
Objektorientierte und Funktionale Programmierung

SS 2013

5 Programmierung mit Java





5 Programmierung mit Java ...

Lernziele

- Kennenlernen wichtiger Teile der Java Klassenbibliothek
- Dateien in Java lesen und schreiben können
- Collection-Klassen kennen und nutzen können
- Einfache graphische Bedienoberflächen erstellen können

Literatur

- [Bi01], Kap. 4.1, 4.5, 6.5, 8.3, 10-13
- [Ba99], Kap. 2.18-2.20
- [BK03], Kap. 4, 5
- [HC05] Band 1, Kap. 12, 7-9; Band 2, Kap. 1, 2
- Java-Klassendokumentation:
<http://java.sun.com/javase/6/docs/api>

5 Programmierung mit Java ...



Inhalt

- Pakete der Java-Klassenbibliothek
- Dateien, Ströme und Serialisierung
- Das Java Collection Framework
- Programmierung graphischer Bedienoberflächen
- Threads
- Applets



5.1 Pakete der Java-Klassenbibliothek

- Die Sprache Java wird von einer (standardisierten) Klassenbibliothek ergänzt
 - Version 1.6 (Standard Edition) enthält 3793 Klassen in 203 Paketen
- Häufig genutzte Pakete:
 - `java.lang`: Klassen, die zum Kern der Sprache Java gehören
 - z.B.: `String`, `StringBuffer`, `Object`, `System`, ...
 - müssen nicht explizit importiert werden
 - `java.io`: Ein- und Ausgabe (Konsole und Dateien)
 - `java.util`: nützliche Hilfsklassen
 - u.a. Java Collection Framework (Container-Klassen)



5.1 Pakete der Java-Klassenbibliothek ...

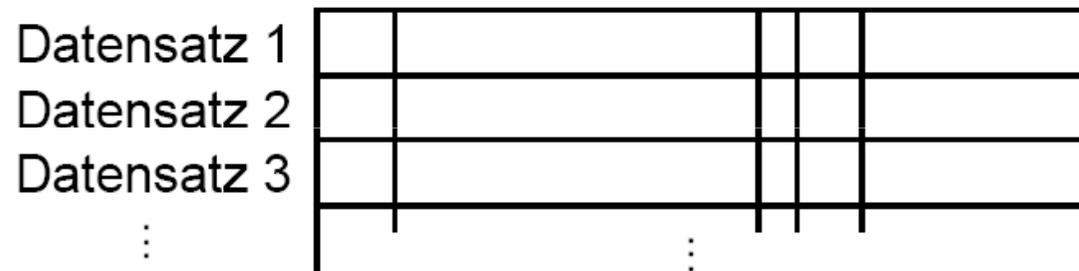
- Häufig genutzte Pakete ...:
 - `java.awt`: Elemente für Bedienoberflächen
 - Fenster, Menüs, Knöpfe, Textfelder ...
 - Graphikobjekte, Bilder, ...
 - Ereignisbearbeitung
 - `javax.swing`: verbesserte Bibliothek für Bedienoberflächen
 - `java.net`: Netzwerkkommunikation
 - `java.sql`: Datenbank-Anbindung
 - `java.beans`: Komponentenmodell
 - ...
- Dokumentation aller Pakete und Klassen im WWW:
 - <http://java.sun.com/javase/6/docs/api>



5.2 Dateien, Ströme und Serialisierung

5.2.1 Die Datenstruktur "Datei" (*file*)

- Eine **Datei** ist eine nach bestimmten Gesichtspunkten zusammengestellte Menge von Daten
- Sie besteht aus einer Folge gleichartig aufgebauter Datensätze



- ein Datensatz besteht aus mehreren Feldern unterschiedlichen Typs
- die Anzahl der Datensätze muss nicht festgelegt werden
- Dateien werden i.d.R. dauerhaft auf Hintergrundspeichern (z.B. Magnetplatte) gespeichert



5.2.1 Die Datenstruktur "Datei" (*file*) ...

Gängige Datei-Organisationen

➤ Sequentielle Datei

- Daten sind fortlaufend gespeichert und können nur in dieser Reihenfolge gelesen werden
- es gibt ein Dateifenster, das bei jedem Lesen bzw. Schreiben um eine Position (einen Datensatz) weiterrückt
- es kann nur jeweils der Datensatz im Dateifenster gelesen bzw. geschrieben werden

Dateifenster

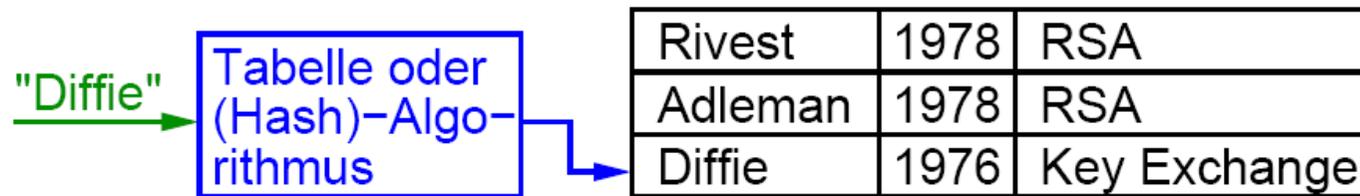
Adleman	1978	RSA
Diffie	1976	Key Exchange
Rivest	1978	RSA
Shamir	1978	RSA

- das Dateifenster kann z.T. auch direkt positioniert werden

5.2.1 Die Datenstruktur "Datei" (*file*) ...

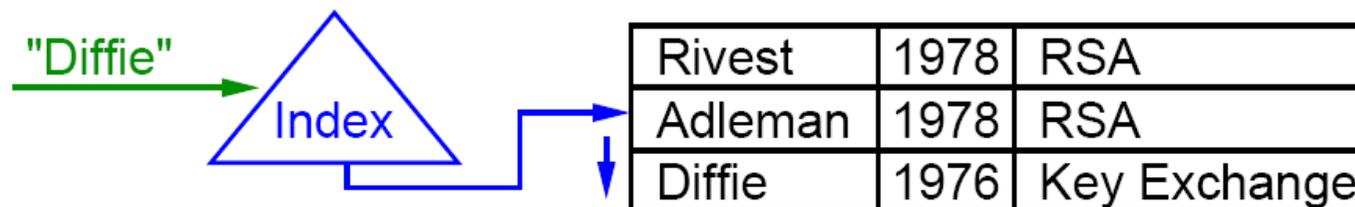
➤ Direkte Datei

- Zugriff auf Datensätze erfolgt über einen Schlüssel, aus dem direkt die Position in der Datei bestimmt wird



➤ Indexsequentielle Datei: Mischform

- Nutzung einer Tabelle (Index), die für einen Schlüssel in die Nähe des Datensatzes führt. Von dort aus sequentielle Suche





5.2.1 Die Datenstruktur "Datei" (*file*) ...

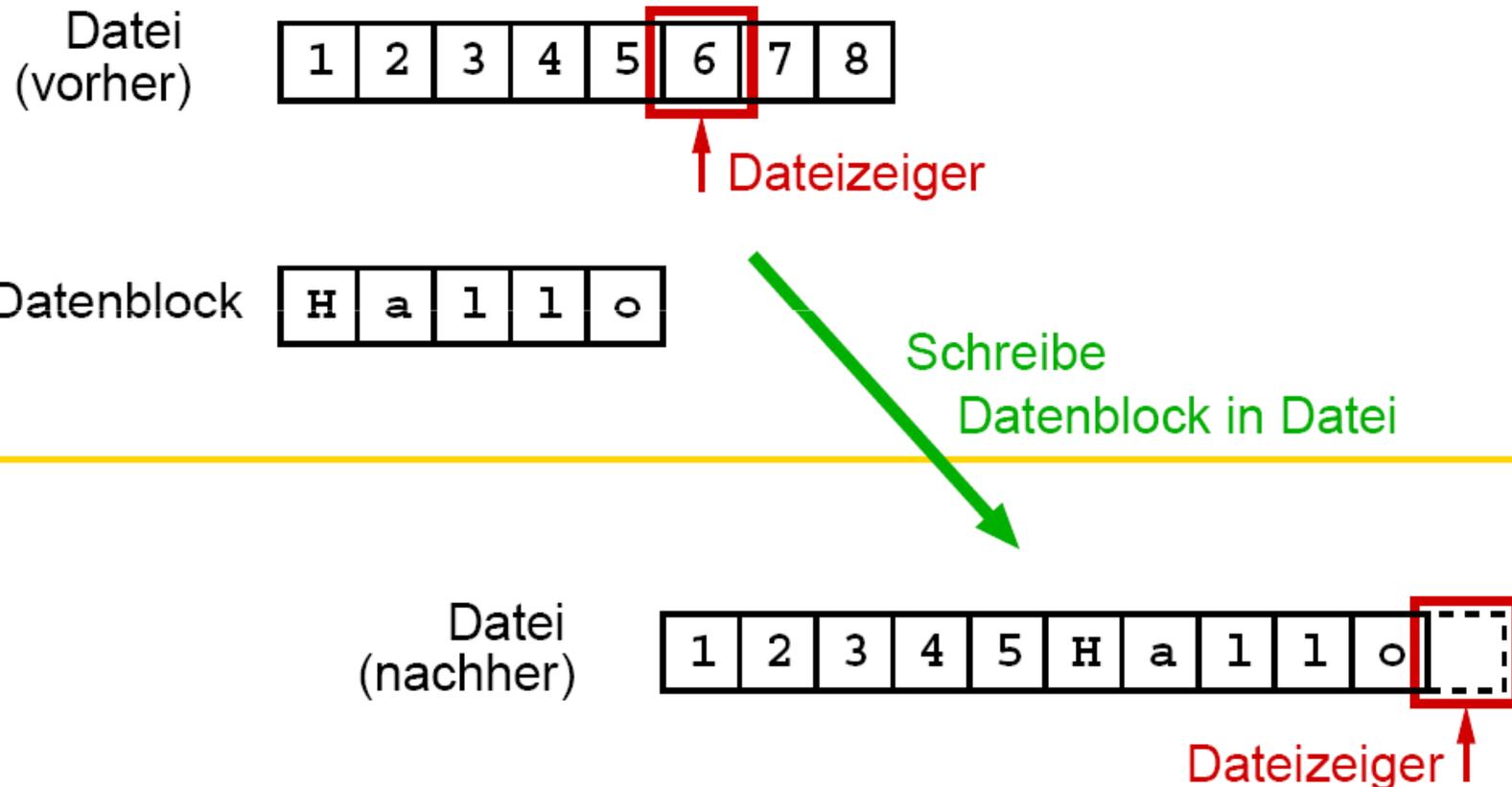
Das Dateimodell von Java

- Eine Datei in Java ist eine (unstrukturierte) Folge von Bytes
 - z.B. Textdatei: Folge von 8-Bit-Zeichen
- Nach dem Öffnen einer Datei verweist ein **Dateizeiger** auf das nächste zu lesende bzw. zu schreibende Byte
- Lese- und Schreiboperationen kopieren einen Datenblock aus der Datei bzw. in die Datei
 - der Dateizeiger wird entsprechend weitergeschoben
- Lesen über das Dateiende hinaus (**End-of-file, EOF**) ist *nicht* möglich
- Schreiben über das Dateiende führt zum **Anfügen** an die Datei
- Der Dateizeiger kann auch explizit positioniert werden



5.2.1 Die Datenstruktur "Datei" (*file*) ...

Beispiel: Schreiben in eine (Text-)Datei





5.2.1 Die Datenstruktur "Datei" (*file*) ...

Grundoperationen auf Dateien

- **öffnen (*open*)** einer durch ihren Namen gegebenen Datei
 - zum Lesen: Dateizeiger wird auf Anfang positioniert
 - zum Schreiben: Dateizeiger wird auf Anfang bzw. Ende positioniert (überschreiben der Datei bzw. Anfügen)
 - i.d.R. wird beim Öffnen auch ein **Dateipuffer** eingerichtet
 - speichert einen Teil der Datei im Hauptspeicher zwischen
 - verhindert, dass jede Datei-Operation sofort auf dem langsamen Hintergrundspeicher ausgeführt werden muss
- **Schließen (*close*)** einer geöffneten Datei
 - Dateien sollten nach Verwendung immer geschlossen werden
 - sonst evtl. Datenverlust: Zurückschreiben des Dateipuffers
 - nach dem Schließen sind keine Operationen mehr zulässig



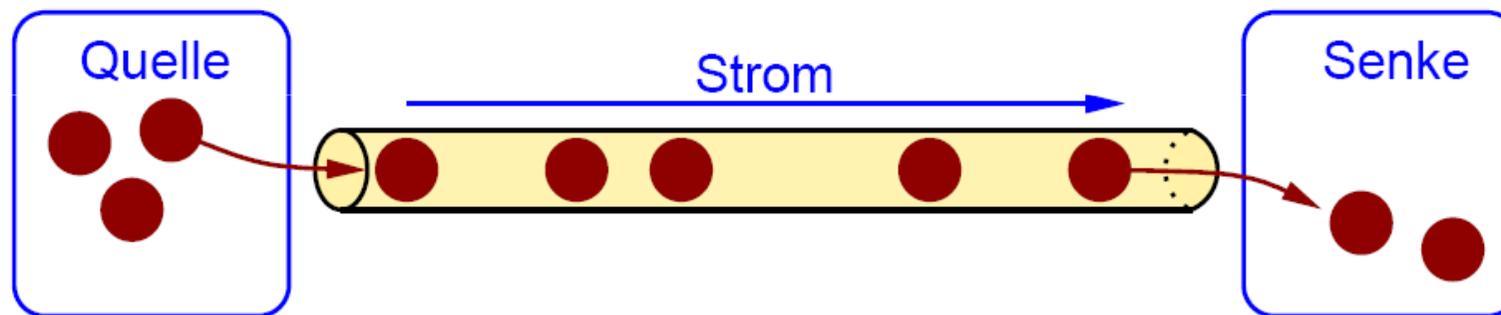
5.2.1 Die Datenstruktur "Datei" (*file*) ...

- **Lesen (*read*)** eines Datenblocks
 - die Daten ab dem Dateizeiger werden in eine Variable (z.B. Byte-Array) kopiert
 - Dateizeiger wird entsprechend weiterbewegt
- **Schreiben (*write*)** eines Datenblocks
 - Inhalt einer Variable (z.B. Byte-Array) wird ab dem Dateizeiger in die Datei kopiert (ggf. angefügt)
 - Dateizeiger wird entsprechend weiterbewegt
- ***flush***: Leeren des Dateipuffers
 - Inhalt des Dateipuffers wird in die Datei zurückgeschrieben
- ***seek***: explizites Positionieren des Dateizeigers
 - ermöglicht wahlfreien Zugriff auf die Datei

5.2 Dateien, Ströme und Serialisierung ...

5.2.2 Ein- und Ausgabe mit Strömen (*Streams*)

- In Java erfolgt jede Ein-/Ausgabe (auch in Dateien) über Ströme
 - sie stellen die Schnittstelle des Programms nach außen dar
- Ein **Strom** ist eine geordnete Folge von Daten mit einer Quelle und einer Senke



- Ströme sind i.d.R. unidirektional (entweder Ein- oder Ausgabe)
- ein Strom puffert die Daten so lange, bis sie von der Senke entnommen werden (Warteschlange)



5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...

Wichtige Strom-Klassen / Schnittstellen im Paket `java.io` ...

- **Abstrakte Basisklassen:** `InputStream`, `OutputStream`
 - allgemeine Ströme für Ein- bzw. Ausgabe
- **Dateiströme:** `FileInputStream`, `FileOutputStream`
 - spezielle Ströme für die Ein-/Ausgabe auf Dateien
- **Bidirektionaler Dateistrom:** `RandomAccessFile`
 - ermöglicht zusätzlich Positionieren des Dateizeigers
- **Filterströme:** `FilterInputStream`, `FilterOutputStream`
 - erhalten Daten von einem anderen Strom und filtern diese bzw. geben gefilterte Daten an einen anderen Strom weiter
 - Filterung: z.B. Umwandlung von Datentypen in Byteströme



5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...

Wichtige Strom-Klassen / Schnittstellen im Paket `java.io` ...

- Schnittstellen `DataInput`, `DataOutput`
 - definieren Operationen zur Ein-/Ausgabe von einfachen Datentypen (`int`, `double`, ...) und Strings
 - implementiert von den Filterströmen `DataInputStream`, `DataOutputStream`
- Schnittstellen `ObjectInput`, `ObjectOutput`
 - definieren Operationen zur Ein-/Ausgabe von Objekten
 - implementiert von den Strömen `ObjectInputStream`, `ObjectOutputStream`
- Strom zur formatierten Text-Ausgabe von Daten: `PrintStream`

5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...



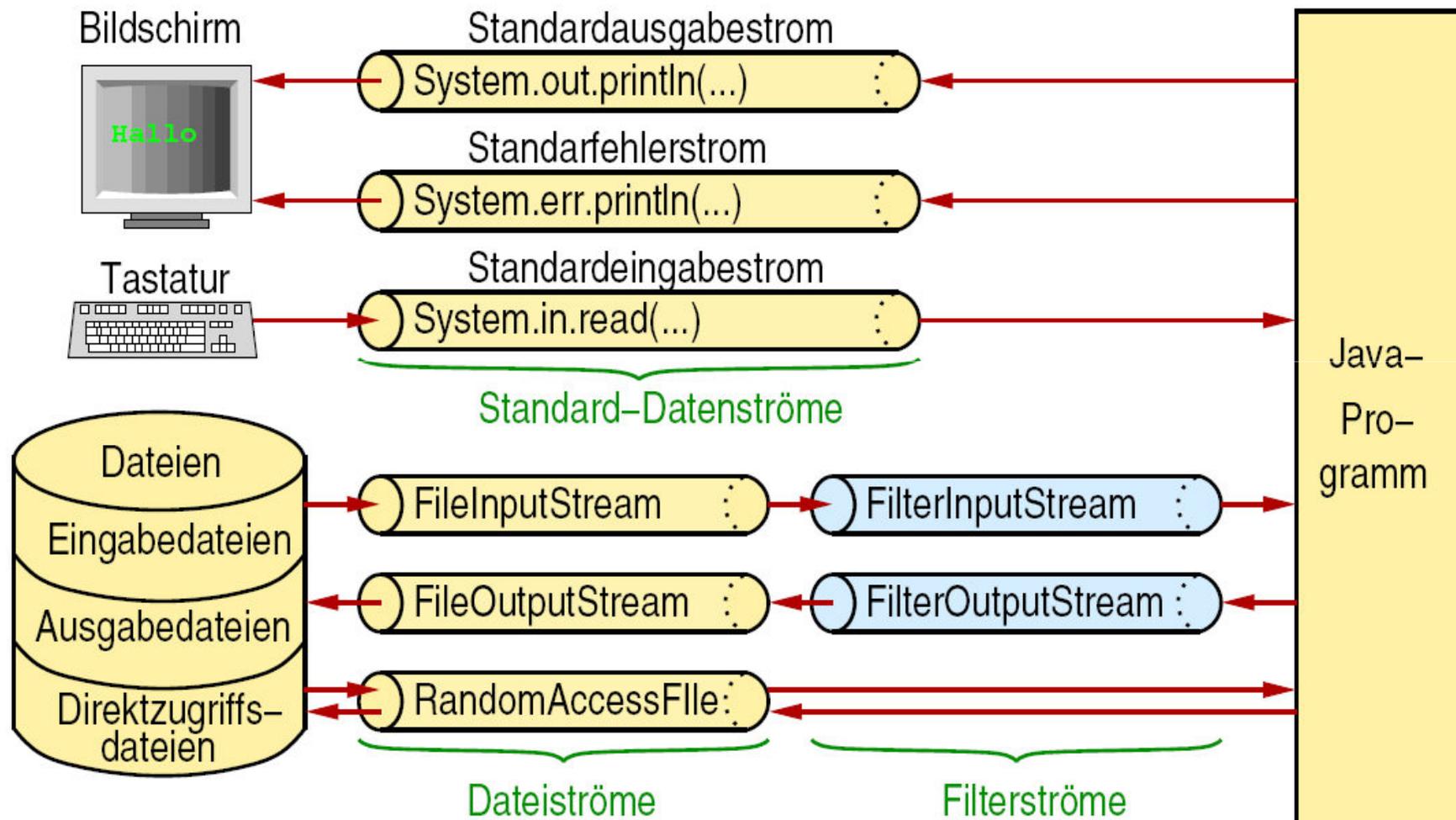
Standard-Datenströme

- Java definiert drei Standard-Datenströme für die Ein-/Ausgabe von / zur Konsole:
 - `InputStream System.in`
 - zum Einlesen von Zeichen von der Tastatur
 - `PrintStream System.out`
 - zur Ausgabe von Zeichen auf den Bildschirm
 - z.B. `System.out.println("Hallo");`
 - `PrintStream System.err`
 - zur Ausgabe von Zeichen auf den Bildschirm
 - speziell für Fehlermeldungen

5.2.2 Ein- und Ausgabe mit Strömen (Streams) ...



Veranschaulichung des Stromkonzepts in Java





5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...

Wichtige Operationen der Klasse `InputStream`

- `abstract int read() throws IOException`
 - liest ein Byte (0 ... 255) aus dem Strom
 - blockiert, falls keine Eingabe verfügbar ist
 - am Stromende (z.B. Dateiende) wird -1 zurückgegeben
- `int read(byte[] buf) throws IOException`
 - liest bis zu `buf.length` Bytes aus dem Strom
 - blockiert, bis eine Eingabe verfügbar ist
 - Ergebnis: Zahl der gelesenen Bytes bzw. -1 am Stromende
- `void close() throws IOException`
 - schließt den Strom: Freigabe belegter Ressourcen



5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...

Wichtige Operationen der Klasse `OutputStream`

- `abstract void write(int b) throws IOException`
 - schreibt das Byte `b` (0 ... 255) in den Strom
 - (nur die unteren 8 Bit von `b` sind relevant)
 - `void write(byte[] buf) throws IOException`
 - schreibt die Bytes aus `buf` in den Strom
 - `void flush() throws IOException`
 - leert den Puffer des Stroms
 - alle noch im Puffer stehenden Bytes werden z.B. auf den Bildschirm oder in die Datei geschrieben
 - `void close() throws IOException`
 - schließt den Strom: Freigabe belegter Ressourcen
-



5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...

Beispiel: Bytes im Eingabestrom zählen (+ WWW: Count.java)

```
import java.io.*;
public class Count {
    // IOException wird nicht gefangen, dies muß deklariert werden
    public static void main(String[] args) throws IOException {
        int count = 0;
        // Zeichen einlesen bis Stromende (^D, ^Z)
        while (System.in.read() != -1)
            count++;
        String msg = "Eingabe hatte " + count + " Bytes\n";
        // Nur zur Demonstration. Ausgabe i.a. mit println()
        System.out.write(msg.getBytes());
    }
}
```

5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...



Dateiströme

- Konstruktoren (u.a.):
 - `FileInputStream(String path)` throws `FileNotFoundException`
 - öffnet Datei mit angegebenem Namen zum Lesen
 - `FileOutputStream(String path)` throws `FileNotFoundException`
 - öffnet Datei mit angegebenem Namen zum Schreiben
 - Datei wird ggf. neu erzeugt
- Operationen:
 - werden von `InputStream` bzw. `OutputStream` geerbt
 - teilweise mit neuen Implementierungen überschrieben



5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...

Beispiel: Datei kopieren

```
public static void copyFile(String from, String to)
    throws IOException {
    // Ein- und Ausgabedateien öffnen
    FileInputStream in = new FileInputStream(from);
    FileOutputStream out = new FileOutputStream(to);
    // Datei byteweise kopieren
    int b = in.read();
    while (b != -1) {
        out.write(b);
        b = in.read();
    }
    // Dateien schließen
    in.close();
    out.close();
}
```

Alternative Codierung der Schleife:

```
int b;
while ((b = in.read()) != -1) {
    out.write(b);
}
```

5.2.2 Ein- und Ausgabe mit Strömen (*Streams*) ...



Beispiel: Datei kopieren ...

```
// Aufruf: java CopyFile <Eingabedatei> <Ausgabedatei>
public static void main(String[] args) {
    if (args.length != 2) {
        System.err.println("Programm benötigt 2 Argumente: " +
            "<Eingabedatei> <Ausgabedatei>");

        return;
    }
    try {
        copyFile(args[0], args[1]);
    }
    catch (IOException e) {
        System.err.println("Fehler beim Kopieren: " + e);
    }
}
```



5.2 Dateien, Ströme und Serialisierung ...

5.2.3 Serialisierung von Objekten

- **Ziel:** einmal erzeugte Objekte sollen auch über das Ende des Programms hinaus gespeichert bleiben
- **Persistenz:** Langfristige Speicherung von Objekten mit ihren Zuständen und Beziehungen, so dass ein analoger Zustand im Arbeitsspeicher wiederhergestellt werden kann
- **Serialisierung:** Umwandlung des Zustands eines Objekts in einen **Byte-Strom** bzw. umgekehrt (Deserialisierung)
 - der Byte-Strom lässt sich dann in eine Datei ausgeben bzw. von dort wieder einlesen
 - das Objekt kann dabei Referenzen auf Arrays und andere Objekte enthalten, die automatisch mit serialisiert werden



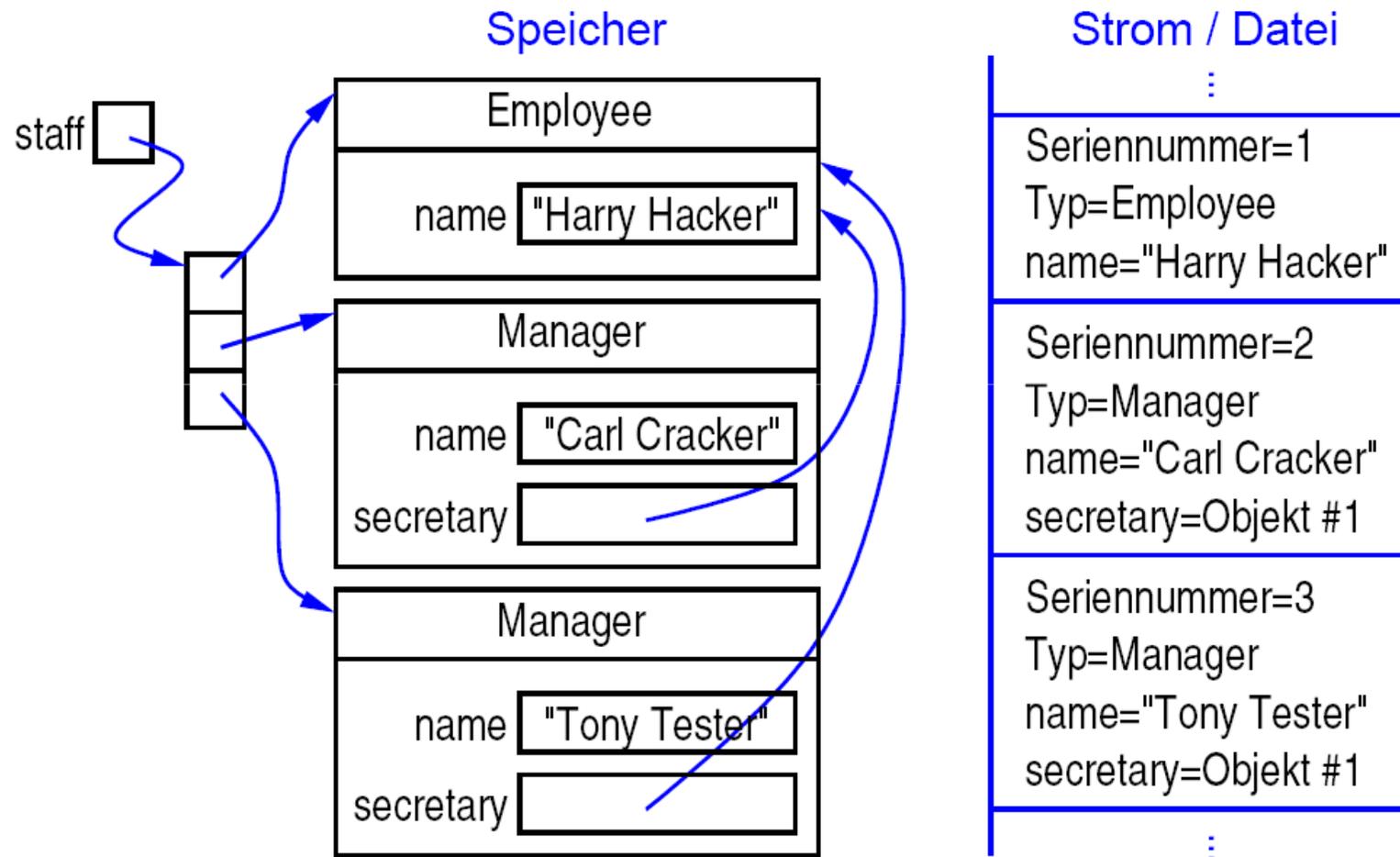
5.2.3 Serialisierung von Objekten ...

Was geschieht bei der Serialisierung eines Objekts / Arrays?

- 1. Erzeuge eine eindeutige Seriennummer für das Objekt und schreibe diese in den Strom
- 2. Schreibe Information zur Klasse in den Strom
 - u.a. Klassenname, Attributnamen und -typen
- 3. Für alle Attribute des Objektes (bzw. Elemente des Arrays):
 - falls keine Referenz: schreibe den Wert in den Strom
 - sonst: Z = Ziel der Referenz
 - falls Z noch nicht in diesen Strom serialisiert wurde:
 - serialisiere Z (Rekursion!)
 - sonst: schreibe die Seriennummer von Z in den Strom

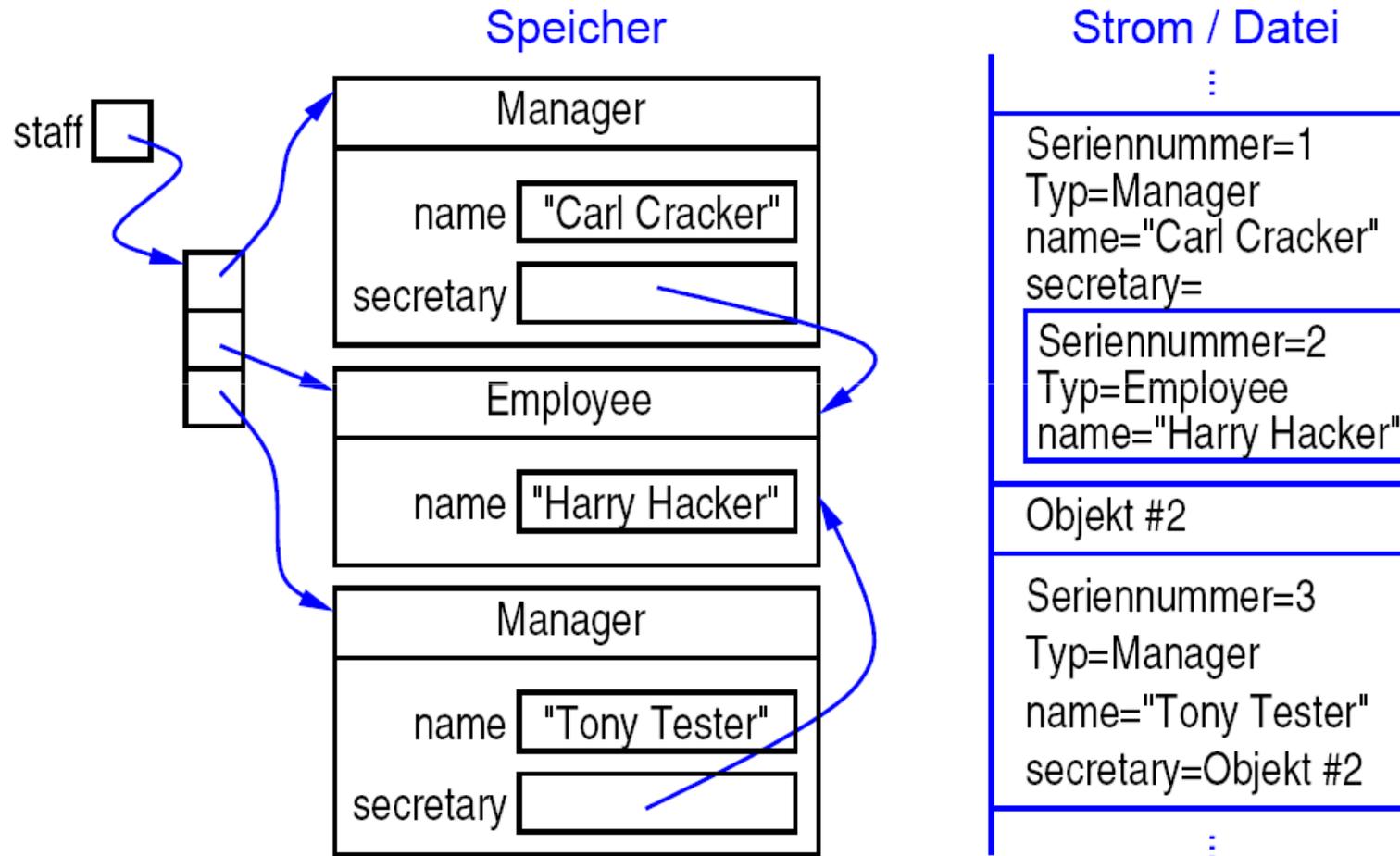
5.2.3 Serialisierung von Objekten ...

Beispiel



5.2.3 Serialisierung von Objekten ...

Beispiel





5.2.3 Serialisierung von Objekten ...

Voraussetzung für die Serialisierbarkeit von Objekten

- Die Klasse muß die Schnittstelle `Serializable` implementieren
 - `Serializable` besitzt weder Methoden noch Attribute (Interface `Serializable` in `java.io`)
 - die Schnittstelle dient nur der Markierung einer Klasse als serialisierbar (*Marker-Interface*)
- Zudem müssen alle Referenzen in dem Objekt wieder auf serialisierbare Objekte verweisen
- Beispiel:

```
class Person implements Serializable {
    private String name; // String ist serialisierbar
    private Address adresse;
}
class Address implements Serializable {
    ...
}
```



5.2.3 Serialisierung von Objekten ...

Die Klasse `ObjectOutputStream`

- Realisiert die Serialisierung von Objekten
- **Konstruktor:** `ObjectOutputStream(OutputStream out) throws IOException`
- `void writeObject(Object obj) throws IOException`
 - serialisiert obj in den Ausgabestrom
- `void reset() throws IOException`
 - löscht alle Information darüber, welche Objekte bereits in den Strom geschrieben wurden
 - nachfolgendes `writeObject()` schreibt Objekte erneut in den Strom
- **zusätzlich:** alle Methoden der Schnittstelle `DataOutput`



5.2.3 Serialisierung von Objekten ...

Was schreibt `writeObject(obj)` in den Ausgabestrom?

- Falls `obj` noch nicht in den Strom geschrieben wurde:
 - neben `obj` werden auch alle von `obj` aus erreichbaren Objekte serialisiert
 - es wird also immer ein ganzer **Objekt-Graph** serialisiert
 - `obj` heißt **Wurzelobjekt** des Objekt-Graphen
 - die Referenzen zwischen den Objekten werden bei der Deserialisierung automatisch wiederhergestellt
- Falls `obj` bereits in den Strom geschrieben wurde (und kein `reset()` ausgeführt wurde):
 - es wird nur ein "Verweis" (Seriennummer) auf das schon im Strom befindliche Objekt geschrieben



5.2.3 Serialisierung von Objekten ...

Die Klasse `ObjectInputStream`

- **Konstruktor:** `ObjectInputStream(InputStream in)` throws `IOException`
- `Object readObject()` throws `IOException`
 - liest das nächste Objekt aus dem Eingabestrom
 - falls Objekt bereits vorher gelesen wurde (ohne `reset()`): Ergebnis ist Referenz auf das schon existierende Objekt
 - sonst: Objekt und alle in Beziehung stehenden Objekte lesen
 - Objekte werden **neu erzeugt**, besitzen denselben Zustand und dieselben Beziehungen wie die geschriebenen Objekte
 - Ergebnis ist Referenz auf das Wurzelobjekt
 - i.d.R. explizite Typkonversion des Ergebnisses notwendig
- **zusätzlich:** alle Methoden der Schnittstelle `DataInput`



5.2.3 Serialisierung von Objekten ...

Die Schnittstellen `DataOutput` und `DataInput`

- Einige Methoden von `DataOutput`:
 - `void writeInt(int v) throws IOException`
 - schreibt ganze Zahl in den Strom (in Binärform: 4 Bytes)
 - `void writeDouble(double v) throws IOException`
 - schreibt Gleitkomma-Zahl (in Binärform: 8 Bytes)
- Einige Methoden von `DataInput`:
 - `int readInt() throws IOException`
 - `double readDouble() throws IOException`
 - bei Leseversuch am Dateiende: `EOFException`
- Weitere Operationen: siehe Java-Dokumentation



5.2.3 Serialisierung von Objekten ...

Beispiel: Studentendatei

```
import java.io.*;
class Name implements Serializable {
    String name
    String vorname;
    public Name(String n, String vn) { ... }
}
class Student implements Serializable {
    Name name
    int matrNr;
    double note;
    public Student(String n, String vn, int mn) {
        name = new Name(n, vn); ...
    }
}
```



5.2.3 Serialisierung von Objekten ...

Beispiel: Studentendatei ...

```
public static void main(String[] args) {  
    ObjectOutputStream oos = null;  
    try {  
        Student s = new Student("Hugo", "Test", 12345678);  
        oos = new ObjectOutputStream(  
            new FileOutputStream("out.ser"));  
        oos.writeObject(s); // Schreibe Objekt s  
        s.setNote(3.7);  
        oos.reset(); // Sonst wird nur eine weitere Referenz  
                    // auf dasselbe Objekt geschrieben  
        oos.writeObject(s); // Schreibe Objekt s nochmal  
    }  
    catch (...) { ... }  
    finally { ... oos.close(); ... }
```



5.2.3 Serialisierung von Objekten ...

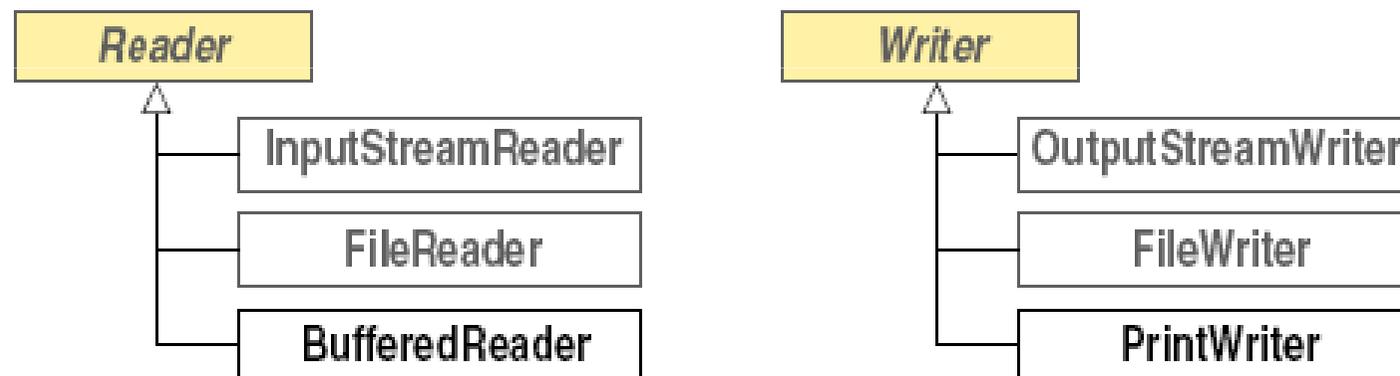
Beispiel: Studentendatei ...

```
ObjectInputStream ois = null;  
try {  
    ois = new ObjectInputStream(  
        new FileInputStream("out.ser"));  
    // Objekte von Datei einlesen und Typ umwandeln  
    Student s1 = (Student) ois.readObject();  
    Student s2 = (Student) ois.readObject();  
    System.out.println(s1);  
    System.out.println(s2);  
    System.out.println(s1 == s2);  
}  
catch (...) { ... }  
finally { ... ois.close(); ... }  
}
```

5.2 Dateien, Ströme und Serialisierung ...

5.2.4 Formatierte Text-Ein-/Ausgabe

- Java-Ströme arbeiten byte-orientiert, nicht zeichen-orientiert
 - d.h. Daten werden im Strom binär übertragen, nicht als Text
- Für die text-basierte Ein-/Ausgabe stellt Java zusätzliche Klassen zur Verfügung, u.a.:



- `Reader` und `Writer` stellen Basis-Methoden für die zeichenweise Ein-/Ausgabe zur Verfügung



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Wichtige *Reader-* und *Writer-Klassen*

- `InputStreamReader`, `OutputStreamWriter`
 - erlauben zeichenweise Ein-/Ausgabe über einen Byte-Strom
 - Umwandlung zwischen Zeichen und Bytes abhängig vom benutzten System-Zeichensatz (z.B. ISO-8859-1, UTF-8)
- `FileReader`, `FileWriter`: **Hilfsklassen**
 - **erzeugen** `InputStreamReader` **auf** `FileInputStream` **bzw.** `OutputStreamWriter` **auf** `FileOutputStream`
- `BufferedReader`: **gepuffertes** `InputStreamReader`
 - erlaubt auch das Lesen von **Textzeilen aus dem Strom**
 - **Methode** `String readLine()` `throws IOException`
- `PrintWriter`: **formatierte Text-Ausgabe** von Daten / Objekten



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Formatierte Ausgabe

- Die Klasse `PrintWriter` bietet zwei überladene Methoden zur Ausgabe von einfachen Datentypen, Strings und Objekten:
 - `void print (... arg), void println (... arg)`
 - beide geben ihr Argument formatiert in den Strom aus
 - Objekte werden über ihre Methode `toString()` in Strings umgewandelt
 - die Methoden werfen keine Exceptions
 - `println()` gibt am Ende noch einen Zeilenvorschub aus
- Anmerkung: `System.out` und `System.err` sind Objekte der Klasse `PrintStream`
 - sie bietet (aus historischen Gründen) fast die gleiche Funktionalität wie `PrintWriter`



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Erzeugung und Nutzung eines `PrintWriters`

- Konstruktoren: `PrintWriter(OutputStream out)`
`PrintWriter(Writer out)`

- Beispiel: komma-separierte Ausgabe in eine Datei:

```
PrintWriter pw;  
pw = new PrintWriter(new FileWriter("dat.txt"));  
// oder ausführlicher:  
// pw = new PrintWriter(new OutputStreamWriter(  
// new FileOutputStream("dat.txt")));  
for (int i=1; i<10; i++) {  
    pw.println(i + "," + i*i + "," + Math.sqrt(i));  
}  
pw.close();
```



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Formatierte Eingabe

- Die Klasse `BufferedReader` erlaubt das Einlesen von Textzeilen aus einem Strom

- Zur weiteren Aufteilung in einzelne Datenfelder kann die Methode

```
String[] split(String regex)
```

der Klasse `String` verwendet werden

- `regex` definiert die Zeichenkette, die die Felder trennt, als **regulären Ausdruck** (= eine Form der Syntaxbeschreibung)

- `regex` ist im einfachsten Fall ein einzelnes Trennzeichen

- Beispiel: komma-separierte Werte:

```
String zeile = "2,Hallo,0.1";
```

```
String[] felder = zeile.split(",");
```

```
// felder[0]="2", felder[1]="Hallo", felder[2]="0.1"
```



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Formatierte Eingabe ...

- Verbleibende Aufgabe: Umwandlung der einzelnen Teilstrings in Werte einfacher Datentypen
 - Lösung: Java definiert für jeden einfachen Datentyp eine entsprechende Wrapper-Klasse
 - z.B.: `Integer` für `int`, `Double` für `double`, `Boolean` für `boolean`
 - Die Wrapper-Klassen besitzen u.a. eine statische Methode, die Strings in Werte des Typs umwandelt, z.B.:
 - `static int parseInt(String s)` in Klasse `Integer`
 - `static double parseDouble(String s)` in Klasse `Double`
 - bei syntaktisch inkorrekten Eingabestrings werfen die Methoden eine `NumberFormatException`
 - (Daneben verpacken diese Klassen einfache Werte in Objekte)
-



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Beispiel: Studentendatei

```
import java.io.*;
class Student {
    String name, vorname;
    int matrNr;
    double note;
    Student(BufferedReader reader) throws IOException {
        try {
            String line = reader.readLine();
            String[] fields = line.split(",");
            name = fields[0];
            vorname = fields[1];
            matrNr = Integer.parseInt(fields[2]);
            note = Double.parseDouble(fields[3]);
        }
    }
}
```



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Beispiel: Studentendatei ...

```
    catch (NullPointerException e) {  
        throw new IOException("Unerwartetes Dateiende");  
    }  
    catch (NumberFormatException e) {  
        throw new IOException("Falsches Elementformat");  
    }  
    catch (IndexOutOfBoundsException e) {  
        throw new IOException("Zu wenig Datenelemente");  
    }  
}  
  
public void writeToStream(PrintWriter pw) {  
    pw.println(name + "," + vorname + "," + matrNr  
               + "," + note);  
    pw.flush();  
}
```



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Beispiel: Studentendatei ...

```
public static void main(String[] args) {  
    PrintWriter pw = null;  
    try {  
        Student s = new Student("Hugo", "Test", 12345678);  
        s.writeToStream(new PrintWriter(System.out));  
        pw = new PrintWriter(new FileWriter("out.txt"));  
        s.writeToStream(pw);  
    }  
    catch (FileNotFoundException e) { ... }  
    catch (IOException e) { ... }  
    finally {  
        if (pw != null) pw.close(); // Keine IOException!  
    }  
    BufferedReader reader = null;  
}
```



5.2.4 Formatierte Text-Ein-/Ausgabe ...

Beispiel: Studentendatei ...

```
try {
    reader = new BufferedReader(new FileReader("out.txt"));
    Student s1 = new Student(reader);
    Student s3 = new Student(new BufferedReader(
        new InputStreamReader(System.in)));
    System.out.println(s1);
    System.out.println(s3);
}
catch (FileNotFoundException e) { ... }
catch (IOException e) { ... }
finally {
    try { reader.close(); } catch (Exception e) { ... }
}
}
```



5.3 Das Java Collection Framework

Container-Klassen

- Dienen zur Speicherung und Verwaltung von Objekten
 - einfachstes Beispiel für einen *Container* ist ein *Array*
- Das Java Collection Framework stellt im Paket `java.util` eine Anzahl vordefinierter *Container* -Klassen bereit
 - Anwendungsklassen können diese Klassen benutzen oder von Ihnen erben
- Zwei grundlegende Arten von *Containern*:
 - **Collection**: reine Sammlung von Objekten (z.B. Studentenliste)
 - **Map**: Abbildung zwischen Objekten (z.B. Telefonverzeichnis)
- Idee des Java Collection Frameworks: Definition gemeinsamer Schnittstellen (= **abstrakte Datenstrukturen**)
 - verschiedene Implementierungen, je nach Verwendungszweck



5.3 Das Java Collection Framework ...

5.3.1 Collections (Sammlungen)

- Prinzipiell gibt es zwei Arten von Sammlungen:
 - **Sequenz (Liste, List)**
 - legt eine bestimmte Reihenfolge ihrer Elemente fest
 - erlaubt, dass dasselbe Element auch mehrfach auftritt
 - **Menge (Set)**
 - ein Element kann nicht mehrfach vorkommen
 - es ist keine bestimmte Reihenfolge der Elemente definiert
- Bei Sequenzen und Mengen ist die Zahl der Elemente nicht im Voraus festgelegt und auch nicht begrenzt
 - d.h. sie können dynamisch (zur Laufzeit) wachsen
 - im Gegensatz zu Arrays, deren Größe bei der Erzeugung festgelegt werden muss



5.3.1 Collections (Sammlungen) ...

Basisoperationen der abstrakten Datenstruktur "Sammlung"

- Einfügen eines Elements x in die Sammlung
 - bei Mengen: kein Mehrfacheintrag
 - bei Sequenzen: ggf. mit Angabe der Einfügestelle
- Löschen eines Elements x aus der Sammlung
- Abfrage, ob x Element der Sammlung ist
- Löschen aller Elemente der Sammlung
- Bestimmung der Anzahl der Elemente in der Sammlung
- Abfrage, ob die Sammlung leer ist
- Durchlaufen aller Elemente der Sammlung
 - bei Mengen: Reihenfolge ist unbestimmt



5.3.1 Collections (Sammlungen) ...

Ein Problem: welchen Typ haben die Elemente ...

- Einerseits wollen wir beliebige Elementtypen zulassen
 - Andererseits müssen wir in der Schnittstelle z.B. bei der Deklaration der Einfügeoperation einen Typ für das Argument angeben
 - Lösungsmöglichkeiten:
 - die Elemente haben den Datentyp Object (bis Java 1.4)
 - alle Klassen sind Unterklassen von Object
 - Nachteil: die Typsicherheit geht verloren
 - **Generische Datentypen** (ab Java 5)
 - die Schnittstellen-Deklaration erhält den Elementtyp als Parameter (**Typparameter**)
 - der Typ des Arguments z.B. beim Einfügen ist dann dieser Parameter
-



5.3.1 Collections (Sammlungen) ...

Einschub: Generische Datentypen (ab Java 5)

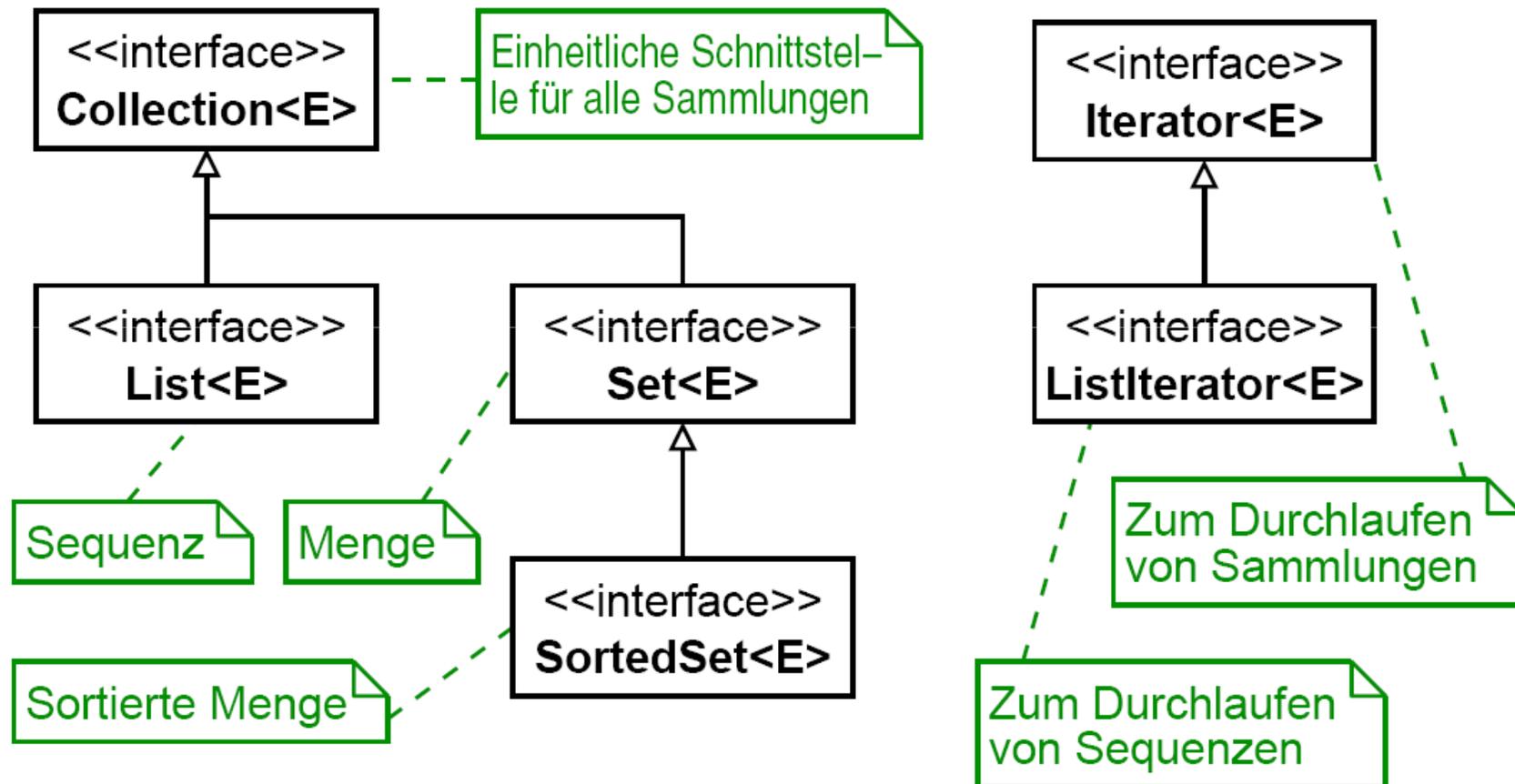
- Klassendeklarationen können mit formalen Parametern für Typen versehen werden
- diese können als Typname in Deklarationen innerhalb der Klasse verwendet werden
- z.B.:

```
public class GenClass<T> {  
    T attribute;  
    public void set(T val) { attribute = val; }  
}
```
- Bei Verwendung der Klasse muß dann ein konkreter Typ als aktueller Parameter angegeben werden
 - z.B.:

```
GenClass<String> gs = new GenClass<String>();
```
- Analog auch für Schnittstellen

5.3.1 Collections (Sammlungen) ...

Schnittstellen für Sammlungen in Java





5.3.1 Collections (Sammlungen) ...

Basis-Operationen der Schnittstelle `Collection<E>`

- `boolean add(E o)`: Einfügen von `o`
 - Fügt (Referenz auf) `o` (vom Typ `E`) der Sammlung hinzu
 - Ergebnis `true`, falls Sammlung verändert wurde
- `boolean remove(Object o)`: Entfernen von `o`
 - Entfernt eine Objektreferenz `e`, für die gilt:
 - `(o == null) ? (e == null) : o.equals(e)`
 - Ergebnis `true`, falls Sammlung verändert wurde
- `boolean contains(Object o)`: ist `o` enthalten?
- Liefert `true`, g.d.w. Sammlung eine Objektreferenz `e` enthält mit
 - `(o == null) ? (e == null) : o.equals(e)`
- `void clear()`: Löschen aller Elemente



5.3.1 Collections (Sammlungen) ...

Basis-Operationen der Schnittstelle `Collection<E>` ...

- `int size()`: Zahl der Elemente
- `boolean isEmpty()`: ist die Sammlung leer?
- `Iterator<E> iterator()`: zum Durchlaufen der Sammlung
 - liefert ein **Iterator**-Objekt zurück

Operationen der Schnittstelle `Iterator<E>`

- `E next()`: gibt das nächste Element zurück
 - falls keines mehr existiert: `NoSuchElementException`
- `boolean hasNext()`: gibt es ein nächstes Element?
- `void remove()`: entferne das zuletzt von `next()` gelieferte Element aus der Sammlung

5.3.1 Collections (Sammlungen) ...



Beispiel Notizbuch-Verwaltung

-  PDA's / Notizen verwalten über Treffen, Termine, Geburtstage
-  Notizen speichern
-  Anzahl der Notizen nicht begrenzt
-  Einzelne Notizen anzeigen
-  Anzahl der Notizen abfragen



Andere Anwendungen: Bibliotheken / Bücher, Zeitschriften, ...

Universitäten / Studenten, ...





5.3.1 Collections (Sammlungen) ...

Beispiel Notizbuch-Verwaltung

```
import java.util.ArrayList; //Klasse ArrayList

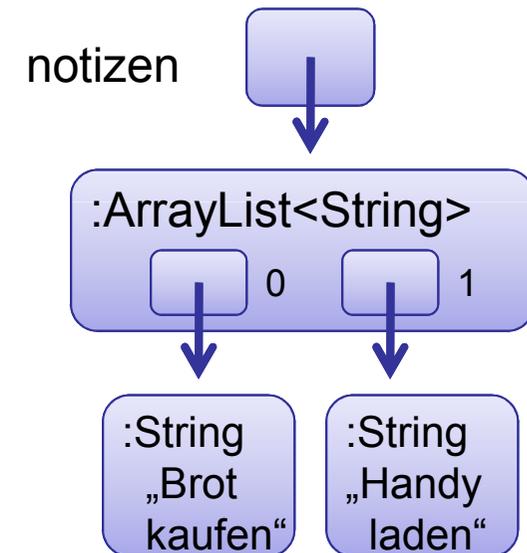
public class Notizbuch {
    //Speicher für beliebige Notizen
    private ArrayList<String> notizen;

    //Initialisierung
    public Notizbuch() {
        notizen = new ArrayList<String>(); }

    //Speichere Notiz
    public void speichereNotiz(String notiz)
    { notizen.add(notiz); }

    //Anzahl der Notizen
    public int anzahlNotizen()
    { return notizen.size(); }

    ... }
}
```



```
-> System.out.println(notizen.get(0));
```



5.3.1 Collections (Sammlungen) ...

Diskussion des Designs

- Design-Entscheidung: eine gemeinsame Schnittstelle für alle denkbaren Sammlungs-Klassen
 - neben Sequenz und Menge auch Keller, Warteschlange, ...
- Vorteil: einheitlicher Zugriff auf alle Sammlungen
 - gibt z.B. eine Methode eine `Collection` zurück, kann darauf unabhängig vom konkreten Sammlungs-Typ in einheitlicher Weise zugegriffen werden
- Problem: nicht alle Sammlungen haben eine Schnittstelle, wie sie in `Collection` definiert ist
 - daher: Operationen zum Ändern der `Collection` sind optional
 - sie werfen ggf. eine `UnsupportedOperationException`
 - Laufzeitfehler statt Fehler zur Übersetzungszeit!



5.3.1 Collections (Sammlungen) ...

Details zur Schnittstelle `Collection`

- Eine `Collection<E>` speichert Referenzen auf Objekte der Klasse `E`
 - damit können Objekte der Klasse `E` und aller Unterklassen von `E` verwaltet werden
 - in den Klassen muss ggf. die Methode `equals()` sinnvoll überschrieben werden
 - für Gleichheit statt Identität
- auch möglich: `Collection<Object>`
 - kann Objekte **beliebiger Klassen** aufnehmen
 - je nach `Collection`-Implementierung auch gleichzeitig in derselben `Collection`
 - entspricht der Realisierung in Java 1.4



5.3.1 Collections (Sammlungen) ...

Details zur Schnittstelle `Collection` ...

- Eine `Collection` speichert **Referenzen** auf Objekte
 - eingefügte Elemente können nachträglich geändert werden
 - Problem bei Mengen: dadurch können zwei Elemente der Menge gleich werden
 - dies ist nicht erlaubt!
 - das Verhalten der gesamten Menge ist dann undefiniert
 - bei sortierten Mengen ergibt sich ein weiteres Problem, wenn sich dadurch die Ordnung der Elemente ändert
- Eine `Collection` speichert Referenzen auf **Objekte**
 - können also z.B. keine Mengen von `double`-Werten erzeugt werden?
 - Lösung: Wrapper-Klassen für einfache Datentypen



5.3.1 Collections (Sammlungen) ...

Einschub: Wrapper-Klassen

- Java definiert für jeden einfachen Typ auch eine zugehörige Klasse
- Ein Objekt dieser Klasse enthält genau einen Wert des zugehörigen Typs
- Beispiel: Klasse `Integer`
 - Konstruktor: `Integer(int value)`
 - Methode `int intValue()` liefert den Wert zurück
 - daneben etliche weitere Methoden zur Umwandlung von/nach `Strings` und in andere Datentypen
- Genau wie `Strings` sind die Wrapper-Klassen "*immutable*"
 - d.h. einmal erzeugte Objekte können nicht verändert werden



5.3.1 Collections (Sammlungen) ...

Beispiel: Collection-Aufbau (☞ WWW: CollectionTest.java)

```
public static void setupCollection(Collection<Integer> c) {  
    int i;  
    for (i=0; i<5; i++) c.add(new Integer(i)); // Einfügen  
    for (i=9; i>=5; i--) c.add(new Integer(i));  
    // Doppeltes Einfügen bleibt evtl. ohne Wirkung  
    c.add(new Integer(3));  
    c.add(new Integer(7));  
    // funktioniert, da Integer equals() überschreibt  
    if (c.contains(new Integer(5))) c.remove(new Integer(5));  
    // ist erlaubt, wirft aber ggf. ClassCastException  
    if (c.contains("Hallo")) System.out.println("OK");  
    System.out.println("Größe: " + c.size() + " Elemente");  
}
```



5.3.1 Collections (Sammlungen) ...

Ergebnisse im Beispiel:

➔ Mit (unsortierter) Menge (HashSet):

➔

2	4	8	9	6	1	3	7	0
---	---	---	---	---	---	---	---	---

➔ Elemente in beliebiger Reihenfolge, keine doppelten

➔ Mit sortierter Menge (TreeSet):

➔

0	1	2	3	4	6	7	8	9
---	---	---	---	---	---	---	---	---

➔ Suchen nach "Hallo" führt zu Exception

➔ Elemente in „natürlicher“ Reihenfolge

➔ Mit Sequenz (ArrayList, LinkedList):

➔

0	1	2	3	4	9	8	7	6	3	7
---	---	---	---	---	---	---	---	---	---	---

➔ Elemente in Einfüge-Reihenfolge (add() fügt hinten an)



5.3.1 Collections (Sammlungen) ...

Verwendung von Iteratoren

- Typischer Code zum Durchlaufen einer Collection:

```
static void printCollection(Collection<Integer> c) {  
    Iterator<Integer> i = c.iterator(); // Iterator holen  
    while (i.hasNext()) {           // Solange noch Elemente da  
        Integer val = i.next();    // nächstes Element holen  
        System.out.println(val);  
    }  
}
```

- siehe WWW: CollectionTest.java

- Die Modifikation der Collection während des Durchlaufens ist nur mit der `remove()`-Operation des verwendeten Iterators erlaubt

- sonst ist der Inhalt / das Verhalten der Collection undefiniert



5.3.1 Collections (Sammlungen) ...

Beispiel zum Löschen von Collection-Elementen

↪ Liste von Veranstaltungen: lösche beendete Veranstaltungen

```
class Veranstaltungen {
    LinkedList<Veranstaltung> list;
    ...;
    public void löscheBeendete() {
        Iterator<Veranstaltung> i = list.iterator();
        while (i.hasNext()) {
            Veranstaltung va = i.next();
            if (va.istBeendet()) {
                i.remove(); // remove() entfernt das Element,
                            // das beim letzten Aufruf von
                            // next() zurückgeliefert wurde
            }
        }
    }
}
```



5.3.1 Collections (Sammlungen) ...

(Zusätzliche) Operationen der Schnittstelle `List<E>`

- `void add(int index, E o)`
`boolean addAll(int index, Collection<E> c)`
 - fügt Element(e) vor der angegebenen Position (= Index) ein
 - `E get(int index)`
 - gibt das Element an der angegebenen Position zurück
 - `int indexOf(Object o)`
`int lastIndexOf(Object o)`
 - gibt die Position des ersten bzw. letzten Vorkommens des Elements o zurück
 - `E remove(int index)`: löscht Element an Position index
 - `E set(int index, E o)`: ersetzt Element bei index
-



5.3.1 Collections (Sammlungen) ...

(Zusätzliche) Operationen der Schnittstelle `List<E>` ...

- `ListIterator<E> listIterator()`
`ListIterator<E> listIterator(int index)`
 - gibt einen `ListIterator` zurück (der bei Pos. `index` startet)

Die Schnittstelle `ListIterator<E>` für Listen

- Ist von der Schnittstelle `Iterator<E>` abgeleitet
- Bietet zusätzlich die Möglichkeit
 - die Liste in beide Richtungen zu durchlaufen
 - `boolean hasNext()`, `E previous()`
 - auf den Index des nächsten / vorigen Elements zuzugreifen
 - `int nextIndex()`, `int previousIndex()`
 - das aktuelle Element zu verändern: `void set(E o)`
 - ein neues Element einzufügen: `void add(E o)`



5.3.1 Collections (Sammlungen) ...

Beispiel zu ListIterator (☞ WWW: IteratorTest.java)

```
import java.util.*;
class MyList<E> extends ArrayList<E> {
    void replace (E old, E newValue) {
        // Durchlaufe Liste vorwärts
        ListIterator<E> it = listIterator();
        while (it.hasNext())
            if (old.equals(it.next()))
                it.set(newValue);
    }
}
public class IteratorTest {
    public static void main(String[] args) {
        MyList<String> l1 = new MyList<String>()
        l1.add("Albert"); l1.add("Bodo"); l1.add("Caesar");
    }
}
```



5.3.1 Collections (Sammlungen) ...

```
l1.add("Caesar"); l1.add("Dora"); l1.add("Emil");
l1.add("Fritz"); l1.add("Gerda"); l1.add("Hans");
// Durchlaufe Liste rückwärts
ListIterator<String> it = l1.listIterator(l1.size());
while (it.hasPrevious()) {
    System.out.print(it.previous() +
        (it.hasPrevious() ? "," : "\n"));
}
System.out.println("Vor replace: " + l1);
l1.replace("Caesar", "Clara");
System.out.println("Nach replace: " + l1);
int i = l1.indexOf("Gerda"); // Element-Index
if (i >= 0) l1.set(i, "Gertrud"); // Element ersetzen
System.out.println("Nach set: " + l1);
}
}
```



5.3.1 Collections (Sammlungen) ...

Beispiel zu ListIterator ...

➤ Ausgabe:

Hans, Gerda, Fritz, Emil, Dora, Caesar, Caesar, Bodo, Albert

Vor replace: [Albert, Bodo, Caesar, Caesar, Dora, Emil, Fritz, Gerda, Hans]

Nach replace: [Albert, Bodo, Clara, Clara, Dora, Emil, Fritz, Gerda, Hans]

Nach set: [Albert, Bodo, Clara, Clara, Dora, Emil, Fritz, Gertrud, Hans]



5.3.1 Collections (Sammlungen) ...

(Zusätzliche) Operationen der Schnittstelle `Set<E>`

- Keine! Die Schnittstelle sichert lediglich die Eindeutigkeit der Elemente einer `Collection` zu

(Zusätzliche) Operationen der Schnittstelle `SortedSet<E>`

- `E first()`: kleinstes Element der Menge
- `E last()`: größtes Element der Menge



5.3.1 Collections (Sammlungen) ...

Nach welchem Kriterium ist ein SortedSet sortiert?

- Erste Möglichkeit: "natürliche Ordnung"
- Alle Elemente müssen Schnittstelle Comparable implementieren:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

- `compareTo()` legt eine Ordnung auf Objekten fest
 - die Methode liefert als Ergebnis
 - < 0 , falls this "kleiner als" o
 - $= 0$, falls this "gleich" o
 - > 0 , falls this "größer als" o
 - sie kann eine `ClassCastException` werfen, falls der Typ von o keinen Vergleich zulässt
 - `String` und alle Wrapper-Klassen implementieren `Comparable`
-



5.3.1 Collections (Sammlungen) ...

Nach welchem Kriterium ist ein SortedSet sortiert? ...

- Zweite Möglichkeit: eigens definierte Ordnung
- Dem Konstruktor der Menge wird ein `Comparator` übergeben:

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- `compare()` legt die Ordnung auf den Objekten fest
 - die Methode liefert als Ergebnis
 - < 0 , falls `o1` "kleiner als" `o2`
 - $= 0$, falls `o1` "gleich" `o2`
 - > 0 , falls `o1` "größer als" `o2`
 - sie kann eine `ClassCastException` werfen, falls der Typ von `o1` und `o2` keinen Vergleich zulässt
-



5.3.1 Collections (Sammlungen) ...

Beispiel zu sortierten Mengen (=> WWW: SortedSetTest.java)

```
class Student implements Comparable<Student> {  
    ...  
    // Vergleiche nach Name, Vorname, Matrikelnummer  
    public int compareTo(Student other) {  
        int cmp = name.compareTo(other.name);  
        if (cmp != 0) return cmp;  
        cmp = vorname.compareTo(other.vorname);  
        if (cmp != 0) return cmp;  
        return (matrNr - other.matrNr);  
    }  
    public boolean equals(Object obj) {  
        // Rückführung auf compareTo() sichert Konsistenz!  
        return (compareTo((Student) obj) == 0);  
    }  
}
```



5.3.1 Collections (Sammlungen) ...

Beispiel zu sortierten Mengen ...

```
class MatrNrComparator implements Comparator<Student> {  
    // Vergleiche nach Matrikelnummer  
    public int compare(Student s1, Student s2) {  
        return (s1.matrNr - s2.matrNr);  
    }  
}  
  
...  
  
// Sortierte Menge mit Standard-Sortierung  
TreeSet<Student> students1 = new TreeSet<Student>();  
  
// Sortierte Menge mit vorgegebener Sortierung  
TreeSet<Student> students2 =  
    new TreeSet<Student>(new MatrNrComparator());
```



5.3 Das Java Collection Framework ...

5.3.2 Implementierung von Sammlungen

- Im Java Collection Framework sind Sammlungen auf vier verschiedene Arten realisiert:
 - mit Hilfe von Arrays (`ArrayList`, `Vector`),
 - als **verkettete Listen** (`LinkedList`),
 - mit Hilfe **sortierter Bäume** (`TreeSet`),
 - und mit Hilfe von **Hashing** (`HashSet`).
- Für Mengen kleiner, nicht-negativer ganzer Zahlen gibt es daneben eine spezielle Bit-Vektor-Implementierung (`BitSet`)
- `BitSet` implementiert die Schnittstelle `Collection` nicht!

5.3.2 Implementierung von Sammlungen ...



Implementierte Schnittstellen

Klasse	Schnittstelle			
	Collection	List	Set	SortedSet
ArrayList	X	X		
Vector	X	X		
LinkedList	X	X		
TreeSet	X		X	X
HashSet	X		X	
BitSet				



5.3.2 Implementierung von Sammlungen ...

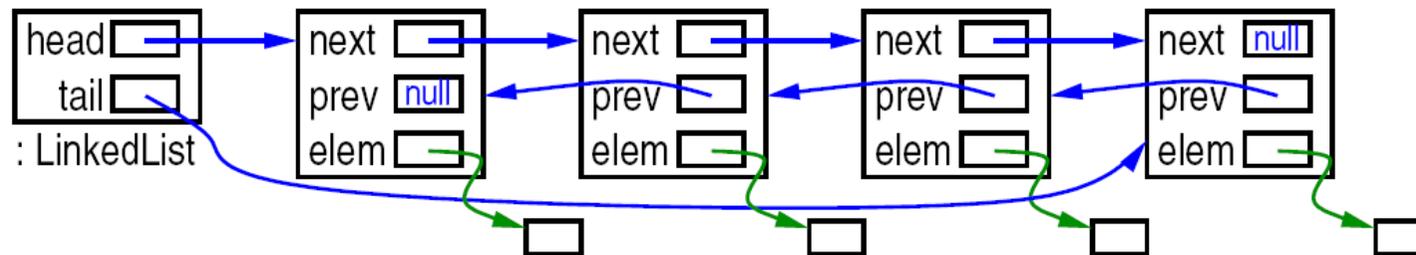
Die Klasse `ArrayList` (bzw. `Vector`)

- Realisiert die Sequenz durch ein Array von Objekt-Referenzen
 - das Array ist immer groß genug, um alle Elemente zu halten
 - seine Größe (*capacity*) wird bei `add()` usw. automatisch angepasst, falls erforderlich
 - Kopie des Arrays in ein größeres Array
 - neues Array i.a. größer als nötig, um häufiges Kopieren zu vermeiden
- Konstruktoren:
 - `ArrayList(int initialCapacity)` :
 - erzeugt Sequenz mit gegebener initialer Kapazität
 - `ArrayList()`: initiale Kapazität ist 10

5.3.2 Implementierung von Sammlungen ...

Die Klasse `LinkedList`

- Realisierung als (doppelt) verkettete Liste (=> EI_I, 11.3):



- jedes Element "kennt" Vorgänger und Nachfolger
- Konstruktor: `LinkedList()`
- Zusätzliche Operationen der Klasse:
 - `addFirst()`, `addLast()`: fügt vorne / hinten an
 - `getFirst()`, `getLast()`: liefert erstes / letztes Element
 - `removeFirst()`, `removeLast()`: entfernt erstes / letztes Element



5.3.2 Implementierung von Sammlungen ...

Die Klasse `LinkedList` ...

- Mit den zusätzlichen Operationen sind Datenstrukturen wie *Stack* (=> EI I, 11.4) oder *Queue* (=> EI I, 11.5) realisierbar
- Beispiel: Auftrags-Warteschlange

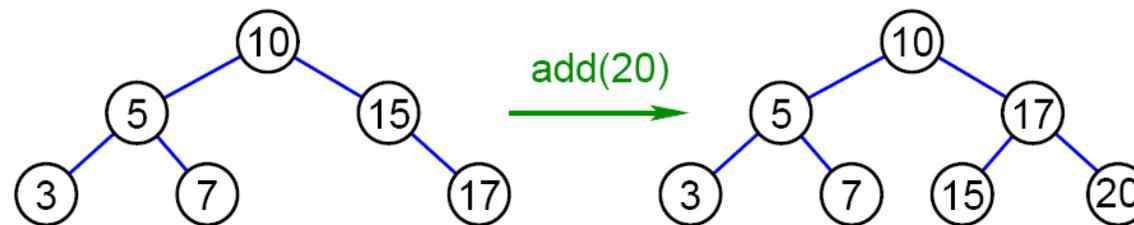
```
class Auftragsbearbeitung {  
    private LinkedList<Auftrag> auftragsListe =  
        new LinkedList<Auftrag> ();  
    public void auftragAnnehmen(Auftrag a) {  
        auftragsListe.addLast(a);  
    }  
    public void naechstenAuftragBearbeiten() {  
        Auftrag a = auftragsListe.removeFirst();  
        // ... bearbeite Auftrag  
    }  
}
```



5.3.2 Implementierung von Sammlungen ...

Die Klasse TreeSet

- Realisierung als (balancierter) binärer Suchbaum (=> EI_I, 12.4)



- jeder Knoten verweist auf (maximal) zwei Unterbäume
- Elemente im rechten (linken) Unterbaum alle größer (kleiner) als aktueller Knoten
- Balancierung verhindert "Entartung" des Baums zur Liste
 - Aufwand für Operationen ist proportional zur Baumhöhe
 - Balancierung garantiert, daß Höhe nur logarithmisch wächst



5.3.2 Implementierung von Sammlungen ...

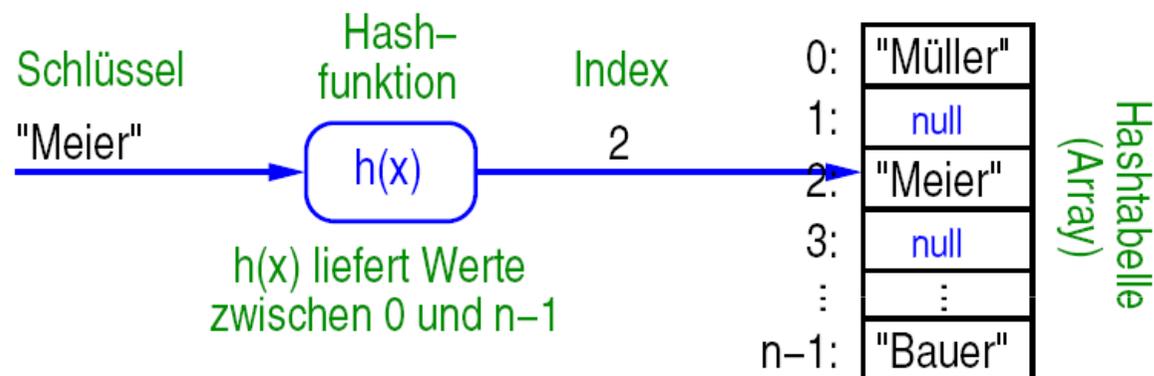
Die Klasse TreeSet ...

- Konstruktoren:
 - `TreeSet()`
 - Menge nach "natürlicher" Ordnung der Elemente geordnet
 - d.h. Methode `compareTo()` der Elemente
 - `TreeSet(Comparator<E> c)`
 - Ordnung in der Menge wird durch `compare()`-Methode von `c` festgelegt

5.3.2 Implementierung von Sammlungen ...

Die Klasse HashSet

- Realisierung durch Hashtabelle (=> EI_I, 14.5)



- Idee des *Hashing*:
 - wähle Hashfunktion, z.B. $h(x) = x \bmod n$, n prim
 - speichere Element e an Platz $h(e)$ der Hashtabelle
- Problem, falls Platz bereits mit anderem Element belegt ist
 - verschiedene Verfahren zur Kollisionsbehandlung
 - Hashtabelle i.a. wesentlich größer als Zahl der Elemente



5.3.2 Implementierung von Sammlungen ...

Die Klasse `HashSet` ...

- Vorteile des *Hashing*:
 - schnelles Einfügen, Löschen und Suchen
- Nachteil: Speicherverbrauch
- Konstruktoren:
 - `HashSet(int initialCapacity, float loadFactor)`
 - initiale Größe der Hashtabelle wird spezifiziert
 - wenn der Füllfaktor größer als `loadFactor` wird, wird eine größere Tabelle angelegt
 - `HashSet(int initialCapacity)`
 - benutzt Standardwert für `loadFactor`: 0.75
 - `HashSet()` : Größe der Hashtabelle ist initial 16



5.3.2 Implementierung von Sammlungen ...

Einschub: Die Methode `hashCode ()` in `Object`

- Die Klasse `Object` definiert eine Methode, um das *Hashing* zu unterstützen: `public int hashCode ()`
 - d.h., Hashwert eines Objekts `o` ist `h(o.hashCode ())`
 - Hashfunktion `h(x)` nur noch für ganze Zahlen notwendig
- Die Standard-Implementierung aus `Object` gibt die Adresse des Objekts als ganze Zahl zurück
- Klassen können diese Methode geeignet überschreiben
 - immer dann, wenn auch `equals ()` überschrieben wird
- `hashCode ()` muß konsistent mit `equals ()` sein, d.h.:
 - `o1.equals (o2) ⇒ o1.hashCode () == o2.hashCode ()`
 - aber nicht notwendigerweise umgekehrt



5.3.2 Implementierung von Sammlungen ...

Vergleich

Klasse	Ausführungszeit				Durchlaufreihenfolge
	add()	remove()	get()	contains()	
ArrayList	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Einfügung
LinkedList	$\mathcal{O}(1)$ *)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Einfügung
TreeSet	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	—	$\mathcal{O}(\log n)$	Ordnung
HashSet	$\mathcal{O}(1)$	$\mathcal{O}(1)$	—	$\mathcal{O}(1)$	unspezif.

*) schneller als ArrayList

- $\mathcal{O}(1)$: konstante Ausführungszeit, unabhängig von der Größe
- $\mathcal{O}(\log n)$: Zeit wächst logarithmisch mit d. Anzahl der Elemente
- $\mathcal{O}(n)$: Zeit wächst linear mit der Anzahl der Elemente



5.3.2 Implementierung von Sammlungen ...

Wann welche Implementierung?

- Sequenzen:
 - wenn mehrfache Einträge erlaubt sein sollen und / oder wenn es auf die Einfügereihenfolge ankommt
 - `LinkedList`: Einfügen und Löschen schneller, wahlfreier Zugriff langsamer als bei `ArrayList`
 - `ArrayList` bei kleineren Sequenzen; wenn häufig wahlfrei zugegriffen wird; bei überwiegend lesendem Zugriff
- Mengen:
 - wenn keine doppelten Einträge vorkommen sollen
 - `HashSet`: Operationen performanter als `TreeSet`; benötigt ggf. mehr Speicher; Durchlaufen i.d.R. ineffizienter als bei `TreeSet`, ohne feste Reihenfolge



5.3 Das Java Collection Framework ...

5.3.3 Maps (Abbildungen)

- *Maps* dienen dazu, Schlüssel auf Werte abzubilden
 - Schlüssel und Werte können dabei beliebige Objekte sein
- Beispiel: Email-Verzeichnis

Schlüssel		Wert
"Meier"	→	"hans.meier@uni-siegen.de"
"Müller"	→	"mueller123@gmx.de"
"Huber"	→	"u94302@aol.com"

- Eine Map realisiert einen **assoziativen Speicher**
 - der Zugriff auf den Speicher erfolgt nicht über eine Adresse (z.B. Index, Referenz) sondern über ein inhaltliches Kriterium (Schlüssel)
-



5.3.3 Maps (Abbildungen) ...

Eigenschaften von Abbildungen

- Für einen Schlüssel gibt es entweder gar keinen oder genau einen Wert in der *Map*
 - falls ein (Schlüssel, Wert)-Paar eingefügt wird, dessen Schlüssel schon in der *Map existiert*:
 - es wird kein neuer Eintrag angelegt, sondern der vorhandene Eintrag geändert
 - dem Schlüssel wird somit ein neuer Wert zugeordnet
- Wird ein Schlüssel gelöscht, so auch der zugehörige Wert
- Eine *Map* kann als Menge von Paaren (Schlüssel, Wert) betrachtet werden, wobei jeder Schlüssel nur einmal vorkommen darf



5.3.3 Maps (Abbildungen) ...

Die Schnittstelle `Map<K, V>`

- Gemeinsame Basisschnittstelle für alle Abbildungen
 - Motivation analog zu `Collection`
- Operationen der Schnittstelle `Map<K, V>`:
 - `clear()`, `size()`, `isEmpty()`: analog zu `Collection`
 - `V put(K key, V value)`
 - trägt `(key, value)` in die Abbildung ein
 - Ergebnis: alter Wert, der `key` zugeordnet war (bzw. `null`)
 - `V get(Object key)`: Wert, der `key` zugeordnet ist
 - `V remove(Object key)`: löscht Abbildung für `key`
 - `boolean containsKey(Object key)`: Schlüssel enthalten?
 - `boolean containsValue(Object value)`: Wert enthalten?



5.3.3 Maps (Abbildungen) ...

Sichten auf Maps

- Eine `Map` stellt drei **Sichten** zur Verfügung, die einen Zugriff auf den Inhalt über die `Collection`-Schnittstelle erlauben:
 - `Set<K> keySet ()` : Menge der Schlüssel
 - `Collection<V> values ()` : Sammlung der Werte
 - keine Menge, da Werte mehrfach vorkommen können
 - `Set<Map.Entry<K, V>> entrySet ()` : Menge der (Schlüssel, Wert)-Paare
 - die Elemente der Menge sind vom Typ `Map.Entry`
 - `Map.Entry<K, V>` ist innere Schnittstelle von `Map<K, V>`:

```
interface Map<K, V> {  
    interface Entry<K, V> { ... }
```
 - Zugriff auf Schlüssel und Wert über die Operationen `getKey ()` und `getValue ()`



5.3.3 Maps (Abbildungen) ...

Sichten auf Maps ...

- Die Sichten sind keine eigenen Objekte, sondern nur andere Sichtweisen der `Map`
 - d.h. Veränderungen der `Map` wirken sich auf die Sichten aus und umgekehrt
- Ein Einfügen in die Sichten ist aber **nicht** möglich
 - Operationen führen zu `UnsupportedOperationException`
- Anwendung der Sichten:
 - Durchlaufen der Schlüssel, Werte bzw. (Schlüssel, Wert)- Paare
 - `Map` selbst unterstützt keine Iteratoren
 - Suchen und ggf. Löschen von Einträgen



5.3.3 Maps (Abbildungen) ...

Implementierungen der Schnittstelle Map

- Java bietet (i.W.) zwei verschiedene Implementierungen von Abbildungen:
 - `TreeMap`: Verwaltung der Schlüsselmenge durch binären Suchbaum
 - Konstruktoren:
 - `TreeMap()`
 - `TreeMap(Comparator<K> c)`
 - `HashMap`: Verwaltung der Schlüsselmenge durch *Hashing*
 - Konstruktoren:
 - `HashMap(int initialCapacity, float loadFactor)`
 - `HashMap(int initialCapacity)`
 - `HashMap()`



5.3.3 Maps (Abbildungen) ...

Beispiel: Email-Verzeichnis (☞ WWW: AddressBook.java)

```
import java.util.*;
import java.io.*;
public class AddressBook implements Serializable {
    private Map<String,Data> addresses;
    private class Data implements Serializable {
        public String fullName;
        public String emailAddress;
        public Data(String name, String email) {
            fullName = name; emailAddress = email;
        }
    }
    public class DoubleEntryException extends Exception {...}
    public class EntryNotFoundException extends Exception {...}
```



5.3.3 Maps (Abbildungen) ...

```
public AddressBook() {
    addresses = new HashMap<String,Data>(100);
}
public void enter(String alias, String name, String email)
                throws DoubleEntryException {
    if (addresses.containsKey(alias))
        throw new DoubleEntryException(...);
    addresses.put(alias, new Data(name, email));
}
public String getEmail(String alias)
                throws EntryNotFoundException {
    Data val = addresses.get(alias);
    if (val == null)
        throw new EntryNotFoundException(...);
    return val.emailAddress;
}
```



5.3.3 Maps (Abbildungen) ...

```
public Collection<String> findByName(String name) {  
    Collection<String> result = new TreeSet<String>();  
    Collection<Data> values = addresses.values();  
    Iterator<Data> i = values.iterator();  
    while (i.hasNext()) {  
        Data val = i.next();  
        if (val.fullName.indexOf(name) != -1)  
            result.add(val.fullName +  
                " <" + val.emailAddress + ">");  
    }  
    return result;  
}
```



5.3.3 Maps (Abbildungen) ...

```
public void deleteByName(String name) {  
    Collection<Data> values = addresses.values();  
    Iterator<Data> i = values.iterator();  
    while (i.hasNext()) {  
        Data val = i.next();  
        if (val.fullName.indexOf(name) != -1)  
            i.remove();  
    }  
}
```



5.3.3 Maps (Abbildungen) ...

```
public String toString() {
    String result = "";
    Set<Map.Entry<String,Data>> entries =
        addresses.entrySet();
    Iterator<Map.Entry<String,Data>> i =
        entries.iterator();
    while (i.hasNext()) {
        Map.Entry<String,Data> entry = i.next();
        Data val = entry.getValue();
        result += entry.getKey() + ": " + val.fullName
            + " <" + val.emailAddress + ">\n"
    }
    return result;
}
```



5.3.3 Maps (Abbildungen) ...

```
public static void main(String[] args) {
    AddressBook emails = new AddressBook();
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
    BitSet withArgs = new BitSet();
    int cmd = 'q';
    String line;
    String[] fields;
    ObjectInputStream ois = null;
    try {
        ois = new ObjectInputStream(
            new FileInputStream("adds.ser"));
        emails = (AddressBook) ois.readObject();
    }
}
```



5.3.3 Maps (Abbildungen) ...

```
catch (Exception e) {  
    System.out.println("Kann 'adds.ser' nicht lesen");  
}  
finally { try { ois.close(); } catch (Exception e) {} }  
  
withArgs.set('a'); withArgs.set('e')  
withArgs.set('f'); withArgs.set('d')  
  
do {  
    try {  
        cmd = in.read();  
        if (withArgs.get(cmd) && (in.read() != ' ')) {  
            System.out.println("Erwarte ' ' nach Kommando!");  
            in.readLine();  
            continue;  
        }  
        line = in.readLine();  
    }  
}
```



5.3.3 Maps (Abbildungen) ...

```
switch (cmd) {  
    case 'a':  
        fields = line.split(",");  
        emails.enter(fields[0], fields[1], fields[2]);  
        break;  
    case 'e':  
        System.out.println(emails.getEmail(line));  
        break;  
    case 'f':  
        Collection<String> res = emails.findByName(line);  
        Iterator<String> i = res.iterator();  
        while (i.hasNext())  
            System.out.println(i.next());  
        break;  
    case 'd':  
        emails.deleteByName(line);  
        break;  
}
```



5.3.3 Maps (Abbildungen) ...

```
case 'p':  
    System.out.print(emails);  
    break;  
case 's':  
    ObjectOutputStream oos = null;  
    try {  
        oos = new ObjectOutputStream(  
            new FileOutputStream("adds.ser"));  
        oos.writeObject(emails);  
    }  
    finally { if (oos != null) oos.close(); }  
    break;  
case 'q':  
    break;  
default:  
    System.out.println("Falsches Kommando");  
    break;
```



5.3.3 Maps (Abbildungen) ...

```
    }  
  }  
  catch (...) { // Fehlerbehandlungen für  
    ...           // aktuelles Kommando  
  }  
} while (cmd != 'q');  
}  
}
```



5.4 GUI-Programmierung

➤ GUI: Graphical User Interface / Graphische Bedienoberfläche



- Bisher: Anwendungen mit textbasierter Schnittstelle
- graphische Schnittstelle eines Programms zum Benutzer

➤ Wichtig: **Software-Ergonomie** (in EI II nicht vertieft!)

- Software soll an Bedürfnisse des Benutzers angepasst sein
- Dazu: Menschen- und aufgabengerechte Gestaltung der
 - Aufgabenverteilung zw. Mensch und Computer (**Arbeitsstrukturierung**)
 - Funktion / Leistung der Anwendungsprogramme (**Software-Gestaltung**)
 - Bedienungsschritte und -abläufe (**Dialoggestaltung**)
 - E/A-Geräte, einschl. der dargestellten Information (**E/A-Gestaltung**)



5.4 GUI-Programmierung ...

Literaturnachtrag

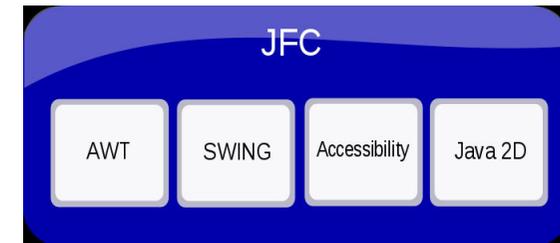
- [Ba99], Kap. 4.1 - 4.6
- Middendorf, Singer, Heid: Java Programmierhandbuch und Referenz, 3. Auflage, dpunkt.verlag, 2003.
Kap. 8 (Oberflächenprogrammierung) und
Kap. 9 (Ereignisbehandlung)
 - online verfügbar, siehe WWW-Seite:

<http://www.dpunkt.de/java/>

5.4 GUI-Programmierung ...

Vorbemerkung: Java-Klassen für GUIs

- Die Java-Klassenbibliothek enthält zwei Pakete, die Klassen für GUIs bereitstellen:



- **AWT** (*Abstract Window Toolkit*, `java.awt`)
 - ursprüngliche Realisierung der Java GUI-Elemente / gehört zu Java Foundation Classes (JFC) als Sammlung von Programmierschnittstellen
 - Klassen greifen auf die GUI-Elemente des jeweiligen Betriebssystems (z.B. Windows) zurück
 - definiert zusätzlich, wie GUI-Elemente mit dem Java- Programm interagieren (Ereignismodell)
- **Swing** (`javax.swing`): Nachfolger von AWT (ab 1997)
 - die GUI-Elemente sind Betriebssystem-unabhängig in Java realisiert
 - Swing-Klassen setzen auf AWT auf

- Im Folgenden: Vorstellung einiger GUI-Elemente von Swing



5.4 GUI-Programmierung ...

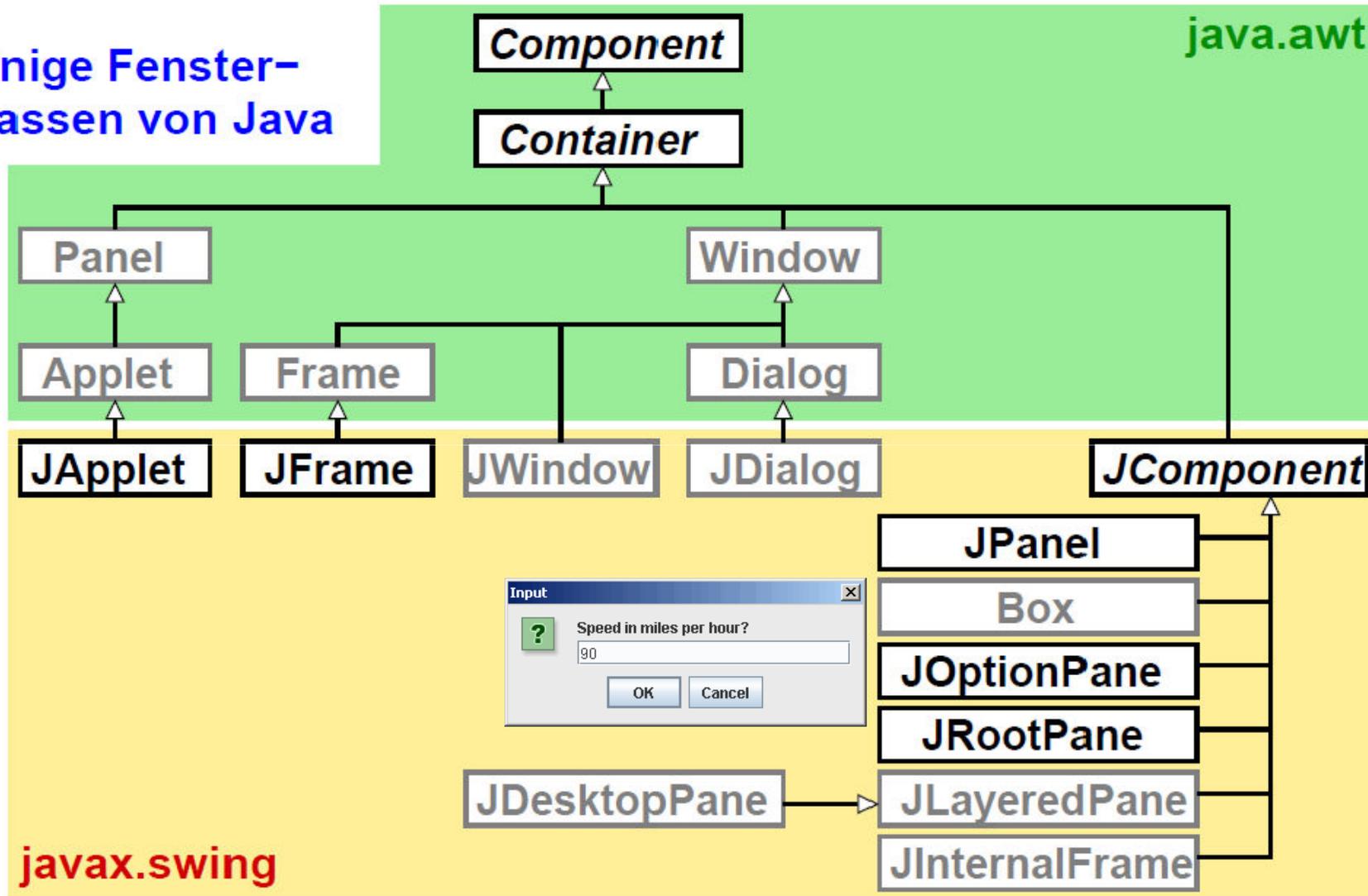
5.4.1 Fenster

- Zur Realisierung von **Dialogen**: Interaktion zwischen Benutzer und System
 - Anwendungsfenster, ggf. mit Unterfenstern, für **Primärdialog**
 - dient der direkten Aufgabenerfüllung
 - wird erst geschlossen, wenn die Aufgabe (d.h. meist: das Programm) beendet wird
 - Dialogfenster / Mitteilungsfenster für **Sekundärdialog**
 - optional und kurzzeitig, wenn situationsabhängig weitere Information vom Benutzer benötigt wird
 - **modaler Dialog** muß beendet sein, bevor mit der Anwendung weitergearbeitet werden kann
 - **nicht-modaler Dialog** kann unterbrochen werden, um andere Aktionen durchzuführen



5.4.1 Fenster ...

Einige Fenster-
klassen von Java

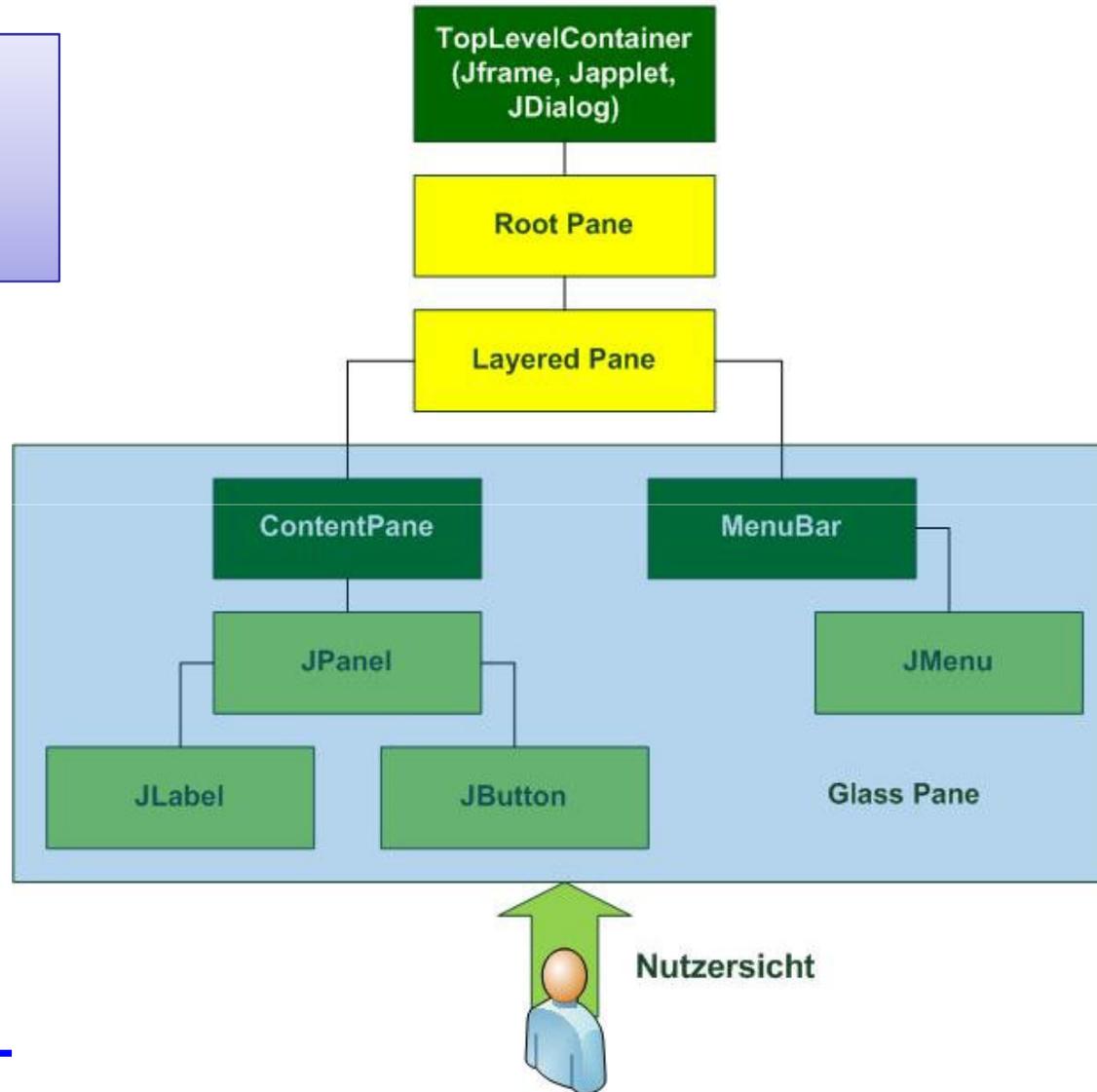




5.4.1 Fenster ...

Top Level Container

Hierarchie eines Top-Level Containers in Java Swing





5.4.1 Fenster ...

Einige Fensterklassen in Java ...

- **Component**: Basisklasse für alle darstellbaren GUI-Komponenten
 - vererbt u.a. folgende Methoden an alle anderen Klassen:
 - `void setSize(int w, int h)`: legt die Größe fest
 - `void setBounds(int x, int y, int w, int h)`: legt Position und Größe fest
 - `void setForeground(Color c)`: Vordergrundfarbe
 - `void setBackground(Color c)`: Hintergrundfarbe
 - `void paint(Graphics gc)`: zeichnet Komponente



5.4.1 Fenster ...

Einige Fensterklassen in Java ...

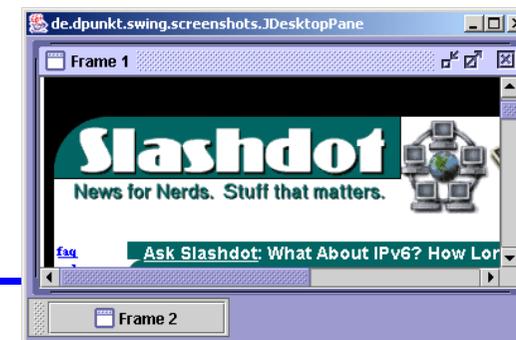
- `Container`: Basisklasse für Komponenten, die andere Komponenten aufnehmen, zusammenfassen und/oder gruppieren
 - wichtigste Methode: `Component add(Component c)`
 - fügt eine Komponente in den Container ein
- `Applet` / `JApplet`: zur Realisierung von Applets (siehe später) (Webseiten, HTML)
 - Java-Programme, die im WWW-Browser ausgeführt werden
- `Frame` / `JFrame`: Anwendungsfenster mit Rahmen und ggf. Menü



5.4.1 Fenster ...

Einige Fensterklassen in Java ...

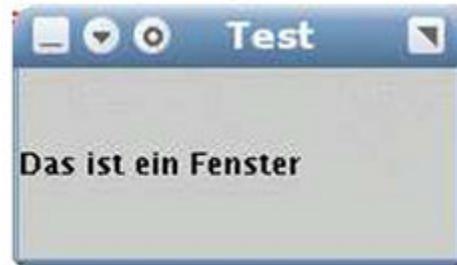
- Dialog / JDialog: Dialogfenster
(Fenster mit Titelzeile, aber ohne Menüleiste)
- JOptionPane: zur Erzeugung von Dialogfenstern
- JPanel / Box: für Gruppierung / Layout von GUI-Elementen
- JRootPane / JLayeredPane: Verwaltung von Fensterinhalten
- JDesktopPane: kann mehrere Unterfenster aufnehmen
- JInternalFrame: Unterfenster



5.4.1 Fenster ...

Anwendungsfenster: JFrame (\Rightarrow WWW: `Frame.java`)

➤ Beispiel:



➤ Java-Code:

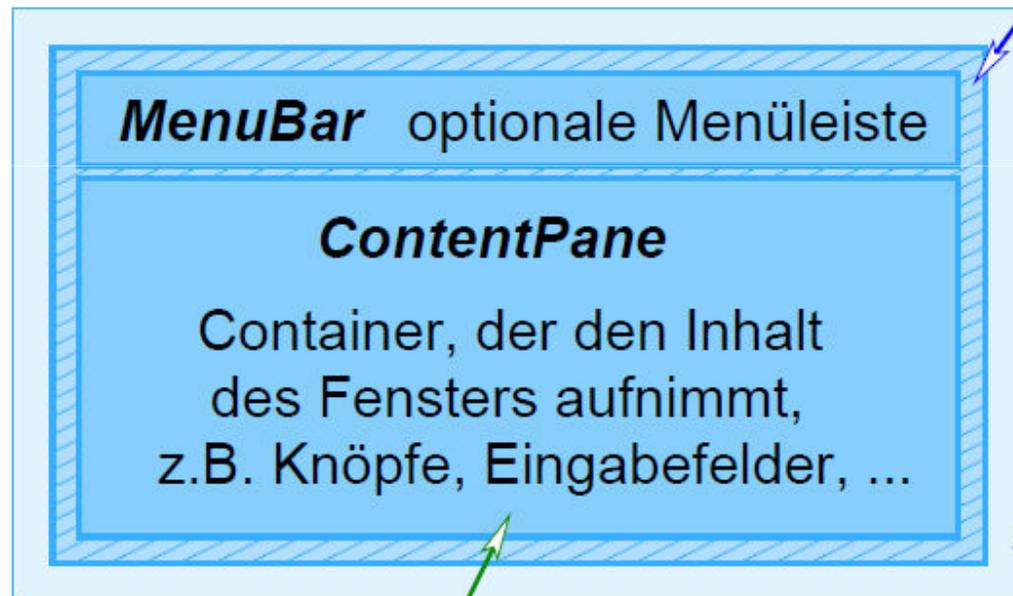
```
JFrame win = new JFrame("Test"); // Fenster-Objekt erzeugen
win.setSize(200, 120); // Größe des Fensters setzen
// Label (Beschriftung) erzeugen und in das Fenster eintragen
JLabel label = new JLabel("Das ist ein Fenster");
win.getContentPane().add(label); // Inhaltsfläche geben lassen
// und Text einfügen

// Programm beenden, wenn Fenster geschlossen wird
win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
win.pack() // Fenster arrangiert eingefügte Komponente
win.setVisible(true); // Fenster auf Bildschirm darstellen
```

5.4.1 Fenster ...

Aufbau eines Swing-Fensters

- Swing-Fenster enthalten genau ein Objekt der Klasse `JRootPane`
- Aufbau der `RootPane` :



LayeredPane

- enthält `MenuBar` und `ContentPane`
- unterstützt / verwaltet mehrere hintereinander liegende Schichten, z.B. für Unterfenster

GlassPane

- liegt unsichtbar über den anderen *Panes*
- verwaltet Mausereignisse

```
win.getContentPane().add(...)
```

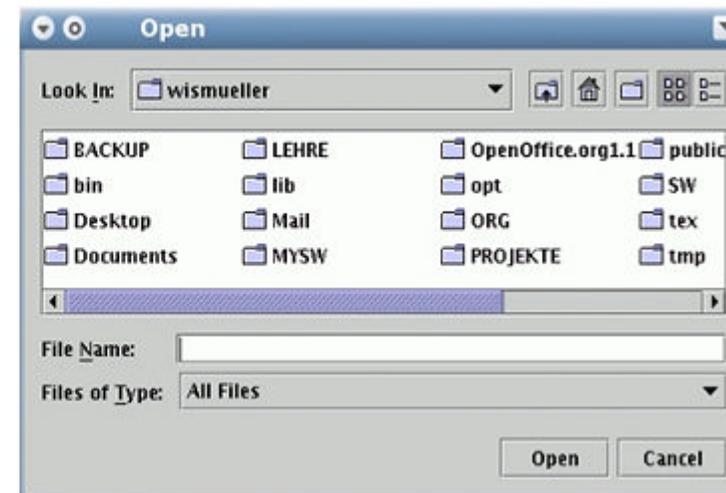
5.4.1 Fenster ...

Modale Dialogfenster (exklusiv zu bedienendes Fenster)

- `JOptionPane`: Informations-/Bestätigungs- und Eingabe-Dialoge



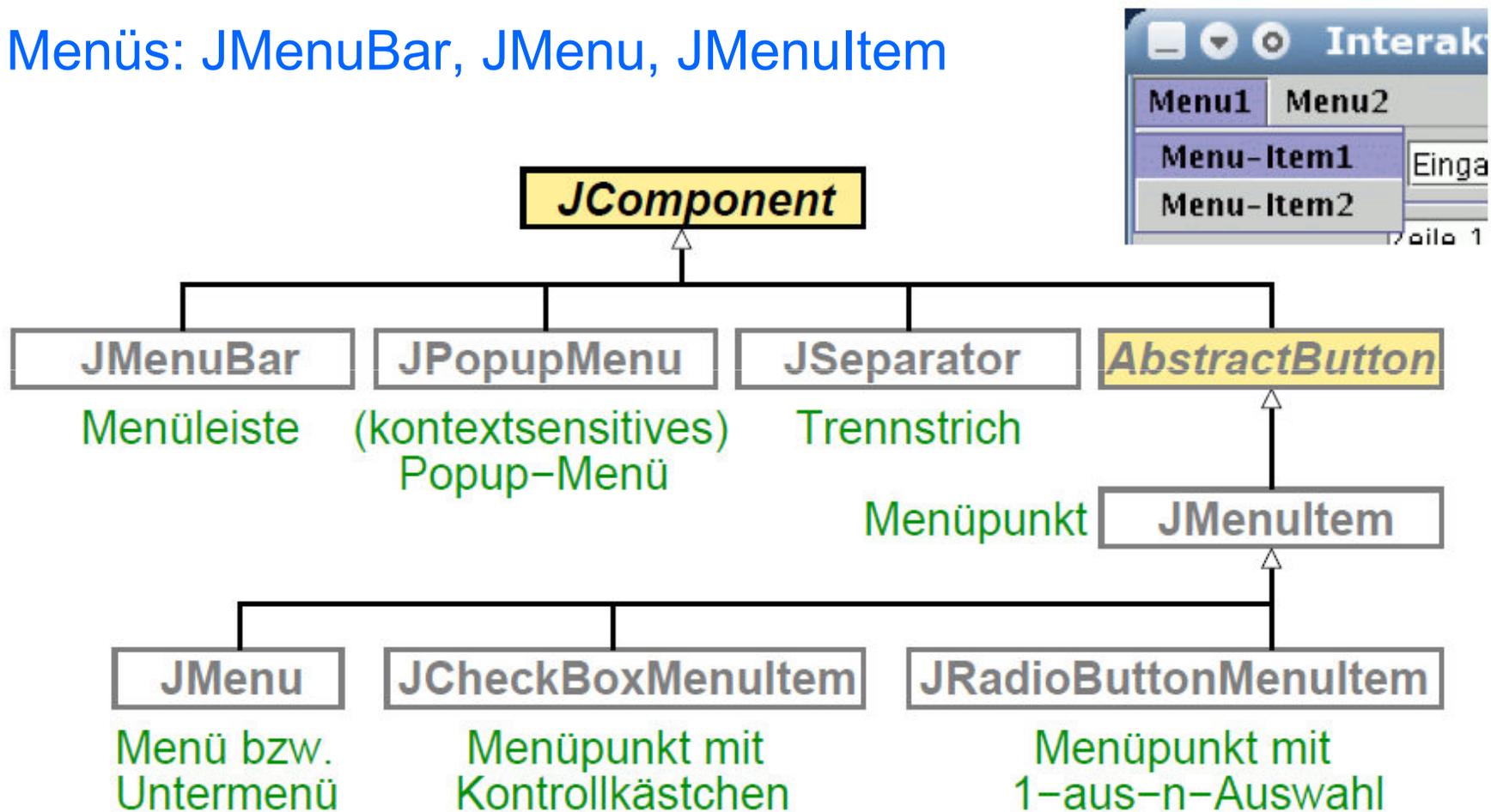
- `JFileChooser`: Dialog zur Dateiauswahl



5.4 GUI-Programmierung ...

5.4.2 Interaktionselemente

Menüs: JMenuItemBar, JMenuItem, JMenuItem





5.4.2 Interaktionselemente ...

Druckknopf (Schaltfläche): `JButton`

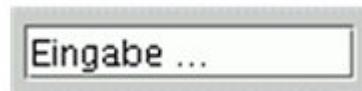
- Zum Auslösen von Aktionen durch den Benutzer



```
JButton but = new JButton("Button");
```

Textfeld (Eingabefeld): `JTextField`

- Zur Eingabe von Texten oder numerischen Daten in einer Zeile

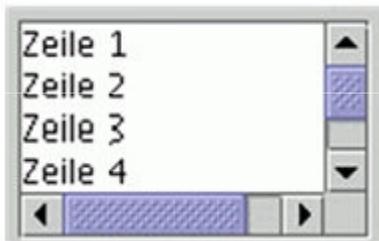


```
JTextField tf = new JTextField  
("Eingabe ...", 10);
```

5.4.2 Interaktionselemente ...

Mehrzeiliges Textfeld (Textbereich): `JTextArea`

- Zur Ein- und/oder Ausgabe mehrzeiliger Texte
- Kann auch mit Rollbalken versehen werden (`JScrollPane`), die bei Bedarf automatisch angezeigt werden

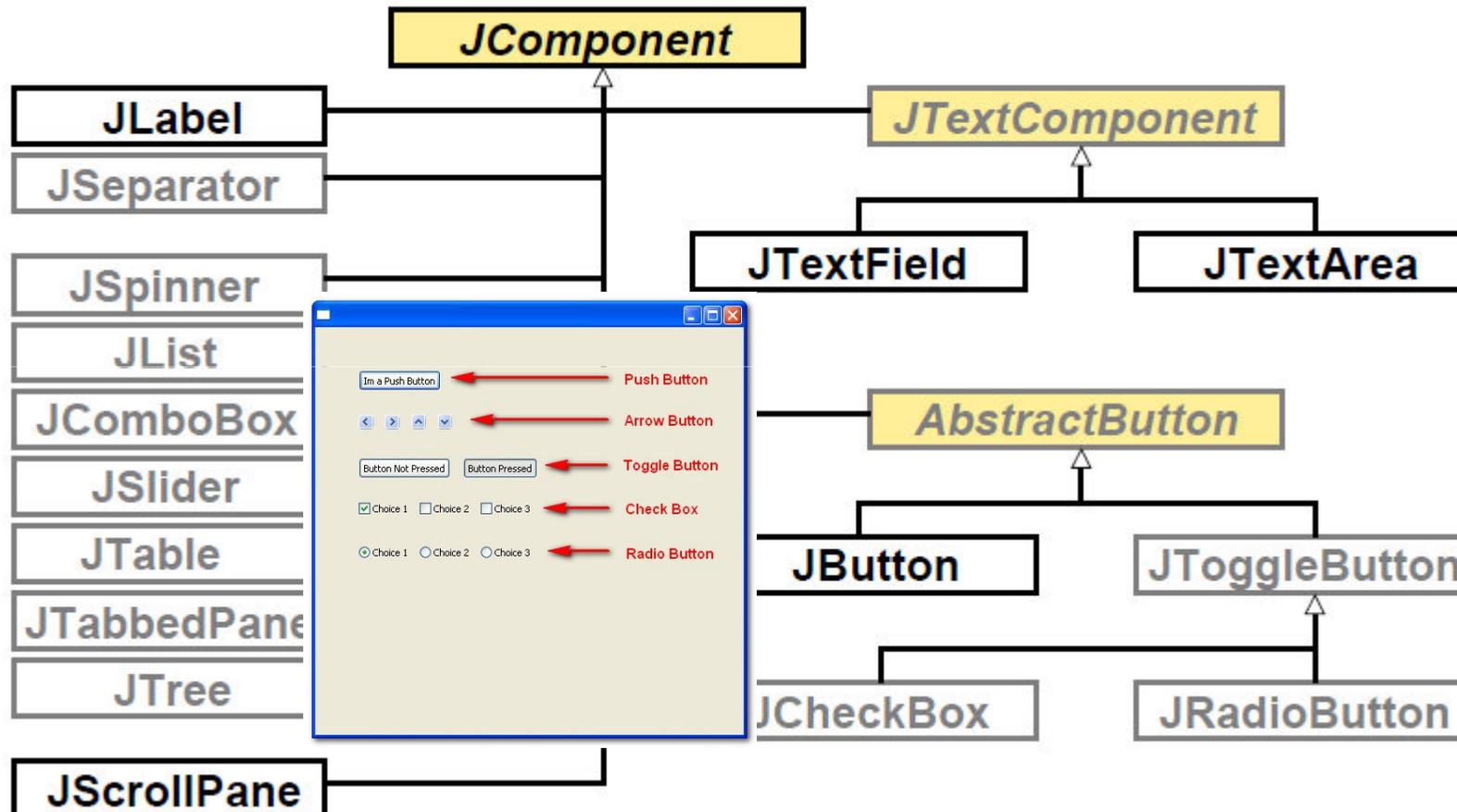


```
JTextArea ta = new JTextArea(4,10);
ta.append("Zeile 1\nZeile 2\n" +
        "Zeile 3\nZeile 4");
win.getContentPane().add(
    new JScrollPane(ta));
```



5.4.2 Interaktionselemente ...

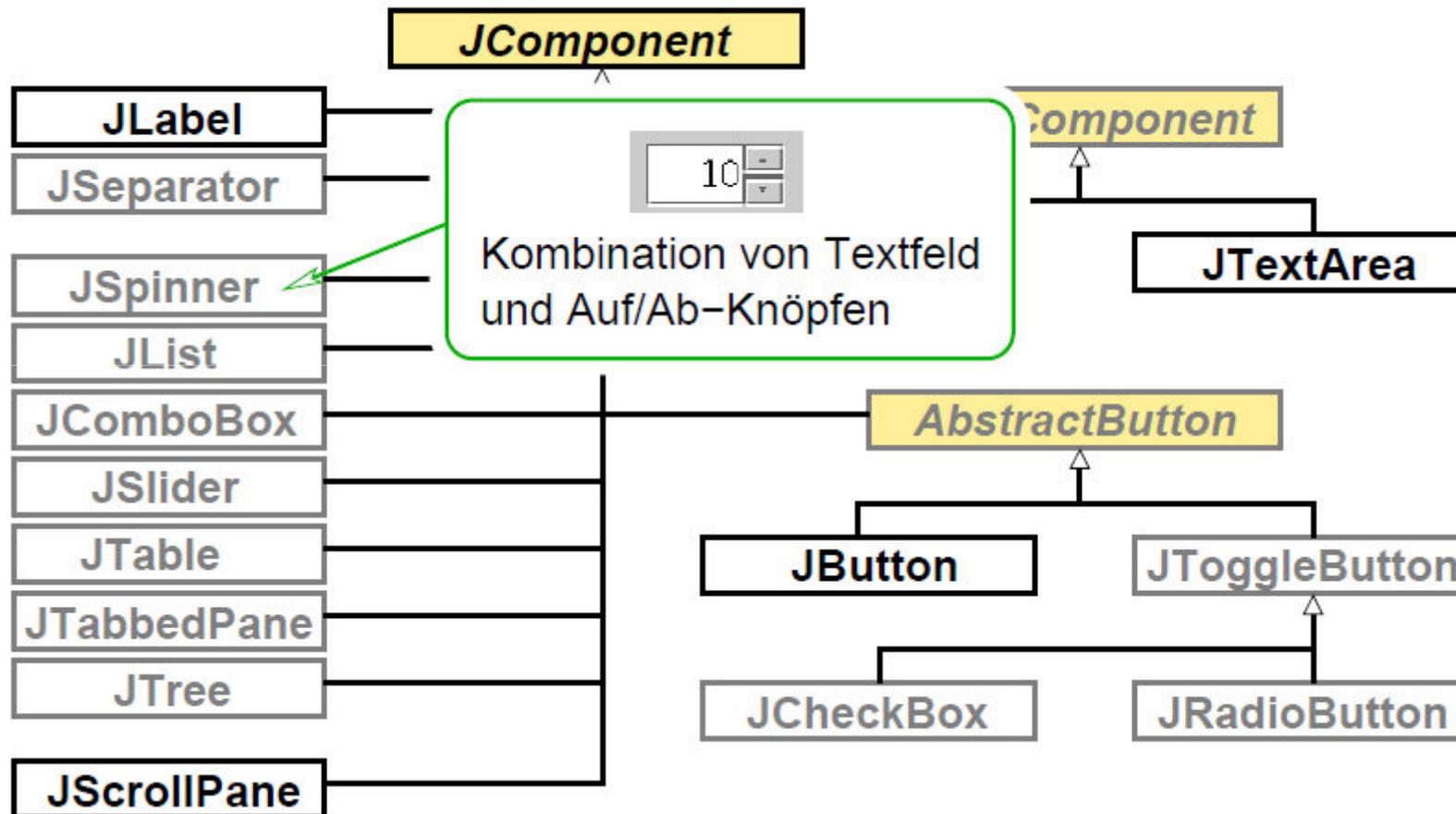
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

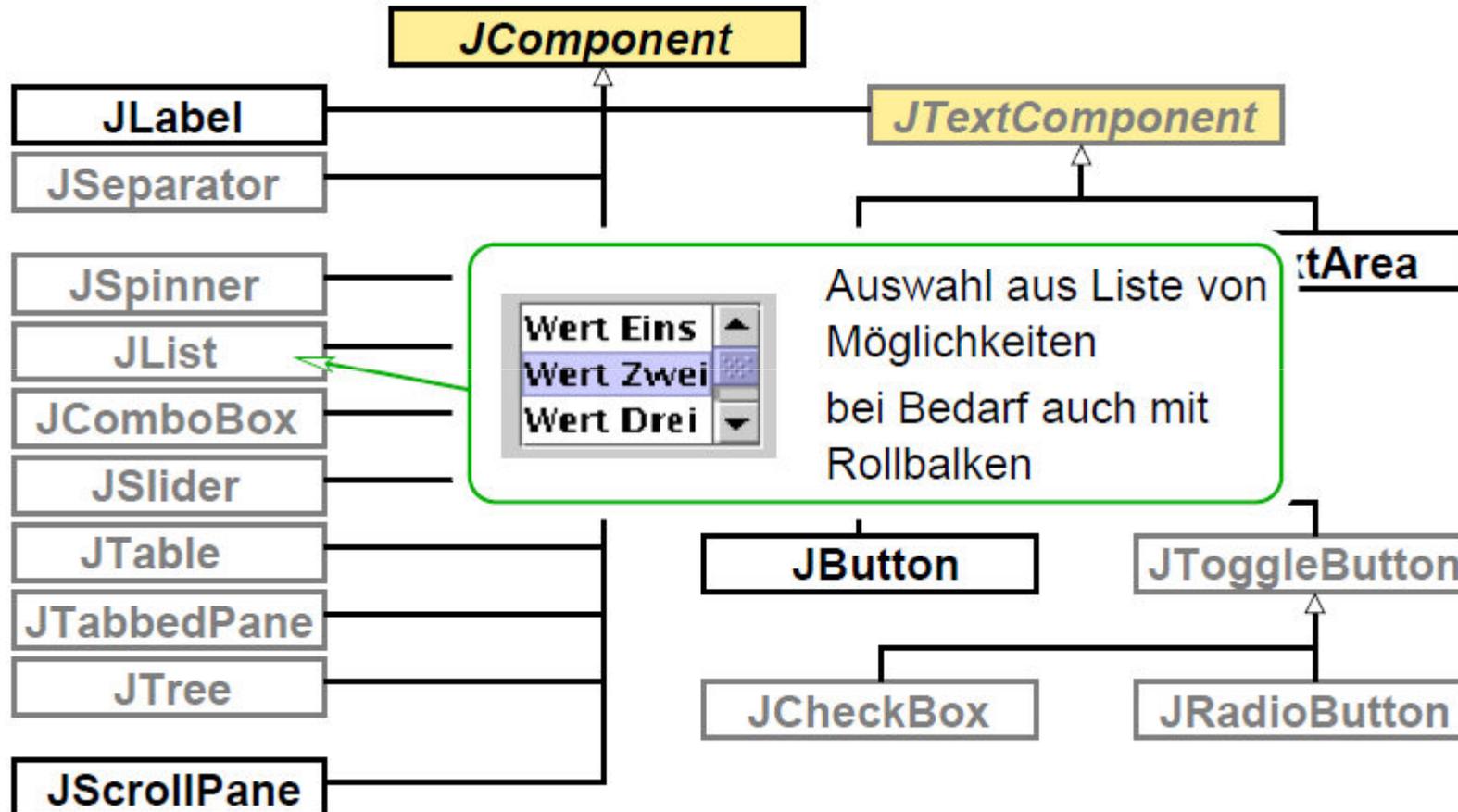
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

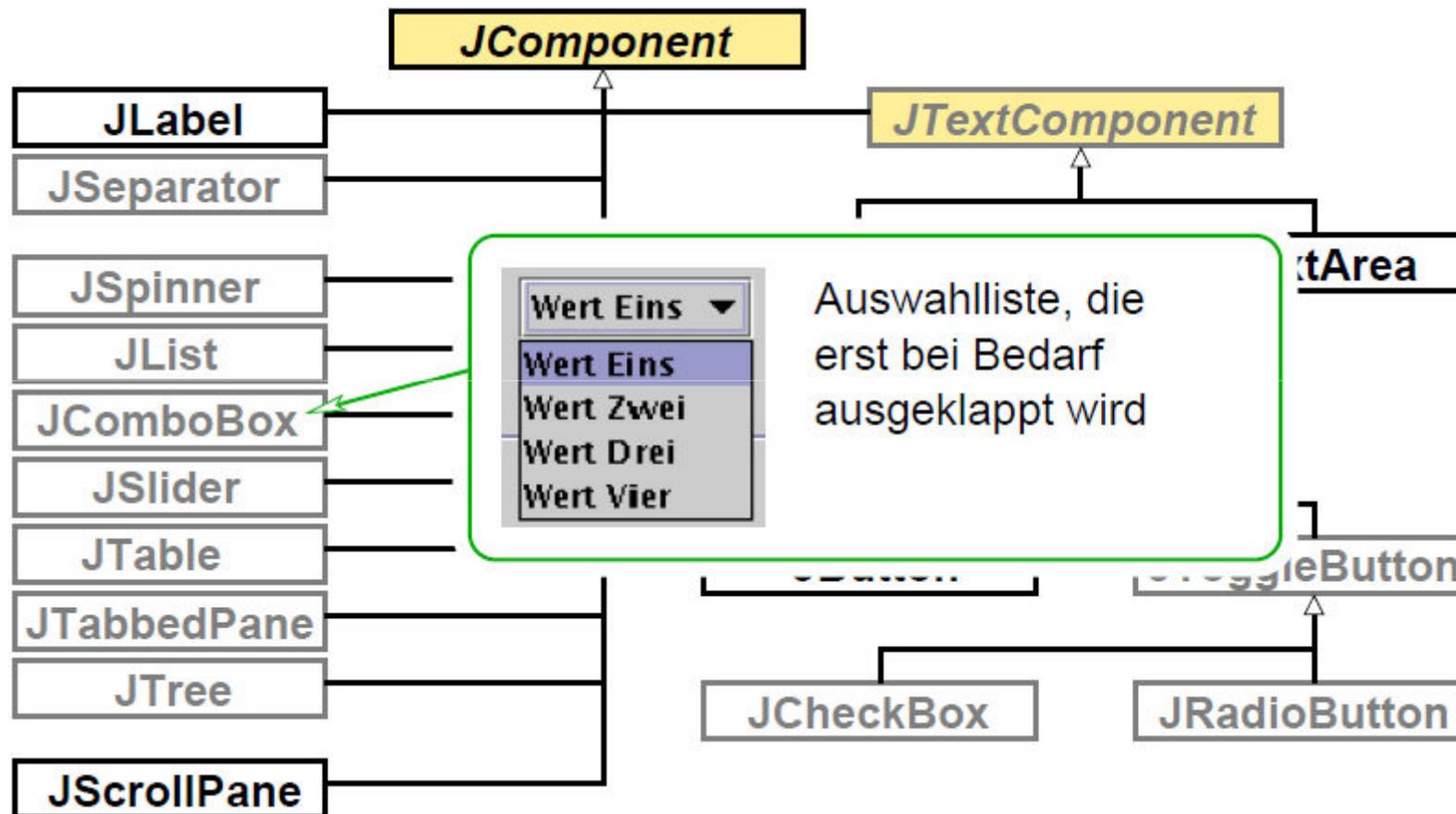
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

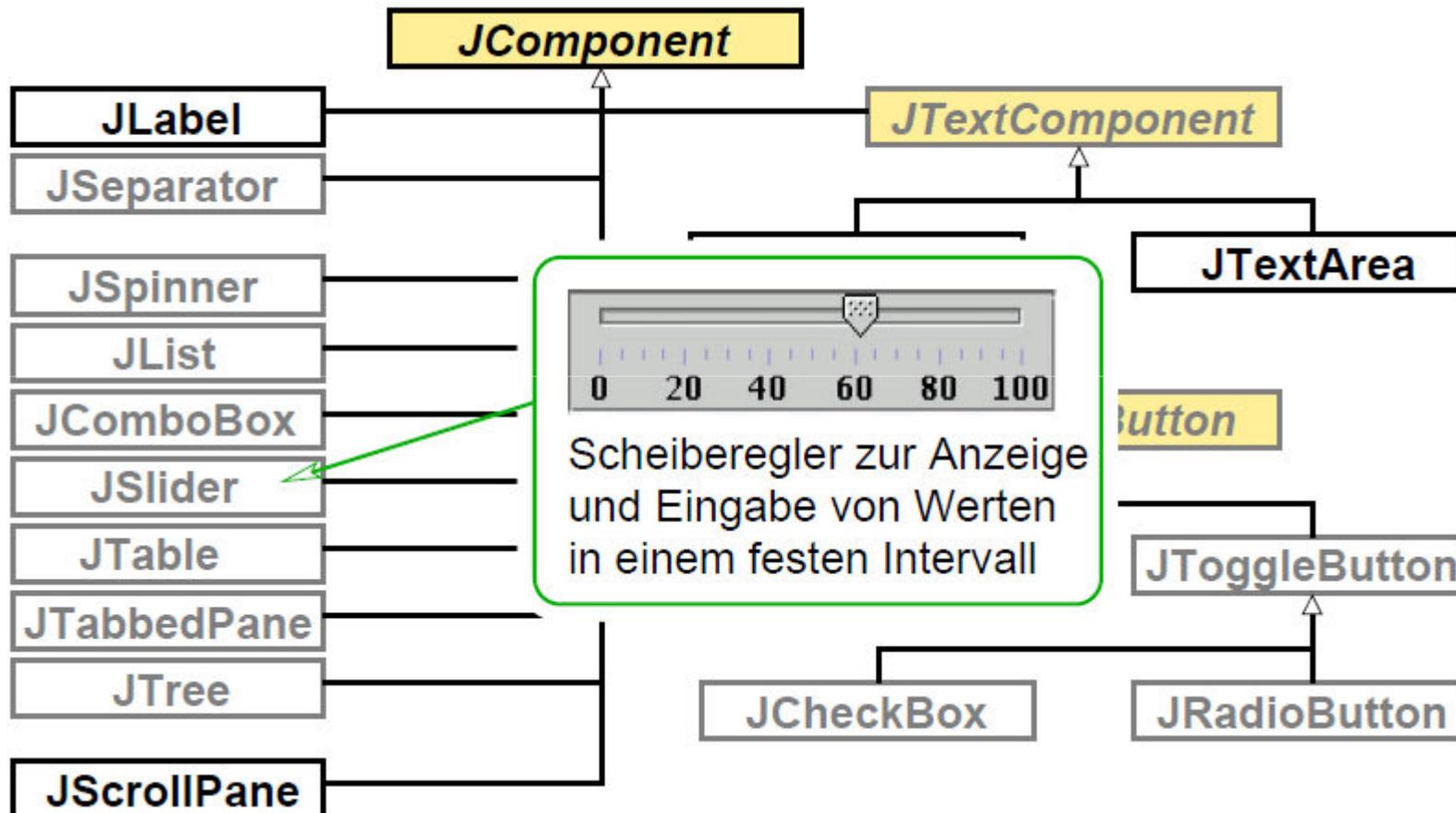
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

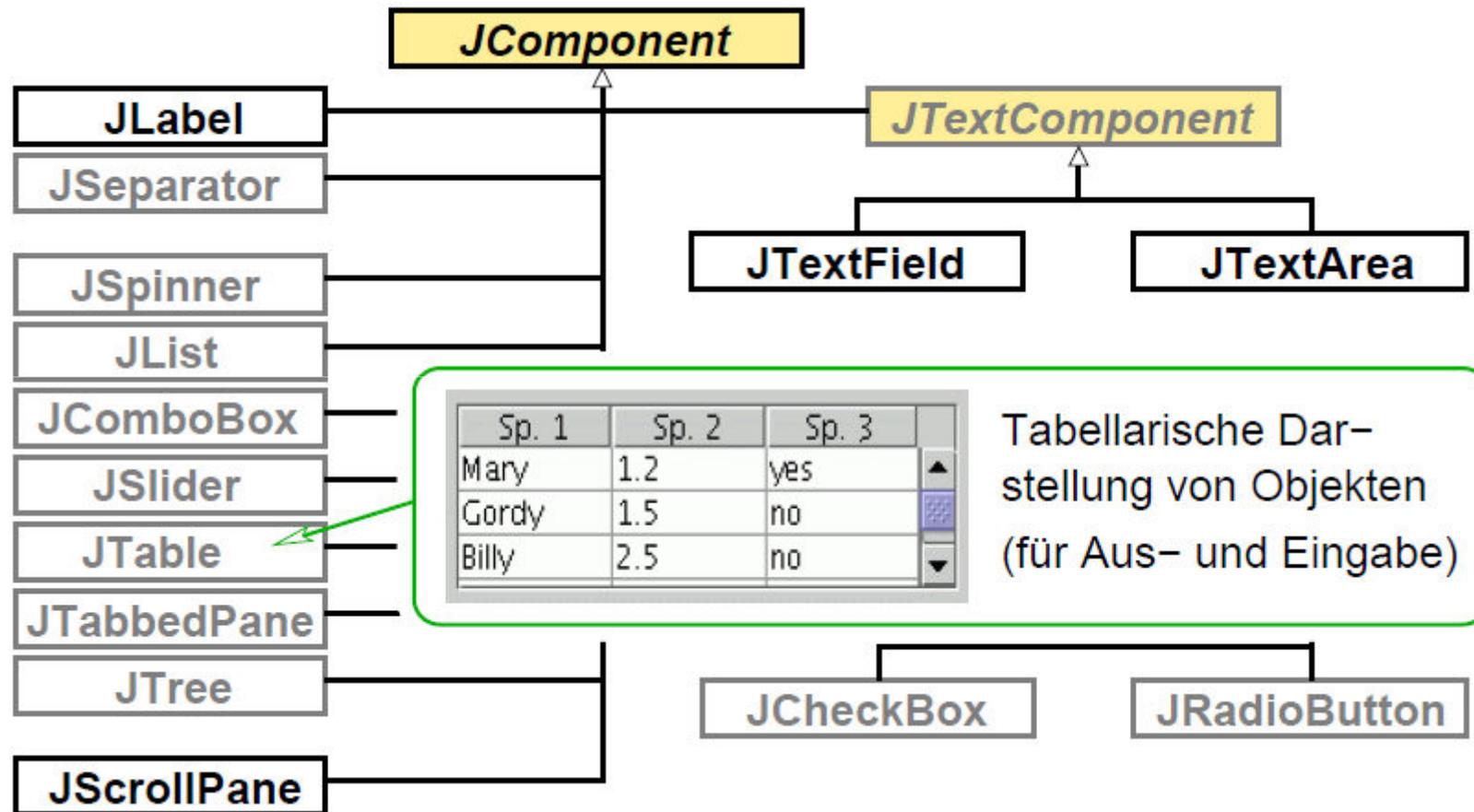
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

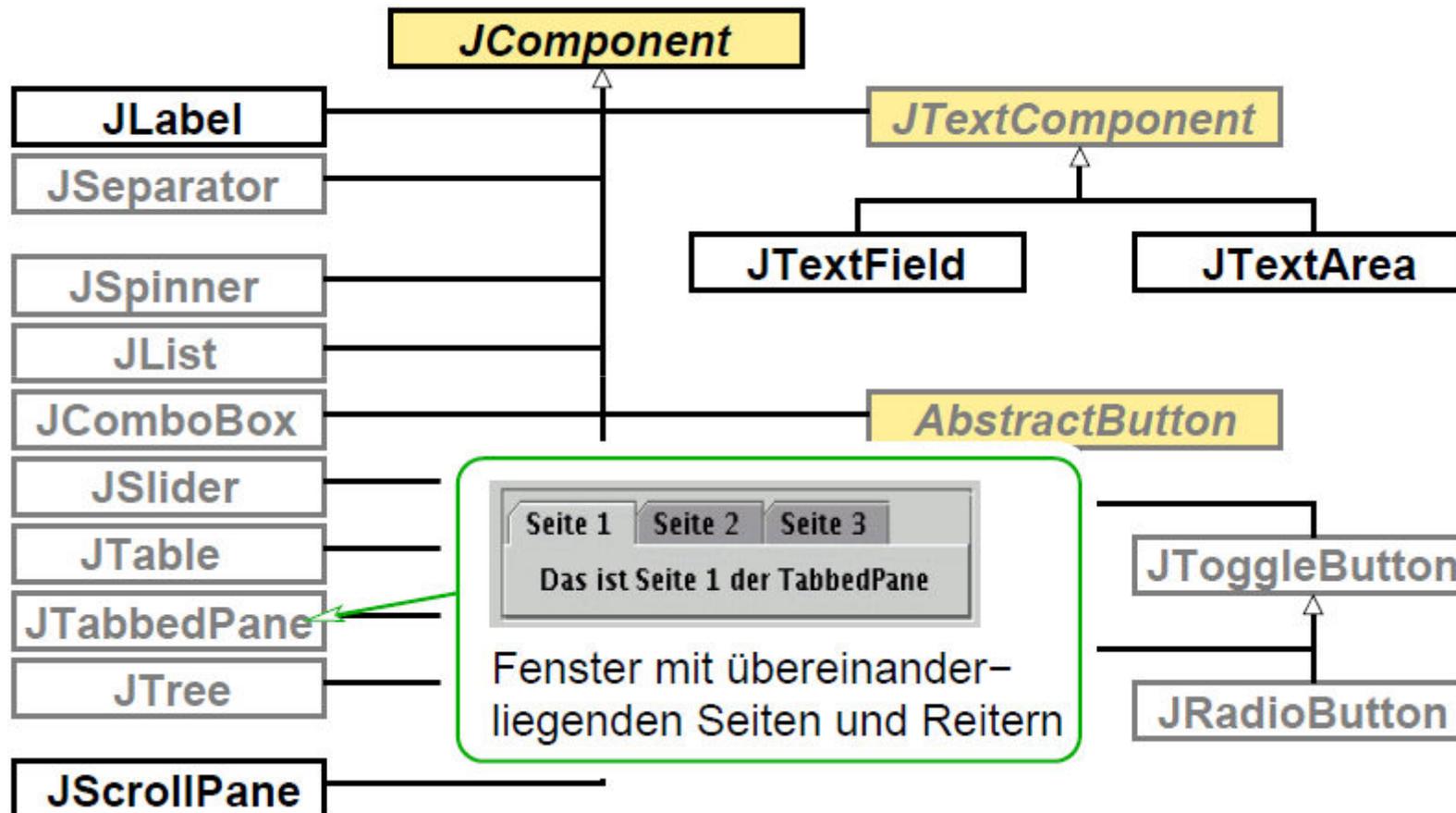
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

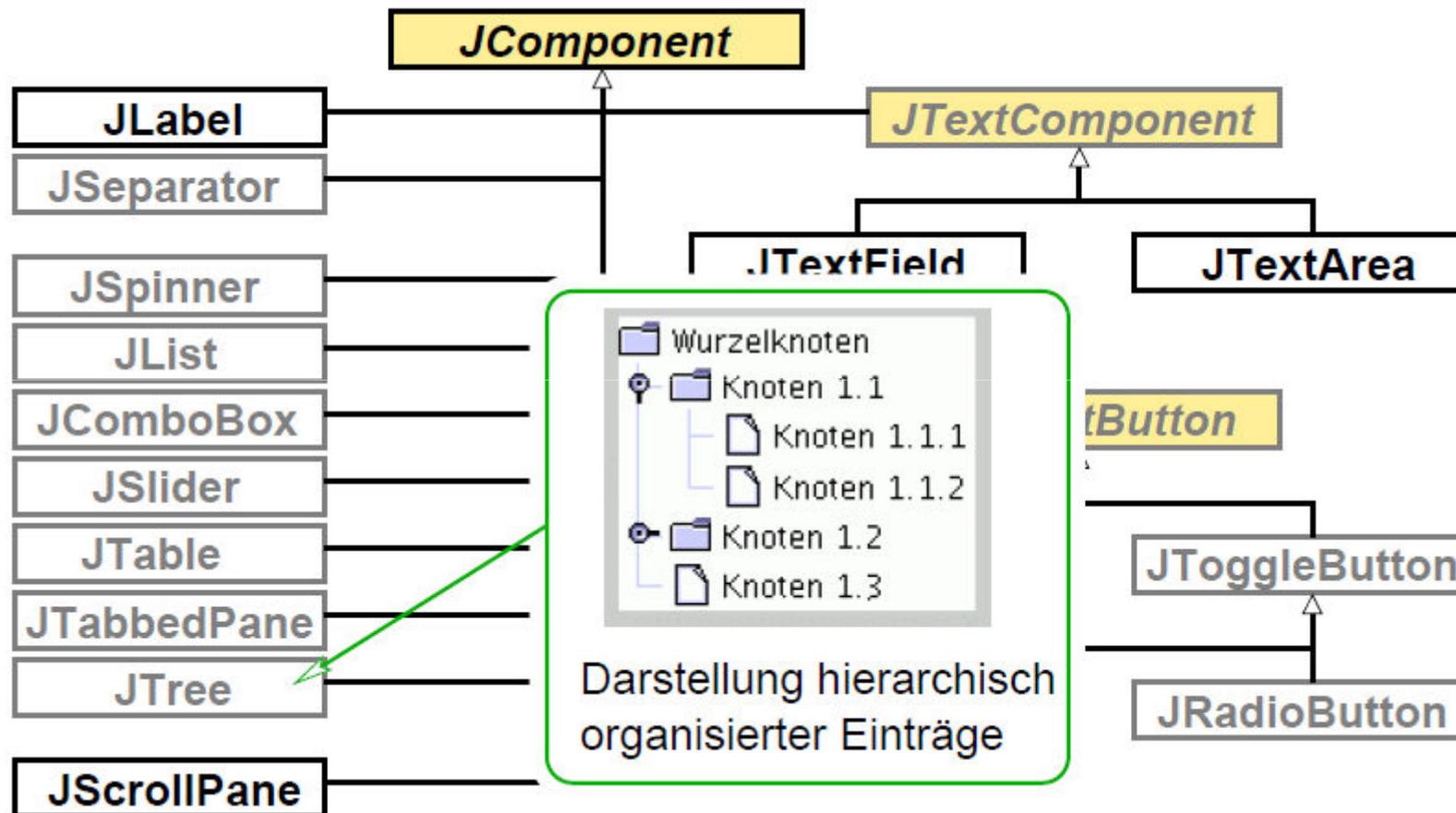
Hierarchie einiger Swing-Klassen für Interaktionselemente





5.4.2 Interaktionselemente ...

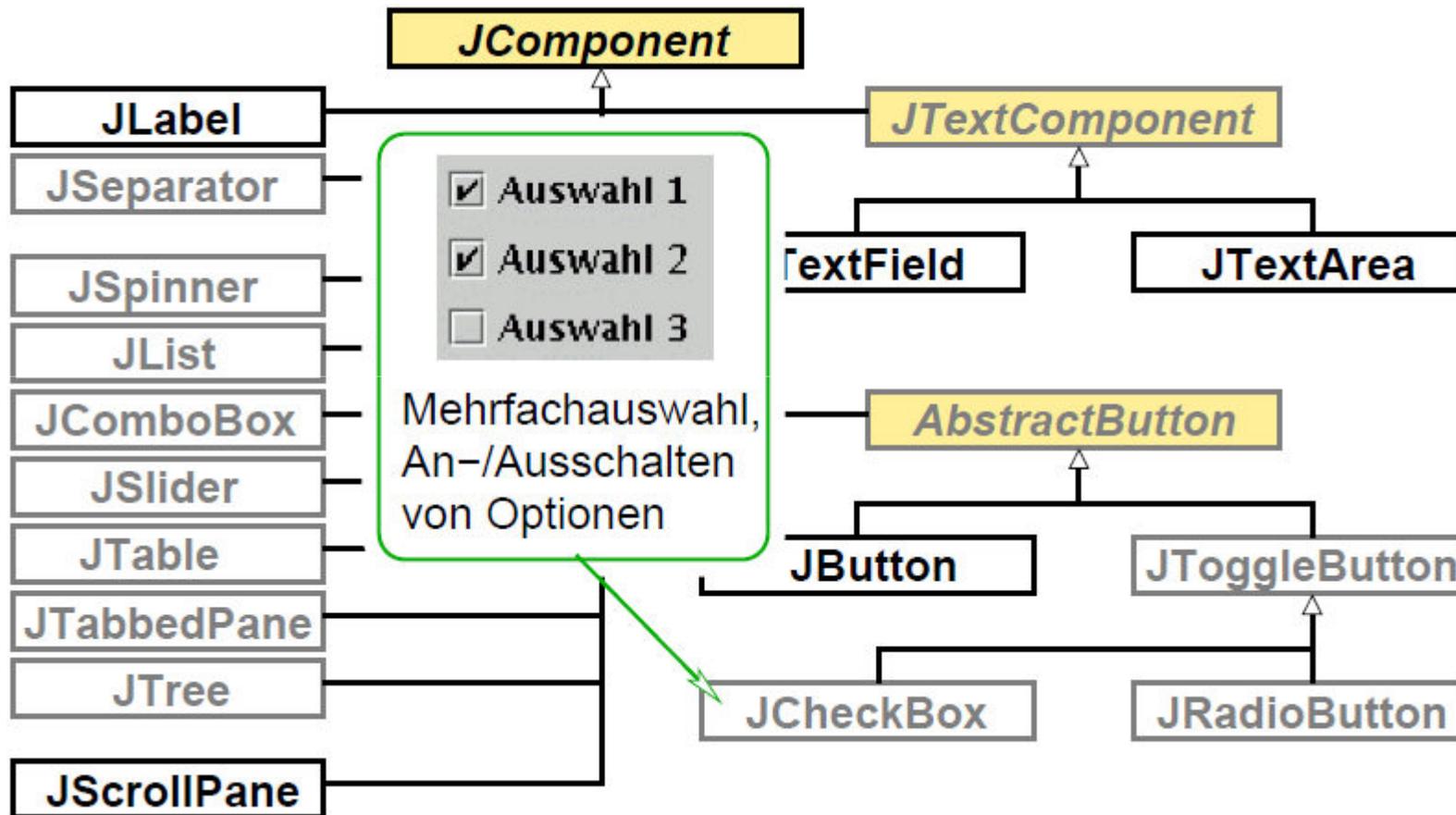
Hierarchie einiger Swing-Klassen für Interaktionselemente





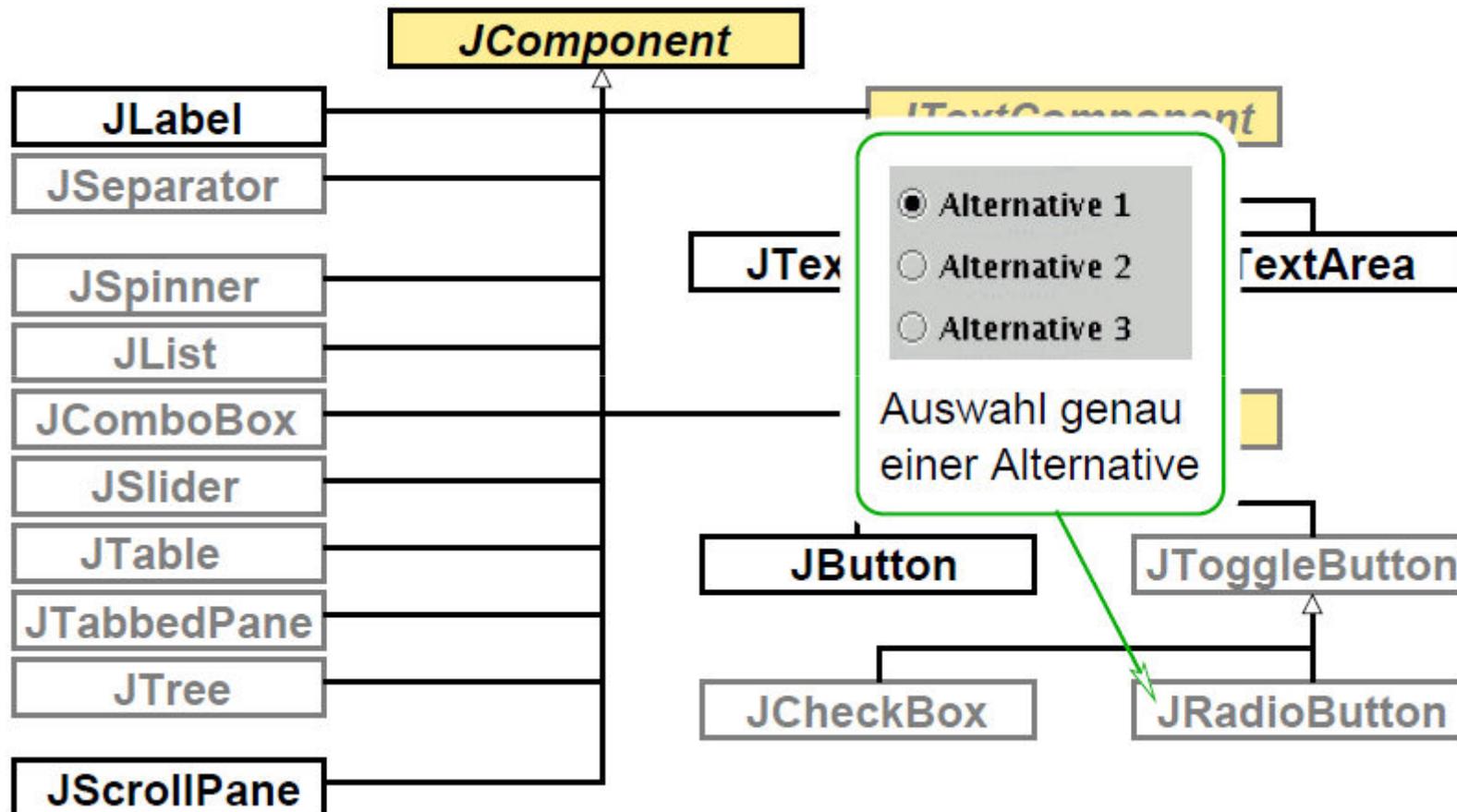
5.4.2 Interaktionselemente ...

Hierarchie einiger Swing-Klassen für Interaktionselemente



5.4.2 Interaktionselemente ...

Hierarchie einiger Swing-Klassen für Interaktionselemente

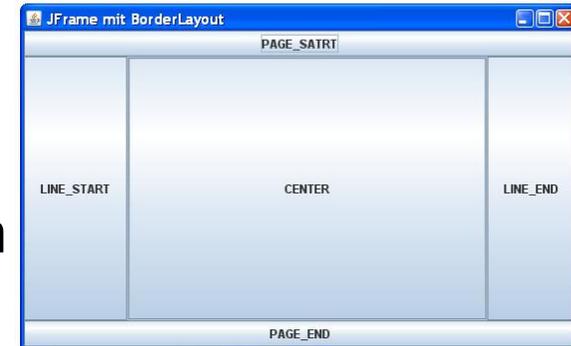




5.4 GUI-Programmierung ...

5.4.3 Layout von Interaktionselementen

- Die Methode `add()` eines Container-Objekts erlaubt das Einfügen von Komponenten in ein Fenster
- Frage: Wie wird dabei das graphische Layout bestimmt?
- Diese Aufgabe wird vom Container an einen LayoutManager delegiert, der festlegt
 - wo eine Komponente in dem Fenster erscheint
 - wie sie sich beim Vergrößern/Verkleinern des Fensters verhält
- Für Teile eines Fensters können auch unterschiedliche Layouts verwendet werden:
 - die Komponenten werden dazu in einen weiteren Container (z.B. `JPanel` oder `Box`) verpackt, der ein anderes Layout nutzt
 - dieser Container selbst ist dabei unsichtbar





5.4.3 Layout von Interaktionselementen ...

Standard-Layout von Swing-Fenstern: BorderLayout

- Kann maximal fünf Komponenten verwalten
 - eine oben, unten, links bzw. rechts und eine in der Mitte



```
JFrame win = new JFrame("Test");
Container cont = win.getContentPane();
cont.add(new JButton("North"), BorderLayout.NORTH);
cont.add(new JButton("South"), BorderLayout.SOUTH);
cont.add(new JButton("East"), BorderLayout.EAST);
cont.add(new JButton("West"), BorderLayout.WEST);
cont.add(new JButton("Center"), BorderLayout.CENTER);
```

5.4.3 Layout von Interaktionselementen ...

Standard-Layout des JPanel-Containers: `FlowLayout`

- Beliebige viele Komponenten werden in einer Zeile angeordnet, ggf. mit Zeilenumbruch bei Änderung der Fenstergröße



```

JPanel panel = new JPanel();
JFrame win = new JFrame("Test");
panel.add(new JButton("Button 1"));
panel.add(new JButton("Butto..."));
...
win.getContentPane().add(panel);

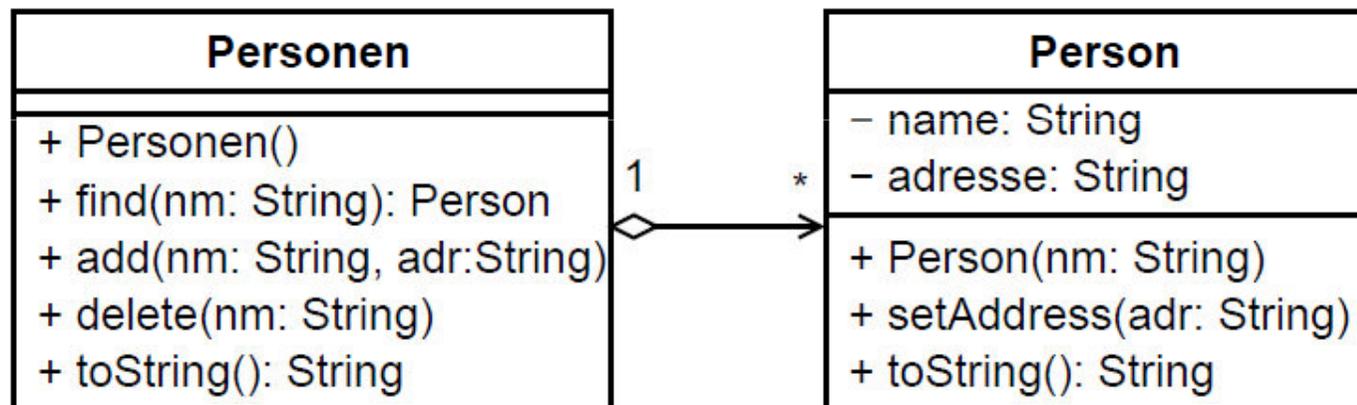
```

- Einem `JPanel`-Objekt kann bei der Erzeugung auch ein anderer `Layout-Manager` übergeben werden
- `JPanel`-Objekte können beliebig verschachtelt werden

5.4 GUI-Programmierung ...

5.4.4 Ein Beispiel

- Es soll ein System zur Verwaltung von Namen und Adressen realisiert werden
- OOD-Klassendiagramm des Fachkonzepts (Anwendungslogik):



- Das Programm soll eine graphische Bedienoberfläche erhalten



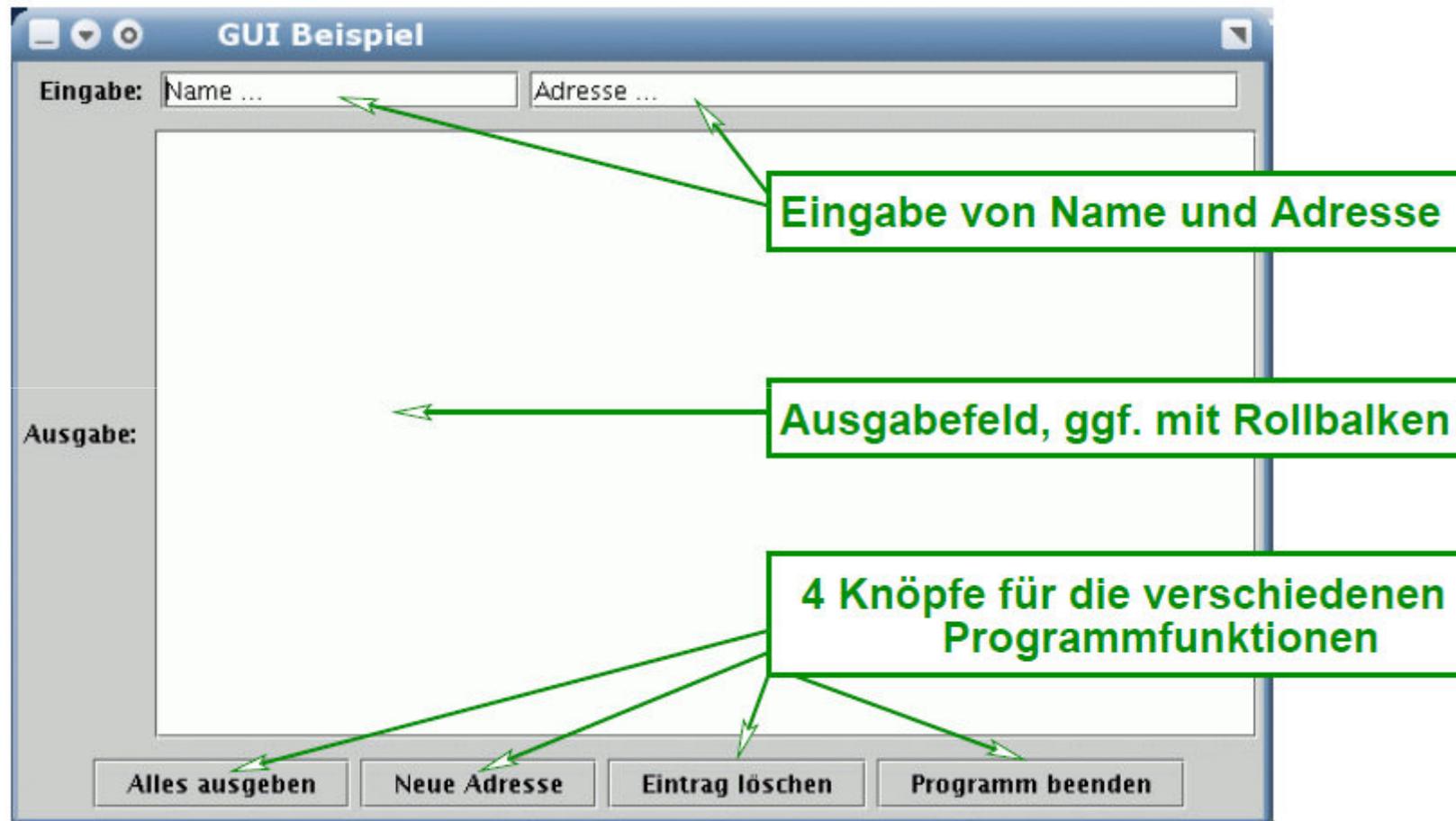
5.4.4 Ein Beispiel ...

Anforderungen an die Bedienoberfläche

- Die Bedienoberfläche soll die folgenden Funktionen unterstützen:
 - Anlegen eines neuen Personen-Eintrags
 - Ändern der Adresse zu einem Namen
 - Ausgabe der Liste aller Personen mit Adressen
 - Löschen einer Person aus der Liste
- Design-Entscheidung:
 - Personennamen sind Primärschlüssel, d.h. mehrere Personen mit dem gleichen Namen sind nicht zulässig
 - das Anlegen und Ändern eines Eintrags wird identisch behandelt

5.4.4 Ein Beispiel ...

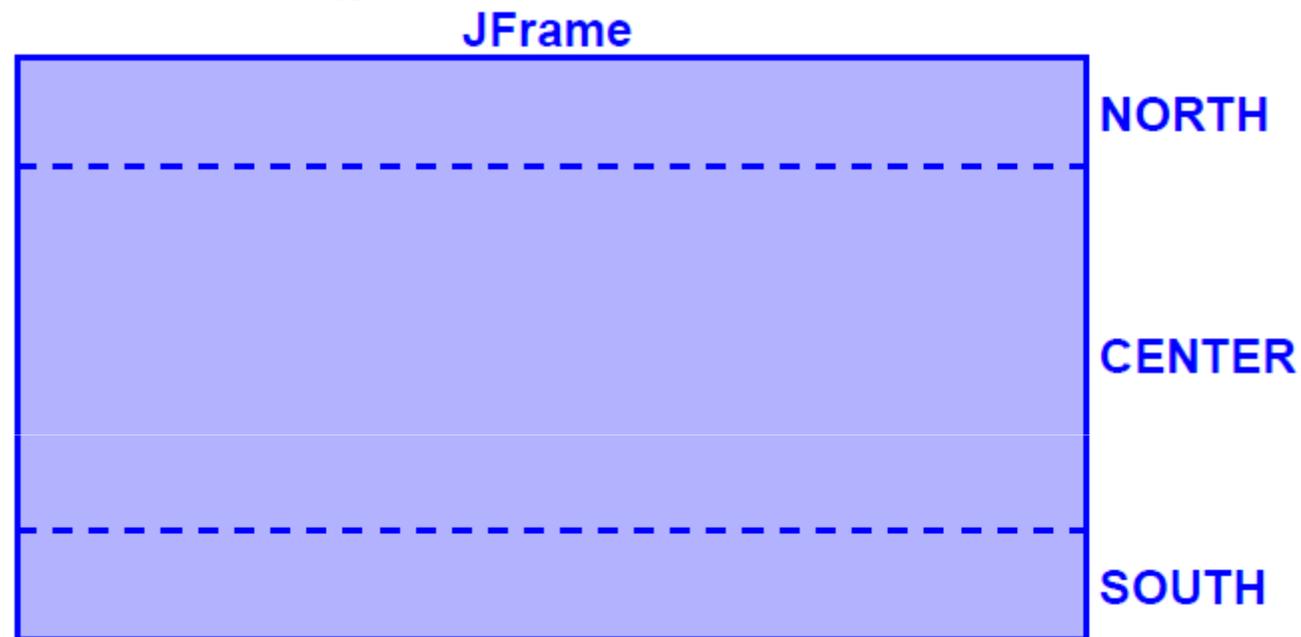
Entwurf des Anwendungsfensters





5.4.4 Ein Beispiel ...

Aufbau des Anwendungsfensters



- JFrame-Fenster mit drei Bereichen: NORTH, CENTER, SOUTH



5.4.4 Ein Beispiel ...

Aufbau des Anwendungsfensters JFrame

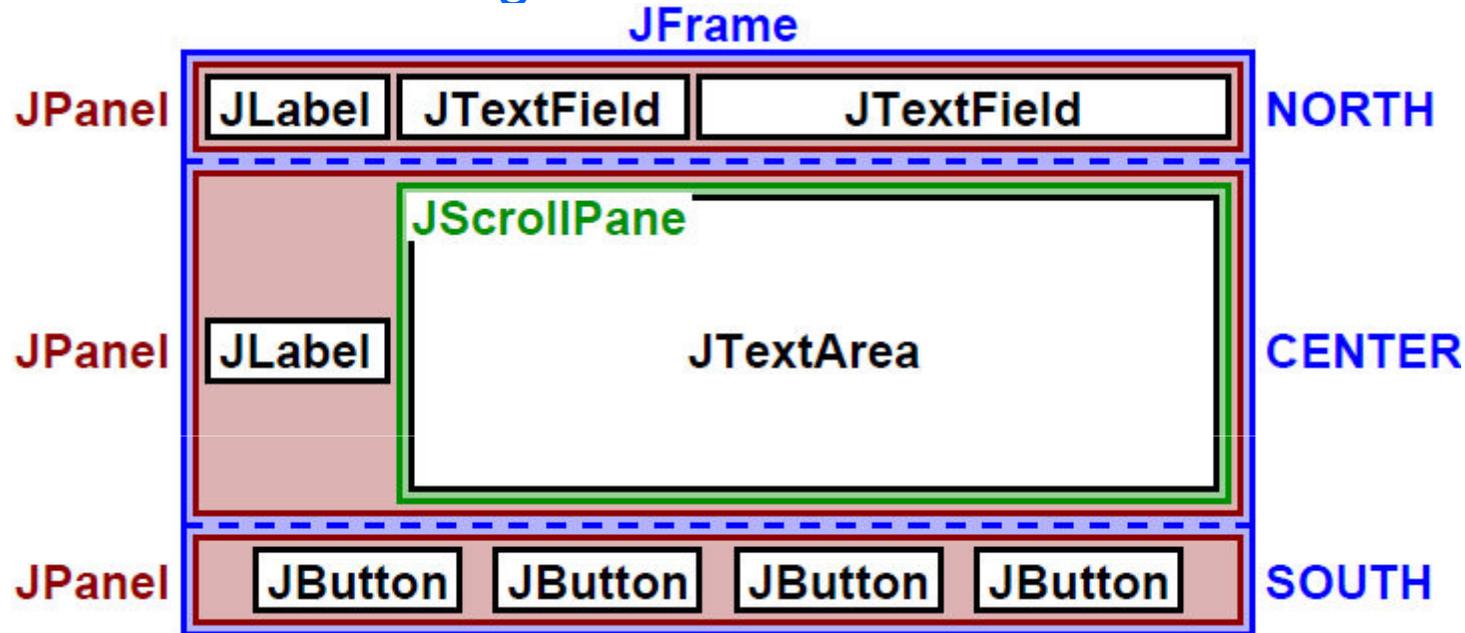


- JFrame-Fenster mit drei Bereichen: NORTH, CENTER, SOUTH
- In jedem Bereich ein JPanel zur Aufnahme mehrerer Elemente



5.4.4 Ein Beispiel ...

Aufbau des Anwendungsfensters



- JFrame-Fenster mit drei Bereichen: NORTH, CENTER, SOUTH
- In jedem Bereich ein JPanel zur Aufnahme mehrerer Elemente
 - die Elemente werden nebeneinander angeordnet



5.4.4 Ein Beispiel ...

Erster Programmentwurf für das GUI

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

class Bsp1GUI extends JFrame { // Wir erben von JFrame
    private JTextField ein1, ein2; // Eingabefelder
    private JTextArea aus; // Ausgabefeld

    public Bsp1GUI() {
        super("GUI Beispiel");
        Container container = getContentPane();
        // Eingabepanel mit Textfeldern erzeugen
        JPanel panel1 = new JPanel();
        panel1.add(new JLabel("Eingabe: "));
        ein1 = new JTextField("Name ...", 16);
        panel1.add(ein1);
    }
}
```



5.4.4 Ein Beispiel ...

```
ein2 = new JTextField("Adresse ...", 32);
panel1.add(ein2);
container.add(panel1, BorderLayout.NORTH); ←
// Ausgabepanel mit mehrzeiligem Textfeld erzeugen

JPanel panel2 = new JPanel();
panel2.add(new JLabel("Ausgabe: "));
aus = new JTextArea(20, 50);
aus.setEditable(false); // nicht editierbar
panel2.add(new JScrollPane(aus)); // Rollbalken

container.add(panel2, BorderLayout.CENTER); ←
// Panel mit vier Knöpfen erzeugen
JPanel panel3 = new JPanel();
JButton button1 = new JButton("Alles ausgeben");
panel3.add(button1);
JButton button2 = new JButton("Neue Adresse");
```



5.4.4 Ein Beispiel ...

```
panel3.add(button2);
JButton button3 = new JButton("Eintrag löschen");
panel3.add(button3);
JButton button4 = new JButton("Programm beenden");
panel3.add(button4);
container.add(panel3, BorderLayout.SOUTH); ←
pack(); // Fenstergröße automatisch berechnen
}

public static void main(String args[]) {
    // Anwendungsfenster erzeugen und sichtbar machen
    Bsp1GUI gui = new Bsp1GUI();
    gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    gui.setVisible(true);
}
}
```



5.4.4 Ein Beispiel ...

Was fehlt noch?

- Wir haben noch nicht programmiert, was beim Drücken der Knöpfe passieren soll
- Zwei Teilaufgaben:
 - Ereignisbehandlung: wir müssen in unserem Programm auf Ereignisse reagieren können, die der Benutzer auslöst
 - Maus- und Tastaturereignisse, z.B. Drücken eines Knopfes, Eingabe in ein Textfeld, ...
 - Anbindung des Fachkonzepts: die Fachklassen müssen an das GUI angebunden werden
 - im Beispiel: `Personen` und `Person`



5.4 GUI-Programmierung ...

5.4.5 Ereignisbehandlung in Java

- Ereignisse werden in Swing /AWT nach dem **Delegationsmodell** bearbeitet
- Die Ereignisquelle (z.B. ein Knopf) delegiert dabei die Verarbeitung des Ereignisses an ein anderes Objekt
 - Objekte registrieren sich bei der Quelle eines Ereignisses als "Ereignis-Abhörer" (*Event Listener*)
 - die Quelle informiert die Objekte dann über diese Ereignisse
 - durch Aufruf einer Methode einer Schnittstelle, die die Objekte implementieren müssen
- Die Ereignisse werden dabei als Objekte modelliert
 - Klassenhierarchie für die verschiedenen Ereignistypen
 - Mausbewegung, Knopf gedrückt, Taste gedrückt, ...



5.4.5 Ereignisbehandlung in Java ...

Delegationsmodell

Ereignisquelle

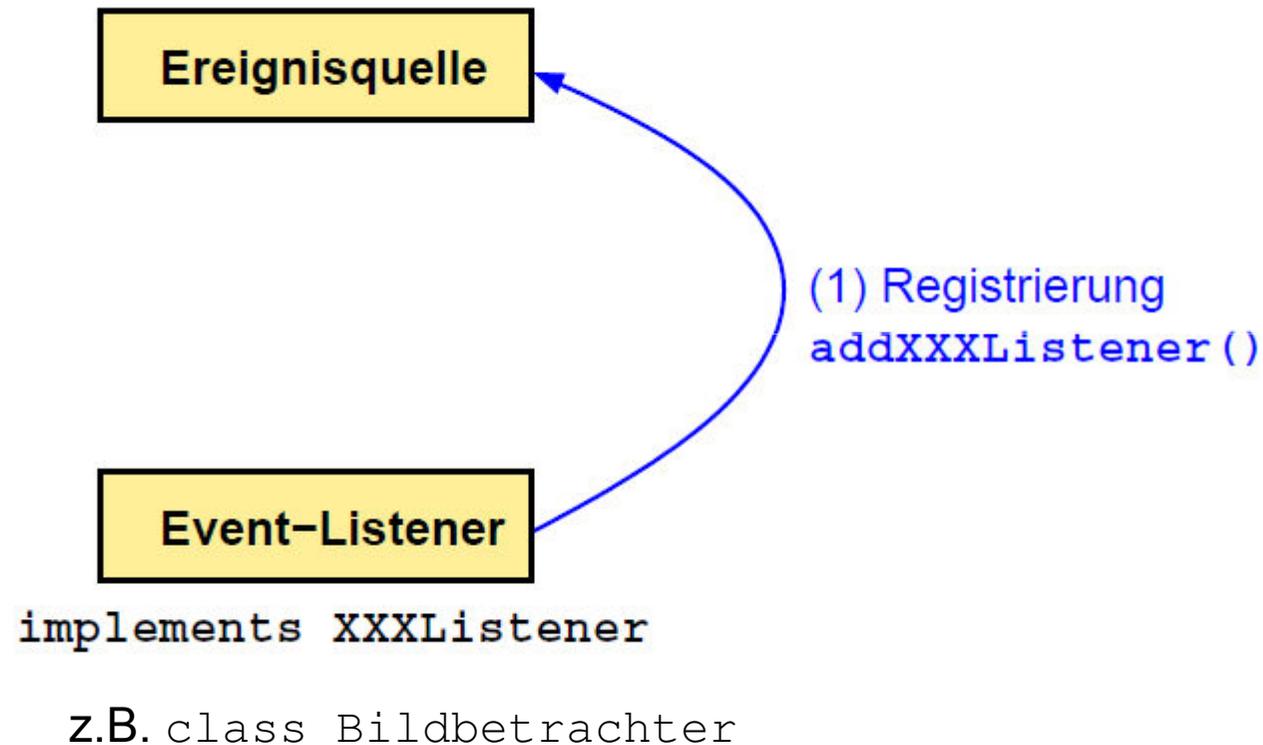
Event-Listener

`implements XXXListener`



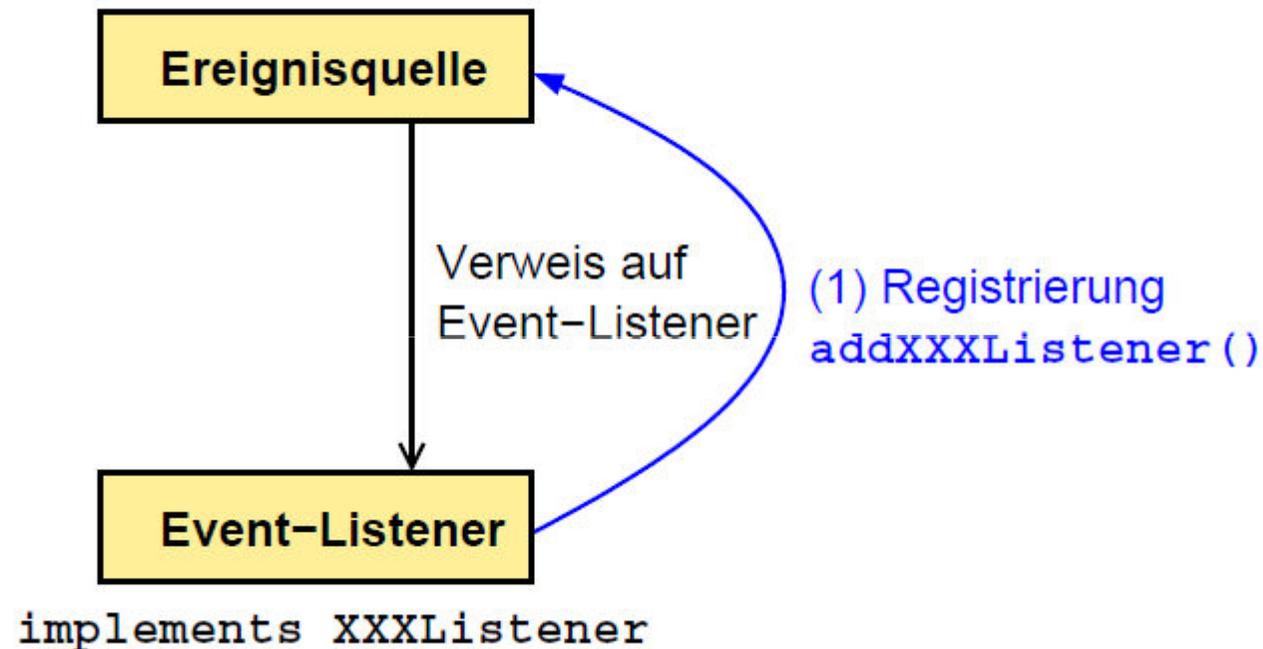
5.4.5 Ereignisbehandlung in Java ...

Delegationsmodell



5.4.5 Ereignisbehandlung in Java ...

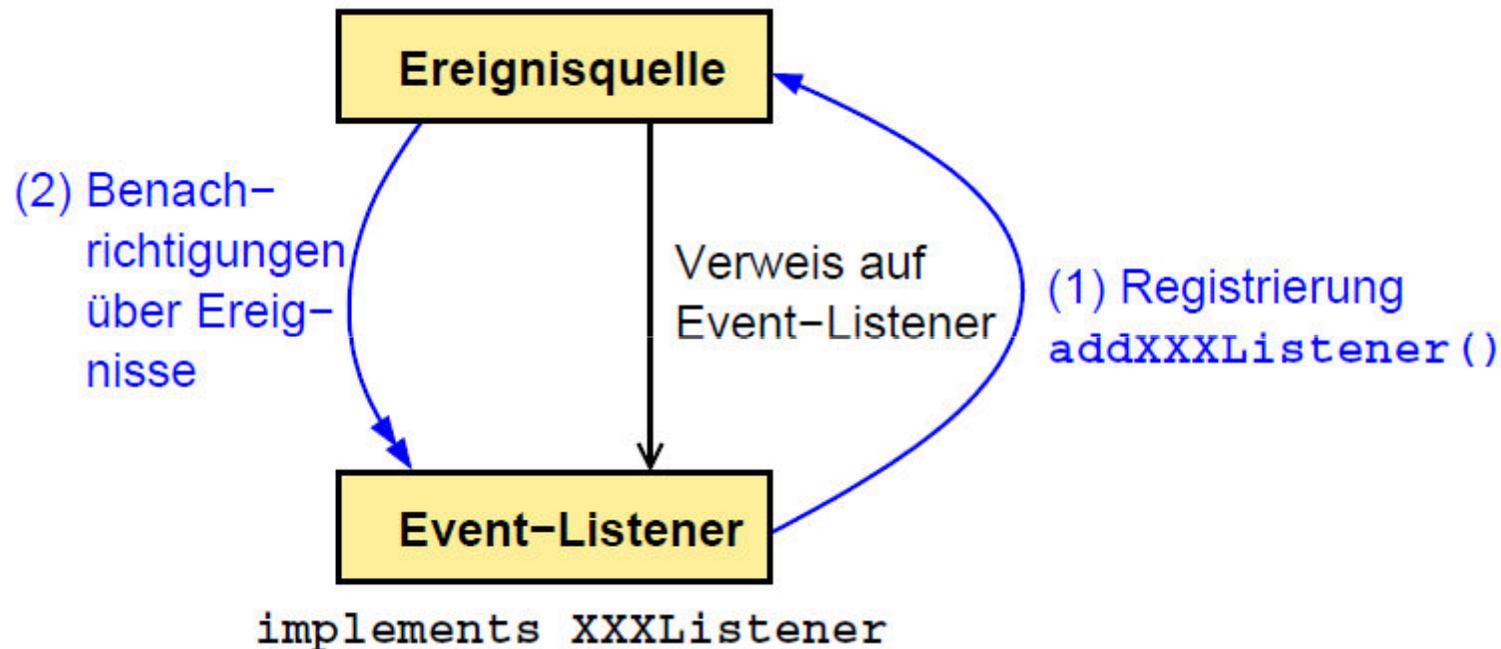
Delegationsmodell



- Für ein Ereignis können sich beliebig viele Listener registrieren

5.4.5 Ereignisbehandlung in Java ...

Delegationsmodell



- Für ein Ereignis können sich beliebig viele Listener registrieren



5.4.5 Ereignisbehandlung in Java ...

Die Ereignisklasse `java.awt.ActionEvent`

- Wichtigste Ereignisklasse in Swing / AWT: zeigt an, daß eine (komponentenspezifische) Benutzer-Aktion stattgefunden hat
 - z.B. Drücken eines Knopfes, Auswahl eines Menüpunkts
- `ActionEvent` erbt von `java.awt.AWTEvent`, das wiederum von `java.util.EventObject` erbt
- Wichtige Methode von `AWTEvent`:
 - `Object getSource()`: liefert Quelle des Ereignisses (z.B. Menüeintrag)
- Wichtige Methode von `ActionEvent`:
 - `String getActionCommand()`: liefert den Kommandostring der auslösenden Swing / AWT-Komponente (z.B. "Öffnen")
 - Voreinstellung für den Kommandostring: Beschriftung der Komponente



5.4.5 Ereignisbehandlung in Java ...

Event-Listener

- Zu jeder Ereignisklasse gibt es eine zugehörige Event-Listener-Schnittstelle
 - Namenskonvention: für ein Ereignis `XXXEvent`
 - heißt die Listener-Schnittstelle `XXXListener`
 - ist die Registrierungsmethode der Swing /AWT-Komponenten
 - `void addXXXListener(XXXListener l)`
 - Ereignisse werden über Methoden der Schnittstelle gemeldet,
z.B.:

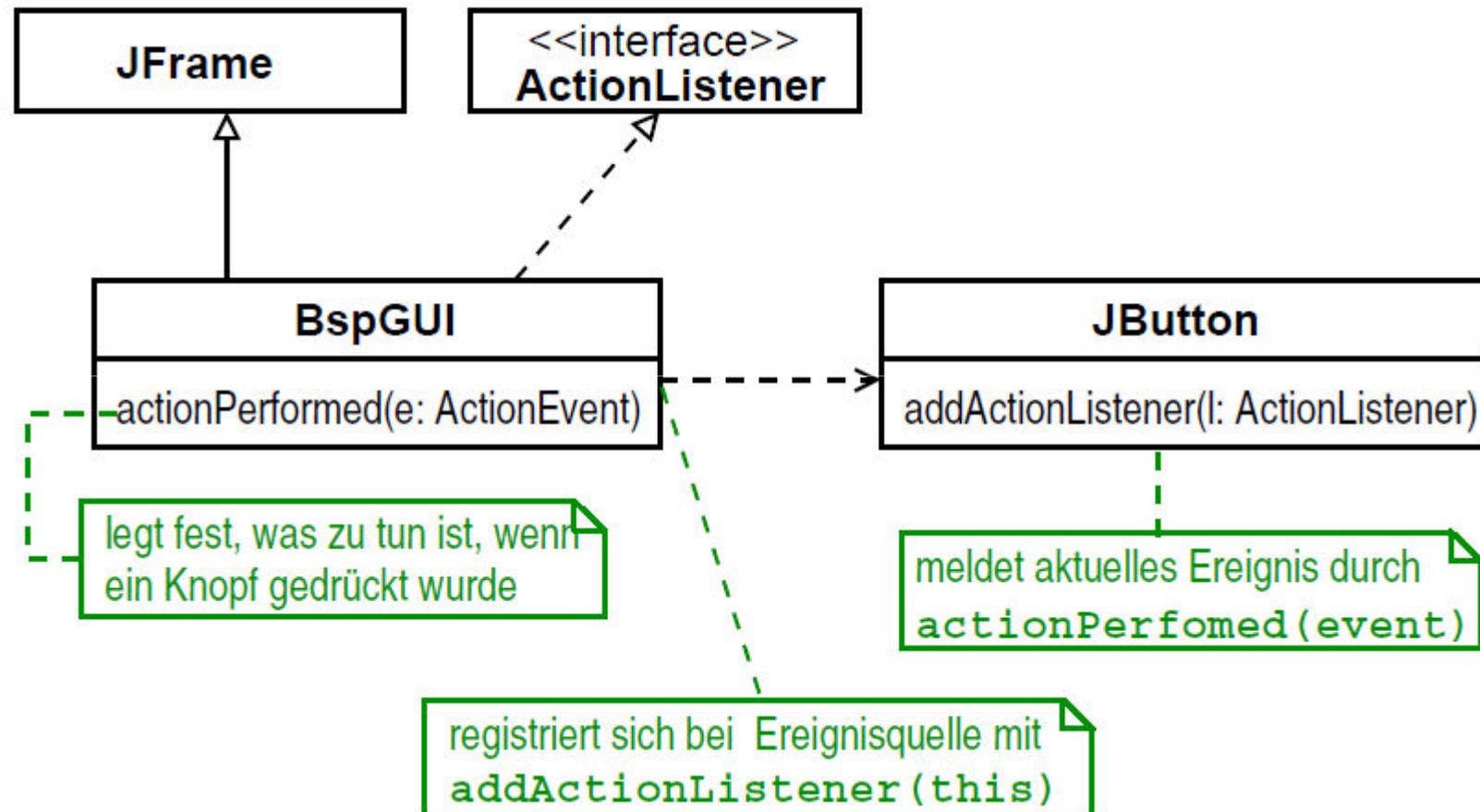
```
interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

 - ggf. auch mehrere Methoden für unterschiedliche Ereignisse
 - Die Listener-Objekte müssen diese Schnittstelle implementieren
-



5.4.5 Ereignisbehandlung in Java ...

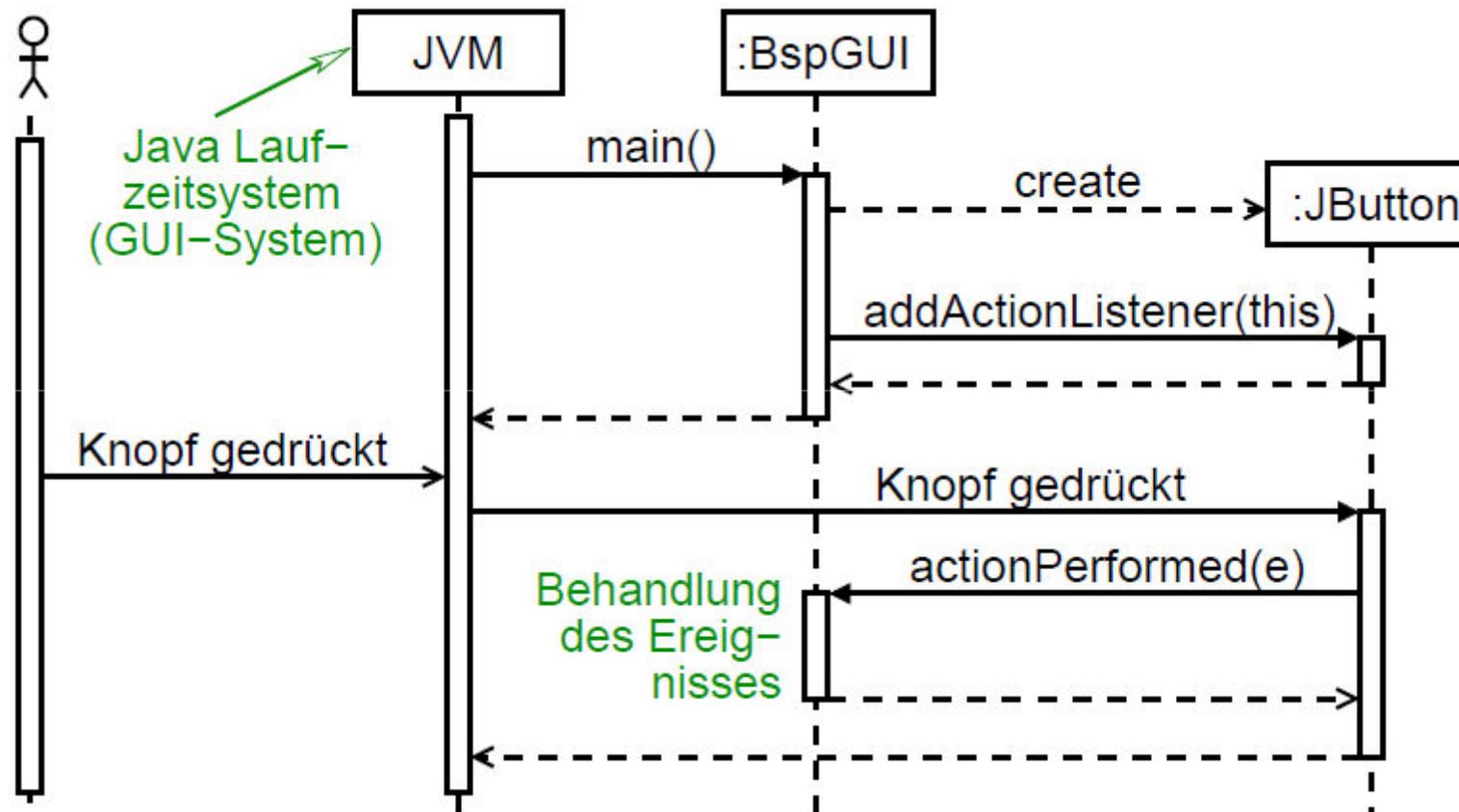
Delegationsmodell für einen Knopf: Klassendiagramm





5.4.5 Ereignisbehandlung in Java ...

Delegationsmodell für einen Knopf: Sequenzdiagramm





5.4.5 Ereignisbehandlung in Java ...

GUI-Bsp. mit funktionierenden Knöpfen

...

```
import java.awt.event.*;
class Bsp2GUI extends JFrame implements ActionListener {
    ... // Bsp2GUI ist ein ActionListener
    public Bsp2GUI () {
        ...
        JPanel panel3 = new JPanel();
        JButton button1 = new JButton("Alles ausgeben");
        button1.addActionListener(this); // als Listener registrieren
        panel3.add(button1);
        JButton button2 = new JButton("Neue Adresse");
        button2.addActionListener(this); // als Listener registrieren
        ...
    }
}
```



5.4.5 Ereignisbehandlung in Java ...

```
// Wird aufgerufen, wenn irgendein Knopf gedrueckt wurde
public void actionPerformed(ActionEvent e) {
    String label = e.getActionCommand();
    if (label.equals("Alles ausgeben")) {
        aus.append("Ausgabeknopf gedrueckt\n");
    }
    else if (label.equals("Neue Adresse")) {
        aus.append("Neue-Adresse-Knopf gedrueckt\n");
        // getText() liefert den eingegebenen String
        aus.append("Name: " + ein1.getText() +
            ", Adresse: " + ein2.getText() + "\n");
    }
    else if (label.equals("Eintrag loeschen")) { ... }
    else { dispose(); } // Fenster schliessen
}
...
```



5.4.5 Ereignisbehandlung in Java ...

Die Knöpfe funktionieren





5.4.5 Ereignisbehandlung in Java ...

Diskussion

- Im Beispiel behandelt eine einzige Methode `actionPerformed()` die Ereignisse **aller** GUI-Elemente des Fensters (zentrale “Verteiler”-Methode)
 - keine gute Code-Struktur
 - unübersichtlich bei komplexeren Fenstern
- Alternativ könnten wir für jeden Knopf eine eigene Listener-Klasse implementieren und registrieren
 - viele Klassen, viel Schreibaarbeit, ...
- Lösung: Verwendung **anonymer Klassen**
 - sie erlauben es, für jedes GUI-Element eine eigene `actionPerformed()`-Methode zu programmieren



5.4.5 Ereignisbehandlung in Java ...

Anonyme Klassen

- Anonyme Klassen sind Klassen ohne Namen
- Sie sind ein Spezialfall innerer Klassen
- Sie besitzen keinen Konstruktor, können aber den Konstruktor ihrer Oberklasse mit Parametern aufrufen
- Sie werden in einer new-Anweisung definiert, die gleichzeitig das einzige Element der Klasse erzeugt
- Sie können auch Schnittstellen implementieren:

```
new <InterfaceName> () { <MethodenImplementierungen> }
```

 - kein `implements ...`
- Anwendung für einfache Aufgaben (bis ca. 5 Anweisungen), die ein einzelnes Objekt lösen kann



5.4.5 Ereignisbehandlung in Java ...

GUI-Beispiel mit anonymen Klassen

```
class Bsp3GUI extends JFrame {  
    ...  
    private JTextArea aus; // ist von innerer Klasse aus zugreifbar!  
  
    public Bsp3GUI() {  
        ...  
        JButton button1 = new JButton("Alles ausgeben");  
        // Erzeuge/registriere Objekt, das ActionListener  
        implementiert  
        button1.addActionListener(new ActionListener() {  
            // Implementierung der Schnittstelle  
            public void actionPerformed(ActionEvent e) {  
                aus.append("Ausgabeknopf gedrueckt\n");  
            }  
        });  
        ...  
    }  
}
```



5.4.5 Ereignisbehandlung in Java ...

Anonyme Klassen ...

➤ Die Anweisung

```
button1.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        aus.append("Ausgabeknopf gedrueckt\n");  
    }  
});
```

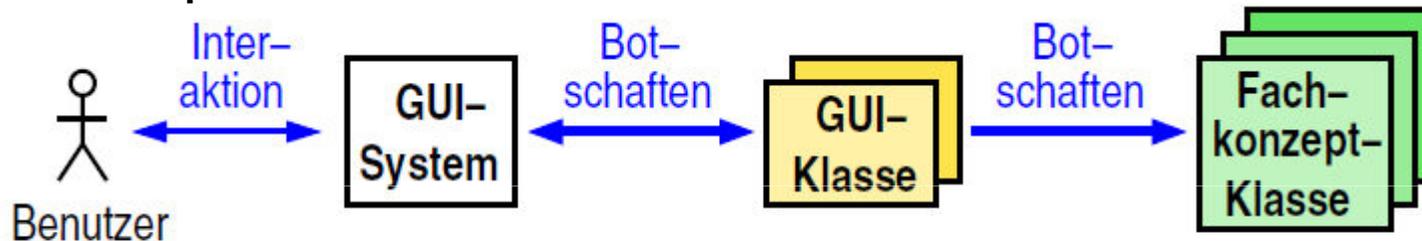
ist gleichbedeutend mit

```
class Listen1 implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        aus.append("Ausgabeknopf gedrueckt\n");  
    }  
}  
button1.addActionListener(new Listen1());
```

5.4 GUI-Programmierung ...

5.4.6 Anbindung des Fachkonzepts

- Grundlegendes Prinzip beim Software-Entwurf: klare Trennung zwischen Benutzungsoberfläche und Fachkonzept



- GUI-Klassen senden Botschaften an die Fachkonzept-Klassen
- Die Fachkonzept-Klassen kennen die GUI-Klassen nicht
 - d.h., sie senden keine Botschaften an die GUI-Klassen
 - sichert Wiederverwendbarkeit des Fachkonzepts



5.4.6 Anbindung des Fachkonzepts ...

Technisches Vorgehen

- Die `main()`-Methode erzeugt das Fachkonzept und das GUI
 - der Konstruktor des GUI erhält eine Referenz auf das Fachkonzept als Parameter
- GUI-Klassen speichern eine Referenz auf das Fachkonzept
- GUI-Klassen rufen Methoden des Fachkonzepts auf
- Fehlerbehandlung:
 - Fachkonzept-Klassen melden Fehler über Exceptions oder Rückgabewerte
 - GUI-Klassen erzeugen Informations-Dialog für Benutzer
 - alternativ: Verwendung des Delegationsmodells bzw. Beobachtermusters (=> 6.4)



5.4.6 Anbindung des Fachkonzepts ...

GUI-Beispiel mit Fachkonzept (=> WWW: Bsp4GUI.java)

```
class Bsp4GUI extends JFrame {
    ...
    private Personen personen; // Referenz auf Fachkonzept
    public Bsp4GUI(Personen pers) {
        super("GUI Beispiel");
        personen = pers; // Fachkonzept-Objekt merken
        ...
        JButton button1 = new JButton("Alles ausgeben");
        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Aufruf einer Operation des Fachkonzepts
                aus.append(personen.toString());
            }
        });
    }
};
```



5.4.6 Anbindung des Fachkonzepts ...

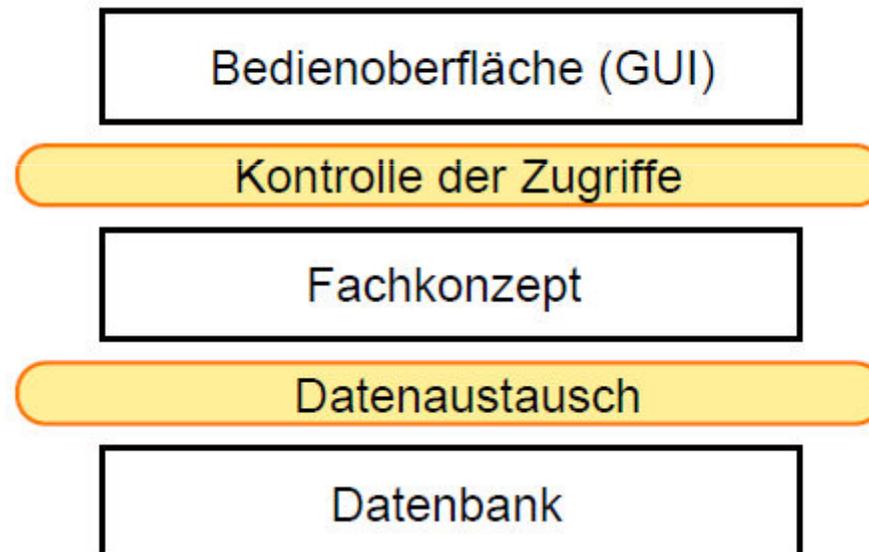
```
...
// Knopf "Neue Adresse"
personen.add(ein1.getText(), ein2.getText());
...
// Knopf "Eintrag loeschen"
personen.delete(ein1.getText());
...
}
public static void main(String args[]) {
    Personen pers = new Personen(); // Fachkonzept erzeugen
    Bsp4GUI gui = new Bsp4GUI(pers); // und an GUI uebergeben
    ...
}
}
```



5.4.6 Anbindung des Fachkonzepts ...

Drei-Schichten-Architektur

- Gängigste Software-Architektur bei kaufmännischen und administrativen Anwendungen
 - trennt GUI, Fachkonzept und persistente Datenhaltung



- Schichten ggf. auch auf mehrere Rechner verteilt

5 Programmierung mit Java ...

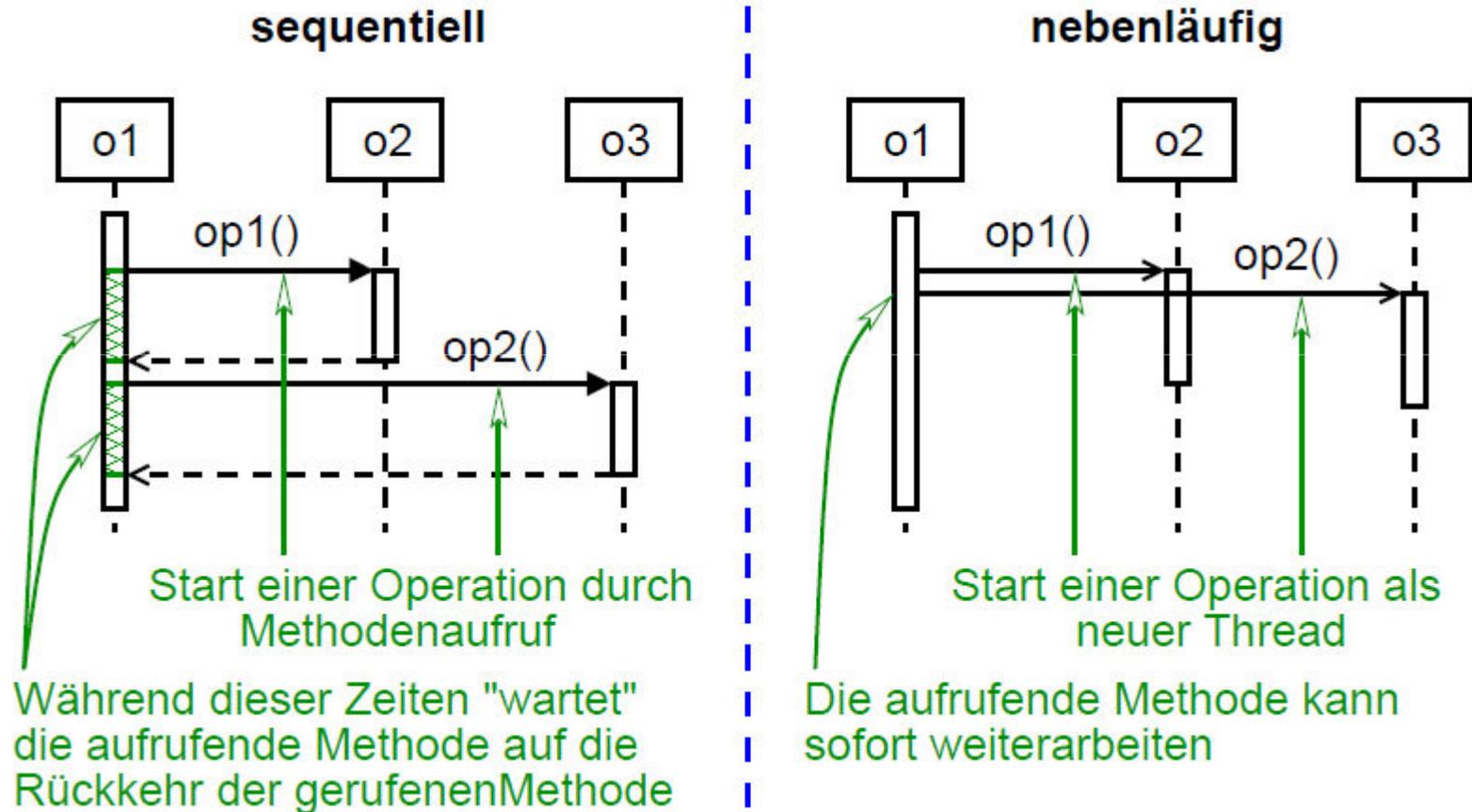


5.5 Threads

- Unsere bisherigen Programme arbeiteten rein **sequentiell**
 - die Anweisungen wurden eine nach der anderen ausgeführt
- Manchmal sollte ein Programm aber auch **nebenläufig** arbeiten können, d.h. mehrere Dinge (scheinbar) gleichzeitig tun, z.B.:
 - Ausgabe von Bild und Ton einer Multimedia-Anwendung
 - gleichzeitige Darstellung mehrerer Animationen
 - Bearbeitung längerer Aufgaben (z.B. Drucken) im "Hintergrund", während mit dem Programm weitergearbeitet wird
- Ein **Thread** ist eine Aktivität (d.h. Ausführung von Programmcode), die nebenläufig zu anderen Aktivitäten ausgeführt wird
 - alle Threads einer Programmausführung arbeiten dabei auf denselben Daten

5.5 Threads ...

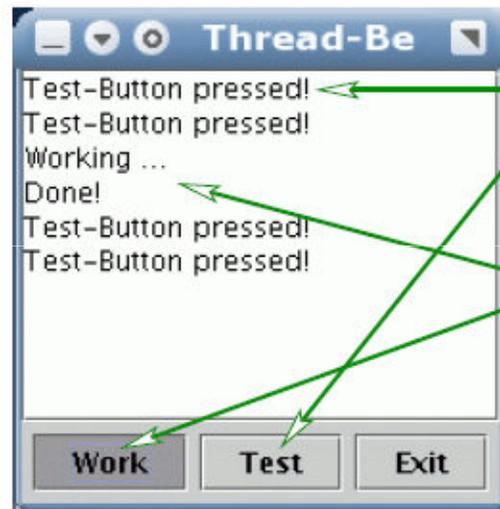
Sequentielles vs. nebenläufiges Programm



5.5 Threads ...

Beispiel zur Motivation (=> WWW: ThreadEx1.java)

- GUI mit einer Funktion, die eine längere Bearbeitungszeit hat, z.B. Drucken eines Dokuments



Drücken des 'Test'-Knopfes führt zu einer Ausgabe im Textfeld.

Drücken des 'Work'-Knopfes führt zum Aufruf einer Methode (doWork()) mit hoher Ausführungszeit.

- Während der Ausführung der Methode reagiert das Programm nicht auf Benutzereingaben (z.B. die anderen Knöpfe)



5.5 Threads ...

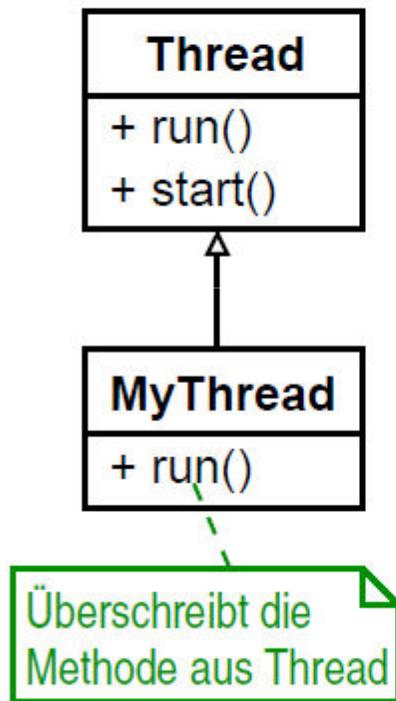
Threads in Java

- Ein Java-Programm startet immer mit genau einem Thread (`main`-Thread), der die Methode `main()` abarbeitet
- Für weitere Threads steht die Klasse `Thread` zur Verfügung
 - von dieser Klasse muß eine Unterklasse definiert werden
- Die wichtigsten Operationen von `Thread` sind:
 - `void run()`: wird beim Start des Threads ausgeführt
 - muß in der Unterklasse überschrieben werden
 - mit dem Code, den der Thread ausführen soll
 - der Thread endet, wenn `run()` zurückkehrt
 - `void start()`: startet den Thread
 - `start()` kehrt sofort zum Aufrufer zurück
 - der Thread führt nebenläufig seine `run()`-Methode aus

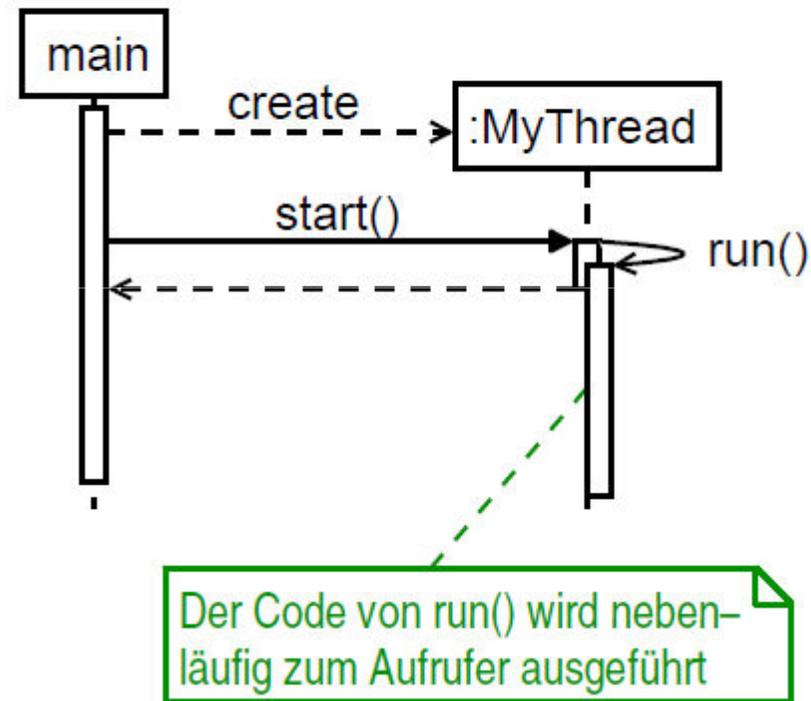
5.5 Threads ...

Threads in Java ...

Klassendiagramm



Sequenzdiagramm





5.5 Threads ...

Beispiel: Berechnung im Hintergrund

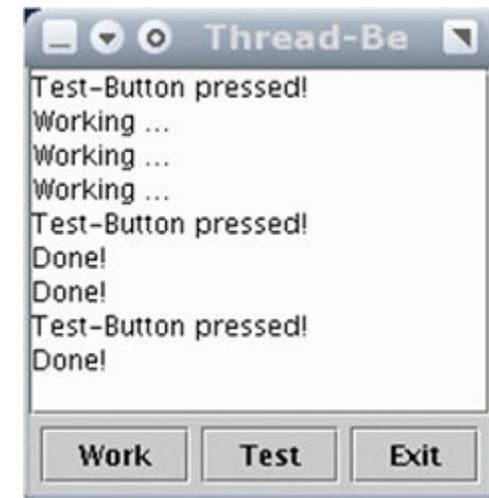
```
class WorkThread extends Thread {
    private JTextArea aus; // Fuer die Ausgabe der Meldungen
    WorkThread(JTextArea t) { aus = t; }
    // Beim Start des Threads nebenlaeufig zum Aufrufer
    ausgefuehrt
    public void run() {
        aus.append("Working ...\n");
        double v = 1.000000001;
        // Schleife simuliert komplexe Berechnung ...
        for (int i=0; i<5000000; i++) { v = v*v; }
        aus.append("Done!\n");
    }
}
```

5.5 Threads ...

Beispiel: Berechnung im Hintergrund ...

```
private void doWork() {
    // Erzeuge ein neues Thread-Objekt
    Thread t = new WorkThread(aus);
    // Fuehre die run-Methode des Objekts in einem neuen Thread
    aus
    t.start();
}
```

- Drücken des "Work"-Knopfes blockiert GUI nicht mehr
 - aber: mehrfaches Drücken startet mehrere nebenläufige Berechnungen!





5.5 Threads ...

Synchronisation von Threads

- Eine Hintergrund-Berechnung wie im Beispiel ist nur möglich, wenn das Ergebnis nicht zum Weiterarbeiten benötigt wird
- Andernfalls kann mit der Methode `join()` auf das Ende des Threads gewartet werden, wenn das Ergebnis gebraucht wird
 - dieses kann / muß in Attributen des Thread-Objekts gespeichert werden
- In vielen Fällen ist auch eine weitergehende Synchronisation der Threads erforderlich (+ Vorlesung "Betriebssysteme I"):
 - wechselseitiger Ausschluß von Methoden
 - verhindert gleichzeitige Ausführung durch mehrere Threads
 - Warten auf Ereignisse, die andere Threads auslösen



5.5 Threads ...



Beispiel: Bankkonto

```
class Konto {  
    ...  
    public boolean abheben(double betrag) {  
        double neuerSaldo = getSaldo() - betrag;  
        boolean ok = true;  
        if (neuerSaldo < 0) {  
            // Bei Ueberziehung: Anfrage an Schufa  
            // (ueber Netzwerk)  
            ok = frageSchufa(neuerSaldo); // kann dauern ...  
        }  
        if (ok)  
            putSaldo(neuerSaldo); // Buchung durchfuehren  
        return ok;  
    }  
}
```



5.5 Threads ...



```
public class Banking extends Thread {
    Konto konto; // Eingabedaten fuer den Thread
    double betrag; // - " -
    Banking(Konto k, double b) { konto = k; betrag = b; }
    public void run() {
        konto.abheben(betrag);
        System.out.println("Kontostand: " + konto.putSaldo());
    }
    public static void main(String args[]) {
        Konto konto = new Konto(10); // Konto mit 10 EUR
        for (int i=0; i<3; i++) { // dreimal 10 EUR abheben
            Banking t = new Banking(konto, 10);
            t.start();
        }
    }
}
```



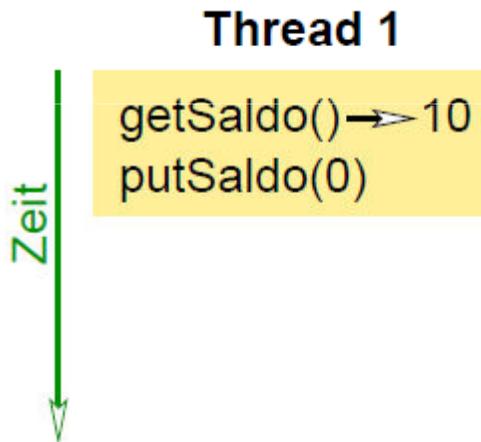
5.5 Threads ...

Beispiel: Bankkonto ...



- **Ausgabe:**

```
Kontostand: 0.0
Kontostand: -10.0 //Warum kommt hier zweimal
Kontostand: -10.0 //derselbe Kontostand??
```
- **Problem:** zeitliche Verzahnung in `abheben()`:





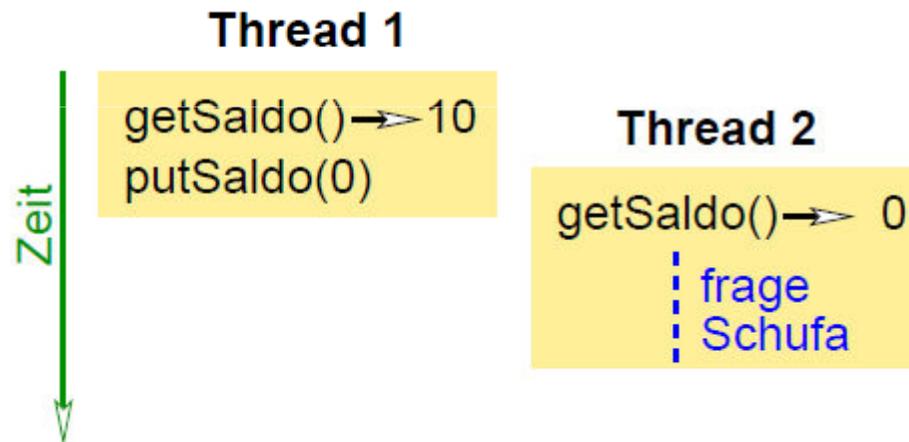
5.5 Threads ...

Beispiel: Bankkonto ...



- **Ausgabe:**

```
Kontostand: 0.0
Kontostand: -10.0 //Warum kommt hier zweimal
Kontostand: -10.0 //derselbe Kontostand??
```
- **Problem:** zeitliche Verzahnung in `abheben()`:





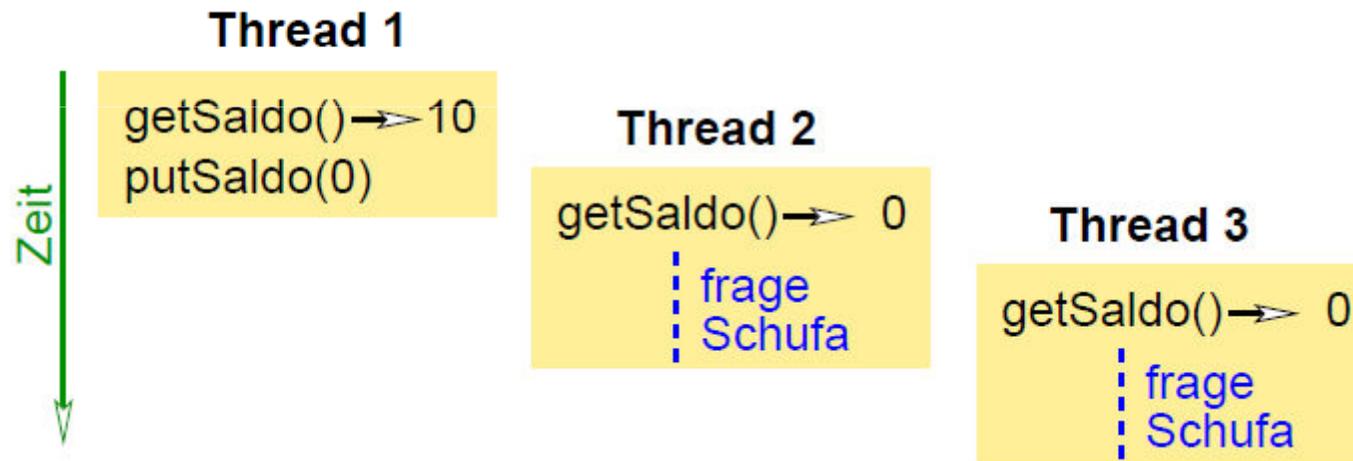
5.5 Threads ...

Beispiel: Bankkonto ...



- **Ausgabe:**

```
Kontostand: 0.0
Kontostand: -10.0 //Warum kommt hier zweimal
Kontostand: -10.0 //derselbe Kontostand??
```
- **Problem:** zeitliche Verzahnung in `abheben()`:





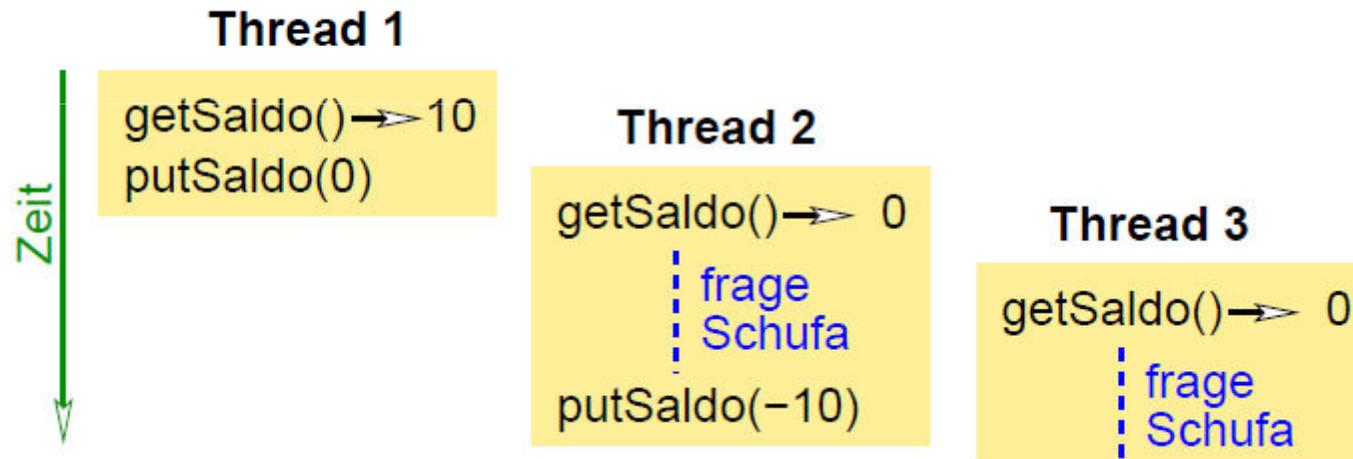
5.5 Threads ...

Beispiel: Bankkonto ...



- **Ausgabe:**

```
Kontostand: 0.0
Kontostand: -10.0 //Warum kommt hier zweimal
Kontostand: -10.0 //derselbe Kontostand??
```
- **Problem:** zeitliche Verzahnung in `abheben()`:





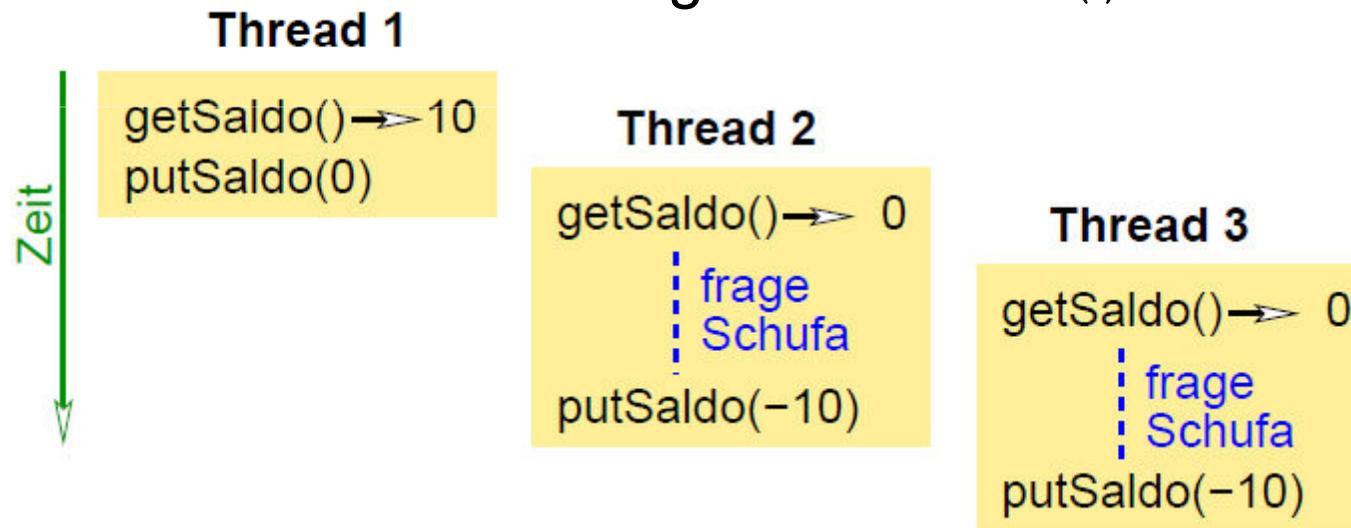
5.5 Threads ...

Beispiel: Bankkonto ...



- **Ausgabe:**

```
Kontostand: 0.0
Kontostand: -10.0 //Warum kommt hier zweimal
Kontostand: -10.0 //derselbe Kontostand??
```
- **Problem: zeitliche Verzahnung in abheben():**



- **Lösung: wechselseitiger Ausschluß für abheben() !**



5 Programmierung mit Java ...

5.6 Applets (+ WWW: `AppletBsp.java`, `AppletBsp.html`)

- Java **Applets** sind Java-Anwendungen, die innerhalb eines WWW-Browsers laufen
 - sie werden normalerweise in HTML-Seiten eingebunden
 - der Code der Klassen wird über das Netz vom WWW-Server geladen (keine lokale Installation erforderlich)
 - sie laufen in einer eingeschränkten Ausführungsumgebung
 - z.B. keine Zugriffe auf Dateisystem möglich
 - Sie erben von `java.applet.Applet` bzw. `javax.swing.JApplet`
 - i.d.R. mit Überschreiben der Methode `public void init()`
 - wird beim Laden des Applets aufgerufen
 - `JApplet` stellt ein Fenster bereit, das ähnlich aufgebaut ist wie `JFrame`, aber innerhalb des WWW-Browsers angezeigt wird
-