



hochschule mannheim

Fakultät für Informatik Programmierung 2 (PR2)

04 - Erweiterte objektorientierte Programmierung

Prof. Dr. Frank Dopatka



Hochschule Mannheim University of Applied Sciences



hochschule mannheim



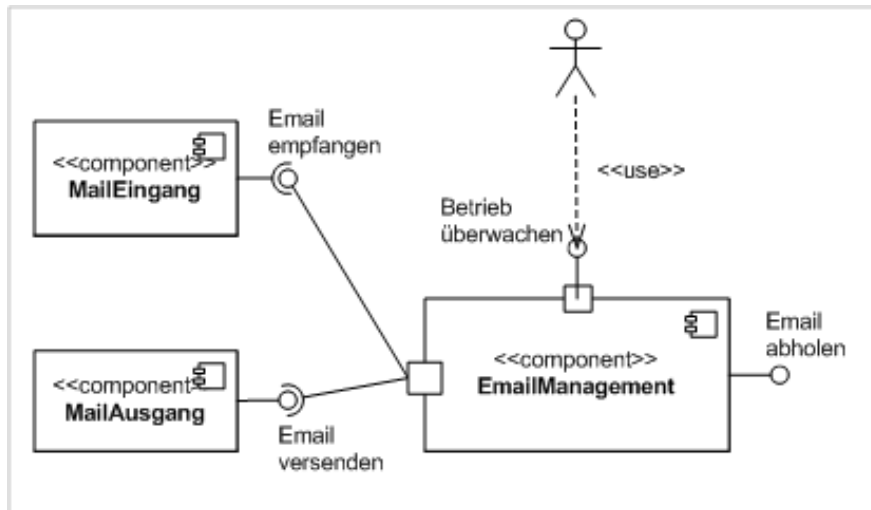
Interfaces



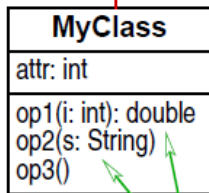
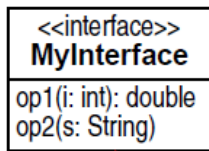
- Schnittstellen definieren „Dienstleistungen“ für aufrufende Klassen, sagen aber nichts über deren Implementierung aus:
 - Es handelt sich um funktionale Abstraktionen, die festlegen was implementiert werden soll, aber nicht wie.
 - Sie realisieren damit das Geheimnisprinzip in der stärksten Form.
- Der Java-Code einer Schnittstelle enthält nur die öffentlich sichtbaren Definitionen, nicht die Implementierung.
- Schnittstellen sind von ihrer Struktur & Verwendung her ähnlich zu abstrakten Klassen:
 - Sie können nicht instanziiert werden.
 - Referenzen auf Schnittstellen sind aber möglich. Die Referenzen können auf Objekte zeigen, welche die Schnittstelle implementieren.
 - Da mehrere Interfaces implementiert werden können, ist dadurch eine Art von Mehrfachvererbung realisierbar.



- Unterschiede zu abstrakten Klassen sind:
 - Alle Methoden sind abstrakt.
 - Es gibt keine Attribute.
 - Es gibt keine Konstruktoren.
 - Interfaces stehen neben der Vererbungshierarchie.
 - Von einem Interface erbt man nicht, man implementiert es
 - Eine Klasse kann beliebig viele Interfaces implementieren, so dass eine Art von Mehrfachvererbung möglich wird.



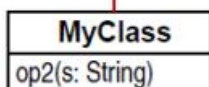
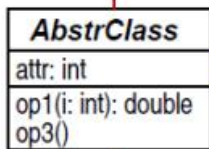
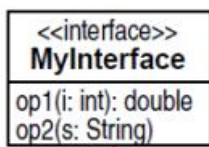
- Klassen können von Schnittstellen „erben“.
- „Vererbt“ werden nur die Signaturen der Operationen.
- Die Klassen müssen diese Operationen selbst implementieren.
- Man spricht in diesem Fall von einer Implementierungs-Beziehung statt von Vererbung.
- **implements** besagt, daß eine Klasse Methoden einer Schnittstelle implementiert.
- Die Klasse erbt die abstrakten Methoden-Definitionen, also deren Signaturen incl. Ergebnistyp.
- Diese müssen dann geeignet implementiert werden.
- Werden nicht alle Methoden implementiert, so bleibt die implementierende Klasse abstrakt.
- Die überschreibenden Methoden müssen öffentlich sein.



```
interface MyInterface {
    public double op1(int i);
    public void op2(String s);
}
```

```
class MyClass
    implements MyInterface {
    private int attr;
    public double op1(int i) { ... }
    public void op2(String s) { ... }
    public void op3() { ... }
}
```

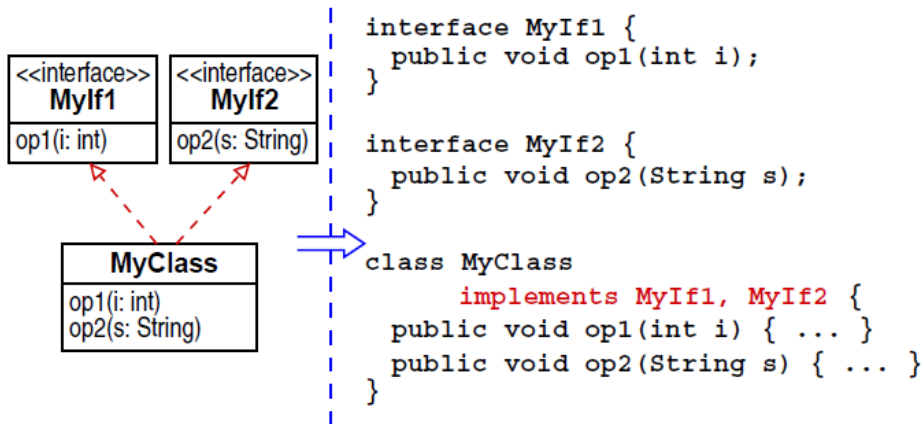
Überschreiben der ererbten (abstrakten) Operationen



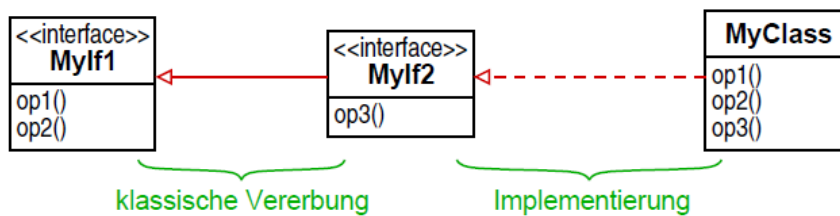
```
interface MyInterface {
    public double op1(int i);
    public void op2(String s);
}
```

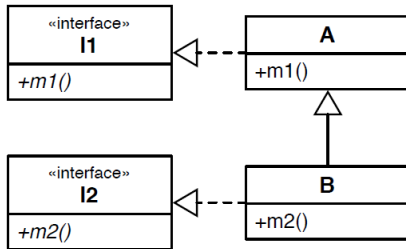
```
abstract class AbstrClass
    implements MyInterface {
    private int attr;
    public double op1(int i) { ... }
    public void op3() { ... }
}
```

```
class MyClass extends AbstrClass {
    public void op2(String s) { ... }
}
```



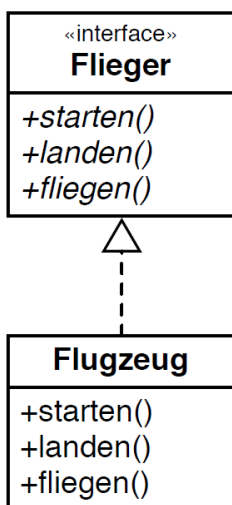
- Schnittstellen können von anderen Schnittstellen erben.
- Die Darstellung in UML und Java erfolgt wie bei der normalen Vererbung.
- Die implementierende Klasse muß die Methoden „ihrer“ Schnittstelle und aller Ober-Schnittstellen implementieren:





```

A a = new A();
B b = new B();
System.out.println(a instanceof I1); // --> true
System.out.println(a instanceof I2); // --> false
System.out.println(b instanceof A); // --> true
System.out.println(b instanceof I1); // --> true
System.out.println(b instanceof I2); // --> true
    
```

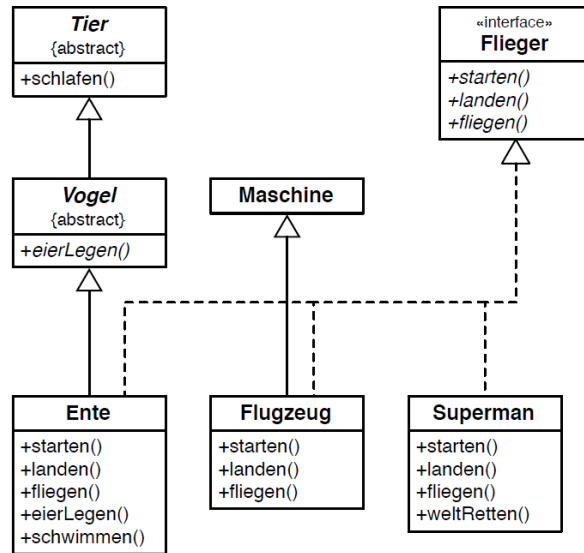


```

public interface Flieger {
    int CONSTANT = 12;
    void starten();
    void landen();
    void fliegen();
}
    
```

```

public class Superman implements Flieger {
    public void fliegen() { ... }
    public void landen() { ... }
    public void starten() { ... }
    public void weltRetten() { ... }
}
    
```



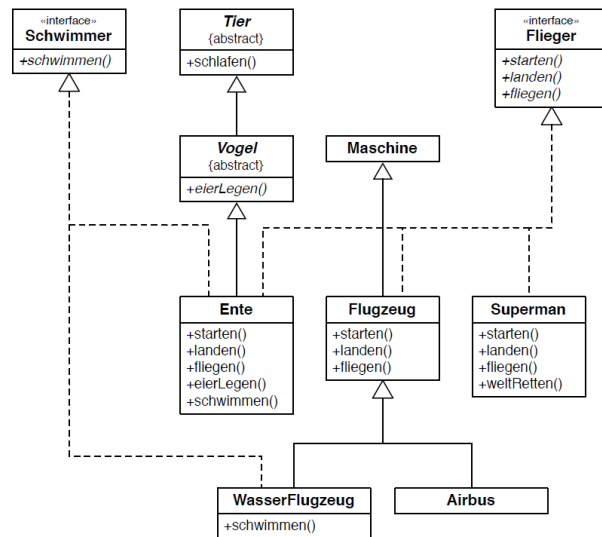
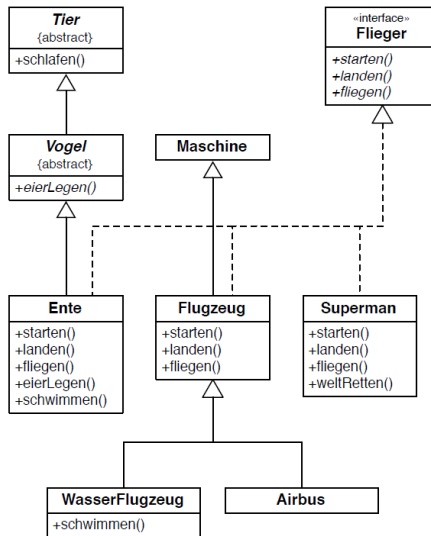
```

public abstract class Tier {
    public void schlafen() { ... }
}

public abstract class Vogel extends Tier {
    public abstract void eierLegen();
}

public class Ente extends Vogel implements Flieger {

    public void eierLegen() { ... }
    public void schwimmen() { ... }
    public void fliegen() { ... }
    public void landen() { ... }
    public void starten() { ... }
}
    
```

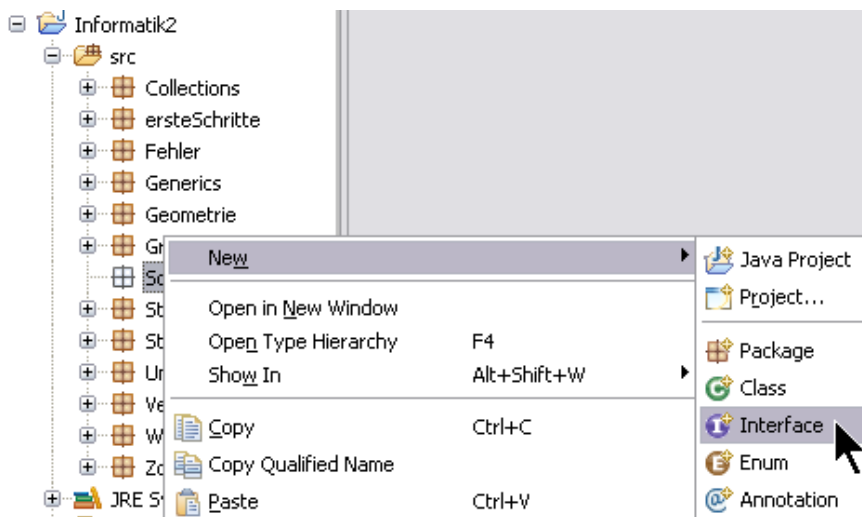



```
public class Ente extends Vogel
    implements Schwimmer, Flieger {

    public void eierLegen() { ... }
    public void schwimmen() { ... }
    public void fliegen() { ... }
    public void landen() { ... }
    public void starten() { ... }
}

public class Wasserflugzeug extends
    Flugzeug implements Schwimmer {

    public void schwimmen() { ... }
}
```




Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected

```
public interface iDrucken {  
    public void drucken(Drucker d);  
}
```

New Java Class

Java Class

 This package name is discouraged. By convention, package names usually start with a lowercase letter

Source folder:

Package:

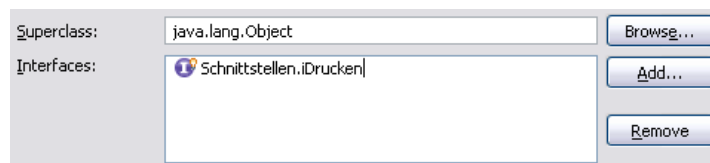
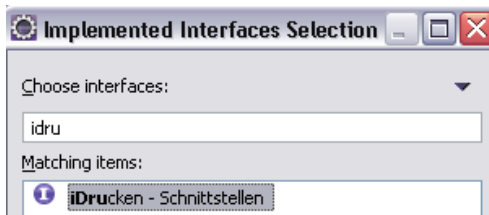
☐ Enclosing type:

Name:

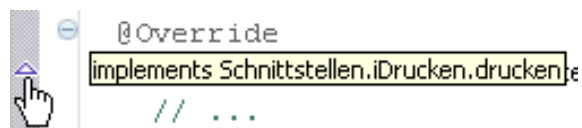
Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:



```
public class WordDatei implements iDrucken {
    @Override
    public void drucken(Drucker d) {
        // ...
    }
}
```





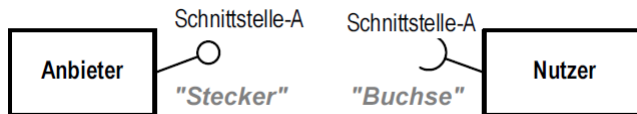
```
public class TestDrucken {  
    public static void main(String[] args) {  
        WordDatei doc=new WordDatei();  
        ExcelDatei xls=new ExcelDatei();  
        Drucker d1=new Drucker();  
  
        IDrucken d;  
        d=doc; d.drucken(d1);  
        d=xls; d.drucken(d1);  
    }  
}
```



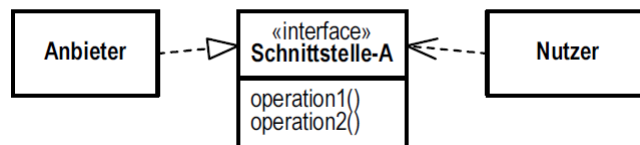
- Interfaces können Default-Methoden enthalten.
 - Dies sind Methoden mit Implementierung, die durch das Schlüsselwort **default** gekennzeichnet werden.
 - Diese Methoden werden an die Klasse weitergegeben, die das Interface implementiert.
 - Sie können von der Klasse mit eigener Implementierung überschrieben werden.

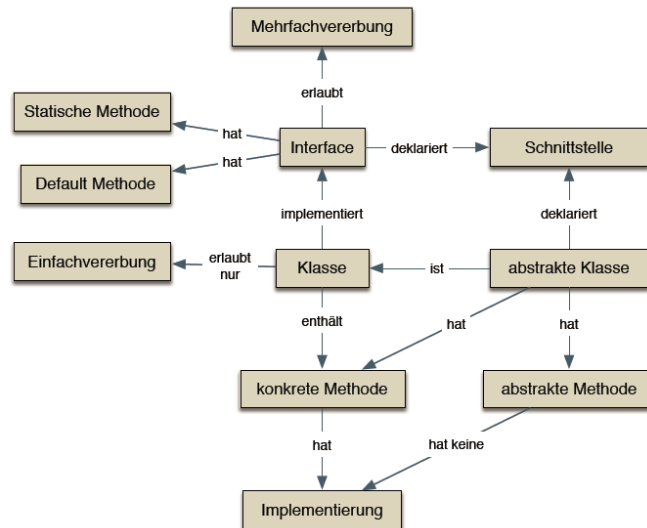


```
public interface Flieger {  
    public void starten();  
    public void landen();  
    public default void sinken() {  
        System.out.println("sinke");  
    }  
    public default void steigen() {  
        System.out.println("steige");  
    }  
}
```



angebotene Schnittstelle *benutzte Schnittstelle*

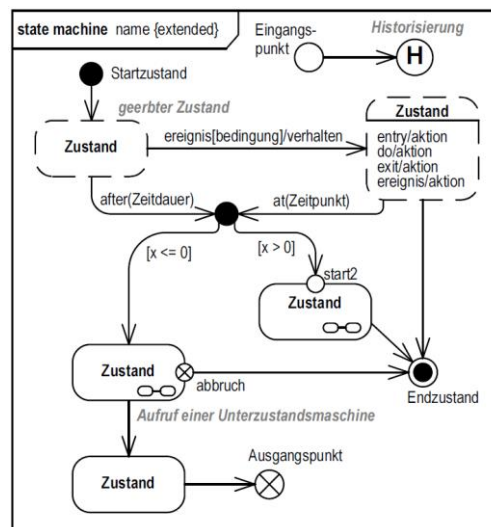


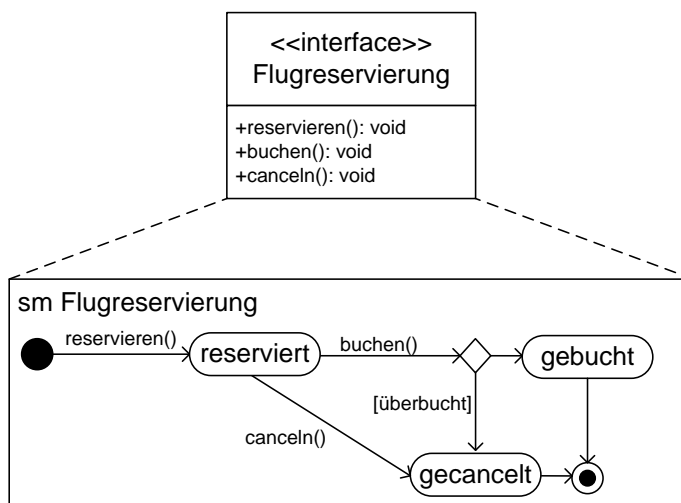
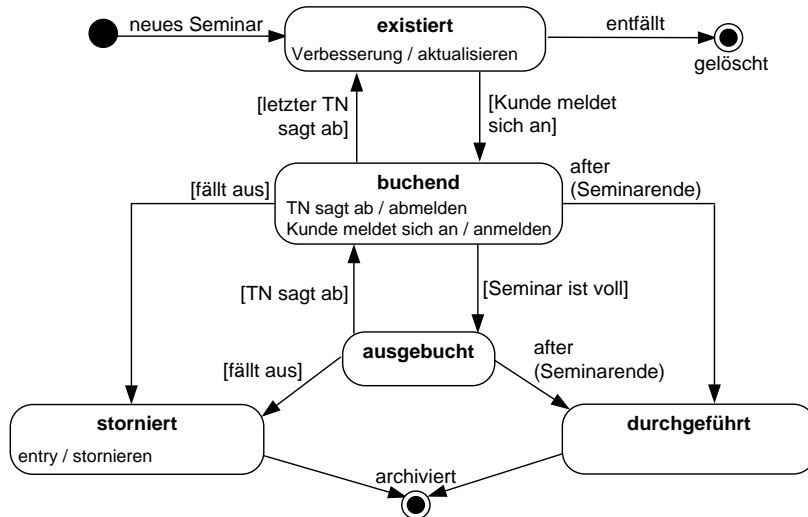


Zustandsdiagramme der UML



- Ein Zustandsdiagramm zeigt die zur Laufzeit erlaubten Zustände eines Zustandsautomaten - z. B. eines Objektes oder auch eines Systems - an und gibt Ereignisse an, die seine Zustandsübergänge auslösen.
- Damit beschreibt ein Zustandsdiagramm einen endlichen Automaten, der sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet.
- Die Zustände in einem Zustandsdiagramm werden durch Rechtecke mit abgerundeten Ecken dargestellt.
- Die Pfeile zwischen den Zuständen symbolisieren mögliche Zustandsübergänge.
- Sie sind mit den Ereignissen beschriftet, die zu dem jeweiligen Zustandsübergang führen.







```
public interface iFlugreservierung {  
    void reservieren();  
    void buchen();  
    void canceln();  
}  
  
public enum zFlugreservierung {  
    reserviert, gebucht, gecancelnt;  
}
```



```
public class TUIException extends RuntimeException {  
    private static final long serialVersionUID = 1L;  
  
    public TUIException(String s) {  
        super(s);  
    }  
  
    public TUIException(String dienst, zFlugreservierung z) {  
        // Gültigkeitsprüfung sinnvoll?  
        super(dienst+" im Zustand "+z+" nicht erlaubt!");  
    }  
}
```



```
import java.util.Calendar;

public class TUI implements iFlugreservierung {
    private zFlugreservierung zustand=null;

    public zFlugreservierung getZustand(){
        return zustand;
    }

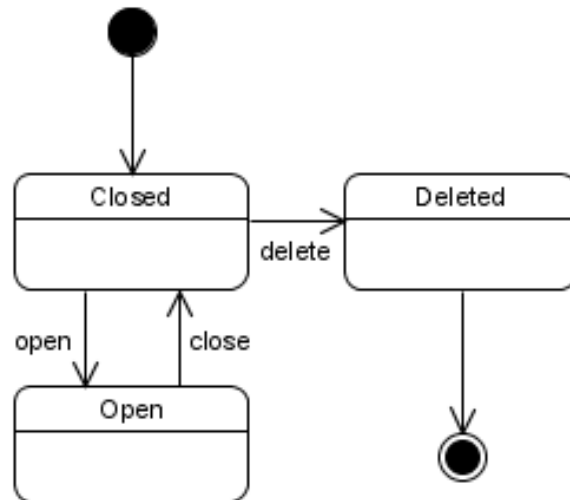
    private void setZustand(zFlugreservierung z) {
        this.zustand=z;
    }

    @Override
    public void reservieren() {
        if (zustand!=null){
            throw new TUIException("Reservieren",getZustand());
        }
        // reservieren...
        setZustand(zFlugreservierung.reserviert);
    }
}
```



```
@Override
public void buchen() throws Exception {
    if (zustand!=zFlugreservation.reserviert){
        throw new TUIException("Buchen",getZustand());
    }
    // buchen ist hier nur an geraden Tagen erfolgreich:
    Calendar cal=Calendar.getInstance();
    if (cal.get(Calendar.DAY_OF_MONTH)%2==0){
        // buchen...
        setZustand(zFlugreservation.gebucht);
    }
    else{
        setZustand(zFlugreservation.gecancelt);
    }
}

@Override
public void canceln() throws Exception {
    if (zustand!=zFlugreservation.reserviert){
        throw new TUIException("Canceln",getZustand());
    }
    // canceln...
    setZustand(zFlugreservation.gecancelt);
}
}
```



```
package StatesMitEnum;

public class File {
    private FileState state;

    public File() {
        state=FileState.closed;
    }
    public FileState getState() {
        return this.state;
    }
}
```

```
package StatesMitEnum;

public enum FileState {
    open, closed, deleted;
}
```



```
public void open() {
    if (state!=FileState.closed)
        throw new RuntimeException("Öffnen nicht erlaubt im aktuellen Zu
System.out.println("Öffne Datei");
    this.state=FileState.open;
}
public void close() {
    if (state!=FileState.open)
        throw new RuntimeException("Schließen nicht erlaubt im aktueller
System.out.println("Schließe Datei");
    this.state=FileState.closed;
}
public void delete() {
    if (state!=FileState.closed)
        throw new RuntimeException("Löschen nicht erlaubt im aktuellen ?
System.out.println("Lösche Datei");
    this.state=FileState.deleted;
}
}
```



```
package StatesMitEnum;

public class TestStateEnum {
    public static void main(String[] args) {
        File f=new File();
        System.out.println(f.getState());
        f.open();
        System.out.println(f.getState());
        f.open();
    }
}
```

- Die Realisierung eines Zustandsautomaten über eine Enum ist „Anfänger-Niveau“.
- Bei größeren Diagrammen ist die Verwendung des Entwurfsmusters „Zustand“ (state-pattern) dringend anzuraten!



Innere Klassen



- Die bisherigen Klassen waren entweder in Paketen organisiert oder in einer Datei.
- Diese Form von Klassen heißt „Top-Level-Klassen“.
- Es gibt darüber hinaus die Möglichkeit, eine Klasse in eine andere Klasse zu schachteln und die beiden Klassen damit noch enger aneinander zu binden.
- Eine Klasse, die so eingebunden wird, heißt „innere Klasse“. Dies sieht prinzipiell wie folgt aus:

```
class Außen {  
    class Innen {  
    }  
}
```
- Die Java-Sprache definiert vier Typen von inneren Klassen, die im Folgenden näher beschrieben werden.



- Die einfachste Variante einer inneren Klasse oder Schnittstelle wird wie eine statische Eigenschaft in die Klasse eingesetzt.
- Wegen der Schachtelung wird dieser Typ im Englischen „nested top-level class“ genannt.
- Die Namensgebung betont mit dem Begriff top-level, dass die Klassen das Gleiche können wie „normale“ Klassen oder Schnittstellen.
- Insbesondere sind keine Exemplare der äußeren Klasse nötig.
- Statische innere Klassen werden nur sehr selten verwendet.



```
public class StaticMember {  
  
    public static class InnereKlasse {  
        public String toString() {  
            return "Ich bin innen";  
        }  
    }  
  
    public String toString() {  
        return "Ich bin außen";  
    }  
}  
  
StaticMember aeussere = new StaticMember();  
StaticMember.InnereKlasse innere = new StaticMember.InnereKlasse();  
System.out.println(aeussere); // -> "Ich bin außen"  
System.out.println(innere); // -> "Ich bin innen"
```



1. Statische innere Klassen und Schnittstellen

- Die Eigenschaften der statischen inneren Klasse besitzen Zugriff auf alle statischen Eigenschaften der äußeren Klasse.
- Ein Zugriff auf Objektvariablen ist aus der statischen inneren Klasse nicht möglich, da sie als extra Klasse gezählt wird, die im gleichen Paket liegt.
- Die innere Klasse muss einen anderen Namen als die äußere haben.



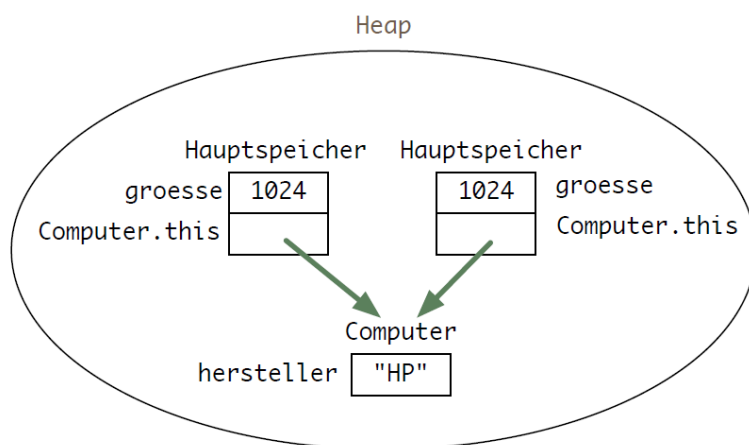
2. Mitglieds- oder Elementklassen

- Eine Mitgliedsklasse (engl. „member class“), auch Elementklasse genannt, ist ebenfalls vergleichbar mit einem Attribut, nur ist es nicht statisch.
- Die innere Klasse kann zusätzlich auf alle Attribute dieses Objektes der äußeren Klasse zugreifen.
- Dazu zählen auch die privaten Eigenschaften!
 - Diese Design-Entscheidung von Java ist sehr umstritten und kontrovers diskutiert!



```
public class Computer {  
  
    String hersteller = "HP";  
  
    class Hauptspeicher {  
        int groesse = 1024;  
    }  
}
```

```
Computer computer = new Computer();  
Computer.Hauptspeicher h1 = computer.new Hauptspeicher();  
Computer.Hauptspeicher h2 = computer.new Hauptspeicher();
```

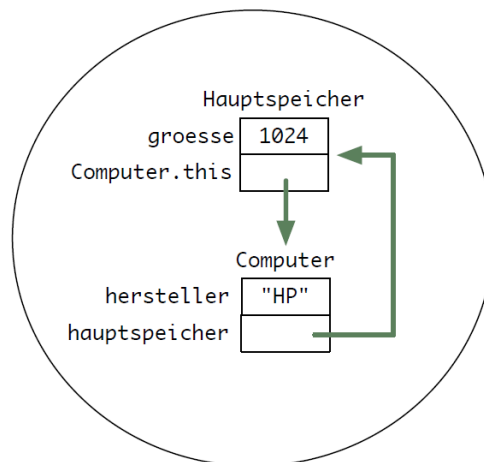




```
public class Computer {  
  
    String hersteller = "HP";  
    Hauptspeicher speicher = new Hauptspeicher();  
  
    class Hauptspeicher {  
        int groesse = 1024;  
  
        void printHersteller() {  
            System.out.println(  
                Computer.this.hersteller);  
        }  
    }  
}  
  
Computer computer = new Computer();
```



Heap





- Ein Exemplar der Hauptspeicher-Klasse hat Zugriff auf alle Eigenschaften der Computer-Klasse.
- Um innerhalb der äußeren Computer-Klasse ein Hauptspeicher-Exemplar zu erzeugen, muss ein Exemplar der Computer-Klasse existieren.
- Das ist eine wichtige Unterscheidung gegenüber den statischen inneren Klassen, denn statische innere Klassen existieren auch ohne Objekt der äußeren Klasse.
- Eine zweite wichtige Eigenschaft ist, dass innere Elementklassen selbst keine statischen Eigenschaften definieren dürfen.
- Elementklassen sind interessant zur Realisierung von Kompositionen und damit zur Realisierung einer „besteht aus“-Beziehung!



- Lokale Klassen sind auch innere Klassen, die jedoch nicht als Eigenschaft direkt in einer Klasse eingesetzt werden.
- Diese Form der inneren Klasse befindet sich in Anweisungsblöcken von Methoden oder Initialisierungsblöcken.
- Lokale Schnittstellen sind nicht möglich!
- Jede lokale Klasse kann auf Methoden der äußeren Klasse zugreifen und zusätzlich auf die lokalen Variablen und Parameter, die mit dem Modifizierer **final** als unveränderlich ausgezeichnet sind.
- Liegt die innere Klasse in einer statischen Funktion, kann sie jedoch keine Objektmethode aufrufen.
- Eine weitere Einschränkung im Vergleich zu den Elementklassen ist, dass die Modifizierer **public**, **protected**, **private** und **static** nicht erlaubt sind.

```
public class LocalBeispiel {  
  
    public static void main(String[] args) {  
  
        class LocalClass {  
            public String toString() {  
                return "Ich bin lokal";  
            }  
        }  
  
        LocalClass local = new LocalClass();  
        System.out.println(local.toString());  
    }  
}
```

Ich bin lokal

```
public class LocalMitVariable {  
  
    public static void main(String[] args) {  
  
        final String ausgabe = "Ich bin lokal";  
  
        class LocalClass {  
            public String toString() {  
                return ausgabe;  
            }  
        }  
  
        LocalClass local = new LocalClass();  
        System.out.println(local.toString());  
    }  
}
```

Ich bin lokal



```
public class LocalMitVariableUndInstanz {  
  
    private String variable = "Ich bin nicht lokal";  
  
    public void doIt() {  
        final String ausgabe = "Ich bin lokal";  
  
        class LocalClass {  
            public String toString() {  
                return ausgabe + ", " + variable;  
            }  
        }  
        LocalClass local = new LocalClass();  
        System.out.println(local.toString());  
    }  
}  
  
Ich bin lokal, Ich bin nicht lokal
```



- Anonyme Klassen gehen noch einen Schritt weiter als lokale Klassen.
- Sie haben keinen Namen und erzeugen immer automatisch ein Objekt.
- Klassendefinition und Objekterzeugung sind zu einem Sprachkonstrukt verbunden. Die allgemeine Notation ist folgende:

```
new KlasseOderSchnittstelle() {  
    // Eigenschaften und Methoden der inneren Klasse  
};
```

- Wenn hinter **new** der bekannte Klassenname A steht, dann ist die anonyme Klasse eine Unterklasse von A.
 - Die Verwendung von **extends** ist nicht erlaubt.
- Wenn hinter **new** der Name einer Schnittstelle S steht, dann erbt die anonyme Klasse von Object und implementiert die Schnittstelle S.
 - Die Verwendung von **implements** ist nicht erlaubt.



- Anonyme innere Klassen werden insbesondere dann verwendet, wenn man einmalig eine Methode einer existierenden Klasse neu definieren will.
- Dies ist u.a. der Fall bei
 - der Reaktion auf einen Button-Klick durch das Implementieren der Methode `actionPerformed` sowie
 - der Implementierung eines neuen Threads durch das Überschreiben der Methode `run`.



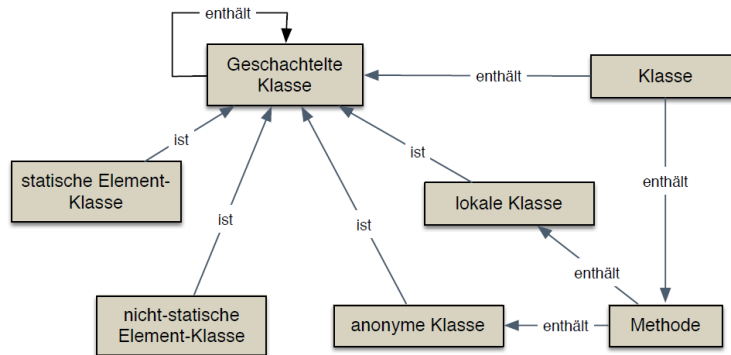
- Im folgenden Beispiel wird die `toString`-Methode eines einzelnen Objektes der vordefinierten AWT-Klasse `Point` überschrieben...

```
import java.awt.Point;

public class BesondererPunkt {

    public static void main(String[] args) {
        Point p=new Point(17,4){
            private static final long serialVersionUID = 1L;
            @Override
            public String toString(){
                return x+"/"+y;
            }
        };
        System.out.println(p);
    }
}
```

<terminated> BesondererPunkt [Java Application]
17/4



GUI mit Swing



- GUI steht als Abkürzung für Graphical User Interface, zu Deutsch grafische Benutzerschnittstelle oder grafische Bedienoberfläche.
- Im Vordergrund steht dabei die Software-Ergonomie:
 - Ziel ist es, dass die Software an die Bedürfnisse des Benutzers angepasst sein soll.
- Eine gute GUI-Programmierung setzt auch Modularität voraus:
 - Das Einbinden neuer Funktionen soll leicht machbar sein.
 - Wie wirkt sich eine „kleine“ Änderungen der GUI bei 1000 Bildschirmseiten aus?
- Wegen einer neuen Datenbank darf nicht die GUI geändert werden:
 - Ein Schichtenmodell ist erforderlich!



- Aufgaben der Schichten:
 - GUI:
 - Dialogführung, Datendarstellung
 - Fachkonzept:
 - modelliert den funktionalen Kern der Software
 - Datenhaltungsschicht:
 - realisiert die Datenspeicherung, z.B. mittels Datenbank und/oder Dateien
- Ziel ist die Unabhängigkeit und Trennung der Schichten, um eine austauschbare, wiederverwendbare Software zu erzeugen.
- Nachteil ist der zusätzliche Aufwand für die Schnittstellenspezifikation.

Bedienoberfläche (GUI)

Kontrolle der Zugriffe

Fachkonzept

Datenaustausch

Datenbank

- Das Abstract Windowing Toolkit AWT wurde mit Java 1.0 eingeführt und hat mit Version 1.1 starke Änderungen erfahren.
- Der Paketname lautet `java.awt`.
- Zentrale Idee des AWT ist die Plattformunabhängigkeit:
 - Ein Toolkit, das auf den Gemeinsamkeiten aller Betriebssystemen basiert, für die eine JRE angeboten wird.
 - Jede Komponente des AWT wird auf eine Komponente der Plattform abgebildet, auf der die JRE läuft.
 - Portierte Programme sehen auf jedem Rechner einheitlich aus.

...bietet folgende Funktionen:

- Zeichnen grafischer Grundelemente wie Linien und Polygone
 - Layoutanordnungen von Elementen
 - Setzen von Farben und Zeichensätzen
 - Behandlung von Ereignissen und Mausbewegung
-
- Die AWT wird heute als Grundlage der Swing-Klassen genutzt.

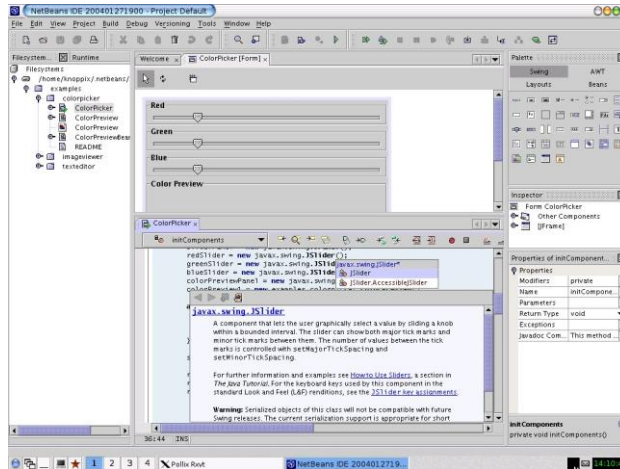


- Die Java Foundation Classes JFC alias Swing wurde mit Java 1.2 als Erweiterung des AWT eingeführt.
- Der Paketname lautet `javax.swing`.
- Die GUI-Komponenten bieten neue grafische Elemente, die vollständig in Java implementiert sind, aber zum Teil auch auf AWT-Komponenten basieren.
- Sie benötigen mehr Ressourcen als AWT-Komponenten, da alle Komponenten aus primitiven Zeichenelementen erzeugt werden.
- Eine direkte Nutzung der Betriebssystem-GUI findet nicht statt.



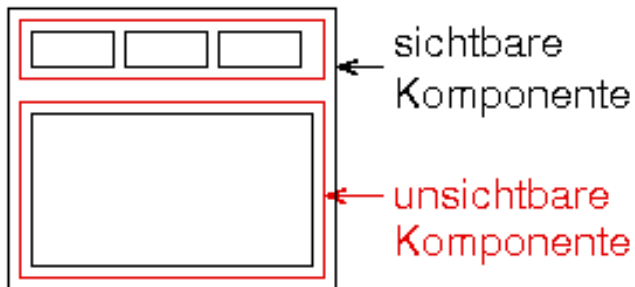
- Swing hat mehr Komponenten als AWT.
- Schaltflächen und Labels in Swing nehmen Symbole auf, die um Text angeordnet werden können.
- Swing-Komponenten können beliebig geformt sein; so kann eine Schaltfläche wie unter MacOS abgerundet sein.
- Jede Swing-Komponente kann einen Rahmen bekommen.

- Für größere Projekte mit GUIs werden in der Regel graphische Editoren wie Netbeans verwendet:

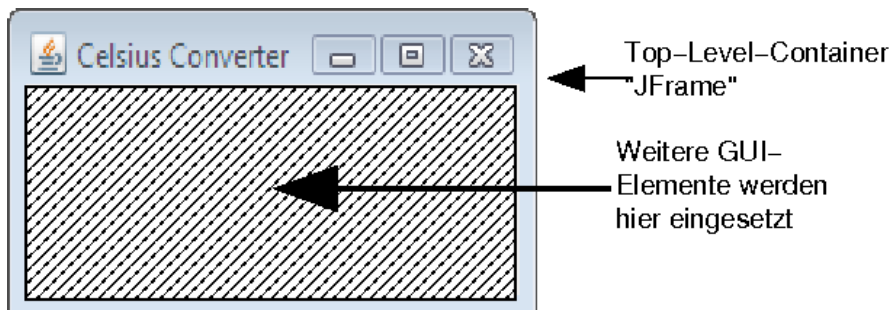


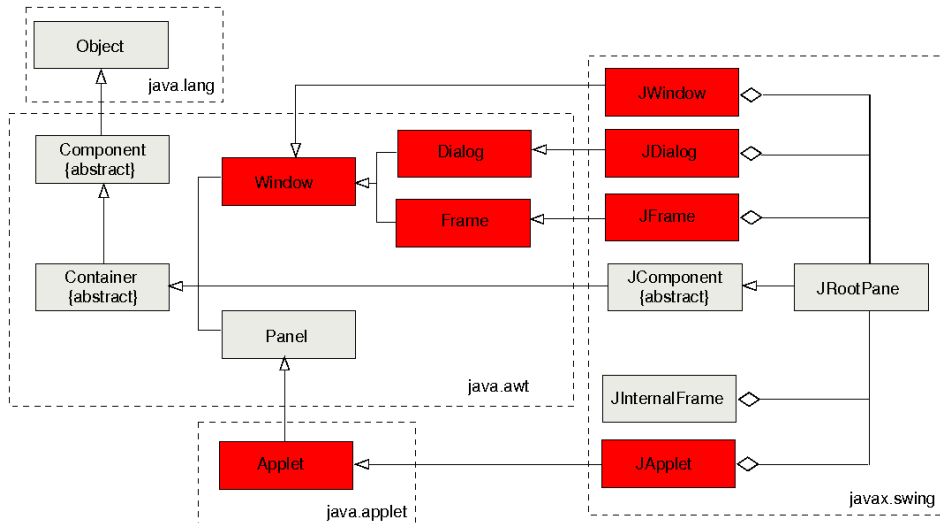
- Persönlich halte ich die GUI-Builder für Java für weitaus weniger ausgereift als die GUI-Builder des .NET-Frameworks.
- Ausserdem sind sie sehr ressourcen-intensiv.
- Es gibt jedoch wichtige Vorteile von Java-GUIs und deren Editoren:
 - Plattformunabhängigkeit
 - Der open-source Gedanke
- Hier wird auf GUI-Builder verzichtet!
- Sie erlernen die direkte textuelle Programmierung und damit den „Hintergrund“ jeder Java-GUI...

- Java-GUIs nutzen kein monolithisches Design, sondern werden aus einzelnen Komponenten zusammengesetzt.
- Bestandteile können einfache grafische Elemente (Linie, Kreis, Text) sein, aber auch komplexere Elemente wie z. B. Boxen, die andere Elemente enthalten.
- Diese Komponenten können sichtbar oder unsichtbar sein.



- Der Top-Level-Container ist Ausgangspunkt jeder grafischen Anwendung.
- In diesen Container werden dann Subcontainer und grafische Komponenten eingefügt.





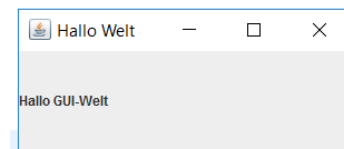
```

import javax.swing.JFrame;
import javax.swing.JLabel;

public class ErsteGUI extends JFrame{
    private static final long serialVersionUID = 1L;

    public ErsteGUI(String titel) {
        super(titel);
        JLabel jl=new JLabel("Hallo GUI-Welt");
        this.add(jl);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new ErsteGUI("Hallo Welt");
    }
}
  
```



- `JLabel jl = new JLabel("Hallo GUI-Welt");`
erzeugt ein neues Label und weist ihm den Text zu.
- `this.add(jl);`
fügt dem Hauptfenster das Label hinzu.
- `pack();`
berechnet die Größe der Anzeige und ordnet die Subcontainer entsprechend dem Layout an.
- Anstelle von `pack()` kann mit den Methoden des Top-Level-Containers `setSize(int width, int height)` die Größe des Fensters manuell festgelegt werden.

- `this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
Damit kann mit folgenden Konstanten als Parameter das Verhalten des Fensters beim Schließen geändert werden:
- `JFrame.DO_NOTHING_ON_CLOSE`
keine Defaultoperation vornehmen
- `JFrame.HIDE_ON_CLOSE`
verstecken des Containers, ohne ihn zu beenden; dies ist der Default-Wert
- `JFrame.EXIT_ON_CLOSE`
sofortiges Beenden
- `JFrame.DISPOSE_ON_CLOSE`
ruft die `dispose()`-Methode auf, die zu überschreiben ist...



```
@Override  
public void dispose(){  
    System.out.println("aufräumen...");  
    System.exit(0);  
}
```

- **this.setVisible(true);**

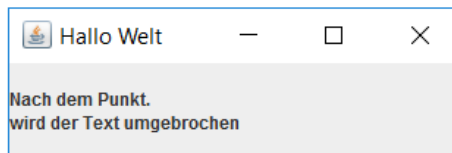
Ein JFrame ist standardmäßig unsichtbar. Mit und dem Wahrheitswert **true** als Parameter wird das Fenster sichtbar, mit **false** wird es unsichtbar.



- Die im Folgenden vorgestellten Methoden der GUI-Komponenten können alle der Java-DOC entnommen werden.
- Jedes einzelne Steuerelement ist eine Klasse mit einer Vielzahl an Methoden.
- Diese Methoden kann niemand auswendig wissen.
- Einige wichtige Methoden werden Sie mit der Zeit automatisch auswendig lernen, wenn Sie sie oft verwenden.
- In Tests und Klausuren wird Ihnen daher immer die JavaDoc zu allen Klassen ausgegeben, so dass Sie Einzelheiten nachlesen können.
- Den Umgang mit einer gegebenen JavaDoc sollten Sie ab jetzt beherrschen!
- Beim Beispiel der ersten Swing-GUI werden die wichtigsten Methoden des JLabels letztmalig vorgestellt...

- Das JLabel stellt einen Anzeigebereich bereit, der Text und/oder ein Bild darstellen kann.
- Es ist eine reine Anzeigekomponente; der angezeigte Inhalt kann vom Endbenutzer nicht verändert werden.
- Text kann mittels Hypertext Markup Language HTML und Cascading Style Sheets CSS formatiert werden, z.B.:

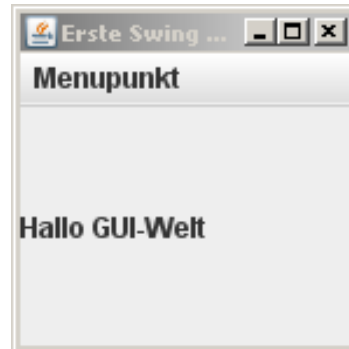
```
JLabel jl=new JLabel("<html>Nach dem Punkt.<br/>" +  
    "wird der Text umgebrochen</html>");
```



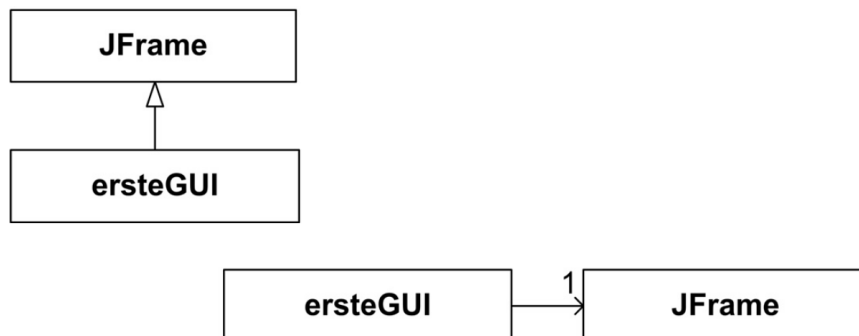
- Konstruktoren:
 - **JLabel ()**
JLabel mit leerem String als Inhalt
 - **JLabel (Icon image)**
JLabel mit Bild als Inhalt
 - **JLabel (String text)**
JLabel mit Text als Inhalt
 - **JLabel (String text, Icon icon, int horizontalAlignment)**
JLabel mit Text und Bild und horizontaler Ausrichtung
- Methoden:
 - **void setText (String text)**
einzeiliger Text in JLabel setzen
 - **void setIcon (Icon icon)**
Bild in JLabel setzen



- JFrame besitzt einen optionalen Menü-Bereich und einen Bereich für den Inhalt.
- Weiterhin kann das Fenster überall auf dem Desktop platziert werden und stellt neben der Titelleiste auch Fenster-Manager-Funktionen für
 - Minimierung,
 - Maximierung,
 - Wiederherstellung und
 - Schließenzur Verfügung.

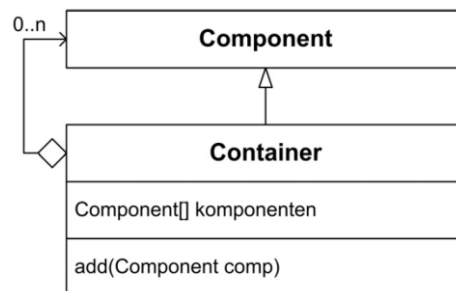


- Im Beispiel der ersten Swing-GUI ist die eigene Klasse ein JFrame.
- Ein alternativer Ansatz besteht darin, dass Ihre GUI ein Objekt der Klasse JFrame über eine Assoziation kennt und steuert.





- Die Grundidee der Benutzeroberflächen in Java ist das Prinzip der Container.
- In einem Container einer Java-GUI kann man ebenso andere Container und/oder Steuerelemente platzieren. Diese werden dann auf dem Bildschirm dargestellt.
- Die Methode zum Hinzufügen eines Containers und/oder eines Steuerelementes lautet `add(Component comp)`.
- Sowohl die Container, als auch die Steuerelemente sind Komponenten der Benutzeroberfläche.
- Das Konzept besteht darin, dass sowohl die Container, als auch die Steuerelemente von der Klasse `Component` vererbt sind.
- Die Vererbung ist stets eine „ist ein“-Beziehung zwischen einer allgemeineren Oberklasse (hier: `Component`) und einer spezielleren Unterklasse (hier: `Container`).

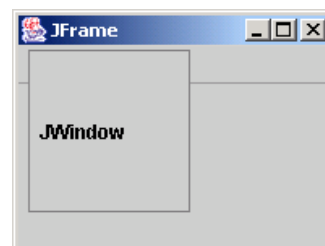




- Da Container aus Containern bestehen und demnach wie eine Zwiebel geschachtelt sind, stellt sich die Frage nach der äußeren Grenze der Schachtelung.
- Eine GUI endet dort, wo der Rahmen des Fensters endet.
- Java hat an dieser Stelle den Begriff des Top-Level Containers eingeführt.
- Bei Objekten dieser Top-Level Container handelt es sich genau um diese äußeren Grenzen. Java kennt 4 Top-Level Container:
 - **JFrame**
 - **JDialog**
 - **JWindow**
 - **JApplet**



- Ein **JDialog** ist ein Dialogfenster, welches einen Titel, einen Rahmen und eine Schaltfläche zum Schließen des Dialogs besitzt.
- Ein Dialog besitzt besondere die Eigenschaft, dass er von einem Hauptfenster – meist von einem **JFrame** – abhängig sein kann.
- Die Abhängigkeit besteht darin, dass man zuerst den Dialog schließen muß, bevor man das Hauptfenster weiter bedienen kann.
- Ein **JWindow** ist ein Fenster ohne Rahmen, welches überall auf dem Desktop platziert werden kann.





- Ein **JApplet** hat ebenfalls keinen Rahmen.
- Es besitzt im Gegensatz zu den anderen Top-Level Containern die besondere Eigenschaft, von einem HTML-Code aus aufrufbar zu sein.
- Damit kann ein JApplet auch in einem Internet Browser ausgeführt werden, sofern dieser ein JRE-PlugIn besitzt.
- Das JApplet wird dann über den Internet Browser automatisch von einem Server nachgeladen und in der VM des Clients mit dem Internet-Browser ausgeführt.



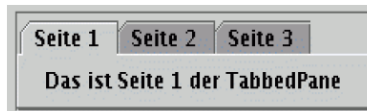
- Die Hauptaufgaben der folgenden Container sind nur das
 - Gruppieren und
 - Anordnender untergeordneten Elemente.
- **JPanel**
Sie werden standardmäßig nicht dargestellt. Die Layouteinstellungen dieser unsichtbaren Elemente bestimmen jedoch, auf welche Weise untergeordnete Elemente angeordnet werden.



- **JSplitPane**
Ermöglicht es, zwei untergeordnete Elemente neben- oder untereinander darzustellen.



- **JTabbedPane**
Ermöglicht die Anordnung von individuellen Elementen mit Registerkarten.



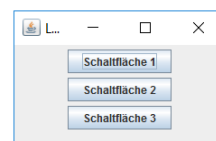
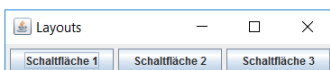
- **JToolBar**
Definiert eine Anordnung von JButtons als Icons mit optionalen Trennern. Diese Anordnung ist andockbar und auf Wunsch frei beweglich.



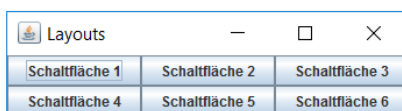
- Sie ordnen die untergeordneten Elemente des Containers nach bestimmten Koordinaten / Layout-Definitionen an.
- Die Zuweisung eines Layout-Managers soll vor dem Zuweisen der Elemente erfolgen, da sonst zusätzlicher Overhead entsteht.
- Zuweisung erfolgt bei der Container-Instanz mit **setLayout(LayoutManager mgr)** oder als Parameter des Container-Konstruktors.

- Im **FlowLayout** werden die einzelnen Elemente hintereinander angeordnet und nur in eine neue Zeile umgebrochen, wenn die maximale Breite des Containers erreicht ist.
- Es ist das Standardlayout aller Container:
 - Nur wenn das Layout eines Containers zuvor geändert wurde, muss mit den Konstruktoren und mit **setLayout()** das FlowLayout wieder eingerichtet werden.

```
private JFrame jf;
private JButton b1 = new JButton("Schaltfläche 1");
private JButton b2 = new JButton("Schaltfläche 2");
private JButton b3 = new JButton("Schaltfläche 3");
public LayoutGUI(String titel) {
    jf=new JFrame(titel);
    JPanel jp=new JPanel();
    jp.setLayout(new FlowLayout());
    jp.add(b1); jp.add(b2); jp.add(b3);
    jf.setContentPane(jp);
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jf.pack(); jf.setVisible(true);
}
```

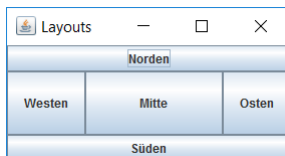


```
private JFrame jf;
private JButton b1 = new JButton("Schaltfläche 1");
private JButton b2 = new JButton("Schaltfläche 2");
private JButton b3 = new JButton("Schaltfläche 3");
private JButton b4 = new JButton("Schaltfläche 4");
private JButton b5 = new JButton("Schaltfläche 5");
private JButton b6 = new JButton("Schaltfläche 6");
public LayoutGUI(String titel) {
    jf=new JFrame(titel);
    JPanel jp=new JPanel();
    jp.setLayout(new GridLayout(2,3));
    jp.add(b1); jp.add(b2); jp.add(b3);
    jp.add(b4); jp.add(b5); jp.add(b6);
    jf.setContentPane(jp);
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jf.pack(); jf.setVisible(true);
}
```



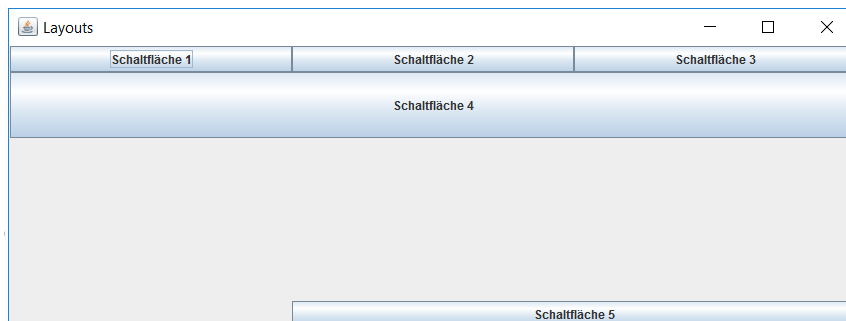
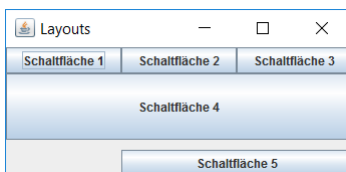


```
private JFrame jf;  
private JButton b1=new JButton("Norden");  
private JButton b2=new JButton("Süden");  
private JButton b3=new JButton("Osten");  
private JButton b4=new JButton("Westen");  
private JButton b5=new JButton("Mitte");  
public LayoutGUI(String titel) {  
    jf=new JFrame(titel);  
    JPanel jp=new JPanel();  
    jp.setLayout(new BorderLayout());  
    jp.add(b1,BorderLayout.NORTH);  
    jp.add(b2,BorderLayout.SOUTH);  
    jp.add(b3,BorderLayout.EAST);  
    jp.add(b4,BorderLayout.WEST);  
    jp.add(b5,BorderLayout.CENTER);  
    jf.setContentPane(jp);  
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    jf.pack(); jf.setVisible(true);  
}
```





```
private JFrame jf;  
private JButton b1=new JButton("Schaltfläche 1");  
private JButton b2=new JButton("Schaltfläche 2");  
private JButton b3=new JButton("Schaltfläche 3");  
private JButton b4=new JButton("Schaltfläche 4");  
private JButton b5=new JButton("Schaltfläche 5");  
public LayoutGUI(String titel) {  
    jf=new JFrame(titel);  
    JPanel jp=new JPanel();  
    jp.setLayout(new GridBagLayout());  
    GridBagConstraints c=new GridBagConstraints();  
    c.fill=GridBagConstraints.HORIZONTAL;  
    c.gridx=0; c.gridy=0; c.weightx=0.5; jp.add(b1,c);  
    c.gridx=1; c.gridy=0; jp.add(b2,c);  
    c.gridx=2; c.gridy=0; jp.add(b3,c);  
    c.gridx=0; c.gridy=1; c.gridwidth=3; c.ipady=40; jp.add(b4,c);  
    c.gridx=1; c.gridy=2; c.gridwidth=2; c.weighty=1.0; c.ipady=0;  
    c.insets=new Insets(10,0,0,0);  
    c.anchor=GridBagConstraints.PAGE_END;  
    jp.add(b5,c);  
    jf.setContentPane(jp);  
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    jf.pack(); jf.setVisible(true);  
}
```





- **JButton**

- Dies sind Schaltflächen, die mit Text und/oder Icons angezeigt werden können.
- JButton ist abgeleitet von AbstractButton.
- Über diese Vererbung erhält ein JButton-Objekt unter Anderem die folgenden Methoden:
 - **getText ()** / **setText ()**
zum Textzugriff auf den Button.
 - **getIcon ()** / **setIcon ()**
zum Zugriff auf das Bild des Buttons.
 - **addActionListener ()**
um auf das Drücken zu reagieren.
 - **doClick ()**
um das Drücken vom Programm aus auszulösen.



- **JButton**

- Das Drücken einer Schaltfläche löst ein Ereignis aus, das behandelt werden muss.
- Die Ereignisbehandlung in Java stellt ein eigenes Kapitel dar!

Einfacher Text





- **JScrollPane**

Verleiht der enthaltenen Komponente die Fähigkeit zu scrollen.



- **JCheckBox**

Erlaubt das An- oder Abwählen von Optionen.



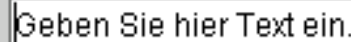
- **JRadioButton**

Erlaubt das Anwählen eines Elementes innerhalb einer definierten Gruppe von mehreren Elementen.



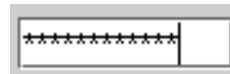
- **JTextField**

Stellt dem Benutzer ein Feld zur Texteingabe zur Verfügung.



- **JPasswordField**

Ableitung von JTextField, bei der anstelle der eingegeben Zeichen des Benutzers sogenannte Echo-Zeichen (Standard: *) angezeigt werden.



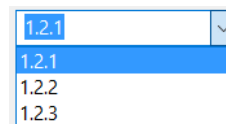
- **JFormattedTextField**

Auch eine Ableitung von JTextField. Es bietet die Möglichkeit, die Eingabe des Benutzers anhand eines Musters / vorgegebenen Formats zu validieren.

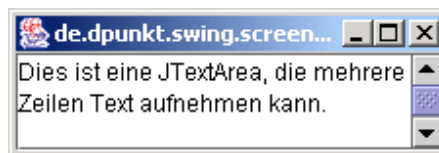


- `TextField`, `PasswordField`, `FormattedTextField` werden von der abstrakten Klasse `TextComponent` abgeleitet.
- Damit werden unter Anderem folgende Methoden zur Textein- und Textausgabe bei den oben genannten Komponenten bereitgestellt:
 - **`String getText()`**
Liefert den Inhalt des Textfelds.
 - **`String getText(int offs, int len)`**
Liefert den Inhalt des Textfelds von `offs` bis `offs + len`. Stimmen die Bereiche nicht, wird eine `BadLocationException` ausgelöst.
 - **`String getSelectedText()`**
Liefert den selektierten Text. Keine Selektion ergibt die Rückgabe `null`.
 - **`void setText(String t)`**
Setzt den Text neu.

- **`JComboBox`**
Stellt ein Drop-Down-Menü dar, mit dem mehrere Elemente platzsparend dem Benutzer zur Auswahl präsentiert werden können.
In dem Editier-Modus kann der Benutzer eigene Auswahlmöglichkeiten hinzufügen.

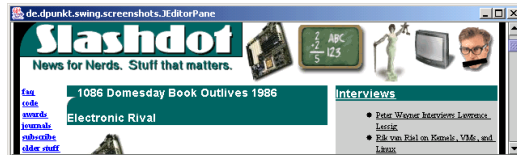


- **`JTextArea`**
 - Feld zur Anzeige von mehrzeiligem Text.
 - Der Text ist nicht speziell formatierbar.
 - Das Feld kann so programmiert werden, dass der Text vom Benutzer modifiziert werden kann.



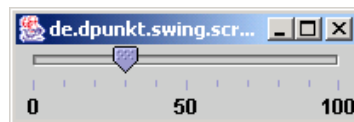
- **JEditorPane**

Textverarbeitungskomponente, die es ermöglicht, RTF- und HTML-formatierten Text anzuzeigen und zu bearbeiten.



- **JSlider**

Mit Hilfe eines JSlider können numerische Werte zwischen einem Minimum und Maximumwert über einen Schieber ausgewählt werden. Benutzer-eigene Angaben sind nicht möglich.



- **JSpinner**

Eine Komponente, die neben einem Eingabefeld zwei Buttons zum Erhöhen bzw. Erniedrigen des eingegebenen Wertes erlaubt. JSpinner unterstützt von sich aus drei Models, die dabei die passenden Editoren mit setzt:

- **SpinnerNumberModel** für Zahlen,
- **SpinnerDateModel** für Datumsangaben
- **SpinnerListModel** für eine Liste beliebiger Daten.

Die Editorkomponente ist in diesen Fällen ein JTextField, die mit dem entsprechenden Wert gefüllt wird.

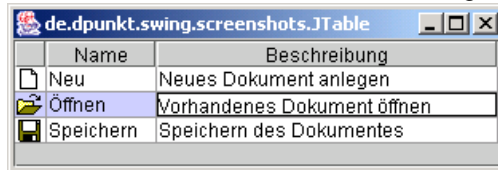


- **JList**

Eine JList zeigt mehrere Zeilen mit Daten gleichzeitig an, wobei kein, ein oder mehrere Elemente selektiert sein können.

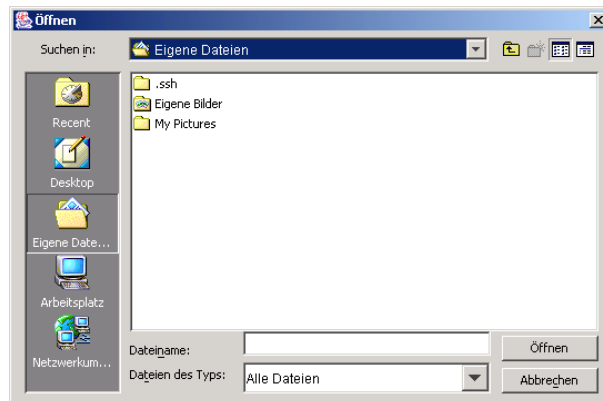
- **JTable**

Eine JTable stellt eine Tabelle in Swing dar.



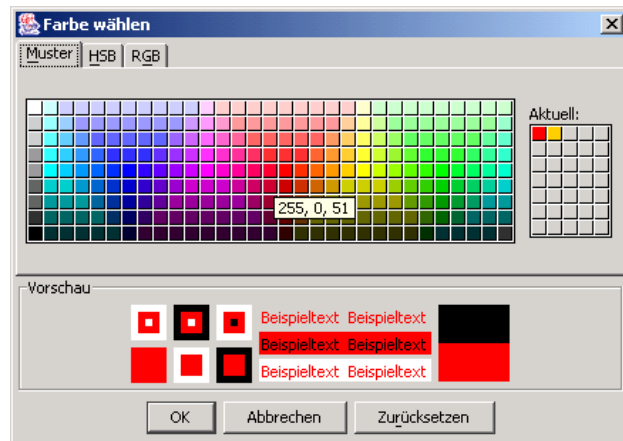
- **JFileChooser**

Ein vorgefertigter Dialog zur Auswahl einer Datei für einen Lese- oder Schreibvorgang.



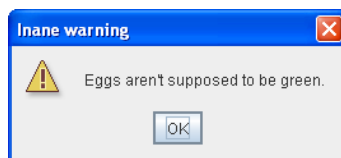
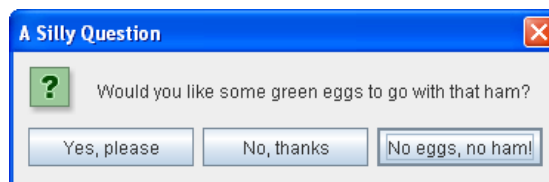
- **JColorChooser**

Ein vorgefertigter Dialog zur Auswahl einer Farbe.



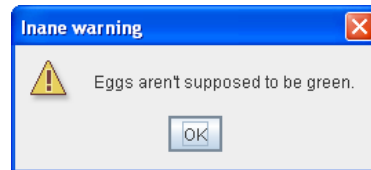
- **JOptionPane**

- Ein Dialog zur Anzeige kurzer Nachrichten. Unterstützt Icons, die Angabe eines Titels und Textes und die Modifikation des Button-Textes durch entsprechende Methoden.





- **JOptionPane**
- Ein Dialog zur Anzeige kurzer Nachrichten. Unterstützt Icons, die Angabe eines Titels und Textes und die Modifikation des Button-Textes durch entsprechende Methoden.



```
JOptionPane.showMessageDialog(  
    frame,  
    "Eggs aren't supposed to be green.",  
    "Inane warning",  
    JOptionPane.WARNING_MESSAGE  
);
```



- Ein Event bezeichnet ein Ereignis, das vom Benutzer einer GUI ausgelöst wird, indem dieser z. B. auf einen Button mit der Maus klickt, den Mauszeiger auf einen Button führt oder eine Taste drückt.
- Ein Event wird vom Betriebssystem an die zugehörige Anwendung weitergeleitet, die dieses Event nach einem bestimmten Modell verarbeitet.
- Dazu werden innerhalb der Anwendung oft „Callback“-Funktionen aufgerufen.
- Eine Rückruffunktion bezeichnet in der Informatik eine Funktion, die einer anderen Funktion als Parameter übergeben wird, und von dieser unter gewissen Bedingungen aufgerufen wird.



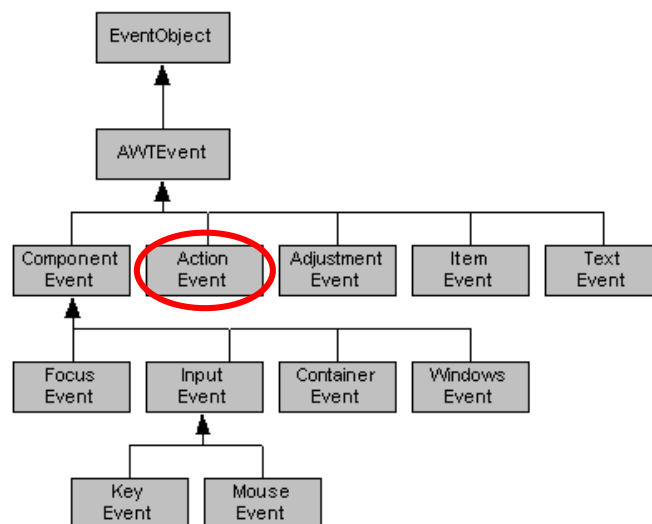
- In Java wird das Delegationsmodell verwendet, das auf dem Beobachter-Entwurfsmuster basiert:
 - Beobachter = Event-Listener
 - Subjekt = Event-Source
- Event-Listener registrieren sich bei den Event-Source, an deren Ereignisse sie interessiert sind...



- Das Auftreten eines Ereignisses in einer Event-Source stellt dann die Zustandsänderung dar, die die Benachrichtigung auslöst:



- Ereignisse werden als Objekte modelliert.
- Es existiert eine Klassenhierarchie für die verschiedenen Ereignistypen.
- Die Idee bei dieser Ordnung ist, ähnliche Ereignisse zu Gruppen zusammen zu fassen.
- Andernfalls wäre jedes Event durch eine eigene Klasse repräsentiert.
- Fenster senden unterschiedliche Events, wenn sie verkleinert oder geschlossen werden.
- Diese Events werden in der Klasse `java.awt.event.WindowEvent` zusammengefasst.
- Die Interpretation von Tastatureingaben hängt von evtl. zusätzlich zu der Eingabe gedrückten Zusatz Tasten ab, die in `...event.KeyEvent` definiert sind.

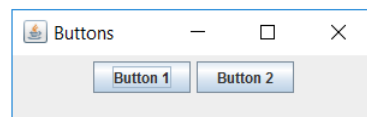


- Alle Ereignisse/Events werden von der Basisklasse `java.util.EventObject` abgeleitet und enden im Namen auf **Event**.
- Wichtigste Methode der Basisklasse ist:
 - `Object getSource()`
welche eine Referenz auf die Event-Source liefert.
- Ist der Name eines Ereignisses `<X>Event`, so ist der des zugehörigen Interfaces `<X>Listener`.
- Die Methoden zur (De-)Registrierung in den Event-Sources heißen
 - `add<X>Listener(<X>Listener listener)`
 - `remove<X>Listener(<X>Listener listener)`

```
public class ButtonGUI {
    private JFrame jf;
    private JButton b1=new JButton("Button 1");
    private JButton b2=new JButton("Button 2");

    public ButtonGUI(String titel) {
        jf=new JFrame(titel);
        JPanel jp=new JPanel();
        jp.add(b1);
        jp.add(b2);
        jf.add(jp);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack(); jf.setVisible(true);
    }

    public static void main(String[] args) {
        new ButtonGUI("Buttons");
    }
}
```





```
public class ButtonGUI implements ActionListener{ // Interface implementieren
    private JFrame jf;
    private JButton b1=new JButton("Button 1");
    private JButton b2=new JButton("Button 2");

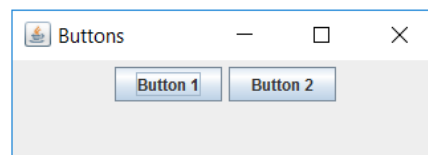
    public ButtonGUI(String titel) {
        jf=new JFrame(titel);
        JPanel jp=new JPanel();
        jp.add(b1);
        b1.addActionListener(this); // Objekt dieser Klasse registrieren
        jp.add(b2);
        b2.addActionListener(this); // Objekt dieser Klasse registrieren
        jf.add(jp);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack(); jf.setVisible(true);
    }
}
```



```
@Override // auf ein ankommendes Event reagieren
public void actionPerformed(ActionEvent ev) {
    Object quelle=ev.getSource();
    if (b1==quelle){
        System.out.println("Button 1 gedrückt!");
    }
    if (b2==quelle){
        System.out.println("Button 2 gedrückt!");
    }
}
```

ButtonGUI [Java Application] C:\Program Files\jre\bin\javaw.exe (21.06.2017, 13:35:04)

```
Button 1 gedrückt!
Button 1 gedrückt!
Button 1 gedrückt!
Button 2 gedrückt!
Button 1 gedrückt!
```





```
public class ButtonGUI{
    private JFrame jf;
    private JButton b1=new JButton("Button 1");
    private JButton b2=new JButton("Button 2");

    public JButton getB1(){ // die Buttons brauchen Getter
        return b1;
    }
    public JButton getB2(){
        return b2;
    }

    public ButtonGUI(String titel) {
        jf=new JFrame(titel);
        JPanel jp=new JPanel();
        EventHandler h=new EventHandler(this); // neuer Handler
        jp.add(b1);
        b1.addActionListener(h); // der neue Handler wird übergeben
        jp.add(b2);
        b2.addActionListener(h);
        jf.add(jp);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack(); jf.setVisible(true);
    }
}
```



```
public class EventHandler implements ActionListener{
    private ButtonGUI gui;

    public EventHandler(ButtonGUI gui){
        this.gui=gui;
    }

    @Override
    public void actionPerformed(ActionEvent ev) {
        Object quelle=ev.getSource();
        if (gui.getB1()==quelle){
            System.out.println("Button 1 gedrückt!");
        }
        if (gui.getB2()==quelle){
            System.out.println("Button 2 gedrückt!");
        }
    }
}
```



```
public ButtonGUI(String titel) {  
    jf=new JFrame(titel);  
    JPanel jp=new JPanel();  
    jp.add(b1);  
    b1.addActionListener(new ActionListener(){  
        @Override // anonyme innere Klasse  
        public void actionPerformed(ActionEvent arg0) {  
            System.out.println("Button 1 gedrückt!");  
        }  
    });  
    jp.add(b2);  
    b2.addActionListener(new ActionListener(){  
        @Override // anonyme innere Klasse  
        public void actionPerformed(ActionEvent arg0) {  
            System.out.println("Button 2 gedrückt!");  
        }  
    });  
    jf.add(jp);  
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    jf.pack(); jf.setVisible(true);  
}
```



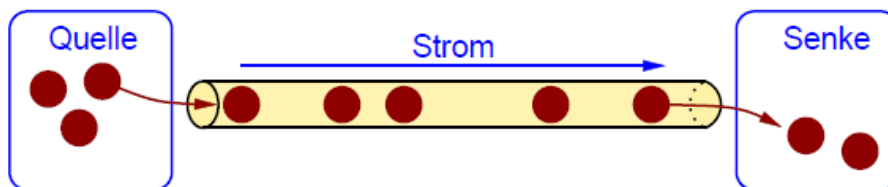
Eingabe / Ausgabe

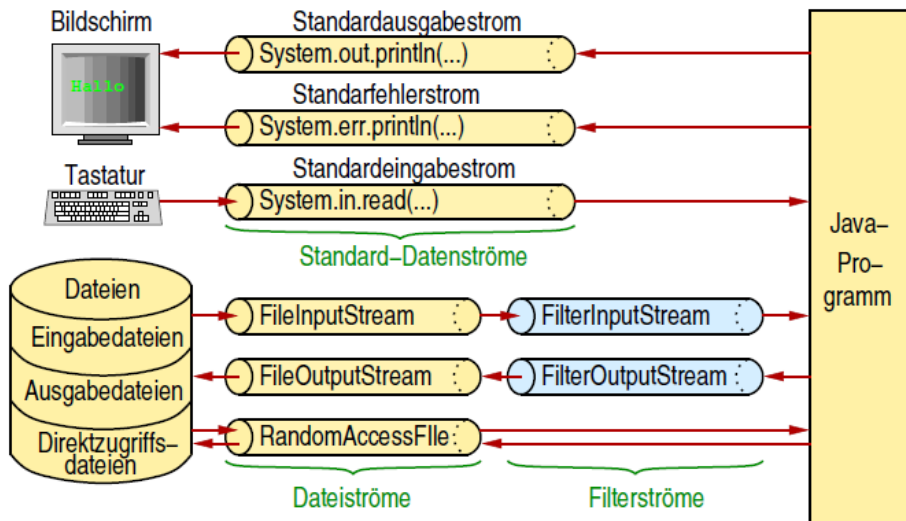
- In klassischen Programmiersprachen ist Eingabe/Ausgabe bzw. input/output bzw. I/O Teil der Sprache.
- In modernen Programmiersprachen wird IO in Bibliotheken ausgelagert:
 - Sockets
 - HTTP
 - Pipes
 - Dateien
- Häufig sind die Schnittstellen für die verschiedenen Datenquellen sehr unterschiedlich.

- Java verwendet ein einheitliches Konzept für nahezu alle I/O-Operationen: Streams (Ströme)
- Sie liefern bzw. konsumieren die Daten in sequentieller Form.
- Man unterscheidet
 - Input Streams
Das Programm liest Bytes von einer Quelle.
 - Output Streams
Das Programm schreibt Bytes in ein Ziel.
 - Reader
Wie InputStream, nur für Zeichen statt Bytes.
 - Writer
Wie OutputStream, nur für Zeichen statt Bytes.

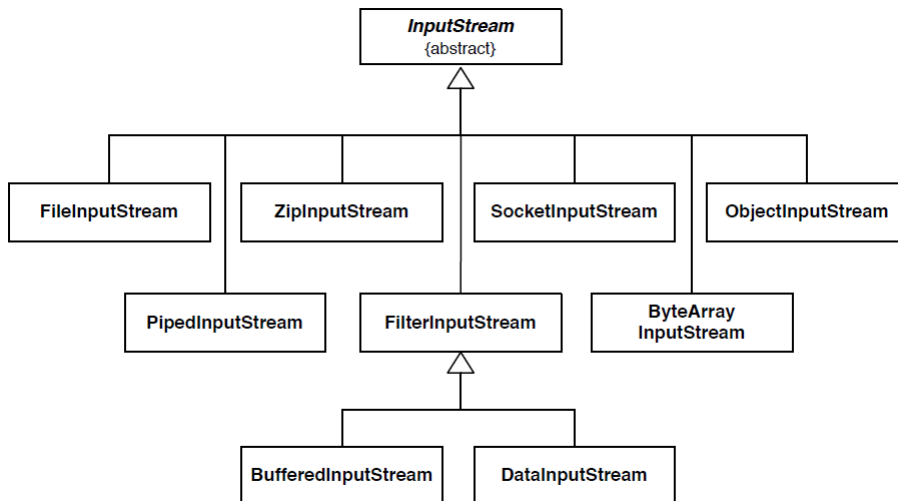
- Java verwendet ein einheitliches Konzept für nahezu alle I/O-Operationen: Streams (Ströme)
- Sie liefern bzw. konsumieren die Daten in sequentieller Form.
- Sie sind eine Schnittstelle des Programms nach außen.
- Man unterscheidet
 - Input Streams
Das Programm liest Bytes von einer Quelle.
 - Output Streams
Das Programm schreibt Bytes in ein Ziel.
 - Reader
Wie InputStream, nur für Zeichen statt Bytes.
 - Writer
Wie OutputStream, nur für Zeichen statt Bytes.

- Ein Strom ist eine geordnete Folge von Daten mit einer Quelle & einer Senke.
- Ströme sind in der Regel unidirektional.
- Ein Strom puffert die Daten so lange, bis sie von der Senke entnommen werden, nach dem Prinzip einer Warteschlange.





- Java definiert drei Standard-Datenströme für die Ein-/Ausgabe von bzw. 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



- FileInputStream für Dateien
- ZipInputStream für Zip-Dateien
- SocketInputStream für Netzwerk-Sockets
- ObjectInputStream zur Serialisierung
- PipedInputStream zum Verbindung von Threads
- ByteArrayInputStream zum Lesen aus einem Byte-Array
- FilterInputStream als Basisklasse für Filter
- BufferedInputStream zum Pufferung
- DataInputStream zum Lesen von Daten



- **abstract int read()**
liest ein einzelnes Byte vom Stream.
- **int read(byte[] b)**
liest vom Stream in das Byte-Array.
- **int read(byte[] b, int off, int len)**
liest maximal len Bytes in das Byte-Array ab position off.
- **long skip(long n)**
überspringt n Bytes.
- **void close()**
schließt den Stream.



```
InputStream fis = new FileInputStream("/tmp/test.txt");

int daten;

while ((daten = fis.read()) != -1) {
    byte b = (byte) daten;
    // jetzt kann man etwas sinnvolles mit den Bytes machen, die aus
    // der Datei gelesen wurden
}

fis.close();
```

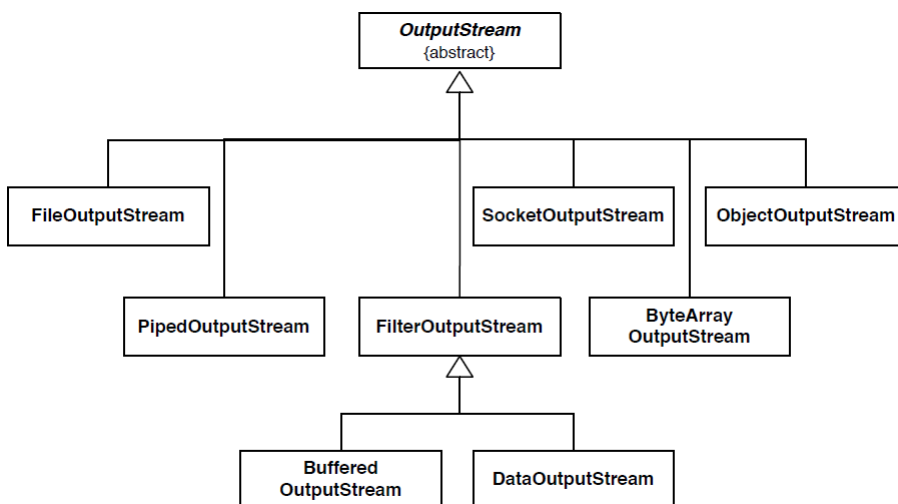


```
InputStream fis = new FileInputStream("/tmp/test.txt");

byte[] daten = new byte[1024];
int bytesRead;

while ((bytesRead = fis.read(daten)) > -1) {
    // jetzt kann man etwas sinnvolles mit den Bytes machen, die aus
    // der Datei gelesen wurden
}

fis.close();
```





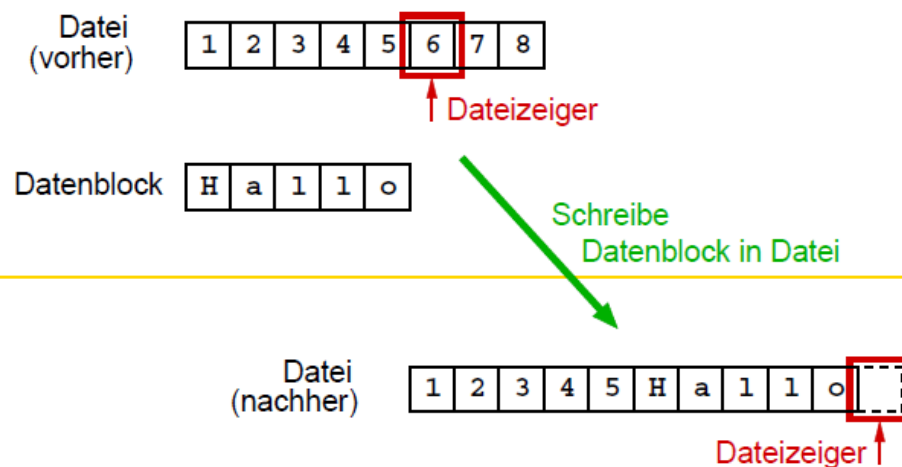
- `FileOutputStream` für Dateien
- `SocketOutputStream` für Netzwerk-Sockets
- `ObjectOutputStream` zur Serialisierung
- `PipedOutputStream` zum Verbindung von Threads
- `FilterOutputStream` als Basisklasse für Filter
- `ByteArrayOutputStream` zum Schreiben in ein Byte-Array
- `BufferedOutputStream` zur Pufferung
- `DataOutputStream` zum Schreiben von Daten



- **`abstract void write(int b)`**
schreibt ein Byte.
- **`void write(byte[] b)`**
schreibt das gesamte Byte-Array.
- **`void write(byte[] b, int off, int len)`**
schreibt len Bytes aus dem Byte-Array beginnend bei off.
- **`void flush()`**
leert interne Puffer und bringt die Daten auf den Datenträger.
- **`void close()`**
schließt den Stream und impliziert ein `flush()`.



- `FileInputStream`
 - `public FileInputStream(String name)`
throws `FileNotFoundException`
 - `public FileInputStream(File file)`
throws `FileNotFoundException`
- `FileOutputStream`
 - `public FileOutputStream(String name)`
throws `FileNotFoundException`
 - `public FileOutputStream(String name, boolean append)`
throws `FileNotFoundException`
 - `public FileOutputStream(File file)`
throws `FileNotFoundException`
 - `public FileOutputStream(File file, boolean append)`
throws `FileNotFoundException`

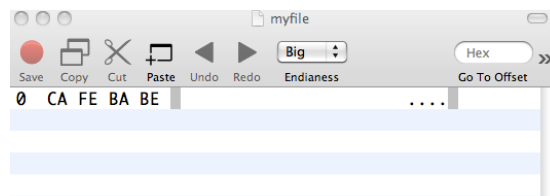




```
OutputStream fos = new FileOutputStream("/tmp/myfile");
```

```
fos.write(0xca);  
fos.write(0xfe);  
fos.write(0xba);  
fos.write(0xbe);
```

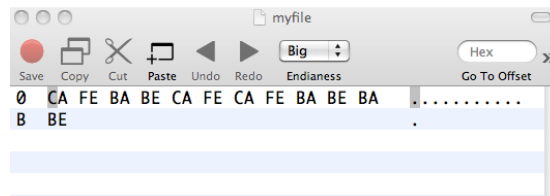
```
fos.close();
```



```
OutputStream fos = new FileOutputStream("/tmp/myfile");
```

```
byte[] daten = { (byte) 0xca, (byte) 0xfe,  
                (byte) 0xba, (byte) 0xbe };
```

```
fos.write(daten);  
fos.write(daten, 0, 2);  
fos.write(daten, 0, 2);  
fos.write(daten, 2, 2);  
fos.write(daten, 2, 2);  
fos.close();
```

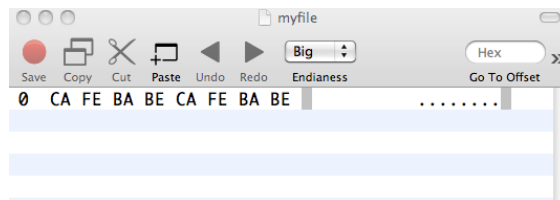




```
byte[] daten = { (byte) 0xca, (byte) 0xfe,  
                (byte) 0xba, (byte) 0xbe };
```

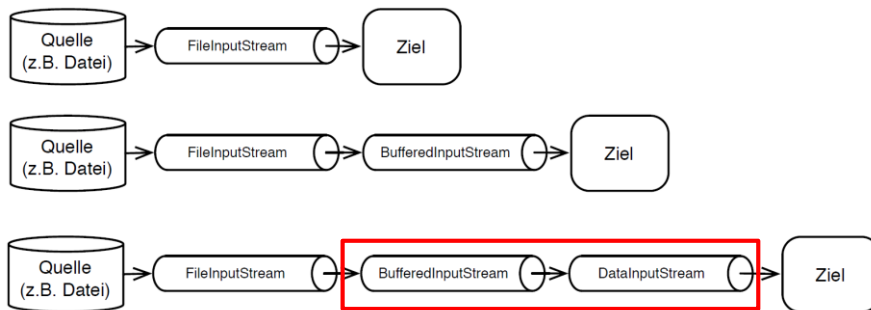
```
OutputStream fos = new FileOutputStream("/tmp/myfile");  
fos.write(daten);  
fos.close();
```

```
OutputStream fos2 = new FileOutputStream("/tmp/myfile", true);  
fos2.write(daten);  
fos2.close();
```



```
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
public class CopyFile {  
    public static void main(String[] args) throws IOException {  
        FileInputStream in = new FileInputStream("C:\\\\haus.jpg");  
        FileOutputStream out = new FileOutputStream("C:\\\\haus2.jpg");  
        int b = in.read();  
        while (b != -1) {  
            out.write(b);  
            b = in.read();  
        }  
        in.close();  
        out.close();  
    }  
}
```

- Zusätzlich zu den Streams, die direkt mit einer Quelle oder Senke (Datei, Socket etc.) verbunden sind, gibt es die Filter-Streams.
- Filter-Streams können hinter einen anderen Stream geschaltet werden und können die Daten entsprechend verändern.
- Dieses Konzept nennt man auch Decorator Pattern.



- **DataOutputStream**
schreibt primitive Datentypen und String in einen anderen Stream und erlaubt so den plattformunabhängigen Datenaustausch.
- **DataInputStream**
liest die von **DataOutputStream** geschriebenen Daten wieder ein.



- `void writeBoolean(boolean v) throws IOException`
- `void writeByte(int v) throws IOException`
- `void writeShort(int v) throws IOException`
- `void writeChar(int v) throws IOException`
- `void writeInt(int v) throws IOException`
- `void writeLong(long v) throws IOException`
- `void writeFloat(float v) throws IOException`
- `void writeDouble(double v) throws IOException`
- `void writeBytes(String s) throws IOException`
- `void writeChars(String s) throws IOException`
- `void writeUTF(String s) throws IOException`



```
DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(
        new FileOutputStream("/tmp/daten")));
```

```
out.writeUTF("** Datendatei **");
out.writeUTF("Datum");
out.writeLong(new Date().getTime());
out.writeUTF("PI");
out.writeDouble(Math.PI);
```

```
out.close();
```

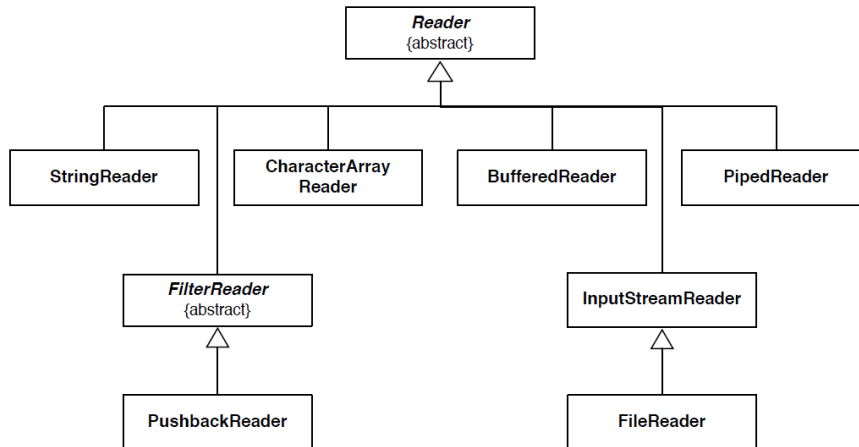
```
00 10 2A 2A 20 44 61 74 65 6E 64  ..** Datend
61 74 65 69 20 2A 2A 00 05 44 61  ate i **..Da
74 75 6D 00 00 01 23 AF EE 3D 90  tum...#..=.
00 02 50 49 40 09 21 FB 54 44 2D  ..PI@.!.TD-
18  .
```



- `boolean readBoolean()` throws `IOException`
- `byte readByte()` throws `IOException`
- `int readUnsignedByte()` throws `IOException`
- `short readShort()` throws `IOException`
- `int readUnsignedShort()` throws `IOException`
- `char readChar()` throws `IOException`
- `int readInt()` throws `IOException`
- `long readLong()` throws `IOException`
- `float readFloat()` throws `IOException`
- `double readDouble()` throws `IOException`
- `String readUTF()` throws `IOException`



```
DataInputStream dis = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("/tmp/daten")));  
  
String header = dis.readUTF();  
String datumTag = dis.readUTF();  
Date datum = new Date(dis.readLong());  
String PITag = dis.readUTF();  
double pi = dis.readDouble();  
dis.close();  
    ** Datendatei **  
    Datum Sun Sep 12 22:49:27 CEST 2010  
System.out.println(header); PI 3.141592653589793  
System.out.println(datumTag + " " + datum);  
System.out.println(PITag + " " + pi);
```



- StringReader, um aus Strings zu lesen
- CharacterArrayReader, um aus char-Arrays zu lesen
- BufferedReader zum gepufferten Lesen
- PipedReader zur Verbindung von Threads
- InputStreamReader zur Verknüpfung von Stream und Reader
- FileReader, um aus Dateien zu lesen
- FilterReader als Filter
- PushbackReader zum lesen und zurückstellen von Zeichen



- **int read()**
liest ein Zeichen.
- **int read(char[] cbuf)**
liest Zeichen in das char-Array.
- **int read(char[] cbuf, int off, int len)**
liest maximal len Zeichen in das char-Array cbuf beginnend bei Position off.
- **long skip(long n)**
überspringt n Zeichen.
- **void close()**
schließt den Reader.



```
Reader fr = new FileReader("/tmp/test.txt");

int daten;

while ((daten = fr.read()) > -1) {
    char c = (char) daten;
    // jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden
}

fr.close();
```



```
Reader fr = new FileReader("/tmp/test.txt");

char[] daten = new char[1024];
int charactersRead;

while ((charactersRead = fr.read(daten)) > -1) {
    // jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden
}

fr.close();
```



```
String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\n" +
    "Nulla laoreet, sem vel mollis imperdiet, sapien mauris\n" +
    "sollicitudin arcu, sed viverra nulla dui at est.\n" +
    "Nunc est erat, semper id sollicitudin ut, pretium ac eros.\n";

Reader sr = new StringReader(text);

int gelesen;
while ((gelesen = sr.read()) > -1) {
    System.out.print((char) gelesen);
}

sr.close();
```



```
String dateiname = "/tmp/text.txt";

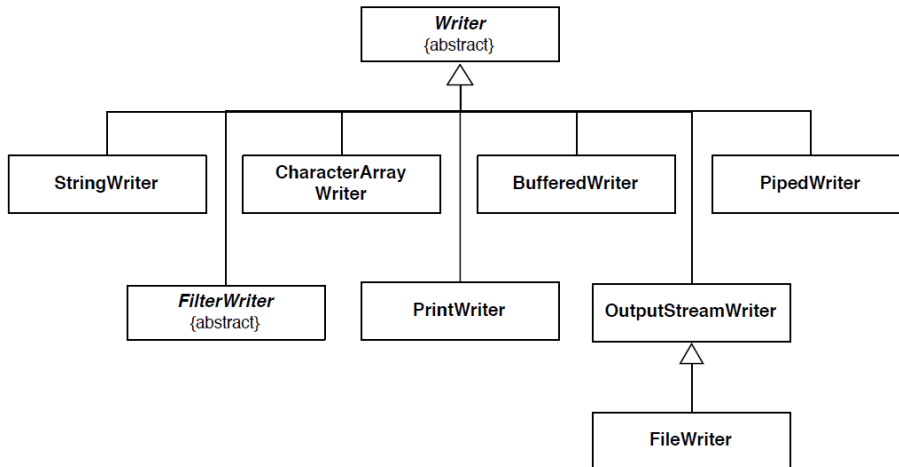
BufferedReader reader = new BufferedReader(
    new FileReader(dateiname));

String zeile;
int nummer = 1;
while ((zeile = reader.readLine()) != null) {
    System.out.printf("%03d %s\n", nummer, zeile);
    nummer++;
}

reader.close();
```



- **BufferedReader(Reader in)**
- **BufferedReader(Reader in, int bufferSize)**
- **String readLine()**



- `StringWriter`, um in Strings zu schreiben
- `CharacterArrayWriter`, um n char-Arrays zu schreiben
- `BufferedWriter` zum gepufferten Schreiben
- `PipedWriter` zur Verbindung von Threads
- `OutputStreamWriter` zur Verbindung von Stream und Writer
- `FileWriter`, um in eine Datei zu schreiben
- `PrintWriter` zur formatierten Ausgabe von Datentypen
- `FilterWriter` als Filter



- **`void write(int c)`**
schreibt ein Zeichen.
- **`void write(char[] cbuf)`**
schreibt alle Zeichen aus dem char-Array.
- **`void write(char[] cbuf, int off, int len)`**
schreibt len Zeichen aus dem char-Array cbuf beginnend bei off.
- **`void write(String str)`**
schreibt den String str.
- **`void write(String str, int off, int len)`**
schreibt len Zeichen aus dem String str, beginnend bei off.
- **`void flush()`**
leert interne Puffer und bringt die Daten auf den Datenträger.
- **`void close()`**
schließt den Writer und impliziert einen flush().



```
Writer fw = new FileWriter("/tmp/mytext");
```

```
fw.write('T');  
fw.write('e');  
fw.write('x');  
fw.write('t');  
fw.write('\n');
```

```
fw.close();
```




```
Writer fw = new FileWriter("/tmp/mytext");
```

```
String daten = "Dies ist ein Text";
```

```
fw.write(daten);
```

```
fw.write(daten, 0, 5);
```

```
fw.write(daten, 0, 5);
```

```
fw.write(daten, 12, 5);
```

```
fw.write(daten, 12, 5);
```

```
fw.write("\n");
```

```
fw.close();
```

Dies ist ein TextDies Dies Text Text



```
import java.io.FileWriter;  
import java.io.IOException;  
import java.io.PrintWriter;
```

```
public class CSV {  
    public static void main(String[] args) throws IOException {  
        PrintWriter pw;  
        pw = new PrintWriter(new FileWriter("dat.txt"));  
        for (int i=1; i<10; i++) {  
            pw.println(i + ";" + i*i + ";" + Math.sqrt(i));  
        }  
        pw.close();  
    }  
}
```

```
dat.txt  
11;1;1.0  
22;4;1.4142135623730951  
33;9;1.7320508075688772  
44;16;2.0  
55;25;2.23606797749979  
66;36;2.449489742783178  
77;49;2.6457513110645907  
88;64;2.8284271247461903  
99;81;3.0  
10|
```



- `BufferedWriter(Writer out)`
- `BufferedWriter(Writer out, int bufferSize)`
- `void newLine()`



- Manche APIs liefern nur Streams, manchmal möchte man die Daten aber mit einem Reader/Writer verarbeiten
 - Sockets
 - Servlet API
- `InputStreamReader` und `OutputStreamWriter` dienen dazu, Streams und Reader/Writer miteinander zu verbinden
 - Sie finden also Verwendung als Filter bzw. Decorator.



```
Socket socket = new Socket("www.web.de", 80);
OutputStream os = socket.getOutputStream();
InputStream is = socket.getInputStream();

BufferedReader reader = new BufferedReader(new InputStreamReader(is));
Writer writer = new OutputStreamWriter(os);

writer.write("GET / HTTP/1.0\r\n\r\n");
writer.flush();

String line;

while ((line = reader.readLine()) != null) {
    System.out.println(line);
}

reader.close();
writer.close();
socket.close();
```



```
HTTP/1.1 200 OK
Date: Sun, 13 Sep 2010 08:47:59 GMT
Server: Apache/2
Last-Modified: Sun, 13 Sep 2010 08:45:25 GMT
ETag: "133970-18cc3-9202d340"
Accept-Ranges: bytes
Content-Length: 101571
Cache-Control: max-age=0
Expires: Sun, 13 Sep 2010 08:47:59 GMT
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de"
id="buster">
<head>
    <title>WEB.DE - E-Mail - Suche - DSL - Modem - Shopping
    ...
```



```
class Student {  
    private String name, vorname;  
    private int matrNr;  
    private double note;  
  
    public Student(String n, String vn, int matrNr) {  
        // Gültigkeitsprüfungen  
        this.name = n; this.vorname = vn;  
        this.matrNr = matrNr; this.note = 0.0;  
    }  
}
```



```
public 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]);  
    }  
    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");  
    }  
}
```



hochschule mannheim

Beispiel zum Laden & Speichern eines Studenten

```
public void writeToStream(PrintWriter pw) {
    pw.println(name + ";" + vorname + ";" + matrNr + ";" + note);
    pw.flush();
}

@Override
public String toString() {
    return name + " | " + vorname + " (" + matrNr + "): " + note;
}
}
```



hochschule mannheim

Beispiel zum Speichern eines Studenten

```
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class TextIOWrite {
    public static void main(String[] args) throws IOException {
        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) {
            System.err.println("Konnte 'out.txt' nicht erzeugen");
        }
        catch (IOException e) {
            System.err.println("Fehler bei der Ein-/Ausgabe: " + e);
        }
        finally {
            if (pw != null) pw.close();
        }
    }
}
```

```
<terminated> TextIOWrite [Java Application]
Hugo;Test;12345678;0.0
```



Beispiel zum Speichern eines Studenten

```
import java.io.BufferedReader; import java.io.FileNotFoundException;  
import java.io.FileReader; import java.io.IOException;
```

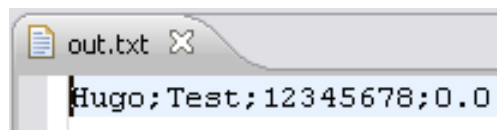
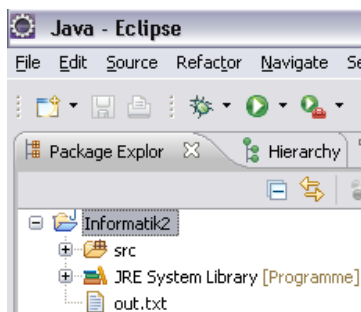
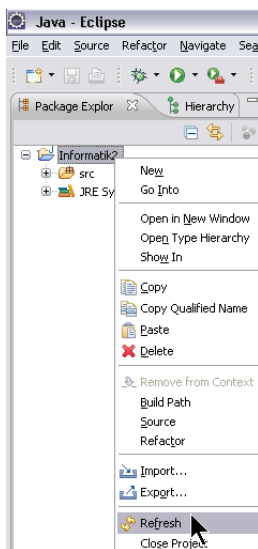
```
public class TextIOread {  
    public static void main(String[] args) {  
        BufferedReader reader = null;  
        try {  
            reader = new BufferedReader(new FileReader("out.txt"));  
            Student s = new Student(reader); System.out.println(s);  
        }  
        catch (FileNotFoundException e) {  
            System.err.println("Konnte 'out.txt' nicht öffnen");  
        }  
        catch (IOException e) {  
            System.err.println("Fehler bei der Ein-/Ausgabe: " + e);  
        }  
        finally {  
            try {  
                reader.close();  
            }  
            catch (Exception e) {  
                System.err.println("Fehler beim Schließen der Datei");  
            }  
        }  
    }  
}
```

<terminated> TextIOread [Java Application]

Hugo Test (12345678): 0.0



Ansehen der txt-Datei in Eclipse



oder F5...



```
PrintStream ps = new PrintStream("/tmp/umleitung.txt");  
System.setOut(ps);  
  
System.out.println("Hallo, das kommt ja gar nicht raus...");  
  
ps.close();
```



- Java ist plattformunabhängig, der Umgang mit Dateien ist aber extrem plattformabhängig.
 - Unterschiedliche Separatoren (/ vs. \)
 - Unterschiedliche Konzepte für Dateisystem-Wurzeln (/ vs. c:\, d:\)
 - Unterschiedliche Berechtigungen (rwx vs. ACLs)
- Die Klasse File abstrahiert von der Plattform und erlaubt es relativ abstrakt in Java mit Dateien zu operieren.
- Alle File-Streams und File-Reader/Writer akzeptieren anstatt eines Strings auch ein File-Objekt.



- `File(String pathname)`
- `String getName()`
- `String getPath()`
- `boolean isAbsolute()`
- `String getAbsolutePath()`
- `File getAbsoluteFile()`
- `String getCanonicalPath()`
- `File getCanonicalFile()`



- `URI toURI()`
- `boolean canRead(), canWrite(),`
- `boolean canExecute(), exists(), isDirectory(),`
`isFile(), isHidden()`
- `long lastModified()`
- `boolean setLastModified(long time)`
- `long length()`
- `boolean createNewFile()`
- `boolean delete()`



- `String[] list(...)`
- `File[] listFiles()`
- `boolean mkdir()`
- `boolean renameTo(File dest)`
- `boolean setWritable(...), setReadable(...),
setExecutable(...), setReadOnly()`
- `long getTotalSpace(), getFreeSpace(), getUsableSpace()`



- Wie transportiere ich Java Objekte über das Netzwerk?
- Wie schreibe ich Java Objekte in Dateien?
- Wie lese ich Java Objekte aus Dateien?
- Wie gehe ich mit ganzen Geflechten von Objektreferenzen um?



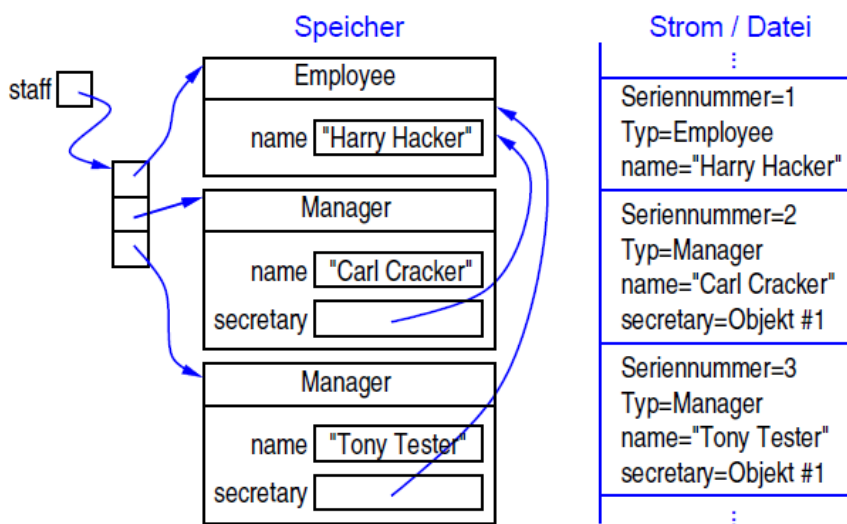
- Mit **DataOutputStream** und **DataInputStream** kann man Objekte in Streams schreiben und aus ihnen lesen, man muss aber die gesamte Logik von Hand schreiben.
- Java-Serialisierung übernimmt dies automatisch und schreibt ganze Objektgeflechte.
- Grundlegende Klassen:
 - **ObjectOutputStream**
serialisiertes Schreiben von Objekten
 - **ObjectInputStream**
serialisiertes Lesen von Objekten (Deserialisierung)
- Serialisierung ist relativ robust gegen Änderungen an den Objekten, sodass serialisierte Objekte auch nach Änderung an der Klasse (häufig) noch deserialisiert werden können.

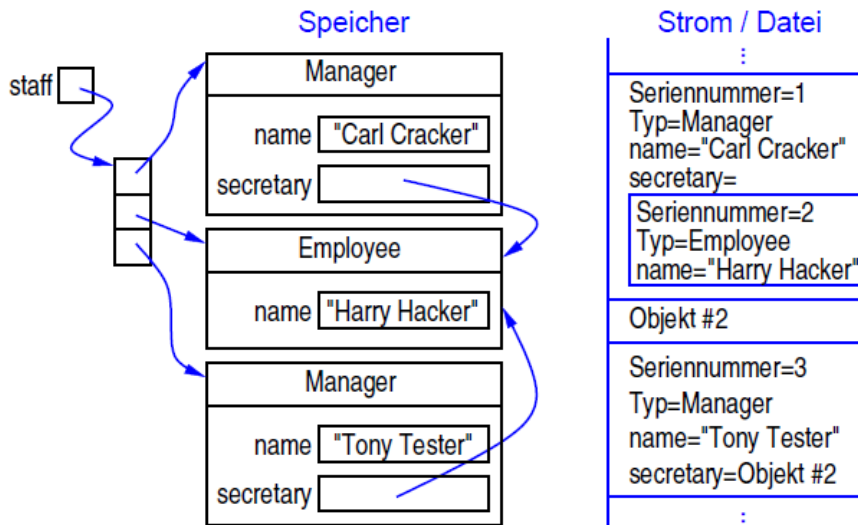


- Objekte können nur serialisiert werden, wenn ihre Klassen das Interfaces **java.io.Serializable** implementieren.
- Superklassen müssen entweder selbst **java.io.Serializable** implementieren oder einen Default-Konstruktor haben.
- Serialisierung erfasst alle Felder, manche Objekte können aber nicht serialisiert werden.
- Mit dem Schlüsselwort **transient** kann man einzelne Attribute von der Serialisierung ausnehmen; diese nennt man dann transiente Attribute.
- Die Java-Laufzeit berechnet für jede Klasse eine spezielle ID um Kompatibilität bei der Serialisierung sicherzustellen, die **serialVersionUID**.
- Man sollte für serialisierbare Klassen eine eigene **serialVersionUID** angeben.



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 Objekts bzw. Elemente des Arrays...
 - falls keine Referenz: schreibe den Wert in den Strom;
 - ansonsten: Z = Ziel der Referenz
 - falls Z noch nicht in diesen Strom serialisiert wurde: serialisiere Z (Rekursion!)
 - ansonsten: schreibe die Seriennummer von Z in den Strom





```

class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    private Name name; private int matrNr; private double note;

    public Student(String n, String vn, int matrNr) {
        // Gültigkeitsprüfungen
        this.name = new Name(n, vn);
        this.matrNr = matrNr; this.note = 0.0;
    }

    void setNote(double note) {
        // Gültigkeitsprüfungen
        this.note = note;
    }

    @Override
    public String toString() {
        return name + " (" + matrNr + "): " + note;
    }
}

```



```
class Name implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name; private String vorname;

    public Name(String name, String vorname) {
        // Gültigkeitsprüfungen
        this.name = name;
        this.vorname = vorname;
    }
    @Override
    public String toString() {
        return name + " " + vorname;
    }
}
```



```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class ObjectIOwrite {
    public static void main(String[] args) throws IOException {
        ObjectOutputStream oos = null;
        try {
            Student s=new Student("Hugo", "Test", 12345678);
            s.setNote(1.3);
            oos=new ObjectOutputStream(new FileOutputStream("out.ser"));
            oos.writeObject(s);
            System.out.println(s);
        }
    }
}
```



Beispiel: Einen Student serialisieren

```
catch (FileNotFoundException e) {
    System.err.println("Konnte 'out.ser' nicht erzeugen");
}
catch (IOException e) {
    System.err.println("Fehler bei der Ein-/Ausgabe: " + e);
}
finally {
    try {
        oos.close();
    }
    catch (Exception e) {
        System.err.println("Fehler beim Schließen der Datei");
    }
}
}
```



Beispiel: Der serialisierte Student in der Datei

out.ser

```
00000000h: AC ED 00 05 73 72 00 0F 53 74 72 C3 B6 6D 65 2E ; i..sr..StrÄme.
00000010h: 53 74 75 64 65 6E 74 00 00 00 00 00 00 01 02 ; Student.....
00000020h: 00 03 49 00 06 6D 61 74 72 4E 72 44 00 04 6E 6F ; ..I..matrNrD..no
00000030h: 74 65 4C 00 04 6E 61 6D 65 74 00 0E 4C 53 74 72 ; teL..namet..LStr
00000040h: C3 B6 6D 65 2F 4E 61 6D 65 3B 78 70 00 BC 61 4E ; Äme/Name;xp..a
00000050h: 3F F4 CC CC CC CC CC CD 73 72 00 0C 53 74 72 C3 ; öiiiiifsr..StrÄ
00000060h: B6 6D 65 2E 4E 61 6D 65 00 00 00 00 00 00 01 ; me.Name.....
00000070h: 02 00 02 4C 00 04 6E 61 6D 65 74 00 12 4C 6A 61 ; ...L..namet..Lja
00000080h: 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 4C ; va/lang/String;L
00000090h: 00 07 76 6F 72 6E 61 6D 65 71 00 7E 00 04 78 70 ; ..vornameq.~..xp
000000a0h: 74 00 04 48 75 67 6F 74 00 04 54 65 73 74 ; t..Hugot..Test
```



Beispiel: Einen Student de-serialisieren

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class ObjectIOread {
    public static void main(String[] args) throws IOException {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream("out.ser"));
            Student s = (Student) ois.readObject();
            System.out.println(s);
        }
    }
}
```



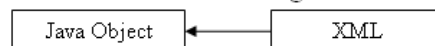
Grenzen der Serialisierung

- Bei einer Änderung einer Klassen-Definition:
 - Die gespeicherten Daten sind nicht mehr als Objekte importierbar.
- Ein Import von anderen Anwendungen nur schwer möglich.
- Lösungsansätze:
 - Speichern des Objekt-Zustandes in einer XML-Datei
 - DOM/SAX-Parser (Document Object Model bzw. Simple API for XML)
 - JAXB (Java Architecture for XML Binding)

Marshalling



Unmarshalling



- Lösungsansätze:
 - Speichern des Objekt-Zustandes in einer Datenbank:
 - JDBC (Java DataBase Connectivity)
 - Hibernate als ORM: Objekt-relationaler Mapper
 - Verwendung einer beliebigen Datenbank, welche JPA (Java Persistence API) unterstützt

```
@Entity
public class Person
{
    ...
}

@Entity
public class Manager extends Person
{
    ...
}
```

Entity	class is "PersistenceCapable"
Table	table to be used when persisting this class
Id	field is being part of the primary key
OneToOne	field is being part of a 1-1 relation
OneToMany	field is being part of a 1-N relation

- Alle IO Operationen von Streams, Readern und Writern sind blockierend.
- Moderne Konzepte wie Memory-Mapped-Files werden nicht unterstützt
- Mit Java 1.4 wurde parallel zu `java.io` mit `java.nio` ein neues I/O-Konzept eingeführt:
 - nio = new IO, gesprochen „neio“

- Die zentralen Elemente von NIO sind
 - Buffer zur Verwaltung der gelesenen und zu schreibende Daten.
Es gibt Buffer für alle primitiven Typen:
 - ByteBuffer, CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer, DoubleBuffer
 - Channels als Quellen bzw. Senken für Daten aus Buffers; diese sind ähnlich zu den Streams.
 - Sie können zum Lesen und Schreiben verwendet werden.
 - Memory mapped files:


```
MappedByteBuffer b=ch.map(MapMode.READ_WRITE, 0, 1024);
```
 - File Locks


```
FileLock lock=fc.lock(start, end, false);
```
 - Asynchrone Socket I/O

