Grundlagen der Informatik

Einführung in die objektorientierte Software-Entwicklung mit Java

2. Teil

Themen des 2. Semesters

- Abstrakte Klassen, Interface, Pakete
- Dateien
- Grundlegende Datenstrukturen
- Rekursives Programmieren
- Grundlegende Algorithmen
- Threads
- Frames

Typkonvertierung von Referenzen

- Eine Typkonvertierung gibt es zwischen Primitiven Datentypen und Referenzdatentypen.
- In Java ist es nicht unbedingt erforderlich, dass der Typ der Referenzvariablen identische mit dem Typ des Objektes ist, auf das die Referenzvariable zeigt.
- Wie bei primitiven Datentypen gibt es auch bei Referenzdatentypen eine implizite, d.h. automatischen Typkonvertierung und eine explizite Typkonvertierung mit Hilfe des cast-Operators.

Implizite Typkonvertierung von Referenzen

- Eine automatische Typkonvertierung findet immer dann statt, wenn eine Referenz auf eine Subklasse (Kindklasse) einer Referenz auf eine Superklasse (Elternklasse) zugewiesen wird.
  Beispiel
  ```
  Kind refKind = new Kind();
  Eltern refEltern = refKind;
  ```
  Die Referenz `refEltern` ist vom Referenztyp `Eltern` und zeigt auf ein Objekt der Klasse `Kind`. Dies ist erlaubt, da ein Kind Objekt alle Eigenschafen besitzt, die auch das Eltern Objekt hat.

- Die Referenz `refEltern` sieht nur die Elternteile des Kind-Objektes. `refEltern` sieht also nicht die zusätzlichen Eigenschaften des Objektes von Kind.
- Durch die Zuweisung `refElten = refKind` wird auf die Referenz `refKind` implizit der cast-Operatorn `(Eltern)` angewandt.
- Man spricht von *impliziter oder automatischer Typkonvertierung.*
- Der Rückgabewert dieser impliziten cast-Operation ist eine Referenz vom Typ `Eltern`, allerdings dann mit eingeschränkter Sichtweise.

Explizite Typkonvertierung von Referenzen

- Eine explizite Typkonvertierung von Referenzen muss immer dann erfolgen, wenn bei einer Zuweisung eine Referenzvariable vom Typ `Eltern`, die auf ein Objekt der Klasse `Kind` zeigt, einer Referenzvariablen vom Typ `Kind` zugewiesen wird.
- Eine explizite Typkonvertierung erfolgt mit Hilfe des cast-Operators.
  Beispiel
  ```
  Kind refKind = new Kind();
  Eltern refEltern = refKind;
  Kind refKind2 = (Kind) refEltern;
  ```


- Die Referenz `refEltern` zeigt auf ein Objekt der Klasse `Kind`. Auf dieser Referenz wird eine explizite cast-Operation ausgeführt. Der Rückgabewert der cast-Operation ist eine Referenz vom Typ `Kind`, die der Referenz `refKind2` zugewiesen wird.
- Eine solche explizite Typkonvertierung mit Hilfe des cast-Operators bezeichnet man auch als Down-Cast. Es gelten folgende Regeln:

- Bei einer Zuweisung `refKind = (Kind) refEltern` ist die explizite cast-Operation nur dann zulässig, wenn die Referenz `refEltern`
  - auf ein Objekt vom Typ `Kind` zeigt
  - oder auf ein Objekt eines Subtyps der Klasse `Kind` zeigt.

Abstrakte Klassen

Nicht immer soll eine Klasse sofort ausprogrammiert werden. Dies ist der Fall, wenn die Oberklasse lediglich Methoden für die Unterklasse vorgeben möchte, aber nicht weiß, wie sie diese implementieren soll.

In Java gibt es hierfür zwei Konzepte: *abstrakte Klassen* und *Schnittstellen* (*interfaces*).

- Es kann sinnvoll sein, dass eine Klasse in einer Vererbungshierarchie so allgemein ist, dass von ihr keine Instanzen (Objekte) gebildet werden können.
- Abstrakte Klassen werden mit dem Modifizierer `abstract` deklariert.
- Von dieser Klasse können keine Instanzen gebildet werden.
- Ansonsten verhalten sich abstrakte Klassen wie normale Klassen, sie enthalten die gleichen Eigenschaften und können auch selbst von anderen Klassen erben.

Beispiele:

- `Tiere` ist die Oberklasse für `Hunde`, `Katzen`, `Schlangen` etc.
- eine abstrakte Klasse `Kleidung` ist die Oberklasse für konkrete Kleidungsstücke, wie `Hose`, `Jacke` etc

```
abstract class Kleidung{
  public void wasche(){
    System.out.prinln("Werde sauber");
  }
 public void trockne(){
   System.out.prinln("Werde trocken");
  }
```


- Durch `abstract` wird ausgedrückt, dass die Klasse Kleidung eine allgemeine Klasse ist, zu der keine konkreten Objekte existieren.
  (dies entspricht der realen Welt, in der es keinen Gegenstand gibt, der nur ein allgemeines Kleidungsstück ist).
- Die abstrakten Klassen werden normal in der Vererbung eingesetzt.
- Eine Klasse kann die abstrakte Klasse erweitern und auch selbst wieder abstrakt sein.
- Erbt eine Klasse von einer abstrakten Klasse, dann ist sie vom Typ aller Oberklassen, auch vom Typ der abstrakten Klasse

- Beispiel

```
class Jacke extends Kleidung{
    ....
}
```

- Es ist also möglich

```
Jacke j1 = new Jacke();
Kleidung j1 = new Jacke();
```


Abstrakte Methoden

- Auch Methoden können abstrakt sein.
- Eine abstrakte Methode definiert nur den Kopf der Methode, die Implementierung muss in einer abgeleiteten Klasse (Unterklasse) nachgeholt werden.
- Eine abstrakte Methode wird durch das reservierte Wort `abstract` gekennzeichnet, der Methodenrumpf der Methode (die nicht vorhandene Implementierung) wird durch ein Semikolon ersetzt.
- Beispiel

```
public abstract class Fahrzeug{
    abstract void belade(double gewicht);
}
```


- Ist mindestens eine Methode innerhalb einer Klasse abstrakt, so ist es automatisch die ganze Klasse.
- Daher muss das reservierte Wort `abstract` ausdrücklich vor den Klassennamen geschrieben werden, sobald eine Methode abstrakt ist.

Vererben von abstrakten Methoden

Wenn eine Klasse abstrakte Methoden erbt, so gibt es für sie zwei Möglichkeiten:

- Die abstrakten Methoden werden überschrieben und implementiert
- Die abstrakten Methoden werden nicht überschrieben. Das bedeutet, dass die erbende Klasse auch wieder abstrakte Methoden enthält, so dass die Klasse ebenfalls abstrakt sein muss.


```
abstract class Tier{
     int alter = -1;

     void setAlter(int a){
          alter = a;
     }

     void ausgabeAlter(){
          System.out.println("Das Alter betraegt:"+alter);
     }

     abstract boolean istSaeuger();

     abstract void ausgabe();
}
```

```
abstract class Saeugetier extends Tier{
     boolean istSaeuger(){
     return true;
     }
}

class Mensch extends Saeugetier{
     void ausgabe(){
     System.out.println("Das ist ein Mensch");
     }
}

class Hund extends Saeugetier{
     void ausgabe(){
     System.out.println("Das ist ein Hund");
     }
}
```



```
public class AbstrakteMethoden{
       public static void main(String args[]){
             Tier m = new Mensch();
             Tier h = new Hund();

             m.ausgabe();
             m.setAlter(26);
             m.ausgabeAlter();

             h.ausgabe();
             h.setAlter(12);
             h.ausgabeAlter();
       }
}
```


Erklärung

- Durch die Klassendefinitionen `Tier, Saeugetier, Mensch, Hund` wurde eine Hierarchie aufgebaut.
- Die abstrakte Oberklasse Tier bestimmt, dass alle –nicht abstrakten- Unterklassen die Methoden `istSaeuger()` und `ausgabe()` implementieren müssen.
- Die Methode `ausgabe()` der Oberklasse `Tier` ist eine Methode, die für jede Tierart eine andere Meldung ausgibt. Daher macht es wenig Sinn, diese schon in `Tier` zu definieren. Die Methode wird `abstract` gesetzt. Damit wird auch die Klasse `Tier` zu `abstract`.

- Auch in der abgeleiteten Klasse `Saeugetier` macht die Implementierung von `ausgabe()` noch keinen Sinn. Hier wird aber die Methode `istSaeuger()` implementiert. Die Klasse kann um beliebig andere Methoden erweitert werden.
  Da `ausgabe()` nicht implementiert wurde, muss die Klasse Saeugetier ebenfalls `abstract` sein.
- Die dritte Klasse ist `Mensch` bzw. `Hund`. Sie erweitert die Klasse `Saeugetier` und implementiert `ausgabe()`. Damit muss sie nicht mehr abstrakt sein und es können Instanzen dieser Klasse angelegt werden.
- Instanzen werden hier durch `Tier m = new Mensch();` bzw. `Tier h = new Hund();` angelegt.
  Dass `h` und `m` vom Datentyp `Tier` sind, ist durch die automatische Typanpassung möglich.

- Durch `Tier m = new Mensch();` wird zunächst der Konstruktor von `Tier` (obwohl `Tier abstract` ist, besitzt die Klasse einen Standardkonstruktor, dieser kann nur nicht mit `new` direkt aufgerufen werden), danach der Konstruktor von `Saeugetier` und dann von `Mensch` aufgerufen.

Interfaces (Schnittstellen)

- In Java kann eine abstrakte Klasse, die ausschließlich abstrakte Methoden und Konstanten definiert, durch eine Schnittstelle ersetzt werden.
- Schnittstellen enthalten also keine Implementierung, sondern nur den Namen und Parameterliste der enthaltenen Methoden.
- Schnittstellen werden durch das reservierte Wort `interface` gekennzeichnet.
- Möchte eine Klasse eine Schnittstelle verwenden, so folgt hinter dem Klassennamen das Schlüsselwort `implements`.
- Eine Klasse kann mehrere Schnittstellen implementieren.

- Sprechweise:
  Klassen werden vererbt und Schnittstellen werden implementiert.

Schnittstellen definieren

- Interfaces müssen wie Klassen mit ihren Eigenschaften und Methoden definiert werden.
- Der äußere Aufbau einer Interfacedefinition ist – bis auf das Schlüsselwort `interface` - mit dem einer Klasse identisch.
- Eigenschaften können allerdings nur als Konstanten definiert werden.
- Alle Variablen sind bei Schnittstellen automatisch `public, static, final`, sie können also von anderen Objekten direkt über den Interface-Namen angesprochen und nicht geändert werden.
- Methoden werden in einem Interface nur deklariert, d.h. es werden der Name, Rückgabetyp und die Parameterliste angegeben. Der Methodenrumpf wird durch ein Semikolon ersetzt.

- Methoden sind innerhalb eines Interfaces damit automatisch `public` und `abstract`. Da dies immer der Fall ist, müssen diese beiden reservierten Schlüssel-Wörter nicht explizit angegeben werden.
- Beispiel für die Definition einer Schnittstelle

```
public interface BonusSparen{
      int BONUS = 100;
      int LAUFZEIT = 1;

      double addBonus(double laufzeit);
}
```


Die Implementierung einer Schnittstelle

Die Klassen, die die zusätzlichen Methoden eines bestimmten Interface erhalten sollen, müssen diese Methoden implementieren.

Durch `implements` wird dem Compiler mitgeteilt, welche Schnittstellenvereinbarung eingehalten werden sollen.

```
class Konto implements BonusSparen {

    protected static double zinssatz;
    protected double kontostand;

    public double addBonus(double laufzeit){
        if (laufzeit >= LAUFZEIT)
            return kontostand+=BONUS;
        else return kontostand;
    }
```



```
    public double getKontostand(){
         return kontostand;
    }

    public void einzahlen(double betrag){
         kontostand += betrag;
    }

    public double auszahlen(double betrag){
         kontostand -= betrag;
         return betrag;
    }

    public static void setZinssatz(double wert){
         zinssatz = wert;
    }

    public void verzinsen(){
         kontostand += kontostand * (zinssatz/100);
    }
}
```



```
public class KontoApplikation{
     public static void main(String args[]){
          Konto.setZinssatz(1.8);
          Konto  k1 = new Konto();
          Konto  k2 = new Konto();
          k1.einzahlen(500);
          k2.einzahlen(1000);
          k1.verzinsen();
          k2.verzinsen();
          k1.addBonus(1.1);
          k2.addBonus(0.8);
          System.out.println(k1.getKontostand());
          System.out.println(k2.getKontostand());
     }
}
```


Bemerkung zu Interfaces

- Durch `implements` *interf1, interf2,…* können mehrere Schnittstellen implementiert werden.
- Auch eine Kombination von Vererbung und Implementierung ist möglich, `class` *Klasse1* `extends` *Klasse2* `implements` *Interface1, Interface2 …*
- Durch die Schnittstellen ist es möglich, verschiedene Klassen, die sonst keine Gemeinsamkeiten haben, die gleichen Methoden bereitzustellen, ohne zwangsweise eine allgemeine Superklasse einzuführen.

Das folgende Beispiel zeigt, wie Konstanten bei Schnittstellen vererbt werden.

```
interface Grundfarben{
      int ROT = 1, GRUEN = 2, BLAU = 3,SCHWARZ = 4;
}

interface Sockenfarben extends Grundfarben{
      int SCHWARZ = 10, BEIGE = 11;
}

interface Hosenfarben extends Grundfarben{
      int BEIGE = 11, SCHWARZ = 20,BRAUN = 21;
}

interface Allefarben extends Sockenfarben, Hosenfarben{
      int BRAUN = 30;
}
```



```
public class VererbteSchnittstellen{
    public static void main(String args[]){
        System.out.println(Grundfarben.SCHWARZ); //Ausgabe 4
        System.out.println(Sockenfarben.SCHWARZ);//Ausgabe 10
        System.out.println(Hosenfarben.SCHWARZ); //Ausgabe 20

        System.out.println(Sockenfarben.ROT);     //Ausgabe 1
        System.out.println(Allefarben.ROT);   //Ausgabe 1
    }
}
```

- Schnittstellen vererben ihre Eigenschaften an die Unter-Schnittstellen, z.B. erbt Sockenfarben die Konstanten `ROT, GRUEN, BLAU` aus `Grundfarben`.
- Konstanten dürfen überschrieben werden. So überschreiben `Sockenfarben` und `Hosenfarben` die Konstante `SCHWARZ` aus `Grundfarben`.
- Ein Aufruf von z.B. `Allefarben.SCHWARZ` ist nicht erlaubt, da dann nicht klar, ob Konstante `SCHWARZ` aus `Hosenfarben` oder `Sockenfarben` gemeint ist.

Das Interface `Cloneable`

- Durch Klonen können Objekte kopiert werden.

  Zur Erinnerung:

  Sei `Klasse1` eine Klasse, dann wird durch `Klasse1 ref1 = new Klasse1();` eine Instanz erzeugt. Das Setzen von `Klasse1 ref2 = ref1;` führt nur dazu, dass jetzt zwei Referenzen auf das selbe Objekt zeigen.

Das folgende Programmbeispiel soll das Kopieren von Referenzen noch einmal verdeutlichen

```
class Klasse1{
 int x;

 public Klasse1 (int x){
  this.x = x;
 }

 public void print(){
  System.out.println("x = "+x);
 }
}

 class KopieReferenz{
   public static void main (String args[]) {
      Klasse1 ref1 = new Klasse1(1);
      Klasse1 ref2 = ref1;
      ref1.print();
      ref2.print();
      ref2.x = 5;
      ref1.print();
      ref2.print();
   }
}
```


- Im Programm zeigen durch `Klasse1 ref2 = ref1;` die Referenzen `ref2` und `ref1` auf das selbe Objekt. D.h. sowohl durch `ref2.x` als auch durch `ref1.x` kann der Wert `x` des Objektes verändert werden.
- Hier fand also eine Kopie der Referenzen auf ein Objekt statt, nicht aber das Kopieren eines Objektes.
- Um Objekte zu kopieren, d.h. um zwei Objekte, deren Werte unabhängig voneinander verändert werden können, zu erhalten, muss das entsprechende Objekt geklont werden.

Das folgende Beispiel zeigt, wie ein Objekt geklont werden kann:


```
class Klasse1 implements Cloneable{ //Klasse erlaubt Klonen
    int x;

    public Klasse1 (int x){
        this.x = x;
    }

    public void print(){
        System.out.println("x = "+x);
    }

    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
}
```



```
class KopieTest{
        public static void main (String args[]) throws
                                CloneNotSupportedException{
        Klasse1 ref1 = new Klasse1(1);
        Klasse1 ref2 = (Klasse1) ref1.clone();
        ref1.x = 5;
        ref1.print();
        ref2.print();
    }
}
```

Das Programm hat nun eine Kopie des Objektes angelegt und die Referenz `ref2` zeigt auf dieses neue Objekt. Damit können beide Objekte unabhängig voneinander verändert werden.

Erklärung zum Clone-Beispiel

- Durch `class Klasse1 implements Cloneable` implementiert die Klasse `Klasse1` das Interface `Cloneable`.
- `Cloneable` ist eine Schnittstelle aus dem Paket `java.lang`.
- Durch das Implementieren dieser Schnittstelle zeigt die Klasse an, dass ihre Objekte kopierbar sind.
- Wird das Interface `Cloneable` implementiert, so muss innerhalb der Klasse die `clone()`-Methode der Klasse `Object` überschrieben werden, da diese mit `protected` geschützt ist. Dies erfolgt hier durch

```
public Object clone() throws CloneNotSupportedException{
         return super.clone();
}
```


- Da die Methode eine Exception vom Typ `CloneNotSupportedException` werfen kann, muss dies explizit im Methoden-Kopf angegeben werden.
- Die Methode `clone()` der Klasse `Object` erzeugt ein neues Objekt und belegt die Datenfelder mit den exakt gleichen Werten wie das Objekt, für das die Methode aufgerufen wird. Es wird eine Referenz vom Typ `Object` auf das neue Objekt zurückgegeben. Diese muss dann noch in den richtigen Typ gecastet werden (im Programm `(Klasse1) ref1.clone();`).

Flache und Tiefe Kopie

- Die clone-Methode der Klasse `Object` erzeugt eine Eins-zu-Eins Kopie von allen Datenfeldwerten. Handelt es sich allerdings bei den Datenfeldern des zu klonenden Objektes nicht um primitive Datentypen (wie im vorherigen Beispiel) sondern um Referenzdatentypen, so kopiert die `clone()`-Methode nur die Referenzen und nicht die Inhalte.
- Die `clone()`-Methode kopiert nur die Referenzen, wenn es sich um Referenzdatentypen handelt. Man spricht von einer *Flachen-Kopie*.
- Sollen aber auch die Inhalte kopiert werden, so spricht man von einer *Tiefen-Kopie*. Dann reicht es nicht aus, in der überschriebenen Methode `clone()` mit `super.clone()` die Methode von `Object` aufzurufen.

Das folgende Beispiel zeigt zunächst eine Flache Kopie.

Ein Objekt der Klasse `Klasse1` soll geklont werden. `Klasse1` besteht aus der primitiven Variable `z` sowie aus der Referenz `ref`, die auf ein Objekt vom Typ `Klasse2` zeigt.

Durch das Klonen wird eine Kopie von z angelegt und die Referenz kopiert, nicht aber der Inhalt des Objektes vom Typ `Klasse2`.

```
//1.flaches Kopieren
class Klasse2 {
    int x = 1;
    int y = 1;
}


class Klasse1 implements Cloneable{
    int z = 1;
    Klasse2 ref = new Klasse2();

    public void print(){
       System.out.println("z = "+z + " x = "+ref.x + " y = "+ ref.y);
    }

    public Object clone() throws CloneNotSupportedException{
         return super.clone();
    }
}
```



```
class KopieTest{

    public static void main (String args[]) throws
                                CloneNotSupportedException{
        Klasse1 orig = new Klasse1();
        Klasse1 kopie = (Klasse1) orig.clone();
        kopie.z = 5;
        System.out.println("Ausgabe nach Veraenderung von z:");
        orig.print();
        kopie.print();
        kopie.ref.x = 2;
        System.out.println("Ausgabe nach Veraenderung von x:");
        orig.print();
        kopie.print();
    }
}
```


Das Programm liefert

```
Ausgabe nach Veraenderung von z:
z = 1 x = 1 y = 1
z = 5 x = 1 y = 1
Ausgabe nach Veraenderung von x:
z = 1 x = 2 y = 1
z = 5 x = 2 y = 1
```

d.h. `z` als primitive Variable wurde in der Kopie verändert, eine Veränderung von `x` in der Kopie führt auch gleichzeitig zum Veränderung von x im Original, da die Variable `ref` eine Referenz auf ein Objekt ist, das auch den Variablen `x` und `y` besteht.

Das folgende Beispiel zeigt, wie eine tiefe Kopie erstellt wird, d.h. eine Kopie, bei der auch die Inhalte des Objektes kopiert werden:


```
//2. Version Tiefes Kopieren
class Klasse2 implements Cloneable{
    int x = 1;
    int y = 1;

    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
}
```



```
class Klasse1 implements Cloneable{
    int z = 1;
    Klasse2 ref = new Klasse2();


    public void print(){
       System.out.println("z = "+z + " x = "+ref.x + " y = "+ ref.y);
    }

    public Object clone() throws CloneNotSupportedException{
       Klasse1 hilf = (Klasse1) super.clone(); //flaches Kopieren
       hilf.ref = (Klasse2) ref.clone();//Kopieren der Objekte,

                                           //auf die Referenz zeigt
       return hilf;
    }
}
```



```
class KopieTest{

    public static void main (String args[]) throws
                                CloneNotSupportedException{
        Klasse1 orig = new Klasse1();
        Klasse1 kopie = (Klasse1) orig.clone();
        kopie.z = 5;
        System.out.println("Ausgabe nach Veraenderung von z:");
        orig.print();
        kopie.print();
        kopie.ref.x = 2;
        System.out.println("Ausgabe nach Veraenderung von x:");
        orig.print();
        kopie.print();
   }
}
```


Jetzt liefert das Programm

```
Ausgabe nach Veraenderung von z:
z = 1 x = 1 y = 1
z = 5 x = 1 y = 1
Ausgabe nach Veraenderung von x:
z = 1 x = 1 y = 1
z = 5 x = 2 y = 1
```

d.h. auch `x` wurde in der Kopie verändert. Hier wurde also nicht die Referenz kopiert, sondern tatsächlich die Inhalte.

Pakete

- Ein Paket stellt eine Klassenbibliothek dar, die einen Namen trägt.
- Eine Paket kann Klassen, Threads (werden später behandelt) Schnittstellen und Unterpakete als Komponenten haben.
- Durch Pakete erhält man größere Einheiten, die für eine Strukturierung von großen objektorientierten Systemen wichtig sind.
- Jedes Paket hat einen eigenen Namen, daher können Klassennamen unterschiedlich vergeben werden. Durch den Paketnamen wird der Klassenname eindeutig.
- Beispiele für ein Paket ist `java.lang`
- Die Pakete in Java sind nach Aufgabenbereiche sortiert.

Erstellen von Paketen

- Zum Anlegen von Paketen werden alle Dateien des Paketes mit der Deklaration des Paketnamens versehen.
- Die Deklaration eines Paketnamens erfolgt mit dem Schlüsselwort `package` gefolgt vom Paketnamen.
- Die Deklaration des Paketnamens muss als erstes in der Datei stehen, nur Kommentare dürfen vorangestellt werden.
- Konvention: Paketnamen werden klein geschrieben.

Beispiel für das Erstellen eines Paketes. Sowohl die Klasse `Paketklasse1` als auch die Klasse `Paketklasse2` gehören jetzt zum Paket `meinPaket.ausgabe`

```
//Paketklasse1.java
package meinPaket.ausgabe;

public class Paketklasse1{
   public static void ausgabe(){
      System.out.println("Hier ist Methode ausgabe1()");
  }
}


//Paketklasse2.java
package meinPaket.ausgabe;

public class Paketklasse2{
   public static void ausgabe(){
      System.out.println("Hier ist Methode ausgabe2()");
   }
}
```


- Der Byte-Code, d.h. die -Dateien `Paketklasse1.class` und `Paketklasse2.class` müssen –vom Arbeitsverzeichnis betrachtet- im Verzeichnis `meinPaket\ausgabe` stehen.
- Pakete können benutzt werden, wenn entweder der vollständige Klassenname angegeben oder das Paket importiert wird.
- Mit `import` *Paketname.Klassenname* können einzelne Klassen des Paketes importiert werden, mit `import` *Paketname.** werden alle Klassen des Paketes importiert.
- Beispiel

```
import meinPaket.ausgabe.*;
public class Test{
    public static void main(String[] args){
    Paketklasse1.ausgabe();
    }
}
```


Die Klasse `Object`

- Die Klasse `Object` ist in der Vererbungshierarchie die höchste Klasse. Sie ist die Elternklasse für alle anderen Klassen.
- Immer wenn eine Klasse definiert wird, dabei aber keine explizite Ableitung mittels `extends` angegeben wird, fügt Java automatisch `extends Object` hinzu.
- Zwei Methoden der Klasse `Object`, die wir bereits kennen gelernt haben sind:

`clone()`	erzeugt eine Kopie des Objektes
`equals`(Object)	Überprüfung auf Gleichheit

Zeichenketten: Die Klassen `String` und `StringBuffer`

- Strings, d.h. Zeichenketten sind in Java Objekte.
- In Java gibt zwei Klassen von Strings, dies sind die Klassen `String` und `StringBuffer`.
- Objekte der Klasse `String` stellen Zeichenketten mit fester Länge, d.h. konstante Zeichenketten, dar.
- Objekte der Klasse `StringBuffer` stellen Zeichenketten dar, die zur Laufzeit des Programms verkürzt oder verlängert werden können. Zudem gibt es verschiedene Methoden zur Textbearbeitung.
- Die Klassen `String` und `StringBuffer` sind aus dem Paket `java.lang`

Die Klasse `String`

Objekte der Klasse sind konstante Zeichenketten.

Sie haben folgende Eigenschaften:

- Die Länge eines Strings steht fest und kann nicht mehr verändert werden.
- Der Inhalt des Strings kann nicht verändert werden.

Eine konstante Zeichenkette mit beispielsweise dem Inhalt „Peter" wird entweder durch die Anweisung

`String name = "Peter";`

oder durch die Anweisung

`String name = new ("Peter");`

erzeugt.

`name` ist also in beiden Fällen eine Referenz auf ein Objekt.

Im ersten Fall spricht man von einer impliziter Erzeugung eines String-Objektes.

Der Unterschied ist, dass im zweiten Fall durch den `new`-Operator auf jeden Fall ein neues String-Objekt erzeugt wird und damit Speicher belegt wird.

Im ersten Fall muss dies nicht unbedingt der Fall sein.

Bei optimierenden Compilern können String-Objekte, die den gleichen Inhalt haben und implizit erzeugt werden, wieder verwendet werden.

Beispiel: Durch

`String name = "Peter";`

`String name2 = "Peter";`

erzeugen optimierende Compiler nur ein Objekt, auf das die zwei Referenzen `name` und `name2` beide zeigen.

Zur Erinnerung:

Strings werden mit der Methode `equals()` verglichen.

```
if (name.equals(name2))
  .......
```

vergleicht die Inhalte der beiden Strings `name` und `name2`.

Auf den nächsten Folien werden einige weitere String-Methoden erläutert:

Länge eines Strings durch `length()`

Die Methode `length()` ermittelt die Länge eines Strings.

```
String a = "Grundlagen";
int l = a.length();
```

ergibt l = 10

Anhängen an Strings mit Hilfe der Methode `concat()`

```
String b = "Informatik";
String titel = "Titel der Vorlesung: ";
titel = titel.concat(a);
titel = titel.concat(" ");
titel = titel.concat(b);
```

Damit ergibt `System.out.println(titel);`
Titel der Vorlesung: Grundlagen Informatik

Stringteile extrahieren mit `charAt()` und `Substring()`

Die Methode `charAt(int index)` liefert das entsprechende Zeichen an der

Stelle, die „index“ genannt wird.

```
String thema = "Java";

char e = thema.charAt(1);
```

liefert das zweite Zeichen (1. Zeichen hat Position 0), d.h. `e` hat den Wert `a`.

```
char e = thema.charAt(thema.length()-1);
```

liefert das letzte Zeichen, d.h. `e` hat den Wert `a`.

Die Methode `substring(a,b)` liefert einen neuen String als Teilstring des Originals. a gibt die Startposition an, b die erste Position, die nicht mehr zum

Substring gehören soll.

```
String subtitel = thema.substring(1,3);

System.out.println(subtitel);
```

ergibt als Ausgabe

av

Anfang und Ende des Strings mit `startsWith()` und `endsWith()`

Die Methode `startsWith(`*anfang*`)` überprüft, ob der String mit *anfang* anfängt.

```
boolean s = titel.startsWith("Titel");

System.out.println(s);
```

ergibt also true.

Die Methode `endsWith(`*ende*`)` überprüft, ob der String mit *ende* aufhört.

```
boolean ende = titel.endsWith ("Informatik");

System.out.println(ende);
```

ergibt true.

Zeichen im String suchen mit `indexOf()`

Die Methode `indexOf(`*zeichen*`)` gibt die Position aus, an der das *zeichen* das erste mal im String vorkommt. Mit `String thema = "Java";`

```
int p = thema.indexOf ('a');
```

ergibt sich für p der Wert 1.

Ist das Zeichen im String nicht enthalten, so liefert die Methode den Wert –1.

Suchen eines Zeichens ab einer bestimmten Position wird durch Angabe eines weiteren Parameters ermöglicht:

```
int p = thema.indexOf ('a',2);
```

ergibt für p den Wert 3.

Zeichen im String suchen -wobei Suche von hinten beginnt- mit `lastindexOf()`

```
int p = thema.lastIndexOf('a');
```

ergibt für p den Wert 3.

Neben einzelnen Zeichen können auch Zeichenketten gesucht werden:

```
int p = titel.lastIndexOf("av");
```


Strings vergleichen, ohne dass auf Groß- und Kleinschreibung geachtet wird: die Methode `equalsIgnoreCase()`

```
String a = "Informatik";
String b = "informatik";

boolean vergleich = a.equalsIgnoreCase(b));
```

liefert für `vergleich` den Wert true.

Stringteile vergleichen mit `regionMatches()`

Mit dieser Methode können Teile einer Zeichenkette mit Teilen einer anderen Zeichenkette verglichen werden.

```
String s1 = "Grundlagen Informatik";
String s2 = "Informatik im 1. Semester";
boolean v = s1.regionMatches(true,11,s2,0,10);
```

ergibt für `v` den Wert true (verglichen wird `Informatik` aus `s1` mit `Informatik` aus `s2`).

Der erste Parameter (`true`) gibt an, dass auf Groß- und Kleinschreibung kein Wert gelegt wird. Wird er weggelassen, so spielt sie eine Rolle.

Der zweite Parameter gibt die Position im String s1 an, ab der verglichen werden soll, es folgt der Vergleichsstring mit der Position. Der letzte Parameter gibt die Anzahl der zu vergleichenden Zeichen an.

Zeichen ersetzen mit `replace()`

Die Methode erzeugt ein neuen String indem die Zeichen aus dem Original ersetzt wurden.

```
String s1 = "Java";
String s2 = s1.replace('a', 'A' );
```

liefert für `s2` JAvA, d.h. das `a` wurden in `A` verändert. `s1` bleibt unverändert.

Groß- und Kleinschreibung:
die Methoden `toUpperCase()` und `toLowerCase()`

Die Methoden erzeugen einen neuen String, in dem alle Kleinbuchstaben in Großbuchstaben verwandelt werden (`toUpperCase()`) bzw. alle Großbuchstaben in Kleinbuchstaben gewandelt werden (`toLowerCase()`).

```
String s1 = "Java";
String s2 = s1.toUpperCase();
```

ergibt für `s2` JAVA,

```
String s2 = s1.toLowerCase();
```

ergibt für `s2` java.

Vergleich zweier Strings auf „Größe“ mit `compareTo()`

Durch den Methodenaufruf `string1.compareTo(string2)` werden die beiden Strings `string1` und `string2` auf ihre lexikographische Größe verglichen.

Der Rückgabewert ist vom Datentyp `int`. Er ist `0` bei Gleichheit, negativ, falls `string1` lexikographisch kleiner ist als `string2` und positiv, falls `string1` lexikographisch größer als `string2` ist.

```
String string1 = "Peter", string2 = "Tina",
       string3 = "Peter" , string4 = "Andrea";
```

Dann ist

`string1.compareTo(string2));` kleiner als 0

`string1.compareTo(string3));` gleich 0

`string1.compareTo(string4));` größer als 0

Leerzeichen am Anfang und Ende entfernen mit `trim()`

Die Methode entfernt Leerzeichen am Anfang und Ende des Strings.

```
String s = "  1234   ".trim();
int i = Integer.parseInt(s);
```


Die Klasse `StringBuffer`

`StringBuffer`-Objekte sind Zeichenketten mit folgenden Eigenschaften:

- Die Länge der Zeichenkette ist nicht festgelegt. Sie vergrößert sich automatisch, wenn im StringBuffer-Objekt weitere Zeichen angefügt werden und der vorhandene Platz nicht ausreicht.
- Der Inhalt einer Instanz der Klasse `StringBuffer` lässt sich verändern.

Das Erzeugen eines `StringBuffer`-Objektes

`StringBuffer`-Objekte müssen immer mit Hilfe des `new`-Operators erzeugt werden.

Es gibt insgesamt drei Konstruktoren für die Klasse `StringBuffer`.

- **`StringBuffer()`**
  Erzeugt `StringBuffer`-Objekt, das zunächst 16 Zeichen aufnehmen kann
- **`StringBuffer(int length)`**
  Erzeugt `StringBuffer`-Objekt, das `length` Zeichen aufnehmen kann
- **`StringBuffer(String str)`**
  Es wird ein `StringBuffer`-Objekt erzeugt und mit einer Kopie des Strings `str` initialisiert. Zusätzlich wird bereits Platz für 16 weitere Zeichen eingeplant.

Die Länge eines `StringBuffer`-Objektes lesen und setzen

Unterschieden wird zwischen den Methoden `length()` und `capacity()`.

- Die Methode `length()` ermittelt die Anzahl der enthaltenen Zeichen.
- Die Methode `capacity()` ermittelt die Kapazität, d.h. den gesamten Speicherplatz (belegter und freier Speicherplatz). Diese Kapazität wird automatisch angepasst, wenn weitere Zeichen eingebaut werden. Man kann sie aber auch manuell einstellen.

Beispiel

```
StringBuffer text1 = new StringBuffer();
StringBuffer text2 = new StringBuffer(20);
StringBuffer text3 = new StringBuffer("Vorlesung");

System.out.println(text1);
System.out.println(text1.length());      //Ausgabe 0
System.out.println(text1.capacity());    //Ausgabe 16

System.out.println(text2);
System.out.println(text2.length());      //Ausgabe 0
System.out.println(text2.capacity());    //Ausgabe 20

System.out.println(text3);               //Ausgabe „Vorlesung"
System.out.println(text3.length());      //Ausgabe 9
System.out.println(text3.capacity());    //Ausgabe 25
```


Zeichen anhängen, einfügen, löschen

- Die Methode `append()` hängt Werte an eine Zeichenkette an. Der interne Platz wird dabei –falls nötig- automatisch vergrößert. Ein neues StringBuffer-Objekt wird nicht erzeugt.
- Durch die Methode `insert()` können Zeichenketten und primitive Datentypen an einer beliebigen Stelle eingefügt werden. Der erste Parameter der Methode gibt die gewünschte Position an (1. Zeichen hat Position 0). Der zweite Parameter gibt den Inhalt an.
- Die Methode `setCharAt()` überschreibt einzelne Zeichen. Der erste Parmameter gibt die Position des zu ersetzenden Zeichens an, der zweite Parameter gibt das neue Zeichen an.

- Die Methode `reverse()` dreht die Reihenfolge der Zeichen um.
- Die Methode `delete()` löscht einen Teil der Zeichenkette. Der erste Parameter gibt die Position des ersten zu löschenden Zeichens an, der zweite Parameter gibt die Position des ersten Zeichens an, das nicht mehr gelöscht werden soll.

Beispiel

```
class BeispielStringBuffer{
    public static void main (String args[]){
    StringBuffer text3 = new StringBuffer("Vorlesung");

    text3.append(" Java");
    System.out.println(text3);

    text3.insert(9," Ueber");
    System.out.println(text3);

    text3.setCharAt(10,'u');
    System.out.println(text3);

    text3.reverse();
    System.out.println(text3);

    text3.delete(4,10);
    System.out.println(text3);

    text3.reverse().append(" Teil").append(" 2");
    System.out.println(text3);
    }
}
```

Ausgabe des Programms

```
Vorlesung Java
Vorlesung Ueber Java
Vorlesung ueber Java
avaJ rebeu gnuselroV
avaJ gnuselroV
Vorlesung Java Teil 2
```


Vergleiche von Zeichenketten als String und StringBuffer

Um zwei StringBuffer-Objekte bzw. ein StringBuffer- und ein String-Objekt auf Gleichheit zu überprüfen, muss das/die StringBuffer-Objekte in Strings umgewandelt werden:

- Umwandeln des/der StringBuffer-Objekte in Strings mit Hilfe der Methode `toString()` und danach Vergleich mit `equals()`:

```
StringBuffer text1 = new StringBuffer("Java");
String text2 = "Java";
System.out.println(text2.equals(text1.toString()));
```


Vergleiche von Zeichenketten als String und StringBuffer

Der Vergleich eines StringBuffer-Objektes mit einem String kann seit Version 1.4 auch mit Hilfe der Methode `contentEquals()` erfolgen:

- Benutzung der Methode `contentEquals(StringBuffer)`

```
System.out.println(text2.contentEquals(text1));
```


Ein weiteres Beispiel

Testen, ob der String `s` ein Palindrom ist.

Palindrome lesen sich von vorne genauso wie von hinten, z.B. OTTO

```
import javax.swing.*;
class Palindrom{
    public static void main (String args[]){
       String s = JOptionPane.showInputDialog (null,
                              "Geben Sie einen String ein");
    boolean isPalindrom =
           s.contentEquals(new StringBuffer(s).reverse());

    if (isPalindrom)
           System.out.println(s+" ist ein Palindrom");
    else
           System.out.println (s +" ist kein Palindrom");

    System.exit(0);
  }
}
```


Bemerkung zum Palindrom-Beispiel

Die Zeile

```
boolean isPalindrom =
          s.contentEquals(new StringBuffer(s).reverse());
```

kann auch ersetzt werden durch

```
boolean isPalindrom =
     new StringBuffer(s).reverse().toString().equals(s);
```

Ein- und Ausgabe in Java

Es gibt unterschiedlichste Datenquellen für die Eingabe (z.B. lokale Dateien, Tastatur, Dateien über das Internet) und unterschiedlichste Datensenken (z.B. Bildschirm, Dateien) in die die Ausgaben geschrieben werden.

Auch sollen Informationen in verschiedenen Einheiten wie z.B. Byte, Zeichen oder Zeilen ausgetauscht werden.

Die Ein- und Ausgabe wird in Java durch die Verwendung von Streams gelöst.

Streams

- Ein Stream ist eine geordnete Folge von Bytes (Bytestrom).
- Ein Stream, der aus einer Datenquelle kommt, wird als Inputstream bezeichnet,
- ein Stream, der in eine Datensenke hineingeht, wird als Outputstream bezeichnet.

In dem Paket `java.io` bündelt das JDK die Stream-Klassen.

Die Einteilung der Stream-Klassen

Alle Stream-Klassen sind von einer der vier abstrakten Basisklassen

`InputStream`, `OutputStream`, `Reader` oder `Writer` abgeleitet.

Klassen, die von `InputStream` oder `OutputStream` abgeleitet sind, arbeiten Byte-orientiert, d.h. sie sind für die Verarbeitung einzelner Bytes (also Werte vom Typ `byte`) verantwortlich. Man bezeichnet sie daher als Bytestream-Klassen.

Da Java zur internen Darstellung von Zeichen Unicode verwendet, gibt es die Klassen `Reader` und `Writer`.

Klassen, die von `Reader` oder `Writer` abgeleitet sind, bezeichnet man als Characterstream-Klassen, da sie verantwortlich für die Verarbeitung von Zeichen (d.h. Werte vom Typ `char`) sind.

Sink- und Springstreams

Die Byte- und Characterstreams können noch weiter anhand ihrer Funktionalität eingeteilt werden:

- Sinkstream-Klasse: Ein Objekt dieser Klasse kann Daten in eine Datensenke schreiben.
- Springstream-Klasse: Ein Objekt dieser Klasse kann Daten direkt aus einer Datenquelle lesen.
- Processingstream-Klasse: Zusätzlich gibt es Filter für die jeweiligen Streams, die jeweils eine Zusatzfunktionalität implementieren, z.B. Puffern von Werten, Einlesen von integer-Werten.

Wie werden Streams erstellt und verwendet?

- Erstellen eines Objektes, das mit Datenquelle oder Datenziel assoziiert ist

  Beispiel:

  Datenquelle ist eine Datei, dann kann ein Eingabestrom erzeugt werden durch

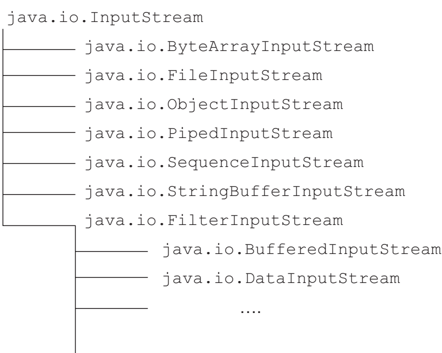
  ```
  FileInputStream fis = new FileInputStream("Test.txt");
  ```
- Benutzen der Methoden von `FileInputStream`, um Daten aus dem Stream zu lesen.
- Schließen des Streams mit `close()`

Eingabe in Java: Byteorientierter Zugriff

Die Klasse `InputStream` dient als Abstraktion für alle konkret möglichen ByteStreams. Man kann einzelne Bytes oder Blöcke von Bytes lesen.

Es gibt keine Methode zur Umformung von Daten.

Die folgende Skizze zeigt einen Ausschnitt der Klassenhierarchie der Byte-Inputstream-Klassen:



Zwei Methoden der Klasse `InputStream`

`void close ();`	Schließen des Streams und Freigabe aller belegten Ressourcen
`int read();`	Lesen eines Bytes. Das Ergebnis wird als Zahl im Bereich von 0 bis 255 zurückgeliefert. Der Wert –1 steht für das Ende der Datei.

Was ist die Aufgabe der Processing-Stream-Klassen?

Sie implementieren jeweils eine Zusatzfunktionalität zur Eingabe.

Zur Anwendung wird jeweils eine Instanz einer solchen Klasse angelegt, die irgendeinen konkreten Eingabestrom benutzt:

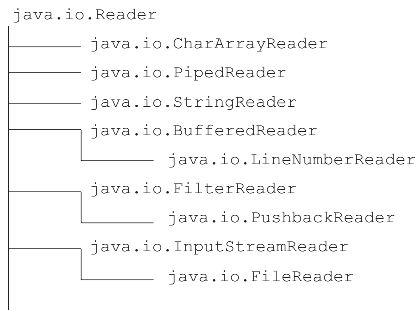
Beispiel

```
FileInputStream fis = new FileInputStream ("Dateiname");
DataInputStream dis = new DataInputStream(fis);

short zahl = dis.readShort();
```


Eingabe in Java: Zeichenorientierter Zugriff

Die Klasse `Reader` dient als Abstraktion für alle konkret möglichen Character-InputStreams.

Die folgende Skizze zeigt die Klassenhierarchie der Character-Inputstream-Klassen:



Erzeugen eines Readers

Der folgende Programmausschnitt zeigt, wie man einen Reader für eine Datei erzeugt:

1. Möglichkeit

```
BufferedReader b = new BufferedReader(
                   new InputStreamReader(
                   new FileInputStream("dateiname")));
```

2. Möglichkeit
   Alternativ kann auch die Klasse `FileReader` als Abkürzung benutzt werden:

```
BufferedReader b = new BufferedReader(
                   new FileReader("dateiname"));
```


Einige wichtige Methoden der Klasse `BufferedReader`

`void close ();`	Schließen des Streams und Freigabe aller belegten Ressourcen
`int read();`	Lesen eines Zeichens. Das Ergebnis wird als Zahl im Bereich von 0 bis 65535 zurückgeliefert. Der Wert –1 steht für das Ende der Datei.
`String readLine();`	Einlesen einer Zeile. Bei Dateiende wird `null` geliefert.

Beispiel: Lesen aus einer Textdatei mit Namen `test.txt`

```
import java.io.*;

public class DateiEinlesen {

 public static void main(String args[])
                 throws java.io.IOException {
      BufferedReader b = new BufferedReader
                   (new FileReader ("test.txt"));


  String s = null;

  while ((s = b.readLine()) != null)
   System.out.println (">" + s + "<");
 }

}
```


Die Standard-Eingabe

Java stellt Standard-Ein- und Ausgaben zur Verfügung.

Die Standard-Eingabe (und auch die Standard-Ausgabe) verwendet einen Bytestream, um Daten von der Tastatur einzulesen.

Über die Referenz `in` können alle Methoden, die die Klasse `InputStream` zur Verfügung stellt, aufgerufen werden.

`in` ist eine Klassenvariable der Klasse `System`.

Beispiel: Lesen von Tastatur

```
import java.io.*;

public class MyReadLine {
      static BufferedReader b = null;

  public static String readln() throws java.io.IOException{
       if (b == null)
       b = new BufferedReader( new InputStreamReader (System.in));
       return b.readLine();
  }

  public static void main (String [] args)
                           throws java.io.IOException{
    String s;
    while ((s = MyReadLine.readln ()) != null)
           System.out.println (">" + s + "<C);
  }

}
```

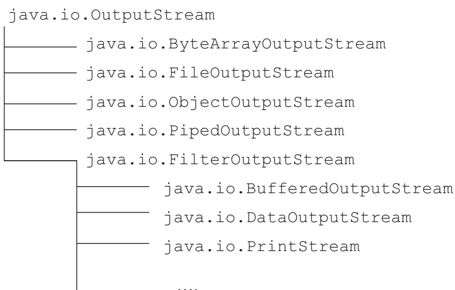

Ausgabe in Java: Byteorientierter Zugriff

Die Klasse `OutputStream` dient als Abstraktion für alle konkret möglichen ByteStreams.

Drei Methoden der Klasse OutputStream

`void close ();`	Schließen des Streams und Freigabe aller belegten Ressourcen
`abstract void write(int b);`	Schreiben einzelner Bytes
`void flush();`	gepufferte Daten werden geschrieben

Die folgende Skizze zeigt einen Ausschnitt der Klassenhierarchie der Byte-Outputstream-Klassen:



Die Standard-Ausgabe

Java stellt Standard-Ein- und Ausgaben zur Verfügung.

Über die Referenz `out` können alle Methoden, die die Klasse `PrintStream` zur Verfügung stellt, aufgerufen werden.

`out` ist eine Klassenvariable der Klasse `System`.

```
static PrintStream out
```

Die wichtigsten (und schon bekannten) Methoden sind `print()` und `println()`.

Die Klasse `PrintStream`

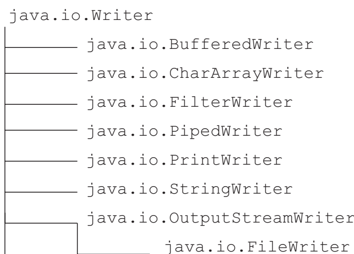
- `PrintStream` ist eine wichtige Klasse für einen Output
- Sie ist bekannt aus `Sytem.out.println()`,
  kann aber auch selbst benutzt werden.
- `System.out` stellt den Standard Output-Stream dar.
- Eine eigene Verwendung finden Sie auf der nächsten Folie.


```
import java.io.*;

public class EigenerPrintStream{
      public static void main(String args[])
                  throws java.io.IOException {
            int i = 10;
            double d = 1.3;

            PrintStream p = new PrintStream(System.out);
            p.println(i);
            p.println(d);

            p.close();
      }
}
```


Ausgabe in Java: Zeichenorientierter Zugriff

Die Klasse `Writer` dient als Abstraktion für alle konkret möglichen Character-OutputStreams.

Die folgende Skizze zeigt die Klassenhierarchie der Character-Outputstream-Klassen:



Beispiel: Schreiben in eine Textdatei test2.txt

```
import java.io.*;

public class DateiSchreiben {

 public static void main(String args[])
                        throws java.io.IOException {
        PrintWriter p =  new PrintWriter(
                        new FileWriter ("test2.txt"));

        p.println("In eine Datei schreiben");
        p.flush();

        p.close();
 }

}
```


Zusammenfassung über die Sink- und Springstream-Klasse

Datenquelle/-senke	zeichenorientiert	byteorientiert
char-Array intern	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
String-Objekt intern	StringReader StringWriter	StringBufferInputStream ----
Pipe intern	PipedReader PipedWriter	PipedInputStream PipedOutputStream
Datei extern	FileReader FileWriter	FileInputStream FileOutputStream

Zusammenfassung der Processingstream-Klassen

Funktion	zeichenorientiert	byteorientiert
Pufferung	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filterung	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Konvertierung zwischen Bytes und Zeichen	InputStreamReader OutputStreamWriter	
Verkettung von Streams	----	SequenzInputStream
Ein-/Ausgabe von Daten primitiver Typen	-----	DataInputStream DataOutputStream
Zeilen zählen	LineNumberReader	LineNumberInputStream
Textausgabe	PrintWriter	PrintStream

Ein weiteres Beispiel: Schreiben und Lesen einzelner Bytes

```
import java.io.*;

public class BytesLesenuSchreiben {

  public static void main(String args[])
                      throws java.io.IOException {
   FileOutputStream fos = new FileOutputStream ("zahlen.txt");
   for (int i = 0; i < 10; i++){
         fos.write(i);
   }
   fos.close();

   FileInputStream fis = new FileInputStream("zahlen.txt");
   for (int i = 0; i< 10;i++)
        System.out.print(fis.read());

   fis.close();

  }
}
```


Datenstrukturen in Java

- Algorithmen und Datenstrukturen sind ein zentrales Thema in der Informatik.
- Algorithmen operieren nur dann effektiv mit Daten, wenn diese für einen Algorithmus geeignet strukturiert sind.
- Die Wahl der richtigen Datenstruktur entscheidet über effiziente Laufzeiten der Algorithmen.

Definition

Eine Datenstruktur ist eine Sammlung von Knoten derselben Klasse oder desselben Typs, deren Organisation und Zugriff klar definiert sind.

Ein Knoten kann ein Objekt oder auch primitive Datentypen, wie eine Zahl sein.

Wir werden uns mit den linearen Datenstrukturen beschäftigen, wobei linear bedeutet, dass es zu jedem Element (Knoten) höchstens einen Vorgänger und höchstens einen Nachfolger gibt.

Arrays

- Ein Array ist die einfachste lineare Datenstruktur.
- In einem Array wird eine endliche Menge von Elementen des selben Typs gespeichert.
- Die einzelnen Elemente heißen Komponenten und können über Indizes angesprochen werden.
- Man bezeichnet die k-te Komponente eines Arrays a mit a[k] oder a_k
- In Java werden die Komponenten eines Arrays ab 0 nummeriert.

Bemerkung zu Arrays

- Die Größe eines Arrays wird zum Zeitpunkt der Erstellung des Arrays festgelegt, danach kann sie nicht mehr verändert werden.

⇨ Ein Array ist eine lineare, statische Datenstruktur.

Nachteile eines Arrays

- Belegung von viel Speicherplatz bei großen Datenmengen
- u. U. aufwendiges Einfügen von Elementen
- Array kann „voll“ sein

Lineare dynamische Datenstrukturen

Dynamische Datenstrukturen passen ihre Größe der Anzahl der Daten an, die sie aufnehmen.

Eine vordefinierte dynamische Datenstruktur in Java ist die Klasse `Vector`.

Die Klasse `Vector`

- Vector ist eine lineare dynamische Datenstruktur
- Die Klasse Vector liegt im Paket `java.util`
- Jedes Exemplar der Klasse Vector vertritt ein Array mit variabler Länge.
- Ein Objekt von Vector vergrößert sich automatisch, wenn neue Elemente hinzukommen.

Erzeugen eines Vektors

Es gibt drei Möglichkeiten, eine Instanz von Vector anzulegen

- `Vector v = new Vector();`
  Ein leerer Vektor mit einer Anfangskapazität von 10 Elementen wird angelegt
- `Vector v = new Vector (int kapazität)`
  Ein leerer Vektor wird angelegt, der Platz für zunächst `kapazität` Elemente bietet
- `Vector v = new Vector (int kapazität, int kapazitätsZuwachs)`
  Ein Vektor mit zunächst Platz für `kapazität` Elemente wird angelegt. Zusätzlich beschreibt die Zahl `kapazitätsZuwachs`, um wie viele Elemente die Kapazität des Vektors jedes Mal erhöht wird, wenn der verfügbare Platz erschöpft ist.

- Ein `Vector` ist nicht auf einen bestimmten Datentyp ausgelegt, d.h. ein Objekt `Vector` kann unterschiedliche Datentypen beinhalten.
- Ein Vector kann alle Objekte als Datentyp haben, nicht aber direkt primitive Datentypen.

Einige Methoden der Klasse `Vector`

- `void add (Object o)`
  Fügt `Object o` in Vektor als letztes Element
- `boolean remove (Object o)`
  Entfernt das erste mit o übereinstimmende Element aus Vektor
- `int size()`
  Gibt Anzahl der Elemente im Vektor zurück
- `int capacity ()`
  Gibt Kapazität des Vektors an.
- `void set (int index, Object o)`
  Das Objekt `o` wird an Stelle `index` im Vektor platziert. Das vorher an dieser Position befindliche Element wird aus dem Vektor entfernt.

Beispiel für die Erstellung eines Vektors

```
import java.util.*;
public class VectorTest{

      public static void main(String args[])  {
             Vector vec = new Vector();
             vec.add ("Kai");
             vec.add ("Monika");
             vec.add ("Karla");
             vec.add ("Kai");
             System.out.println(vec);             [Kai, Monika, Karla, Kai]

             vec.add (1, "Ute");                  [Kai, Ute, Monika, Karla, Kai]
             System.out.println(vec);

             vec.remove  ("Kai");                 [Ute, Monika, Karla, Kai]
             System.out.println(vec);
             vec.remove(0);
             System.out.println(vec);             [Monika, Karla, Kai]
```



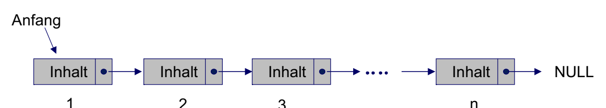
```
            //ein weiteres Vektorelement ist ein Integer Wert
            vec.set (1,new Integer(3));           [Monika, 3, Kai]

            System.out.println(vec);
            System.out.println(vec.size());
            System.out.println(vec.capacity());

      }
}
```

Zusammenfassend ergibt das Programm

```
[Kai, Monika, Karla, Kai]
[Kai, Ute, Monika, Karla, Kai]
[Ute, Monika, Karla, Kai]
[Monika, Karla, Kai]
[Monika, 3, Kai]
3
10
```


Einfach verkettete Liste

- Eine einfach verkettete Liste ist eine lineare dynamische Datenstruktur.
- Sie besteht aus einer Folge von Zellen / Knoten.
- Jeder Knoten enthält neben dem „Daten-Element" einen Zeiger (Referenz) Next auf den folgenden Knoten der Liste.
- Die Liste hat einen Anfangszeiger, der auf den ersten Knoten zeigt.
- Der Zeiger des letzten Knotens zeigt auf NULL.



Entfernen und Einfügen eines Elementes erfolgen durch einfaches Kopieren der Referenzen.

Durch Garbage Collection wird die entfernte Zelle aus dem Speicher entfernt.

Zunächst betrachten wir eine eigene Implementierung einer verketteten Liste.

Anschließend untersuchen wir, was in Java als Standard hierfür gegeben ist.

Knoten einer Liste in Java

```
public class Knoten {
       Object inhalt;
       Knoten next;

       // Konstruktor:
       Knoten (Object inhalt) {
              this.inhalt = inhalt;
       }
       Knoten (Object inhalt, Knoten next) {
              this.inhalt = inhalt;
              this.next = next;
       }
}
```

`next` ist wieder vom Datentyp `Knoten`

Implementierung einer verketteten Liste

```
public class Liste {
    private Knoten anfang;
    private Knoten cursor;

    //Ermittlung der Laenge der Liste
    int laenge () {
    Knoten cur = anfang;
    int l = 0;
    while (cur != null) {
        l++;
        cur = cur.next;
        }

    return l;
    }
```



```
//Hilfsmethoden zum korrekten Setzen der Referenz Cursor
boolean istGueltigePosition (int p) {
    return (p >= 1) && (p <= laenge () );
}

void setzeCursor (int p){
    cursor = null;
    if (istGueltigePosition (p) ) {
        Knoten cur = anfang;
        for (int i = 1; i < p;i++)
            cur = cur.next;
        cursor = cur;
    }
}

//Methode, die Inhalt des Objektes eines Knotens liefert
Object inhalt (int p){
    setzeCursor (p);
        return cursor.inhalt;
}
```



```
// Methode zum Suchen eines Objektes
int suche (Object e){
    cursor = null;
    int p = 0;
    Knoten z = anfang;
    int l = 0;
    while (z != null) {
        l++;
        if (z.inhalt.equals(e))
            {p = l; cursor = z; break;}
        z = z.next;
    }
    return p;
}

//Methode zum Löschen des p-ten Knotens in Liste
void loesche (int p) {
    if (istGueltigePosition(p)){
        if (p == 1) // Loesche 1. Zelle
            anfang = anfang.next;
        else {
            setzeCursor(p-1);
            cursor.next = cursor.next.next;
        }
    }
}
```



```
//Methoden zum Einfügen eines Objektes in Liste
void einsetzenNach (int p, Object e){
    setzeCursor (p);
    if (cursor != null) {
        Knoten z = new Knoten (e, cursor.next);
        cursor.next = z;
    }
}

void einsetzenVor (int p, Object e){
    if (p > 1)
        einsetzenNach (p-1,e);
    else {// Einsetzen am Anfang
        Knoten z = new Knoten (e, anfang);
        anfang = z;
    }
}

void einsetzenAnfang (Object e){
    Knoten z = new Knoten (e,anfang);
    anfang = z;
}
}
```


Die Klasse `LinkedList`

- Java stellt mit `LinkedList` eine Klasse zur Verfügung, die ein einfaches Anlegen und Pflegen einer verketteten Liste möglich macht.
- `LinkedList` ist eine Klasse aus dem Paket `java.util`
- Mit `LinkedList list = new LinkedList();` wird eine leere Liste erzeugt.
- Durch die Methoden
  `void addFirst (Object Objektname);`
  `void addLast (Object Objektname);`
  `void add (int index, Object Objektname);`
  können Objekte in Liste eingefügt werden

- Durch die Methoden
  `void removeFirst();`
  `void removeLast();`
  `void remove (int index);`
  können Objekte aus Liste gelöscht werden.
- Die Methode `size()` gibt Länge der Liste an.

Beispiel für eine verkettete Liste mit `LinkedList`

```
import java.util.*;

public class LinkedListTest{
    public static void main(String args[])  {

        LinkedList list = new LinkedList();
        list.addFirst ("Kai");
        list.addLast ("Monika");
        list.add (1,"Karla");
        list.add (0,"Monika");

        System.out.println(list);
        list.remove("Monika");
        System.out.println(list);
        list.remove (1);
        System.out.println(list);

    }
}
```

Ausgabe des Programms

```
[Monika, Kai, Karla, Monika]
[Kai, Karla, Monika]
[Kai, Monika]
```


Die Datenstrukturen Keller und Schlange

- Keller und Schlange sind spezielle Datenstrukturen, bei denen Operationen nur an einem Ende ausgeführt werden können.
- Sie können mit Hilfe von Arrays oder einfach verketteten Listen dargestellt werden.
- Java bietet mit der Klasse `Stack` eine Datenstruktur für einen Keller.
- In Java gibt es erst ab JDK 1.5 eine Unterstützung für `Queues`.

Die Datenstrukturen Keller (Stack)

Ein Keller oder Stack hat die Eigenschaften

- Knoten sind in einer Folge angeordnet
- Nur der Anfang des Stacks ist zugänglich
- Knoten werden am Anfang hinzugefügt
- Knoten werden vom Anfang entfernt.
- Der Keller kann nicht durchsucht werden, nur der Knoten am Anfang ist sichtbar.

Aufgrund dieser Eigenschaften bezeichnet man einen Keller aus als

LIFO-Liste (Abkürzung für Last In First Out).

Eine Anwendung von Stacks ist die Verwaltung rekursiver Methoden.

Die wichtigsten Methoden für einen Stack sind die Methoden `push()` und `pop()` zum Einfügen und Entfernen aus dem Keller.

Um die Funktionalität von diese beiden Methoden zu verstehen, implementieren wir eine Klasse `Keller` mit den beiden Methoden.

`Keller` soll durch ein Array dargestellt werden.

Implementierung eines Kellers

```
class Keller{

    private int laenge;
    private Object[] kellerarray;
    private int top; //index der nächsten freien Zelle

    //Konstruktor
    Keller (int laenge){
        this.laenge = laenge;
        kellerarray = new Object[laenge];

        top = 0; //Setzen von top auf die erste Position
    }
```


Die Methoden `push()` und `pop()`

```
    void push (Object x){
        if (top >= laenge)
            System.out.println("Keller voll");
        else {
            kellerarray[top] = x;
            top++;
        }
    }

    Object pop(){
        if (top > 0){
            top--;
            return kellerarray[top];
        }
        else
            return null;
    }

}
```


In Java gibt es für die Datenstruktur Keller die Klasse `Stack`.

- `Stack` ist aus dem Paket `java.util`
- `Stack` ist abgeleitet von `Vector`
- Die wichtigsten Methoden sind
  - `public Object push (Object Element);`
    `//Nimmt Element und setzt es an den Anfang des Kellers`
  - `public Object pop() throws EmptyStackException;`
    `//Gibt oberstes Element zurück und entfernt es aus Stack`
  - `public Object peek() throws EmptyStackException;`
    `//Gibt oberstes Element zurück, entfernt es nicht aus Stack`
  - `public boolean empty();`
    `//true, falls Keller keine Elemente enthält`

Die Datenstrukturen Schlange (Queue)

Eine Schlange oder Queue hat die Eigenschaften

- Knoten sind in einer Folge angeordnet
- Es gibt zwei Enden, den Anfang (head) und das Ende (tail)
- Knoten werden am Ende hinzugefügt
- Knoten werden vom Anfang entfernt.
- Die Schlange kann von Anfang bis zum Ende durchsucht werden

Aufgrund dieser Eigenschaften bezeichnet man eine Schlange aus als

FIFO-Liste (Abkürzung für First In First Out).

In Java gibt es zur Zeit keine Klasse, die die Datenstruktur einer Schlange repräsentiert.

Ab JDK 1.5 gibt es ein Interface `QUEUE` in dem Paket `java.util`

Eine Schlange kann mittels einer verketteten Liste oder mit Hilfe eines Arrays implementiert werden. Wir werden hier –wie schon beim Stack- das Array nehmen.

Die zwei wichtigsten Methoden sind

`enqueue(`*`element`*`)` zum Einfügen in eine Queue und `dequeue()` zum Entfernen aus einer Queue.

Implementierung einer Schlange

```
class Schlange{

    private int laenge;
    private Object[] schlangearray;
    private int head, tail;
    private boolean istleer;

    //Konstruktor
    Schlange (int laenge){
        this.laenge = laenge;
        head = tail = 0;
        istleer = true;
        schlangearray = new Object[laenge];

    }
```


Die Methode `enqueue()`

```
    void enqueue (Object x){
        if ((tail == head) && (istleer == false)){
            System.out.println("Queue ist voll");

        }
        else{
            schlangearray[tail] = x;
            tail++;
            istleer = false;

            if (tail == laenge)
                tail = 0;
        }

    }
```


Die Methode `dequeue()`

```
    Object dequeue(){
        if ((tail == head) &&(istleer == true)){
            System.out.println("Queue ist leer");
            return null;
        }

        else {
            Object inhalt = schlangearray[head];
            head++;
            if (head == laenge)
                head = 0;
            if (head == tail)
                istleer = true;

            return inhalt;

        }
    }
}
```


Algorithmen

Algorithmen sind Verfahren zur schrittweise Lösung von Problemen.

Sie sind Grundlage für jedes Computer-Programm.

Definition:

Ein Algorithmus ist eine endliche Folge von Anweisungen. Ein Algorithmus benötigt eine Menge von Werten (diese kann auch leer sein) als Input und generiert eine Menge von Werten als output (diese darf nicht leer sein).

Ein Algorithmus hat fünf Eigenschaften

1. endlich
2. exakt
3. elementar, d.h. alle Operationen sind auf elementare Operationen zurückzuführen
4. Input: hat kein oder mehrere Input-Daten
5. Output: hat ein oder mehrere Output-Daten

Kriterien für die Qualität eines Algorithmus

- Korrektheit (zwingend notwendig)

Darüber hinaus sollte ein Algorithmus

- einfach zu verstehen sein
- einfach zu implementieren sein
- eine geringe Laufzeit und wenig Plattenplatz benötigen.

Man unterscheidet zwischen rekursiven und iterativen Algorithmen.

Rekursive Algorithmen sind Algorithmen, die sich selber aufrufen.

Iterative Algorithmen enthalten dagegen Abschnitte, die innerhalb einer einzigen Ausführung des Algorithmus mehrfach durchlaufen werden.

Als Vergleich ein einfaches Beispiel:

Es soll die Fakultät von einer natürlichen Zahl n berechnet werden,

d.h. z.B. 5! = 1*2*3*4*5 = 120

Es gilt 0! = 1.

Zunächst wird der Algorithmus iterativ beschrieben, danach rekursiv.

Iterative Lösung

```
import javax.swing.JOptionPane;

public class FakultaetIterativ{

    public static long fakultaet (int n){
        long wert = 1;
        for (int i = 1; i <=  n; i++)
            wert = wert * i;
        return wert;
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(JOptionPane.showInputDialog(null,
                    "Bitte eine Zahl eingeben"));

        System.out.println(n+"! = " + fakultaet (n));

        System.exit(0);
    }
}
```


Der rekursive Algorithmus sieht in Java wie folgt aus:

```
    public static long fakultaet (int n){
        if (n == 0)
            return 1;
        else
            return n*fakultaet (n-1);

    }
```


Um die rekursive Methodenaufrufe besser zu verstehen, wird als Hilfsmethode die Methode `blanks()` implementiert. Diese Methode rückt die Aufrufe ein, so dass die jeweilige Stufe des Aufrufs erkannt wird.

```
import javax.swing.JOptionPane;

public class FakultaetRekursiv2{

    //Hilfsmethode um Rekursive Aufrufe der Methode
    //fakultaet() auf den verschiedenen Stufen anzuzeigen

    static String blanks (int zahl){
        StringBuffer b = new StringBuffer ();
        for (int i = 0; i < zahl*2;i++)
            b.append(' ');
        return b.toString();
    }
```



```
    public static long fakultaet (int n){
        System.out.println(blanks(20-n)+
                "Aufruf fakultaet("+ n +")");
        long wert;
        if (n == 0)
            wert = 1;
        else
            wert = n* fakultaet(n-1);
        System.out.println(blanks(20-n) +
                "Rueckkehr fakultaet ("+n+") = " + wert);
        return wert;
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(JOptionPane.showInputDialog(null,
                "Bitte eine Zahl eingeben"));
        System.out.println();
        System.out.println(fakultaet(n));

        System.exit(0);
    }
}
```


Der Aufruf von `fakultaet(6)` ergibt dann

```
                    Aufruf fakultaet(6)
                      Aufruf fakultaet(5)
                        Aufruf fakultaet(4)
                          Aufruf fakultaet(3)
                            Aufruf fakultaet(2)
                              Aufruf fakultaet(1)
                                Aufruf fakultaet(0)
                                Rueckkehr fakultaet (0) = 1
                              Rueckkehr fakultaet (1) = 1
                            Rueckkehr fakultaet (2) = 2
                          Rueckkehr fakultaet (3) = 6
                        Rueckkehr fakultaet (4) = 24
                      Rueckkehr fakultaet (5) = 120
                    Rueckkehr fakultaet (6) = 720
720
```


Ein anderes Beispiel:

Berechnung des größten Wertes innerhalb eines Arrays mit Integer Werten.

Die Größe des Arrays wird vom Nutzer festgelegt, das Array wird mit Zufallszahlen zwischen 0 und 100 gefüllt.


```
import javax.swing.JOptionPane;

public class SucheMaximum{

    public static int suchemax (int[] a, int l, int r){
        if (l == r)
            return a[l];
        else{
            int m = suchemax (a,l+1,r);
            if (a[l] > m)
                return a[l];
            else
                return m;
        }
    }
```

```
    public static void main(String[] args) {
        int n = Integer.parseInt(JOptionPane.showInputDialog(null,
                  "Größe des Arrays"));
        int a[] = new int[n];

        //Fuellen des Arrays mit Zufallszahlen
        for (int i = 0; i<a.length;i++)
            a[i] = (int) Math.round (100*Math.random())  ;

        for ( int i = 0; i<a.length;i++)
            System.out.print(a[i]+" ");
        System.out.println();
        System.out.println("Maximum ist: „
                +suchemax(a,0,a.length-1));
        System.exit(0);

    }
}
```

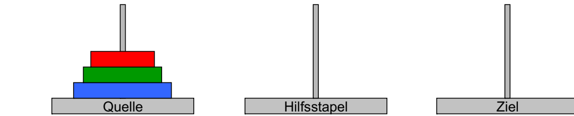

Das Standard Beispiel für Rekursive Algorithmen:

Die Türme von Hanoi

Das Problem:

Drei senkrechte Stangen sind auf einem Brett befestigt. Auf einer der Stangen befinden sich n durchlöcherte Scheiben.

Die Aufgabe besteht darin, den Stapel von Scheiben auf eine andere Stange zu bringen. Dabei darf jeweils nur eine Scheibe bewegt werden und es darf nie eine größere auf eine kleinere Scheibe zu liegen kommen.



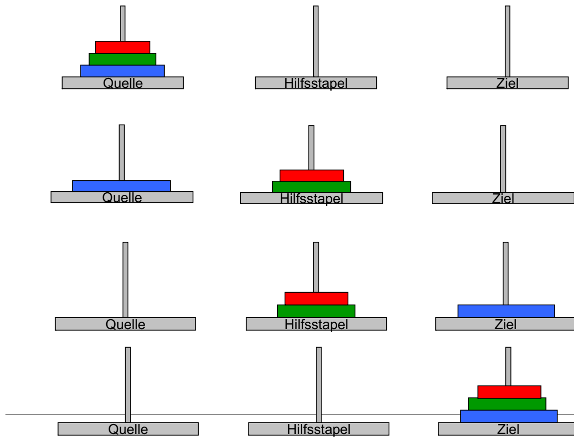
Lösung durch Unterteilung in Teilprobleme

Die drei Stapel werden als Quelle, Hilfsstapel und Ziel bezeichnet.

Der rekursive Algorithmus läuft wie folgt

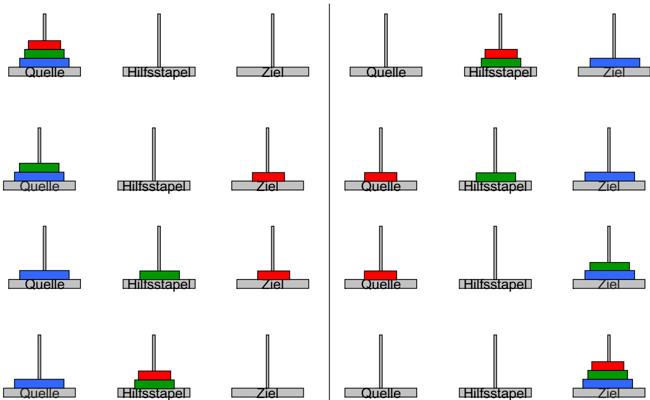
n = 1 Transportiere den Stapel von Quelle zum Ziel

n > 1
- I. Transportiere den oberen Stapel mit n-1 Scheiben vom Quellstapel auf den Hilfsstapel unter Zuhilfenahme des Zielstapels.
- II. Transportiere die oberste Scheibe von der Quelle zum Ziel.
- III. Transportiere die (n-1) Scheiben auf dem Hilfsstapel zum Ziel. Hierbei kann der Quellstapel als Hilfsstapel benutzt werden.



Der Algorithmus

```
public class TuermeVonHanoi{

    public static void ziehe (int quelle, int hilf, int ziel, int n){
        if (n ==1)
            System.out.println("Bewege oberste Scheibe von Stapel
                    "+quelle+" auf Stapel " + ziel);
        else{
            ziehe( quelle, ziel, hilf, n-1);
            System.out.println("Bewege oberste Scheibe von Stapel
                    "+quelle+" auf Stapel " + ziel);
            ziehe(hilf, quelle, ziel, n-1);

        }
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        ziehe(0,1,2,n);

    }
}
```

Der Aufruf `java TuermeVonHanoi 3` liefert dann

```
Bewege oberste Scheibe von Stapel 0 auf Stapel 2
Bewege oberste Scheibe von Stapel 0 auf Stapel 1
Bewege oberste Scheibe von Stapel 2 auf Stapel 1
Bewege oberste Scheibe von Stapel 0 auf Stapel 2
Bewege oberste Scheibe von Stapel 1 auf Stapel 0
Bewege oberste Scheibe von Stapel 1 auf Stapel 2
Bewege oberste Scheibe von Stapel 0 auf Stapel 2
```


Suchalgorithmen

Formulierung des Suchproblems

In einem Behälter A befinden sich eine Reihe von Elementen.

Prüfe, ob ein Element $e \in A$ existiert, das eine bestimmte Eigenschaft P(e) erfüllt.

Behälter sind z.B. Arrays, Einfach verkettete Listen, Bäume etc.

Wir werden im folgenden alle Algorithmen auf Arrays betrachten.

Die Werte im Array seien Zahlen.

Das Suchproblem lautet also für Arrays:

Sei a[] ein Array der Länge n.

Prüfe, ob Element e im Array vorhanden ist, d.h. existiert ein i für das gilt:
a[i] == e für i = 0,…,n-1.

Lineare Suche

Die lineare Suche ist der einfachste Suchalgorithmus. Das Array wird der Reihe nach durchsucht.

Der Algorithmus lautet

```
static int linSuche (int[] a, int e) {
  for (int i = 0; i < a.length; i++){
     if (a[i] = = e)
       return i;
   }
  return -1; //ungültiger Index
}
```

Es gilt $T_{best} = O(1)$ $T_{average} = T_{worst} = O(n)$

Binäre Suche

Die binäre Suche ist ein Suchalgorithmus auf einem sortiertem Array a[].

Definition: a heißt geordnet (oder sortiert) wenn gilt:

$$\forall \quad i: 0 \le i < n-2 \text{ ist } a[i] \le a[i+1]$$

Idee der Binären Suche

Sei `0 = min`, `n - 1 = max` und sei `a[]` ein sortiertes Array.

Wähle Index `m` mit `min` ≤ `m` ≤ `max` (meist `m = (min+max)/2`)

- Falls `e = a[m]` => Fertig, Element gefunden
- Falls `e < a[m]` => Suche weiter im Bereich `[min,m-1]`, d.h. `max = m-1`
- Falls `e > a[m]` => Suche weiter im Bereich `[m+1;max]`, d.h. `min = m+1`

Abbruch, falls `min > max`. Dann ist `e` kein Element des Arrays.

Der Algorithmus zur Binären Suche

```
static int binSuche (int[] a, int e, int min, int max) {
      if (min > max)
            return -1; //Elemente e nicht in Array vorhanden
      int m = (min + max) /2;
      if (e == a[m])
            return m;
      if (e < a[m])
            return binSuche (a,e,min,m-1);
      //falls e > a[m]
      return binSuche (a,e,m+1,max);
   }
```


Die binäre Suche ist für große Datenmengen weit effizienter als die lineare Suche.
Es gilt:

$T_{average} = T_{worst} = O(\log_2 n)$

Im besten Fall ist nur ein Aufruf des Suchalgorithmus nötig.

Aber: Die Elemente müssen sortiert vorliegen.

Daher oft bei großen nicht sortierten Datenmengen sinnvoll:
Kopiere alle Elemente in Array, sortiere Array, führe binäre Suche durch.

In Java gibt es im Package `java.util.Arrays` die Methode `binarySearch()`, die die binäre Suche in sortierten Arrays durchführt.

Sortierverfahren

Gegeben: Unsortierte Folge $a_1,\dots,a_n$ von Elementen

Gesucht: Sortierte Folge mit $a_i \le a_{i+1}$

Wir werden die verschiedenen Sortieralgorithmen auf Arrays betrachten, d.h. das Problem lautet

Sei a ein Array mit a[0],...,a[n-1].

Gesucht wird ein sortiertes Array mit a[i] ≤ a[i+1]; i = 0,...,n-2

Es gibt eine ganze Reihe von Sortieralgorithmen.

Wir werden den BubbleSort, QuickSort sowie den MergeSort betrachten.

BubbleSort

BubbleSort ist ein elementares Sortierverfahren.

Idee des BubbleSort

Bubble-Sort bringt in mehreren Durchläufen jeweils das kleinste Element des Restarrays an die erste Position. Im ersten Durchlauf ist das Restarray gleich dem ganzen Array. Im zweiten Durchlauf fehlt gegenüber dem ersten Durchlauf das erste Element. In jedem Durchlauf bringt man das kleinste Element nach vorne, in dem vom letzten Element her jedes Element mit seinem Vorgänger verglichen wird. Falls es kleiner als sein Vorgänger ist, wird es vertauscht.

Der Algorithmus kann abgebrochen werden, wenn nichts mehr vertauscht werden muss.

Zunächst benötigt man eine Methode, die zwei Elemente vertauscht

```
public void tausch (int x, int y, int [] a){
    int z = a[x];
    a[x] = a[y];
    a[y] = z;
}
```


Der Algorithmus BubbleSort läuft dann wie folgt

```
public void bubble (int[] a) {
    boolean b = true;
    int j = a.length;
    while(b){ //true
        b = false;
        for (int i = a.length-1;i > a.length-j;i--)
            if (a[i] < a[i-1]){
                tausch (i, i-1, a);
                b = true;
            }
        j--;
    }
}
```

`b` gibt an, ob noch etwas vertauscht wird. Falls `false`, wird der Algorithmus abgebrochen.

BubbleSort ist ein recht langsames Sortierverfahren.

„Divide and Conquer“ Algorithmen

Divide and Conquer Algorithmen beschreiben folgende Problemlösestrategie:

- Zerlege das Problem P in die Teilprobleme $P_1,...,P_n$
- Löse das Problem rekursiv für jede Teilmenge
- Setze die Lösung L von P als Kombination der Lösungen $L_1,..., L_n$ der Teilmengen zusammen.

Ein Vertreter dieser Algorithmen ist das Sortierverfahren QuickSort.

QuickSort

QuickSort wurde 1962 von dem britischen Informatiker C.A.R.Hoare entwickelt. Damals waren noch keine schnellen Sortierverfahren bekannt.

QuickSort ist ein rekursiver Algorithmus. Rekursive Algorithmen galten früher als ineffizient.

Der Algorithmus QuickSort arbeitet wie folgt:

Gegeben sei das unsortierte Array a[0],…,a[n-1] der Länge n.

Divide: Teile die Elemente des Arrays a[l],…,a[r] in zwei Teilarrays a[l],…,a[p] und a[p+1],..,a[r], so dass gilt $a[i] \leq a[j]$ für alle $l \leq i \leq p$ und $p+1 \leq j \leq r$. (Im ersten Schritt ist l = 0 und r = n-1)

a[p] bezeichnet man als Pivot.

Diesen Schritt nennt man Partitionierung.

Conquer: Sortiere die beiden Teilarrays rekursiv indem QuickSort für die Teilarrays aufgerufen wird.

Wie erfolgt die Partitionierung?

Von links her wird gesucht, bis erstes Element a[i] ≥ Pivot gefunden, von rechts her wird gesucht, bis erstes Element a[j] < =Pivot gefunden. Dann werden diese Elemente vertauscht, wobei i um 1 erhöht und j um 1 erniedrigt wird. Es wird weitergesucht, solange bis j <= i ist.

Bemerkung zur Wahl des Pivot-Elementes

Als Pivot Element wählt man meist ein Element in der Mitte des Arrays, d.h. Pivot = a[(l+r)/2]

Der Algorithmus QuickSort

```
public  void quick (int[] a, int begin, int end) {
       if (begin >= end)
          return;

       int pivot = a[(begin+end)/2];
       int l = begin; int r = end;

       while(l <= r){
          while (a[l] < pivot)
             l++;
          while(a[r] > pivot)
             r--;
          if (l <= r){
             tausch(l,r,a);
             l++; r--;
          }
       }
          quick (a, begin, r);
          quick (a, l, end);
 }
```

Partitionierung (Anmerkung zum Block von `int pivot` bis zur schließenden Klammer der while-Schleife)

MergeSort

Merge-Sort ist ein weiterer Divide and Conquer Algorithmus.

Merge Sort teilt die ursprüngliche Menge der Datensätze in zwei Hälften auf. Diese werden sortiert und dann zusammengemischt. Dabei vergleicht man immer wieder die vordersten Elemente der sortierten Hälften und entnimmt das kleinste der beiden. Auf diese Weise verschmelzen (merge) die zwei geordneten Listen zu einer gemeinsamen geordneten Liste, die alle Elemente der ursprünglichen zwei Listen enthält.

Der Algorithmus MergeSort arbeitet wie folgt

- *Divide*, d.h. teile Datensätze in zwei Teilmengen mit jeweils n/2 Elemente
- *Conquer* durch Sortierung der beiden Teilmengen rekursiv durch Aufruf von MergeSort
- *Merge*, d.h. mische beide sortierten Teilmengen.

Der Algorithmus MergeSort

```
public void mergeSort (int[] a, int l, int r) {
       if (l < r){
          int m = (l+r+1)/2;
          mergeSort(a,l,m-1);
          mergeSort(a,m,r);
          merge(a,l,r,m);
       }
}
```


Der Teilalgorithmus `merge` stellt sich wie folgt dar:

```
public void merge (int[] a, int l, int r, int m){
       int [] temp = new int[r-l+1];

       for (int i=0, j=l, k=m; i < temp.length; i++)
          if ((k > r) || ((j < m) && (a[j] < a[k]))){
             temp[i] = a[j];
             j++;
          }
          else{
             temp[i] = a[k];
             k++;
          }
       for (int i=0; i < temp.length;i++)
          a[l+i] = temp[i];
}
```


Nebenläufigkeit in Java: Threads

- Moderne Betriebssysteme sind multitasking-fähig, d.h. auch wenn der Rechner nur über einen einzigen Prozessor verfügt, können mehrerer Programme quasi gleichzeitig ausgeführt werden.
- Diese quasi-Parallelität bezeichnet man als Nebenläufigkeit.
- Sie wird durch das Betriebssystem durch Prozessorzuteilung gewährleistet.
- Java ist eine Programmiersprache, die nebenläufige Programmierung direkt unterstützt.
- Das Konzept beruht auf sog. Threads (Fäden). Threads sind quasi parallel ablaufenden Aktivitäten.

Threads

Es gibt zwei Methoden einen Thread in Java zu erzeugen:

1. Durch direkte Ableitung von der Klasse `Thread`
2. Übergabe eines Objektes, dessen Klasse die Schnittstelle `Runnable` implementiert, an ein Objekt der Klasse `Thread`.

Thread Erzeugung durch direkte Ableitung von der Klasse `Thread`

- Die Klasse `Thread` befindet sich im Paket `java.lang`.
- Die Methode `run()` der Klasse `Thread` muss überschrieben werden.
- Der Programmcode, der nebenläufig ablaufen soll, ist in der überschriebenen Methode `run()` enthalten.
- Erzeugt wird ein Thread mit Hilfe des `new`-Operators.
- Gestartet wird ein Thread durch den Methodenaufruf `start()`
- `start()` reserviert die notwendigen Systemressourcen und ruft die Methode `run()` auf.

Beispiel für Thread Erzeugung durch Ableitung von Klasse `Thread`

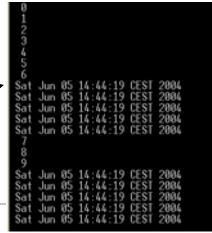
```
import java.util.*;

class DateThread extends Thread{
    public void run(){
        for (int i = 0; i < 10;i++)
            System.out.println(new Date());
    }
}

class ZaehlThread extends Thread{
    public void run(){
        for (int i = 0; i < 10;i++)
            System.out.println(i);
    }
}
```

1. Threadklasse, Methode `run()` muss überschrieben werden

2. Threadklasse

```
public class ThreadBeispiel1{
    public static void main(String[] args){

        //Erzeugung des ersten Threads durch new
        DateThread t1 = new DateThread();

        //Starten des Threads, start() ruf run() auf
        t1.start();

         //Erzeugung und Starten des zweiten Threads
        ZaehlThread t2 = new ZaehlThread();
        t2.start();
    }
}
```

eine mögliche Ausgabe



Thread Erzeugung durch Implementierung der Schnittstelle `Runnable`

- Das Interface `Runnable` befindet sich im Paket `java.lang`.
- `Runnable` deklariert als einzige Methode `run()`
- Erzeugung eines Threads:
  Mit dem `new`-Operator wird eine Instanz der Klasse `java.lang.Thread` generiert. Beim Konstruktoraufruf wird als Übergabeparameter eine Referenz auf ein Objekt übergeben, dessen Klasse die Schnittstelle `Runnable` implementiert.

Beispiel für Thread Erzeugung durch Implementierung der Schnittstelle `Runnable`

```
import java.util.*;

class DateThread implements Runnable{
    public void run(){
        for (int i = 0; i < 10;i++)
            System.out.println(new Date());

    }
}

class ZaehlThread implements Runnable{
    public void run(){
        for (int i = 0; i < 10;i++)
            System.out.println(i);

    }
}
```



```
public class ThreadBeispiel2{
    public static void main(String[] args){

        //Erzeugen des Threads
        Thread t1 = new Thread(new DateThread());
        t1.start();
        Thread t2 = new Thread(new ZaehlThread());
        t2.start();
    }
}
```


Kurzzeitiges Anhalten (Schlafen) von Threads

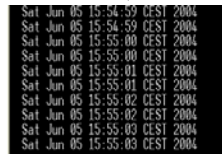
- Mit der Klassenmethode `sleep()` der Klasse `Thread` kann ein Thread für eine bestimmte Zeit angehalten werden.
- `static void sleep(long millis) throws InterruptedException`
  Der aktuell ausgeführte Thread wird mindestens `millis` Millisekunde angehalten. Unterbricht ein anderer Thread den schlafenden Thread, so wird vorzeitig eine `InterruptedException` ausgelöst.
- Da `sleep()` die Exception `InterruptedException` werfen kann, muss der Methodenaufruf innerhalb eines `try catch` Blockes stehen, in dem in die Exception behandelt wird.
- Beispiel: Applikation soll 1 Sekunde schlafen

```
try {
   Thread.sleep(1000);
} catch (InterruptException e){}
```



```
import java.util.*;

class DateThread implements Runnable{
    public void run() {
        for (int i = 0; i < 10;i++){
            System.out.println(new Date());
            try{
            Thread.sleep (500);

            }catch(InterruptedException e){ }
        }
    }
}

public class ThreadBeispiel3{
    public static void main(String[] args){
        Thread t1 = new Thread(new DateThread());
        t1.start();

    }
}
```

Thread schläft 0,5 Sekunden



Beenden von Threads

Ein Thread ist beendet, wenn eine der folgenden Fälle eintritt:

- Die `run()` Methode wurde ohne Fehler beendet.
- In der `run()` Methode tritt eine Exception auf, die die Methode beendet.

Ein Thread kann auch von außen abgebrochen werden. Dies erfolgt mit der veralteten Methode `stop()`, die allerdings nicht mehr verwendet werden sollte.

Thread durch die Methoden `interrupt()` und `isInterrupted()` beenden

```
class ThreadInterrupt extends Thread{
    public void run(){
        System.out.println("Start");
        while (true){
            if (isInterrupted())
                break;
            System.out.println("Der Thread schläft nicht");
            try{
                Thread.sleep(500);
            }
            catch (InterruptedException e){
            interrupt();
            }
        }
        System.out.println("Ende");
    }
```

ergibt true, falls von außen die interrupt() Methode für diesen Thread aufgerufen wird

```
    public static void main(String[] args){
        ThreadInterrupt t = new ThreadInterrupt();
        t.start();
        try{
        Thread.sleep(2000);
        } catch (InterruptedException e){}
        t.interrupt();
    }
}
```

Aufruf der Methode interrupt()

```
Start
Jetzt schlaeft der Thread nicht
Jetzt schlaeft der Thread nicht
Jetzt schlaeft der Thread nicht
Jetzt schlaeft der Thread nicht
Ende
```

Jede halbe Sekunde wird ein Text ausgegeben. Nach 2 Sekunden wird der Thread unterbrochen

Warten auf das Ende der Aktivität eines Threads mit Hilfe der Methode `join()`

```
class ThreadJoin extends Thread{
    int wert = 0;

    public void run(){
        wert = 1;
    }

    public static void main(String[] args) throws InterruptedException{
        ThreadJoin t = new ThreadJoin();
        t.start();
        t.join();
        System.out.println(t.wert);

    }
}
```

Die Ausgabe ergibt 1.
Ohne `t.join()` ergibt sich dagegen 0.

Threads können Prioritäten erhalten

```
public class ThreadBeispielPrio{
    public static void main(String[] args){
        DateThread t1 = new DateThread();
        t1.setPriority(Thread.MAX_PRIORITY);

        System.out.println("Prioritaet t1: "
                + t1.getPriority());
        t1.start();
        ZaehlThread t2 = new ZaehlThread();
        System.out.println("Prioritaet t2: "
                + t2.getPriority());
        t2.start();
    }
}
```

```
Prioritaet t1: 10
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Sat Jun 05 21:55:14 CEST 2004
Prioritaet t2: 5
0
1
2
3
4
5
6
7
8
9
```


Bemerkung zu der Vergabe von Prioritäten bei Threads

- Hat ein Thread eine höhere Priorität als ein anderer Thread, so bedeutet das nur, das er im Durchschnitt mehr Rechenzeit erhält.
- Ist ein Thread sehr rechenintensiv, so kann mit Hilfe der Methode `yield()` erreicht werden, dass Threads mit gleicher oder niedriger Priorität den Prozessor erhalten.
- Prioritäten werden gesetzt mit der Methode
  `public final void setPriority (int newPriority)`
- Die gesetzte Priorität erhält man durch
  `public final int getPriority()`
- Für die Prioriät eines Thread gibt es 3 Konstanten in der Klasse Thread:
  `MAX_PRIORITY = 10, NORM_PRIORITY = 5, MIN_PRIORITY = 1`

Zugriff auf gemeinsame Ressourcen- Synchronisation und Monitorkonzept

- Benutzen mehrere Threads dieselben Daten, so kann es zu Synchronisations-problemen kommen, die inkonsistente Daten zur Folge haben.
- Beispiel: Thread1 schreibt Daten, bekommt während des Schreibens den Prozessor entzogen. Thread2 bekommt Prozessor zugewiesen und liest die – noch nicht fertig geschriebenen Daten.
- Man nennt dies Race Condition, d.h. das Ergebnis hängt von der Reihenfolge, in der die Threads ausgeführt werden, ab.
- Daher muss eine Abarbeitungsreihenfolge definiert werden, d.h. eine Synchronisation ist erforderlich.

- Zur Vermeidung von Race Conditions wendet man das Prinzip des wechselseitigen Ausschlusses an.
- Ein kritischer Abschnitt ist eine Folge von Befehlen, die ein Thread nacheinander vollständig abarbeiten muss, auch wenn er vorübergehend den Prozessor an einen anderen Thread abgibt. Kein anderer Thread darf einen kritischen Abschnitt betreten, solange der erste Thread mit der Abarbeitung der Befehlsfolge noch nicht fertig ist.

Das Monitorkonzept

- Monitore wurden 1974 von C.A.R.Hoare als Synchronisationsmittel eingeführt. Sie ermöglichen eine wechselseitigen Ausschluss
- Grundidee von Monitoren: Daten, auf denen die kritischen Abschnitte arbeiten, und die kritischen Abschnitte selbst werden in einem zentralen Konstrukt zusammengefasst.
- Die grundlegenden Eigenschaften eines Monitors sind:
  - Kritische Abschnitte, die auf denselben Daten arbeiten, sind Methoden eines Monitors
  - Ein Prozess betritt einen Monitor durch Aufruf einer Methode des Monitors
  - Nur ein Prozess kann zur selben Zeit den Monitor benutzen.

Das Monitorkonzept und Java

- In Java wird der wird der wechselseitige Ausschluss mit dem Monitorkonzept realisiert.
- Das Monitorkonzept wird mit Hilfe des reservierten Wortes `synchronized` umgesetzt.
- Sowohl Methoden als auch zu synchronisierende Blöcke können das Schlüsselwort `synchronized` aufweisen.

Monitore und Klassenmethoden

Werden eine oder mehrere Klassenmethoden mit `synchronized` versehen, so wird ein Monitor um diese Methoden gebaut. Damit kann nur ein einziger Thread zu einer bestimmten Zeit eine der synchronisierten Methoden bearbeiten.

Beispiel:

```
public class Beispiel1{
     …
     public static synchronized void methode1(){
          //kritischer Abschnitt
     }
     public static synchronized void methode1(){
          //kritischer Abschnitt
     }
     …
}
```

Bemerkung: Klassenmethoden ohne `synchronized` gehören nicht zum Monitor.

Monitore und Objektmethoden

Werden eine oder mehrere Objektmethoden mit `synchronized` versehen, so wird ein **Monitor pro Objekt** um diese Methoden gebaut. Damit kann nur ein einziger Thread zu einer bestimmten Zeit eine der synchronisierten Methoden bearbeiten.

Beispiel:

```
public class Beispiel2{
     …
     public synchronized void methode1(){
          //kritischer Abschnitt
     }
     public synchronized void methode1(){
          //kritischer Abschnitt
     }
     …
}
```

Bemerkung: Die Klasse `StringBuffer` verfügt über mehrer synchronisierte Methoden

Beispiel für die Anwendung von Monitoren in Java

```
public class Punkt{
     private float x, y;
     …Konstruktoren…
     public synchronized void setzePunkt(float x, float y){
          this.x = x;
          this.y = y;
     }
     public synchronized Punkt liesPunkt(){
          return new Punkt(x,y)
     }
}
```

Ohne `synchronized` könnte eine Unterbrechung eines Threads, der gerade die Methode `liesPunkt()` ausführt, durch einen anderen Thread, der die Methode `setzePunkt()` ausführt,dazu führen, dass die Methode `liesPunkt()` die x-Koordinate des alten Punktes und die y-Koordinate des neuen Punktes liefert.

Monitore und einzelne Blöcke

Manchmal ist das Synchronisieren einer gesamten Methode nicht gewollt. Nur einzelne Anweisungen bilden den kritischen Abschnitt.

Beispiel:

```
public class Beispiel3{
     …
     public void methode1(){
          …
          synchronized (schlüssel){
          //kritischer Abschnitt
          }
     }
}
```

schlüssel ist eine Referenz auf ein Objekt

Synchronisation mit Reihenfolge

Bis jetzt war es nicht möglich, eine definierte Reihenfolge der Threads zu erreichen. Dies ist aber oft notwendig, wenn die Thread auf die selben Daten zugreifen.

Um eine Reihenfolge zu erzielen gibt es in Java die Objektmethoden `wait()` und `notify()` der Klasse `Object`.

`void wait() throws InterruptedException`

Durch den Aufruf von `wait()` wird der Thread, der eine synchronisierte Methode oder Block bearbeitet, in den Zustand „blocked" überführt und verlässt den Monitor. Er wartet auf ein `notify()` um weiterarbeiten zu können.

`void notify()`

Mit Hilfe der Methode `notify()` wird der „wartende" Thread wieder aufgeweckt und gelangt in den Zustand „ready to run". Jetzt kann er wieder den Monitor zugewiesen bekommen.

Beispiel für `wait()` und `notify()`

```
class Pipe{
    static final int MAXPIPE = 4;
    private  Vector pipevector = new Vector();

    public synchronized void write (int wert){
        while(pipevector.size() == MAXPIPE)
            try{
                wait();
            } catch (InterruptedException e){}

        //Schreiboperationen durchführen
        notify();   //Einen eventuell wartenden Leser wecken, da
                    //sich Element in Pipe befindet
    }
```



```
    public synchronized int read(){
        while (pipevector.size() == 0) //es gibt kein Element zu lesen
            try{
                wait();
            } catch (InterruptedException e){}

        //Leseoperationen durchführen
        //Element aus Pipe entfernen
        notify(); //einen eventuell wartenden Schreiber wecken
    }
}
```

Deadlocks

- Das Prinzip des wechselseitigen Ausschlusses, das durch Monitore realisiert wird, kann sog. Deadlocks erzeugen.
- Ein Deadlock kommst z.B. vor, wenn ein Thread A eine Ressource belegt, die ein anderer Thread B benötigt. Thread B belegt aber auch eine Ressource, die Thread A benötigt.
  Beide Thread können somit nicht weiterarbeiten und befinden sich in einem dauernden Wartezustand.

Beispiel für ein Deadlock

```
class T1 extends Thread{
        public void run(){
            synchronized(Deadlock.a){
                System.out.println("T1: Schluessel fuer a bekommen");
                try{
                    sleep(1000);
                } catch (InterruptedException e){}
                synchronized (Deadlock.b){
                    System.out.println("T1:
                                        Schluessel fuer b bekommen");
                }
            }
        }
}
```



```
class T2 extends Thread{
        public void run(){
            synchronized(Deadlock.b){
                System.out.println("T2: Schluessel fuer b bekommen");

                synchronized (Deadlock.a){
                    System.out.println("T2:
                                Schluessel fuer a bekommen");
                }
            }
        }
}

class Deadlock{
    static Object a = new Object(),b = new Object();
    public static void main(String args[]){
        new T1().start();
        new T2().start();
    }
}
```


Die Applikation liefert das Ergebnis:

```
T1: Schluessel fuer a bekommen
T2: Schluessel fuer b bekommen
```

Danach sind die Monitore blockiert.
T1 blockiert „a“, T2 blockiert „b“ . Beide verlassen Monitore nicht, da ihre run() Methode noch nicht abgeschlossen wurde.
T1 wartet auf Zutritt für Monitor „b“, T2 wartet auf Zutritt für Monitor „a“.

Grafikprogrammierung in Java

Ohne eine ansprechende graphische Benutzeroberfläche haben Computer-Programme in der heutigen Zeit keinen Bestand mehr.

Java bietet ein breites Spektrum von Klassen und Methoden zur Gestaltung von grafischen Benutzeroberflächen an.

Hierzu dient in Java die Klassenbibliothek der Java Foundation Classes (JFC).

Die Java Foundation Classes

Die JFC sind in verschiedenen Klassenbibliotheken, die teilweise aufeinander aufbauen und zusammenarbeiten, aufgeteilt.

Diese Klassenbibliotheken sind

- AWT (Abstract Window Toolkit)
  Älteste in Java verfügbare Klassenbibliothek zur Programmierung von grafischen Benutzeroberflächen. Ist heute überholt, bildet aber mit einigen Klassen das Fundament des JFC.
  Das AWT greift auf Funktionen des Betriebssystems zu, auf dem das Programm läuft.
  Die Komponenten des AWT bezeichnet man auch als schwergewichtige Komponenten.

- Swing
  Swing Komponenten sind vollständig in Java implementiert. Es werden keine Funktionen des Betriebssystems verwendet. Die Komponenten bezeichnet man als leichtgewichtige Komponenten. Das Aussehen ist daher vom verwendeten Betriebssystem unabhängig.
  Diese Klassenbibliothek ist sehr nützlich und wird zusammen mit dem JDK ausgeliefert.

- Java 2D
  Unterstützt die Erstellung und Bearbeitung von 2D-Grafiken und Bildern.

- Drag and Drop
  Ermöglich Verschieben von Daten oder Grafiken eines Java Programms in ein anderes Programm

- Accessibility
  Wird zur Unterstützung für Leute mit Behinderungen eingesetzt, z.B. spezielle Lesegeräte etc.

Fenster unter grafischen Benutzeroberflächen

Die Grundlage für eine grafische Benutzeroberfläche bildet das Fenster (Frame).

Beispiel: Frame erstellen und öffnen

```
import java.awt.Frame;

public class ErsterFrame{
    public static void main(String[] args){
        Frame f = new Frame ("Titel: Ein Frame");
        f.setSize(400,300);
        f.setVisible(true);
    }
}
```

das Fenster erscheint am linken oberen Bildschirmrand

Die Klasse Frame

Die Klasse Frame befindet sich im Paket `java.awt` und ist abgeleitet von der Klasse `Window`.

Ein Frame ist Window mit Titelleiste.

```
class java.awt.Frame extends Window implements MenuContainer
```

Konstruktoren für Frame

- `Frame()`
  erzeugt ein neues Frame-Objekt, das am Anfang unsichtbar ist
- `Frame(String)`
  erzeugt ein neues Frame-Objekt, das am Anfang unsichtbar ist.
  Das Frame-Objekt hat einen Fenster-Titel.

Nach Aufruf des Konstruktors ist das Fenster vorbereitet und muss noch sichtbar gemacht werden.

Dazu werden Methoden der Klasse `Window`, `Container` oder `Component` verwendet.

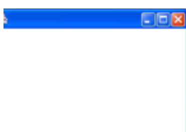
- `void setVisible(true)`
  zeigt Fenster an, bzw. holt es in Vordergrund
- `void show()`
  zeigt Fenster an
- `void setSize(int width, int height)`
  verändert die Höhe und Weite der Komponente

Beispiel: Zwei Frames mit unterschiedlicher Position

```
import java.awt.*;

public class ZweiFrames extends Frame{
    public ZweiFrames(int x, int y){
        super ();
        setSize (x,y);
    }
    public static void main(String args[]){
        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        ZweiFrames f1 = new ZweiFrames(200,100);
        f1.show();
        ZweiFrames f2 = new ZweiFrames(300,200);
        f2.show();
        f1.setLocation((d.width-f1.getSize().width)/2,
                (d.height-f1.getSize().height)/2);
        f1.show();
    }
}
```

ermittelt Größe des Bildschirms

verändert Position des Frames



Frame `f2` wird standardgemäß in der linken oberen Bildschirmecke angezeigt.

Frame `f1` wird in der Bildschirmmitte angezeigt.

Bemerkung: Die Fenster können nicht mit dem x in der Titelleiste geschlossen werden.

Die Methode
`void setLocation(int x, int y)`
setzt die Komponenten an die Position x,y.

Die Bildschirmgröße wird abgefragt durch die Methode `getScreenSize()` der Klasse `Toolkit` aus dem Paket `java.awt`

```
Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
```


Die paint()-Methode

- Die `paint()`-Methode dient zum Zeichnen z.B. in ein Fenster.
- `void paint(Graphics g)` ist eine Methode der Klasse `Component` und steht damit allen von dieser Klasse abgeleiteten Klassen zur Verfügung. Sie muss überschrieben werden.
- Diese Methode wird von der virtuellen Maschine stets aufgerufen, wenn ein Neuzeichnen einer Komponente erforderlich ist.
- Durch das Überschreiben der Methode `paint()` kann innerhalb der Methode direkt auf z.B. den Fensterinhalt gezeichnet werden.
- In der Methode `paint()` wird ein Objekt vom Typ `Graphics` übergeben.
- `Graphics` ist eine Klasse im Paket `java.awt` und enthält viele Methoden zum Zeichnen, z.B. Zeichenketten, Kreise, Grafiken, Linien etc.

Beispiel für das Zeichnen eines Strings in ein Fenster

```
import java.awt.*;

public class PaintBeispiel extends Frame{

    public PaintBeispiel(){
        setSize (400,100);
    }
    public void paint(Graphics g){
        g.drawString("Gezeichnet wird ein String",100,60);
    }

    public static void main(String[] args){
        new PaintBeispiel().show();
    }
}
```

Neuzeichnen durch `repaint()`

- Ein Neuzeichnen wird mit der Methode `repaint()` initiiert.
  Danach kann der Programmcode in der `paint()`-Methode das Fenster neu füllen.

Ereignisse beim AWT

- Bis jetzt konnten wir noch kein von uns erzeugtes Fenster -durch Klicken in der Titelleiste- schließen.
- Beim Arbeiten mit grafischen Oberflächen interagiert der Benutzer mit Komponenten (z.B. Drücken eine Buttons).
- Das grafische System beobachtet die Aktionen des Benutzers und informiert die Applikation über die anfallenden Ereignisse. Die Applikation kann dann entsprechend reagieren.
- Diese Botschaften werden in sog. Event-Klassen unterteilt.

Listener

- Wird ein Ereignis auf der grafischen Oberfläche ausgelöst (z.B. Drücken eines Buttons), so gibt es eine Reihe von Interessenten, die informiert werden möchten.
- Damit ein Interessent nicht über alle Ereignisse, die ausgelöst werden, informiert wird, muss er sich explizit an der –für ihn wichtigen- Ergebnisquelle anmelden. Diese informiert ihn dann, wenn sie ein Ereignis aussendet.
- Dieses Anmelden geschieht durch die Implementierung eines Listeners.

Listener implementieren

- Ein Listener ist eine Schnittstelle, die von den Interessenten implementiert wird.
- Der Interessent muss die Methoden der Schnittstelle implementieren.
- Um ein Fenster korrekt zu schließen, muss das `WindowListener`-Interface implementiert werden.
- Die Methoden, die `WindowListener` definiert, werden mit der Methode `addWindowListener()` der Klasse Frame an ein Fenster gebunden.

Die folgenden zwei Beispiele zeigen, wie ein Fenster geschlossen werden kann.

Beispiel: Fensterschließen 1. Variante

```
import java.awt.*;
import java.awt.event.*;

public class SchliessFrame1 extends Frame implements WindowListener{
    public SchliessFrame1(){
        setSize(400,300);
        addWindowListener(this);
        show();
    }
    //Implenentierung der Methoden des WindowListener
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }

    public void windowClosed(WindowEvent e){};
```

Bindung der Methoden des WindowListener an das Fenster

Diese Methode ist für das Korrekte Schließen zuständig


```
    public void windowDeiconified (WindowEvent e){};
    public void windowActivated(WindowEvent e){};
    public void windowIconified (WindowEvent e){};
    public void windowDeactivated(WindowEvent e){};
    public void windowOpened(WindowEvent e){};
    public static void main(String[] args){

    new SchliessFrame1();
    }
}
```

Diese Methoden werden im Interface deklariert und müssen daher in der Klasse, die das Interface implementiert, implementiert werden.

Damit auch wirklich nur die notwendige Methode `windowClosing()` implementiert werden muss, ist die folgende Variante 2 besser geeignet.

Beispiel: Fensterschließen 2. Variante. Nutzung von Adapterklassen

```
import java.awt.*;
import java.awt.event.*;

public class SchliessFrame2 extends Frame {
    public SchliessFrame2(){
        setSize(400,300);
        addWindowListener(new FensterSchliessAdapter());
        show();
    }
    public static void main(String[] args){
        new SchliessFrame2();
    }
}

class FensterSchliessAdapter extends WindowAdapter{
    public void windowClosing (WindowEvent e){
        System.exit(0);
    }
}
```


Beispiel: Fensterschließen 2.b Variante. Nutzung von Adapterklassen

```
import java.awt.*;
import java.awt.event.*;

public class SchliessFrame3 extends Frame {
    public SchliessFrame3(){
        setSize(400,300);
        addWindowListener(new WindowAdapter(){
            public void windowClosing (WindowEvent e){
                System.exit(0);
            }
        }
        );
        show();
    }
    public static void main(String[] args){
        new SchliessFrame3();
    }
}
```

Das Zeichnen eines speziellen Bildes in ein Fenster

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

public class SetzeIcon extends Frame {
    public SetzeIcon(String titel){
        setSize(400,300); setLocation(50,50);setTitle(titel);

        ImageIcon imageicon = new ImageIcon("T4.gif");
        bild = imageicon.getImage();

        addWindowListener(new WindowAdapter(){
            public void windowClosing (WindowEvent e){
                System.exit(0);
            }
        }
        );
        show();
    }
```


Das Zeichnen eines speziellen Bildes in ein Fenster

```
public void paint(Graphics g){
    g.drawImage(bild,100,100,this);
}
```

Position im Fenster

```
    private Image bild;
    public static void main(String[] args){
        new SetzeIcon("Eine Grafik setzen");
    }
}
```

Die Größe des Bildes kann verändert werden

```
public void paint(Graphics g){
    g.drawImage(bild,100,100,35,40,this);
}
```

Position im Fenster   Breite, Höhe von `bild`

Bemerkung: `ImageIcon` ist eine Klasse aus dem Paket `javax.swing` und muss daher hier importiert werden.

Farben

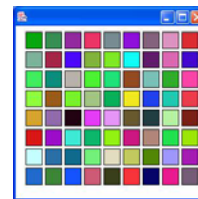
- Zur Verwendung von Farben gibt es in Java die Klasse `Color` aus dem Paket `java.awt`
- Die Klasse stellt viele Methoden zur Erzeugung von Color-Objekten zur Verfügung.
- Es existieren viele bereits vordefinierte Farben in dieser Klasse.
- Ein möglicher Konstruktor ist
  `Color(int r, int g, int b).`
  Durch diesen wird ein Color-Objekt mit den Grundfarben Rot, Grün und Blau erzeugt. Die Werte liegen zwischen 0 und 255.

- Es gibt in der Klasse Color viele bereits vordefinierte Farben, wie

`white`	mit r = g = b =255,
`black`	mit r = g = b = 0,
`red`	mit r = 255, g = b = 0,
`green`	mit r = 0, g = 255, b = 0,
`blue`	mit r = 0, g = 0, b = 255,
`yellow`	mit r = 255, g = 255, b = 0,
`darkGray`	mit r = 128, g = 128, b = 128

Beispiel

Das folgende Programm soll in einem Fenster Quadrate malen, die mit einer zufällig gewählten Farben gefüllt werden. Die Ränder der Quadrate sollen in Schwarz ausgegeben werden.




```
/*aus Ullenboom: Java ist auch eine Insel*/

import java.awt.*;
import java.awt.event.*;

public class FarbenSpiel extends Frame{

    public FarbenSpiel(){
        setSize(300,300);

        addWindowListener(new WindowAdapter(){
            public void windowClosing (WindowEvent e){
                System.exit(0);
            }
        }
        );
    }
```



```
    final private int random(){
        return (int)(Math.random()*256);
    }

    public void paint(Graphics gr){
        for(int x = 20; x < getSize().width - 25; x = x+30)
            for (int y = 40; y < getSize().height-25;y = y+30){
                int r = random(), g = random(), b = random();

                gr.setColor(new Color(r,g,b));
                gr.fillRect(x,y,25,25);
                gr.setColor(Color.black);
                gr.drawRect(x,y,25,25);
            }
    }

    public static void main(String[] args){
        new FarbenSpiel().show();
    }
}
```

Bemerkung zum Programm

- Die Methode `drawRect (int x, int y, int width, int height)` zeichnet die Außenlinie eines Rechtecks in der Vordergrundfarbe. Die Ecken des Rechtecks liegen bei `(x,y), (x,y+height), (x+wight,y)` und `(x+wight, y+height)`
- `fillRect (int x, int y, int width, int height)` zeichnet ein gefülltes Rechteck in der Vordergrundfarbe. Die Ecken des Rechtecks liegen bei `(x,y), (x,y+height-1), (x+wight-1,y)` und `(x+wight-1, y+height-1)`

Hinzufügen von Komponenten

- Ein Container, z.B. ein Frame, nimmt Komponenten auf und setzt sie mit Hilfe eines Layout-Managers an die richtige Position im Container.
- Das Hinzufügen von Komponenten erfolgt durch die Methode `add()`

Buttons unter AWT

- Eine Schaltfläche, d.h. ein Button, wird eingesetzt, wenn der Anwender eine Aktion auslösen soll.
- Ein Button sollte eine Beschriftung haben.
- Ein Button ist ein Objekt der Klasse `Button` aus dem Paket `java.awt`
- Mit `Button b = new Button ("Beenden");` wird ein Button mit der Beschriftung „Beenden" erzeugt.
- Mit `add()` wird er dem Container hinzugefügt.
- Der Layout-Manager bestimmt die Position des Buttons.
- Um eine Funktionalität hinter diesen Button zu legen, muss ein Listener erzeugt werden, der am Button horcht und das entsprechende Ereignis bearbeitet.

Beispiel 1a: Erzeugen eines Buttons (noch ohne Funktionalität)

```
import java.awt.*;
public class SetzeButton1 extends Frame {

    public SetzeButton1(){

        Button b = new Button ("Beenden");
        add(b);


    public static void main(String[] args){
        SetzeButton1 button = new SetzeButton1();
        button.pack();
        button.setVisible(true);
    }
}
```

Beispiel 1b: Erzeugen eines Buttons mit der Funktionalität, dass das Fenster geschlossen wird.

```
import java.awt.*;
import java.awt.event.*;

public class SetzeButton1 extends Frame {
    public SetzeButton1(){
        Button b = new Button ("Beenden");
        add(b);

        ActionListener a = new ActionListener(){
            public void actionPerformed(ActionEvent e){
                System.exit(0);
            }
        };

    b.addActionListener(a);
    }
    …..
```

Implementierung eines Listeners

Anmeldung des Listeners an Button

Beispiel 2: Der Button wird unten in den Frame gesetzt

```
import java.awt.*;
import java.awt.event.*;

public class SetzeButton2 extends Frame {
    public SetzeButton2(){
        Button b = new Button ("Beenden");

        add(b,BorderLayout.SOUTH);

        setSize(300,200);
        ……

    }

    public static void main(String[] args){
        SetzeButton2 button = new SetzeButton2();
        button.setVisible(true);
    }
}
```

Veränderung der Position in den „Süden" des Frames.

Bemerkung zum Einfügen von Komponenten

- Komponenten, wie Buttons, sollten nie direkt in den Frame gesetzt werden.
- Es sollte ein Panel erzeugt werden, das in den Frame passt.
- In dieses Panel werden die Buttons dann hinzugefügt.
- Ein `Panel` ist der einfachste Container in AWT.

Beispiel 3: Erzeugen zweier Buttons im Panel

```
import java.awt.*;
import java.awt.event.*;

public class SetzeButton extends Frame {

    public SetzeButton(){
        Panel p = new Panel();
        Button b = new Button ("1. Button");
        p.add(b);
        p.add(new Button("2. Button"));
        add("South",p);
        setSize(300,200);
        ….
    }
    public static void main(String[] args){
        SetzeButton button = new SetzeButton();
        button.setVisible(true);
    }
}
```

Der Layout-Manager

- Der Layout-Manager bestimmt die Anordnung der Komponenten im Container.
- Layout-Manager werden sowohl von AWT als auch von Swing benutzt.
- Die verschiedenen Layout-Manager befinden sich im Paket `java.awt`.
- Swing kann auch die speziellen Layout-Manager aus dem Paket `javax.swing` benutzen.
- Mit der Methode
  `setLayout(new java.awt.`*`Layoutmanager (Parameter)`*`);`
  wird der Layout-Manager bestimmt.
- Im Paket `java.awt` gibt es die 5 Layout-Manager `FlowLayout, BorderLayout, GridLayout, CardLayout, GridBagLayout.`

BorderLayout

- Dieser Layout- Manager ist voreingestellt bei `(J)Frame` und `(J)Window`.
- Die Komponenten eines Containers werden auf die vier Randbereiche (oben,unten, links, rechts) sowie den Mittelbereich aufgeteilt.

Beispiel BorderLayout-Manager

```
import java.awt.*;

public class BorderLayoutBeispiel extends Frame {
    public BorderLayoutBeispiel(){
        setSize(300,200);
        //setLayout(new BorderLayout());
        add(BorderLayout.NORTH, new Button ("Norden"));
        add(BorderLayout.SOUTH, new Button ("Süden"));
        add(BorderLayout.WEST, new Button ("Westen"));
        add(BorderLayout.EAST, new Button ("Osten"));
        add(BorderLayout.CENTER, new Button ("Mitte"));

        setVisible(true);
    }

    public static void main(String[] args){
        new BorderLayoutBeispiel();
    }
}
```


FlowLayout

- Das FlowLayout positioniert die Komponenten in einer Zeile von links nach rechts.
- Ist die Zeile mit Komponenten gefüllt, wird in eine neue Zeile umgebrochen.
- Die Klasse `FlowLayout` ist der Standard-Layout-Manager für die Container `Applet, Panel, JPanel.`
- Der Konstruktor ist entweder Parameterlos oder die Parameterliste besteht aus drei Parametern
  `public FlowLayout(int align, int hgap, int vgap)`

FlowLayout

- `align` gibt an, ob die Komponenten je Zeile linksbündig, rechtsbündig oder zentriert eingefügt werden. Erlaubt sind die Werte `FlowLayout.RIGHT, FlowLayout.LEFT, FlowLayout.CENTER.`
- `hgap` ist der horizontale Abstand in Pixel
- `vgap` ist der vertikale Abstand in Pixel.

rechtsbündig mit
Abstand 10 Pixel

Beispiel FlowLayout-Manager

```
import java.awt.*;

public class FlowLayoutBeispiel extends Frame {

    public FlowLayoutBeispiel(){
        setSize(300,200);
        setLayout(new FlowLayout(FlowLayout.RIGHT,10,10));

        add(new Button ("Button 1"));
        add(new Button ("Button 2"));
        add(new Button ("Button 3"));

        setVisible(true);
    }

    public static void main(String[] args){
        new FlowLayoutBeispiel();
    }
}
```


Oberflächenprogrammierung mit Swing

- Die Swing-Klassenbibliothek baut auf AWT auf.
- Die Containerklassen `Frame, Dialog, Window` und `Applet` sind von AWT übernommen worden und durch Ableitung erweitert worden.
- Die Klassen in Swing heißen `JFrame, JDialog, JWindow, JApplet.`
- Swing unterscheidet sich von AWT z.B. beim Einfügen von Komponenten in einen Container.

Einfügen von Komponenten

- Die Klassen des Swing-Pakets haben einen sog. ContentPane, an die die GUI-Komponenten angehängt werden um zur Laufzeit angezeigt zu werden.
- Das ContentPane ist ein Objekt der Klasse `Container` im Paket `java.awt.`
- Die Methode `getContentPane()` gibt eine Referenz auf die ContentPane des jeweiligen Swing-Objekts. An dies kann man die GUI-Komponente mit der Container-Methode `add()` anhängen.

Das folgende Beispiel zeigt die Erzeugung eines `JFrame`s mit einem Button und der Funktionalität „Schließ Fenster“.

Beispiel JFrame

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

public class ErsterJFrame extends JFrame{
        public static void main(String[] args){
            JFrame f = new JFrame ("Ein JFrame");
            JButton b = new JButton ("Button 1");
            f.getContentPane().setLayout(new FlowLayout());
            f.getContentPane().add(b);

            f.addWindowListener(new WindowAdapter(){
                public void windowClosing (WindowEvent e){
                    System.exit(0);
                }
            });
            f.setSize(300,200);
            f.setVisible(true);
        }
}
```

Einige GUI-Komponenten von Swing

- `JPanel`

  Stellt eine Oberfläche bereit, auf der GUI-Elemente platziert werden können oder auf der gezeichnet werden kann. Sie ist unsichtbar, kann aber Ereignisse erkennen.
  Das `JPanel` ist eine wesentliche Unterstützung bei der Strukturierung komplexer GUI-Fenster.
- `JButton`
- `JLabel`

  Dient zur kurzen Textdarstellung.

- `JCheckBox`
  Klasse stellt Button dar, der vom Anwender wahlweise an- oder ausgeschaltet werden kann.
- `JComboBox`

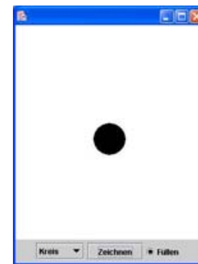
  Die Klasse implementiert eine Auswahlbox.
- `JList`

  Listendarstellung mit Auswahlmöglichkeit. Sie hat eine vertikale Bildlaufleiste und ermöglicht die gleichzeitige Auswahl mehrere Elemente.
- `JRadioButton`

  Button, der wahlweise an- oder ausgeschaltet werden kann.
- `JScrollPane`

  Klasse implementiert einen Scroll Balken.

Ein Beispiel für ein GUI mit ComboBox und RadioButton



Der Anwender soll auswählen könne, ob ein Kreis, Quadrat oder Dreieck gezeichnet wird.

Außerdem soll er auswählen können, ob die Figur gefüllt oder nicht gefüllt wird.


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

public class AuswahlZeichnen extends JFrame{

        private JComboBox figurWahl = new JComboBox();
        private JButton startbutton = new JButton("Zeichnen");
        private JRadioButton fuellknopf =
                               new JRadioButton("Füllen",false);
        private Bild leinwand = new Bild();

        //Konstruktor
        public AuswahlZeichnen(){

            //Hinzufügen von leinwand zum ContentPane in Mitte
            getContentPane().add(leinwand, BorderLayout.CENTER);
```



```
            //Steuerpanel
            JPanel steuerflaeche = new JPanel();

            //Hinzufügen zur ComboBox
            figurWahl.addItem("Kreis");
            figurWahl.addItem("Quadrat");
            figurWahl.addItem("Dreieck");

            //Hinzufügen zum Panel steuerflaeche
            steuerflaeche.add(figurWahl);
            steuerflaeche.add(startbutton);
            steuerflaeche.add(fuellknopf);

            //Hinzufügen des Panels steuerflaeche zum ContentPane
            getContentPane().add(steuerflaeche,
                                   java.awt.BorderLayout.SOUTH);
```



```
            //Ereignisbearbeitung: Registierung des Beobachters am
            //startbutton.
            startbutton.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent evt){

                    leinwand.setfuellstatus(fuellknopf.isSelected());
                    leinwand.setAuswahl(figurWahl.getSelectedIndex());
                    leinwand.repaint();
                }
            });
            setSize(300,400);
        }
```



```
        //die main Methode
        public static void main(String args[]){
            AuswahlZeichnen fenster = new AuswahlZeichnen();
            fenster.show();

            //Schließen des Frames
            fenster.addWindowListener(new WindowAdapter(){
            public void windowClosing (WindowEvent e){
                System.exit(0);
            }
        }
        );

        }
}
```

Die Klasse `Bild`

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

public class Bild extends JPanel{

        int auswahl = 1;
        boolean status = false;

        public Bild(){
        auswahl = 10;
        show();
        }
```



```
    //die Methode paint zum Zeichnen der Objekte
    public void paint(Graphics g){
        g.setColor(Color.white);
        g.fillRect(0,0,getSize().width, getSize().height);
        g.setColor(Color.black);

        switch(auswahl){
        case 0:
                          if (status )
                g.fillOval(120,150,50,50);
            else
                g.drawOval(120,150,50,50);
            break;

        case 1:
            if (status)
                g.fillRect(120,150,50,50);
            else
                g.drawRect(120,150,50,50);
            break;
```



```
            case 2:
                //Zeichne Dreieck
                break;
            case 10:
                break;
            }
        }

        public void setfuellstatus(boolean f){
            status = f;
        }

        public void setAuswahl(int a){
            auswahl = a;
        }
}
```