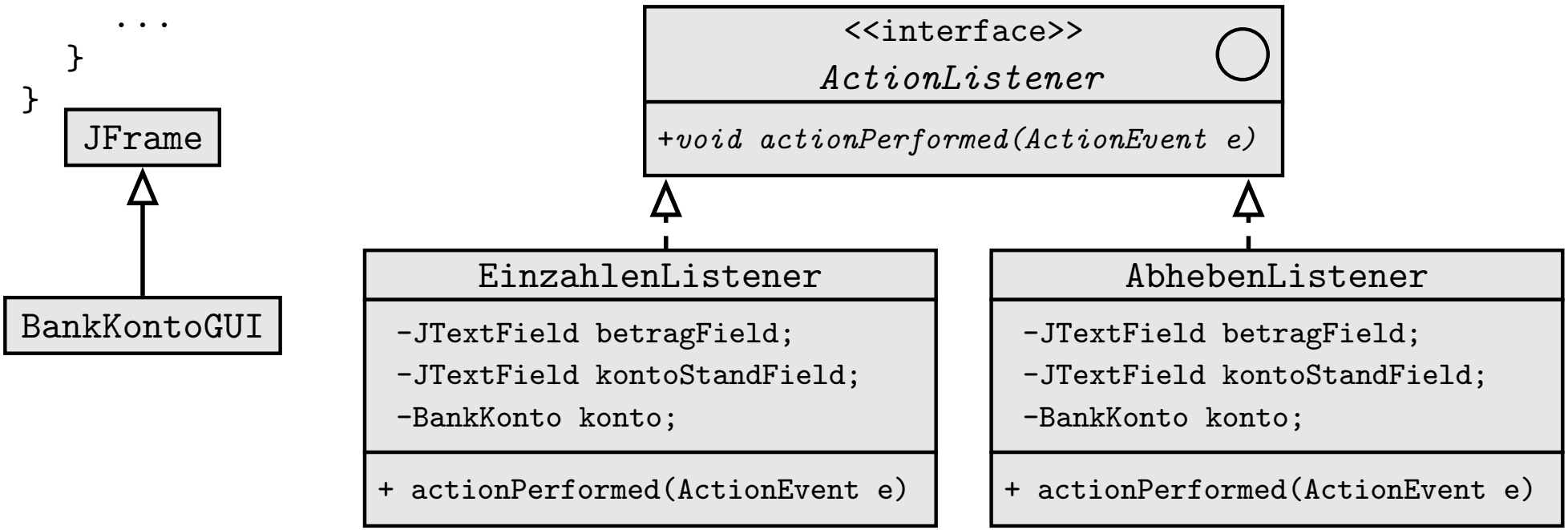
Variante 1: BankKontoGUI als ActionListener (2)

Implementierung des Interfaces: in der Klasse BankKontoGUI

```
public class BankKontoGUI extends JFrame implements ActionListener {
  ...
  @Override
  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource(); // Welcher Knopf wurde gedrueckt?
    if (source == this.einzahlenButton) {
      // Einlesen des Betrags
      double betrag = Double.parseDouble(this.betragField.getText());
      this.konto.einzahlen(betrag);    // Einzahlen auf Konto
      this.kontoStandField.setText(    // Ausgabe neuer Kontostand
                          Double.toString(this.konto.getKontoStand()));
    }
    else if (source == this.abhebenButton) {
      // Einlesen des Betrags
      double betrag = Double.parseDouble(this.betragField.getText());
      this.konto.abheben(betrag);     // Abheben vom Konto
      this.kontoStandField.setText(   // Ausgabe neuer Kontostand
                          Double.toString(this.konto.getKontoStand()));
    }
  }
}
```

Variante 2: Eigene Klassen für beide ActionListener (1)

```
...
public class BankKontoGUI extends JFrame {
   ...
   // Konstruktor
   public BankKontoGUI(BankKonto konto) {
      ...
      // Listener registrieren
      this.einzahlenButton.addActionListener(
            new EinzahlenListener(betragField,kontoStandField,konto));
      this.abhebenButton.addActionListener(
            new AbhebenListener(betragField,kontoStandField,konto));
      ...
   }
}
```



Variante 2: Eigene Klassen für beide ActionListener (2)

```
class EinzahlenListener implements ActionListener {
  private JTextField betragField;
  private JTextField kontoStandField;
  private BankKonto konto;
  // Konstruktor
  public EinzahlenListener(JTextField betragField
                           ,JTextField kontoStandField
                           ,BankKonto konto) {
    this.betragField = betragField;
    this.kontoStandField = kontoStandField;
    this.konto = konto;
  }
  // Implementierung der Knopfdruckbehandlung
  @Override
  public void actionPerformed(ActionEvent e) {
    // Einlesen des Betrags
    Double betrag = Double.parseDouble(this.betragField.getText());
    this.konto.einzahlen(betrag);   // Einzahlen auf Konto
    this.kontoStandField.setText(   // Ausgabe neuer Kontostand
           Double.toString(this.konto.getKontoStand()));
  }
}
```

Variante 2: Eigene Klassen für beide ActionListener (3)

```
class AbhebenListener implements ActionListener {
  private JTextField betragField;
  private JTextField kontoStandField;
  private BankKonto konto;
  public AbhebenListener(JTextField betragField
                         ,JTextField kontoStandField
                         ,BankKonto konto) {
    this.betragField = betragField;
    this.kontoStandField = kontoStandField;
    this.konto = konto;
  }
  // Implementierung der Knopfdruckbehandlung
  @Override
  public void actionPerformed(ActionEvent e) {
    // Einlesen des Betrags
    double betrag = Double.parseDouble(this.betragField.getText());
    this.konto.abheben(betrag);      // Abheben vom Konto
    this.kontoStandField.setText(    // Ausgabe neuer Kontostand
          Double.toString(this.konto.getKontoStand()));
  }
}
```

Vergleich von Variante 1 und Variante 2

Vorteile von Variante 2:

- Der ist Code besser strukturiert, da die Aufgaben den einzelnen Klassen zugeordnet sind.

Vergleich von Variante 1 und Variante 2

Vorteile von Variante 2:

- Der ist Code besser strukturiert, da die Aufgaben den einzelnen Klassen zugeordnet sind.
- Keine Abfrage notwendig, welcher Button gedrückt wurde.

Vergleich von Variante 1 und Variante 2

Vorteile von Variante 2:

- Der ist Code besser strukturiert, da die Aufgaben den einzelnen Klassen zugeordnet sind.
- Keine Abfrage notwendig, welcher Button gedrückt wurde.
- Jede Klasse behandelt ein Ereignis. Daher ist man sicher, dass alle Ereignisse behandelt werden. Bei Variante 1 muss die Methode `actionPerformed` alle Ereignisse behandeln.

Vergleich von Variante 1 und Variante 2

Vorteile von Variante 2:

- Der ist Code besser strukturiert, da die Aufgaben den einzelnen Klassen zugeordnet sind.
- Keine Abfrage notwendig, welcher Button gedrückt wurde.
- Jede Klasse behandelt ein Ereignis. Daher ist man sicher, dass alle Ereignisse behandelt werden. Bei Variante 1 muss die Methode `actionPerformed` alle Ereignisse behandeln.

Nachteile von Variante 2:

1. Parameterübergabe und Attribute für die Komponenten der GUI, die verändert werden. Bei mehr Komponenten wird dies unübersichtlich.

Vergleich von Variante 1 und Variante 2

Vorteile von Variante 2:

- Der ist Code besser strukturiert, da die Aufgaben den einzelnen Klassen zugeordnet sind.
- Keine Abfrage notwendig, welcher Button gedrückt wurde.
- Jede Klasse behandelt ein Ereignis. Daher ist man sicher, dass alle Ereignisse behandelt werden. Bei Variante 1 muss die Methode `actionPerformed` alle Ereignisse behandeln.

Nachteile von Variante 2:

1. Parameterübergabe und Attribute für die Komponenten der GUI, die verändert werden. Bei mehr Komponenten wird dies unübersichtlich.
2. Mehr Code durch die Definition der Klassen. Die Klassen werden im Grunde nur einmal verwendet, daher ist „unklar“, ob der Aufwand „lohnt“.

Vergleich von Variante 1 und Variante 2

Vorteile von Variante 2:

- Der ist Code besser strukturiert, da die Aufgaben den einzelnen Klassen zugeordnet sind.
- Keine Abfrage notwendig, welcher Button gedrückt wurde.
- Jede Klasse behandelt ein Ereignis. Daher ist man sicher, dass alle Ereignisse behandelt werden. Bei Variante 1 muss die Methode `actionPerformed` alle Ereignisse behandeln.

Nachteile von Variante 2:

1. Parameterübergabe und Attribute für die Komponenten der GUI, die verändert werden. Bei mehr Komponenten wird dies unübersichtlich.
2. Mehr Code durch die Definition der Klassen. Die Klassen werden im Grunde nur einmal verwendet, daher ist „unklar“, ob der Aufwand „lohnt“.

Beide Nachteile werden durch die beiden weiteren Varianten addressiert.

Innere Klassen

In unserem Beispiel (Variante 2) war das Problem, dass wir aus der Klasse `EinzahlenListener` aus nicht auf die (privaten) Attribute von `BankKontoGUI` zugreifen können.
Java unterstützt geschachtelte Klassen:

- Man kann Klassen innerhalb von Klassen definieren.
- Die innere Klasse darf auf die Attribute der äußeren Klasse zugreifen.
- Objekte der inneren Klasse existieren nur innerhalb eines Objekts der äußeren Klasse

```
class OuterClass {
  ...
  class InnerClass {
  ...
  }
...
}
```

Variante 3: Innere Klassen für beide ActionListener

```
public class BankKontoGUI extends JFrame {
  private JTextField kontoStandField, betragField;
  private BankKonto konto;
  ...
  public BankKontoGUI(BankKonto konto) {...
    this.einzahlenButton.addActionListener(new EinzahlenListener());
    this.abhebenButton.addActionListener(new AbhebenListener());
    ...                                          }
  /* innere Klassen */
  class EinzahlenListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e)
      double betrag = Double.parseDouble(betragField.getText());
      konto.einzahlen(betrag);
      kontoStandField.setText(Double.toString(konto.getKontoStand()))
  }
  class AbhebenListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e)
      double betrag = Double.parseDouble(betragField.getText());
      konto.abheben(betrag);
      kontoStandField.setText(Double.toString(konto.getKontoStand()))
  }
}
```

Anonyme Klassen

Eine Schwäche der bisherigen Implementierung ist, dass die beiden Klassen für `EinzahlenListener` und `AbhebenListener` definert werden, aber nur jeweils genau eine Instanz davon erzeugt wird.

Anonyme Klassen

Eine Schwäche der bisherigen Implementierung ist, dass die beiden Klassen für `EinzahlenListener` und `AbhebenListener` definert werden, aber nur jeweils genau eine Instanz davon erzeugt wird.

Schöner wäre: Erzeuge das Objekt, ohne der Klasse einen Namen zu geben.

Anonyme Klassen

Eine Schwäche der bisherigen Implementierung ist, dass die beiden Klassen für `EinzahlenListener` und `AbhebenListener` definert werden, aber nur jeweils genau eine Instanz davon erzeugt wird.

Schöner wäre: Erzeuge das Objekt, ohne der Klasse einen Namen zu geben.

Für innere Klassen geht das mit sogenannten **anonymen Klassen**

Nebenbedingung: Die Klasse ist Subklasse oder implementiert genau ein Interface

Anonyme Klassen (2)

Syntax

```
class OuterClass {
...
  new ParentClassOrInterface() {
    ...
    }
...
```

wenn die innere Klasse Unterklasse von `ParentClassOrInterface` oder das Interface `ParentClassOrInterface` implementiert

Variante 4: Anonyme Klassen für beide ActionListener

```
public class BankKontoGUI extends JFrame {
 ...
 public BankKontoGUI(BankKonto konto) {
  ...
  /* Listener registrieren */
  this.einzahlenButton.addActionListener(
   new ActionListener() {     // anonyme Klasse
    @Override
    public void actionPerformed(ActionEvent e) {
     double betrag = Double.parseDouble(betragField.getText());
     konto.einzahlen(betrag);
     kontoStandField.setText(Double.toString(konto.getKontoStand()));
    }
   }                                          );
  this.abhebenButton.addActionListener(
   new ActionListener() {     // anonyme Klasse
    @Override
    public void actionPerformed(ActionEvent e) {
     double betrag = Double.parseDouble(betragField.getText());
     konto.abheben(betrag);
     kontoStandField.setText(Double.toString(konto.getKontoStand()));
    }
   }                                          );
 }
```

Zusammenfassung

- Grundlegende Konstrukte der GUI-Programmierung mit AWT/Swing
- Trennung von Ansicht und Modell
- Ereignisgesteuerte Programmierung
- Innere Klassen
- Anonyme Klassen