

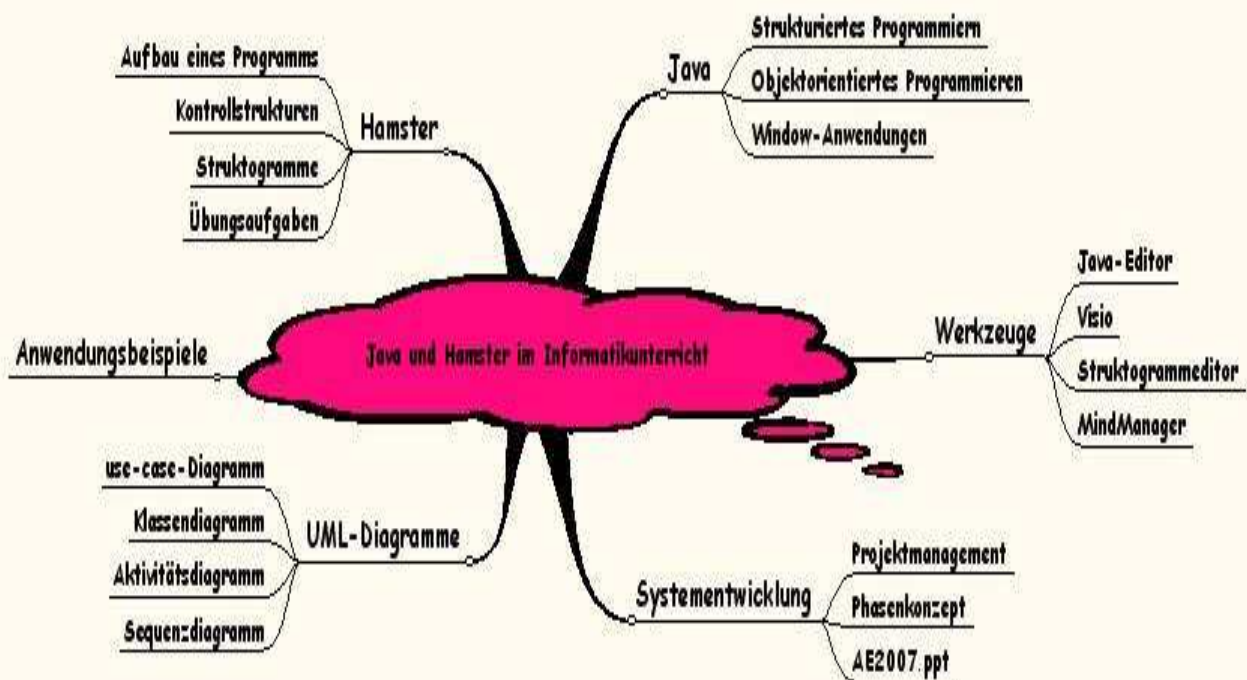
# Skript für den Informatikunterricht an der Max-Weber-Schule



Jochen Pellatz



Version 3.0 (2011)



Teil A: Programmieren lernen mit dem Java-Hamster-Modell

Teil B: Strukturiertes Programmieren mit Java und dem Java-Editor

Teil C: Objektorientierte Programmentwicklung mit Java und UML



# Programmieren lernen mit dem Hamstermodell



## Teil A. Einführung in die Hamsterprogrammierung

Das Hamstermodell ist eine auf der Programmiersprache JAVA basierendes Werkzeug, welches die Grundlagen der Programmierung auf einfache und anschauliche Weise vermittelt.

Der Hamster wurde an der Universität Oldenburg von Dietrich Bowles entwickelt. Die Syntax der Hamstersprache ist an den Programmiersprachen JAVA und C++ orientiert. Mit der Version 2 bietet der Hamster auch die Möglichkeit, dass objektorientierte Programmieren zu erlernen.

Die gesamte Software für das Hamstermodell ist frei zugänglich. Programm und Installationsanleitung finden sich unter [www.java-hamster-modell.de](http://www.java-hamster-modell.de).

Einige Ideen zu diesen Skript stammen von dem Kollegen Ulrich Helmich, der auf seiner Webseite [www.u-helmich.de](http://www.u-helmich.de) einen Hamsterkurs veröffentlicht hat.

## 0. Installationshinweise und Funktionsweise

### Installation

Laden Sie die aktuelle Version des Hamstersimulators von der angegebenen Internetseite. Folgen Sie den Installationshinweisen. Zuerst muss eine Java-Laufzeitumgebung auf Ihrem Rechner installiert werden, ohne die der Hamstersimulator nicht funktioniert. Der Hamstersimulator wird mit der Datei **hamstersimulator.bat** oder **hamstersimulator.jar** gestartet. Beide Dateien befinden sich im Unterverzeichnis **hamstersimulator-v27-01**.

### Funktionsweise

Ein Hamsterfeld kann unter mit der Endung **ter** gespeichert werden. Bsp. `territorium1.ter`  
Ein im Editor erstelltes Hamsterprogramm wird mit der Endung **ham** gespeichert.

Bsp. `Programm1.ham`.

Nach erfolgreicher Compilierung entstehen zwei neue Programme mit der Endung **class** und **java** (Also hier: `Programm1.java` und `Programm1.class`).

Soll ein fertiges Hamsterprogramm geladen werden, dann muss das Programm mit der Endung **class** aufgerufen werden.

## Übersicht über die Hamster-Befehle

Funktion	Beschreibung	Typ
<code>vor()</code>	Der Hamster geht genau 1 Feld weiter	void
<code>linksUm()</code>	Der Hamster dreht sich um 90° nach links	void
<code>nimm()</code>	Der Hamster nimmt ein Korn auf	void
<code>gib()</code>	Der Hamster legt ein Korn ab	void
<code>vornFrei()</code>	Liefert TRUE, falls der Hamster nicht vor einer Wand steht	boolean
<code>kornDa()</code>	Liefert TRUE, falls das Feld, auf dem der Hamster gerade steht, mindestens ein Korn enthält.	boolean
<code>maulLeer()</code>	Liefert TRUE, falls der Hamster kein Korn im Maul hat.	boolean



# Programmieren lernen mit dem Hamstermodell



## 1. Erste Schritte

### Hinweise zu den Aufgaben des Skriptes

Legen Sie einen Ordner mit ihrem Namen und Anfangsbuchstaben ihres Vornamens an.  
Bsp.: Sie heißen Sebastian Weber → Name ihres Ordners: *WeberS*

Schreiben Sie in die oberste Reihe jedes Hamsterprogrammes ihren Namen als Kommentar.  
Ein Kommentar für eine Zeile beginnt mit //  
Bsp.: *// Sebastian Weber Aufgabe 1*

Speichern Sie die Aufgaben und Territorien unter der Aufgabennummer in ihrem Ordner ab.  
Bsp.: *aufgabe1.ham* und *aufgabe1.ter*

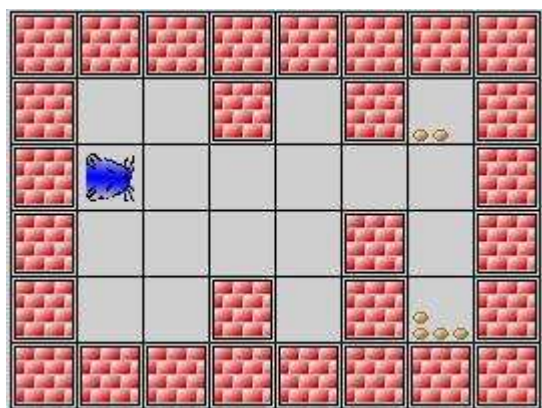
Verwenden Sie innerhalb des Dateinamens keine Leerzeichen und Sonderzeichen. Der Name sollte immer mit einem Buchstaben beginnen.

Gelöste Aufgaben werden mit Punkten bewertet. Die Punkte können nur vergeben werden, wenn Sie sich an obige Regeln halten. Zum ‚Bestehen‘ des Hamsterkurses ist eine bestimmte Anzahl an Punkten erforderlich.

Um der angeborenen Trägheit vieler Schüler entgegenzuwirken, die sich fertige Aufgaben einfach von Mitschülern in ihren Ordner kopieren (wobei sie hoffentlich so schlau sind, den Namen in der Kommentarzeile zu ändern), werde ich mir die Lösungen von Zeit zu Zeit von Schülern persönlich erläutern lassen.

### Aufgabe 1 (1 Punkt)

Erstellen Sie folgendes Hamsterterritorium und speichern Sie es in Ihrem Ordner unter *aufgabe1.ter*





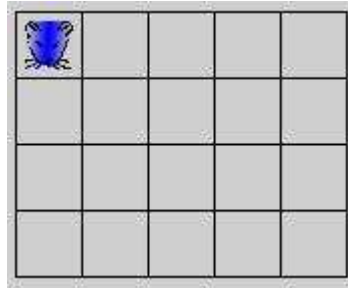
# Programmieren lernen mit dem Hamstermodell



## Aufgabe 2 (1 Punkt)

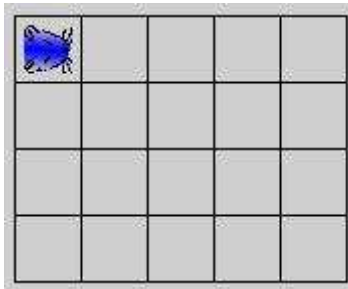
Erzeugen Sie ein Territorium mit 5 Spalten und 4 Zeilen. Der Hamster steht in der linken oberen Ecke mit Blickrichtung Süd.

Lassen Sie den Hamster ein Mal um das ganze Feld bis zu seinem Ausgangspunkt laufen.



## Aufgabe 3 (1 Punkt)

Wie Aufgabe 2. Der Hamster steht jedoch in Blickrichtung Ost.



## 2. Prozeduren

Sicher ist ihnen aufgefallen, dass Aufgabe 3 etwas aufwändig ist, weil der Hamster sich jetzt in jeder Ecke nach rechts drehen muss. Da es aber in der Hamstersprache keine Anweisung *rechtsUm()* gibt, müssen wir dies mit einer dreifachen Linksdrehung lösen. Wenn dies nun häufiger vorkommt, ist dies ziemlich viel Schreibarbeit. Dafür gibt es nun die Möglichkeit eine **Prozedur** zu verwenden.

Mit einer Prozedur schreiben wir uns so zu sagen eine neue Anweisung, die aus mehreren Befehlen der Hamstersprache besteht. Wir kennen bereits die Prozedur *main()*, die jedes Hamsterprogramm enthält. Ähnlich sieht jetzt unsere neue Prozedur *rechtsUm()* aus.

Bei der Vergabe des Namens für eine Prozedur sollten wir uns an die Java-Konvention halten:

Anfangsbuchstabe des ersten Wortes klein, Anfangsbuchstabe der folgenden Worte groß.

Bsp.: *linksUm()*      *rechtsUm()*      *sucheKorn()*      *kommNachHaus()*

Der Aufbau einer Prozedur ist immer ähnlich:

```
void Name der Prozedur ( )  es muss immer ein Klammerpaar folgen
{                             Block auf
    Anweisungen;             die Anweisungen der Prozedur stehen zwischen Blockklammern
}                             Block zu
```



# Programmieren lernen mit dem Hamstermodell



Hier nun der Quellcode des kompletten Programms zu Aufgabe 3 mit einer Prozedur *rechtsUm()*:

```
void main( )
```

```
{
```

```
    vor( ); vor( ); vor( ); vor( );
```

```
    rechtsUm( );
```

```
    vor( ); vor( ); vor( );
```

```
    rechtsUm( );
```

```
    vor( ); vor( ); vor( ); vor( );
```

```
    rechtsUm( );
```

```
    vor( ); vor( ); vor( );
```

```
    rechtsUm( );
```

```
}
```

```
void rechtsUm( )
```

```
{
```

```
    linksUm( ); linksUm( ); linksUm( );
```

```
}
```

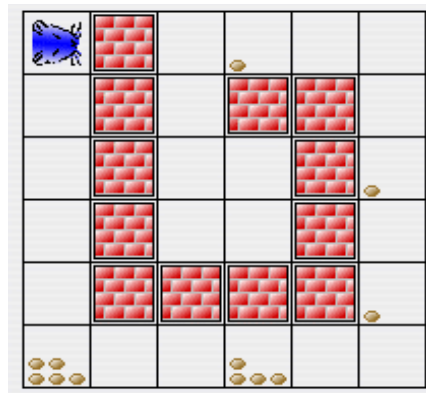
vor
vor
vor
vor
rechts um
vor
vor
vor
rechts um
vor
vor
vor
vor
rechts um
vor
vor
vor
rechts um

Darstellung als  
Struktogramm

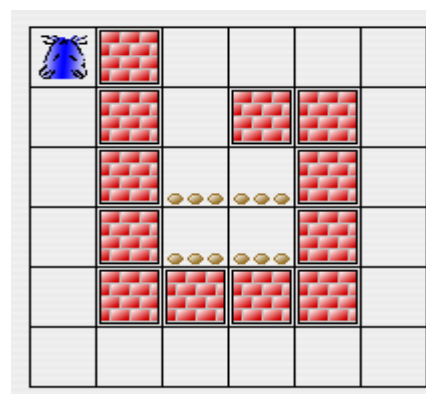
**„lineare Struktur“**

## Aufgabe 4 (3 Punkte) (ohne Fleiß kein Preis)

Erstellen Sie ein neues Territorium und speichern Sie es in Ihrem Ordner unter einem neuen Namen ab:



Schreiben Sie nun ein Programm unter Verwendung der Prozedur *rechtsUm()*, welches zu folgender Situation führt:





# Programmieren lernen mit dem Hamstermodell



## 3. Wiederholungskonstrukte (Schleifen)

Der Hamster steht an der linken Wand mit Blickrichtung nach Ost. Er soll bis zu nächsten Wand laufen, kehrt machen und zum Ausgangspunkt zurück laufen.

Jetzt haben wir ein Problem: Wenn wir die Größe des Hamsterterritoriums nicht kennen, wissen wir nicht, wie oft der Hamster vor gehen soll. Geht er zu oft vor, läuft er gegen die Wand und stirbt.

Die Lösung des Problems besteht darin, dass wir den Hamster die Anweisung `vor( )` so lange ausführen lassen, wie das Feld vor ihm frei ist. Um dieses zu prüfen, gibt es die Funktion `vornFrei( )` in der Hamstersprache.

Frei übersetzt könnten wir dem Hamster jetzt folgendes mitteilen:

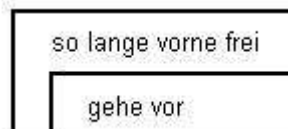
So lange das Feld vor dir frei ist, laufe vor!

In die Hamstersprache übersetzt sieht das so aus:

```
while (vornFrei( ))  
{  
    vor( );  
}
```

Die Anweisung `vor( )` wird nun so lange ausgeführt, wie die Bedingung in der Klammer erfüllt ist.

Die Darstellung einer while Schleife im Struktogramm sieht so aus:

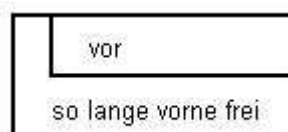


### Aufgabe 5 (2 Punkte)

Der Hamster steht in der linken oberen Ecke eines beliebig großen Territoriums mit Blickrichtung Ost. Er soll analog zu Aufgabe 3 einmal um das ganze Feld laufen bis zu seinem Ausgangspunkt.

In der Programmierung gibt es noch andere Schleifen. Insbesondere ist hier die `do ....while – Schleife` und die `for – Schleife` zu nennen.

Die `do...while` Schleife funktioniert ähnlich wie die `while-` Schleife. Die Prüfung der Bedingung erfolgt aber erst am Ende, so dass die Schleife mindestens ein Mal durchgeführt wird.



Die `for – Schleife` wird später behandelt.



# Programmieren lernen mit dem Hamstermodell



## 4. Fallabfrage – if ... else

In jeder Programmiersprache gibt es die Möglichkeit, bestimmte Teile des Quelltextes nur dann ausführen zu lassen, wenn eine bestimmte Bedingung erfüllt ist. Man spricht hier auch von Verzweigungen oder bedingten Anweisungen. Am besten schauen wir uns dazu ein einfaches Beispiel an:

```
if(vornFrei())
    vor();
else
    linksUm();
```



In Java werden solche Verzweigungen im Programmfluss mit Hilfe von IF-ELSE-Anweisungen realisiert.

Die Struktogrammdarstellung einer Verzweigung ist oben dargestellt.

Betrachten wir noch einige Quelltextbeispiele für korrekte IF-ELSE-Anweisungen:

---

```
if (vornFrei()) vor();
```

Ein sehr einfaches Beispiel. Der Hamster geht vorwärts, wenn vor ihm ein Feld frei ist.

---

```
if (! vornFrei())
{
    linksUm();
    linksUm();
}
```

Wieder sehr einfach. Diesmal werden allerdings zwei Anweisungen ausgeführt, wenn die Bedingung erfüllt ist. Das Zeichen ! bedeutet 'nicht' und kehrt die Bedingung um.

---

```
if (vornFrei())
    vor();
else
{
    linksUm();
    if (vornFrei()) vor();
}
```

Ein etwas komplizierteres Beispiel mit der ELSE-Variante. im IF-Zweig steht nur eine einzelne Anweisung, daher sind keine geschweiften Klammern notwendig. Im ELSE-Zweig stehen zwei Anweisungen, daher sind Klammern erforderlich. Und noch etwas neues: Im ELSE-Zweig ist die zweite Anweisung wiederum eine IF-Anweisung. Wir dürfen IF- bzw. IF-ELSE-Anweisungen also schachteln.

Es können auch mehrere Bedingungen miteinander verknüpft werden. Dazu gibt es die logischen Operatoren && (und), || (oder), == (ist gleich), > (größer) oder < (kleiner). Um eine Aussage zu negieren (verneinen), verwendet man den ! – Operator.

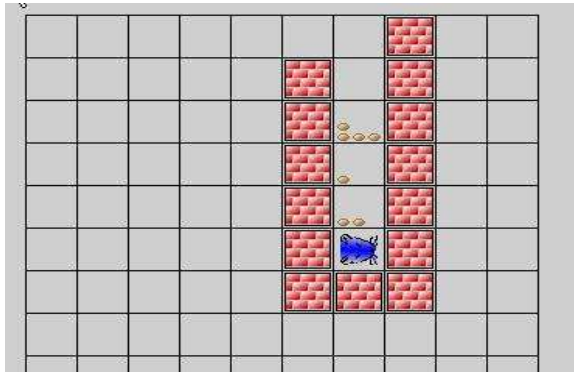


# Programmieren lernen mit dem Hamstermodell



## Aufgabe 6 ( 4 Punkte)

Der Hamster sitzt am Ende eines Ganges unbekannter Länge. Seine Blickrichtung ist unbekannt. Auf einigen Feldern des Ganges liegt jeweils ein Korn. Der Hamster soll alle Körner im Gang aufsammeln und in die linke obere Ecke des Feldes gehen und dort stehen bleiben.



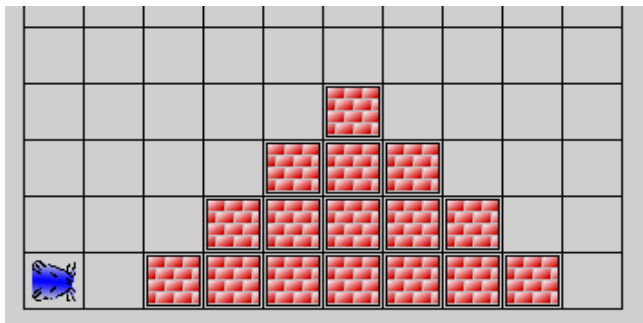
## Aufgabe 7 (4 Punkte)

Der Hamster steht vor einem Berg mit 4 Stufen. Er soll die Stufen aufsteigen und auf der anderen Seite wieder absteigen.

Lösen Sie die Aufgabe mit Prozeduren.

Die Prozedur main sieht so aus:

```
void main()
{
    steigAuf();
    steigAb();
}
```



## Aufgabe 8 (5 Punkte)

Der Hamster steht vor einem Berg unbekannter Höhe. Er soll herauf und auf der anderen Seite wieder herunter steigen.





# Programmieren lernen mit dem Hamstermodell



## Aufgabe 9 ( 5 Punkte )

Der Hamster befindet sich in einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern in der linken unteren Ecke mit Blickrichtung Ost. Es befinden sich wahllos eine unbekannte Anzahl Körner im Raum. Der Hamster soll alle Körner aufsammeln und dann stehen bleiben.

Lösen Sie die Aufgabe, indem Sie das Feld **reihenweise** abgrasen.

## 5. Boolesche Prozeduren

Zur Hamstersprache gehören die Prozeduren *vornFrei*, *kornDa* und *maulLeer*. Dieses sind so genannte boolesche Prozeduren, da sie einen Wert vom Typ boolean liefern. Ein boolescher Wert ist ein Wahrheitswert, der entweder wahr (true) oder falsch (false) sein kann. So kann die Antwort auf die Frage *vornFrei* entweder true oder false sein. Wir können uns eigene boolesche Prozeduren schreiben.

Beispiel: Wir wollen eine Prozedur *linksFrei* schreiben, die testet ob links vom Hamster eine Mauer ist.

```
void main( )
{
    if (linksFrei( ))                // Aufruf der Prozedur linksFrei
        linksUm( );
    else
        vor( );
}

boolean linksFrei( )
{
    linksUm( );
    if (vornFrei( ))
    {
        rechtsUm( );                // Prozedur rechtsUm nicht explizit aufgeführt
        return true;                // liefert den Wahrheitswert true zurück
    }
    else
    {
        rechtsUm( );
        return false;
    }
}
```

Dem Namen einer booleschen Prozedur wird mit das Wort boolean vorangestellt. Dies bedeutet, dass die Prozedur einen booleschen Wert (true oder false) zurückliefert. Das zurück liefern geschieht mit dem Wort return. Return muss immer die logisch letzte Anweisung in einer Prozedur sein. In diesem Fall muss sich der Hamster vorher allerdings erst wieder in seine Ausgangsposition drehen.

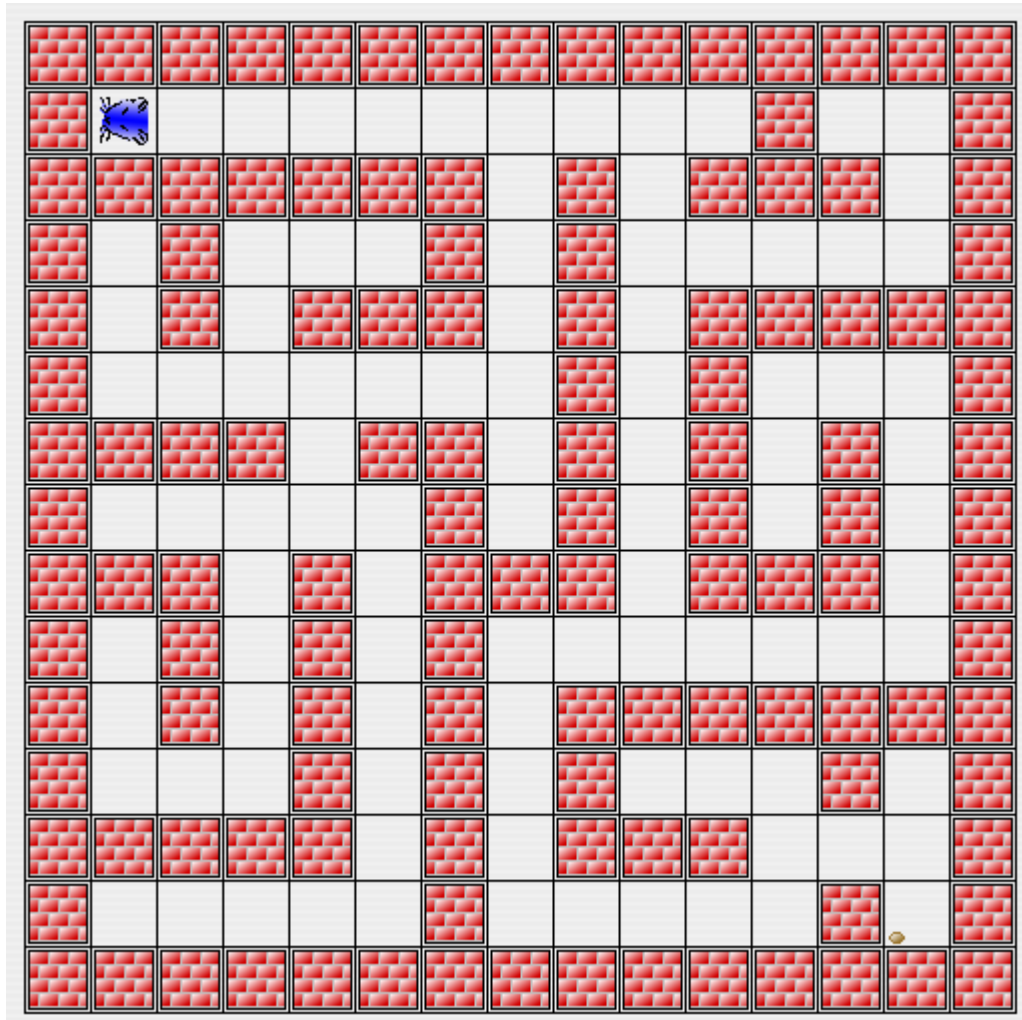


# Programmieren lernen mit dem Hamstermodell



## Aufgabe 10 (8 Punkte)

Erstellen Sie ein Territorium mit einem Labyrinth wie zum Beispiel das Folgende:



Wichtig: Der Hamster muss alle Felder erreichen können. Dazu dürfen die Wege nicht mehr als 2 Felder breit sein. Irgendwo in dem Labyrinth befindet sich ein einziges Korn. Das soll der Hamster finden und auf dem entsprechenden Feld soll er stehen bleiben.

Ihr Programm soll jetzt für alle denkbaren Labyrinth funktionieren! Und es soll möglichst kurz sein. Ideal wären 20 Zeilen Quelltext oder weniger. Das ist aber nur sehr schwer zu schaffen. Überlegen Sie mal, wie man aus einem Labyrinth garantiert herausfindet! Sie können z. B. eine boolesche Prozedur verwenden, die testet, ob links oder rechts vom Hamster eine Mauer ist. (z.B. linksFrei)



# Programmieren lernen mit dem Hamstermodell



## 6. Verwenden von Variablen

Problem: Der Hamster steht irgendwo mitten in einem Territorium. Er hat kein Korn im Maul. Er soll in Blickrichtung bis zur Wand laufen und wieder auf sein Ausgangsfeld zurückkehren.

Damit der Hamster diese Aufgabe lösen kann, muss er Zählen lernen.

Problemlösung: Der Hamster zählt die Anzahl der Schritte bis zur Wand, macht kehrt und zählt die gleiche Anzahl Schritte wieder zurück.

Wir benötigen eine **Variable**, die sich die Anzahl der Schritte merkt. Wir nennen diese Variable z.B. *steps*. Zu einer Variablen gehört immer ein bestimmter Datentyp. Dieser gibt an, welche Art von Daten in der Variable gespeichert werden. Im wesentlichen unterscheidet man drei Arten von Variablen:

int	-	integer, ganze Zahlen
float	-	Fließkommazahlen, d.h. Zahlen mit Nachkommastelle
char	-	character, Zeichen (Buchstaben, Ziffern, Sonderzeichen)

Für den Schrittzähler benötigen wir eine Variable vom Typ int.

Zu Beginn müssen wir die Variable dem Programm bekannt machen. Man nennt dies **Deklariieren**. Dazu gehören Datentyp und Name der Variablen.

```
int steps;
```

Wenn wir der Variablen einen Wert zuweisen wollen, verwendet man das Gleichheitszeichen. Dies kann auch schon bei der Deklaration geschehen.

```
steps = 0;           // steps bekommt den Wert 0.
```

Hier das Programm

```
void main( )
{
    int steps = 0;           //Initialisierung bei Deklaration
    while (vornFrei( ))
    { vor( );
      steps++;               //dies entspricht der Anweisung steps = steps+1
    }
    kehrt( );
    while(steps >0)
    { vor( );
      steps--;               //dies entspricht der Anweisung steps = steps-1
    }
}
```



# Programmieren lernen mit dem Hamstermodell



## 7. Funktionen mit Parametern

Angenommen wir benötigen in einem Programm sehr häufig die Anweisung `vor( )`. Dabei soll sich der Hamster jeweils mehrere Schritte vorwärts bewegen. Es ist aber mühsam, mehrmals hintereinander die Anweisung `vor( )` zu schreiben. Schön wäre es, wenn wir dem Hamster mit einer einzigen Anweisung mitteilen könnten, wie viele Schritte er vor gehen soll: Etwa so: `vor(4)`, d.h. Hamster gehe vier Schritte vor! Dieses geht in der Tat, wenn wir eine entsprechende Prozedur schreiben.

### a) Funktion mit **Übergabewert**

```
void main()
```

```
{  
    vor(4);  
}
```

Die Anweisung `vor(4)` ruft die Funktion **vor** auf und übergibt ihr die Anzahl der Schritte, indem sie die Anzahl in die Klammer schreibt.

```
void vor(int s)
```

```
{  
    int a=0;  
    while(a < s)  
    { vor();  
        a++;  
    }  
}
```

Bei der Deklaration der Funktion `vor` steht der Datentyp und ein beliebiger Bezeichner für eine Variable in der Klammer. Unter diesem Bezeichner wird der übergebene Wert in der Funktion angesprochen. Man nennt eine solche Funktion eine Funktion mit **Übergabeparameter**.

Beachte: Diese Funktion ist nicht identisch mit der Hamsteranweisung `vor( )`, obwohl sie den gleichen Namen hat. Sie unterscheidet sich von `vor( )` dadurch, dass sie in der Klammer einen Übergabeparameter besitzt. Man nennt Funktionen, die unter gleichem Namen mehrfach existieren **überladene** Funktionen. Dies benötigt man eher bei der objektorientierten Programmierung.

### b) Funktion mit **Rückgabewert**

```
int zaehleSchritte( )
```

```
{  
    int steps=0;  
    while(vornFrei( ))  
    {  
        steps++;  
        vor( );  
    }  
    return steps;  
}
```

Der Datentyp vor dem Funktionsnamen gibt den Type des Rückgabewertes an. Die Funktion liefert einen Wert vom Datentyp integer zurück.

Mit **return** wird der Wert der Variablen `steps` an den aufrufenden Programmteil zurückgeliefert.

### Aufruf im Hauptprogramm

```
void main( )
```

```
{ int s;  
    s = zaehleSchritte( );  
}
```

Der Rückgabewert wird der Variablen `s` zugewiesen



# Programmieren lernen mit dem Hamstermodell



## Aufgabe 11 ( 3 Punkte)

Der Hamster steht in der linken oberen Ecke eines 10 x 10 großen Territoriums mit Blickrichtung Ost. Er soll 5 Mal um das ganze Feld herum laufen und auf seiner Ausgangsposition stehen bleiben. Lösen Sie das Problem mit folgenden Vorgaben:

1. Verwenden Sie eine überladene Funktion *vor* für das Laufen entlang der Wand.
2. Verwenden Sie eine überladene Funktion *linksUm* für die Rechtsdrehung.
3. Die main-Funktion besteht aus einer Schleife, in der die Anweisungen für das Laufen entlang der Mauer und die Rechtsdrehung viermal ausgeführt werden. Zählen Sie die Zahl dieser Anweisungen in einer Variablen hoch. Bei 5 Umrundungen sind das  $5 \times 4 = 20$  Durchläufe.

Bsp.:       $z = 0$   
              solange  $z < 20$   
                  vor(9)  
                  linksUm(3)  
                   $z++$

Dies ist Pseudocode und entspricht nicht der korrekten Syntax.

## Aufgabe 12 (4 Punkte)

Der Hamster hat eine große Anzahl Körner im Maul und steht irgendwo in einem beliebig großen Feld. Seine Aufgabe ist es, die Größe des Feldes in Zeilen und Spalten zu ermitteln und eine entsprechende Anzahl Körner auf den ersten beiden Feldern abzulegen. (1. Feld – Anzahl Spalten, 2. Feld – Anzahl Zeilen)

Verwenden Sie für die Ermittlung der Zeilen und Spalten Funktionen mit Rückgabewert.

## Aufgabe 13 ( 7 Punkte) (Abwandlung von Aufgabe 9)

Der Hamster befindet sich in einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern in der linken unteren Ecke mit Blickrichtung Ost. Es befinden sich wahllos eine unbekannte Anzahl Körner im Raum. Der Hamster soll alle Körner aufsammeln und dann stehen bleiben. Der Hamster soll das Feld *spiralförmig* abgrasen.

## Aufgabe 14 ( 10 Punkte) (Erweiterung von Aufgabe 10)

Der Hamster steht am Eingang eines Labyrinthes wie in Aufgabe 10. Im Labyrinth befinden sich beliebig viele Körner. Der Hamster soll alle Körner fressen und wieder auf sein Ausgangsfeld zurückkehren und alle Körner vor sich ablegen.

## 8. Der Hamster lernt Lesen und Schreiben

Mit den Funktionen **schreib( )** und **liesZahl( )** bzw. **liesZeichenkette( )** oder **liesZahl( )** kann der Hamster einen Antwort ausgeben oder eine Information einlesen. Beispiele::

**schreib("Ich bin am Ziel!")**      gibt einen Text aus.

**schreib("Ich habe "+n+" Körner im Maul")**      gibt einen Text und den Wert einer Variablen aus.

**zahl=liesZahl("Gib eine Zahl ein: ")**      fordert den Benutzer auf, einen Wert für die Variable zahl einzugeben.



# Programmieren lernen mit dem Hamstermodell



## Teil B. Objektorientierte Hamsterprogrammierung

Um den Unterschied zum imperativen Hamstermodell im Teil A zu erkennen, analysieren wir zunächst noch einmal das imperative Modell des vorhergehenden Teils:

- Es existiert nur ein einzelner Hamster
- Es gibt die vier Befehle *vor*, *linksUm*, *gib* und *nimm* sowie die Testbefehle *vornFrei*, *maulLeer* und *kornDa*.
- Der Hamster besitzt drei Eigenschaften(Attribute):
- er sitzt auf einer bestimmten Kachel im Territorium
- er schaut in eine bestimmte Blickrichtung
- er hat eine bestimmte Anzahl Körner im Maul.
- Der Hamster besitzt keinen Namen.
- Der Hamster wird vor dem Ausführen des Programms initialisiert, d.h. er wird auf eine bestimmte Kachel gesetzt, seine Blickrichtung wird festgelegt und er bekommt eine bestimmte Anzahl von Körnern ins Maul. Er wird nicht erst im Hamsterprogramm ‚geboren‘.

Der objektorientierte Ansatz weicht davon ab:

- Es können mehrere Hamster existieren
- Jeder Hamster kennt außer den genannten noch einige zusätzliche Befehle
- Hamster können Namen haben.
- Jeder Hamster besitzt drei Attribute Position, Blickrichtung und Körnerzahl.
- Zusätzliche Hamster werden während der Laufzeit des Programms ‚geboren‘.
- Auf einer Kachel können sich mehrere Hamster befinden.
- Wenn ein Hamster einen Fehler macht, gibt es einen Programmabbruch

### 1. Wir erzeugen einen neuen Hamster

Es gibt einen Standardhamster, der bereits vor Beginn des Programms initialisiert wurde, sowie zusätzliche Hamster, die im Programm erzeugt werden.

Wir wollen dem Standardhamster den Namen *willi* geben und einen neuen Hamster namens *heidi* erzeugen.

```
Hamster willi = Hamster.getStandardHamster( );           //Standardhamster
```

```
Hamster heidi = new Hamster( );                          //Erzeugen eines neuen Hamsters
```

Da der neue Hamster hat noch keine Eigenschaften. Deshalb muss er zunächst initialisiert werden.

```
heidi.init(3, 4, OST, 5)
```

heidi sitzt auf der Kachel mit der Koordinate 3 / 4 (4. Reihe, 5. Spalte), blickt nach Ost und hat 5 Körner im Maul. (Statt der Himmelsrichtung können auch die Zahlen 0 bis 3 verwendet werden, wobei gilt: 0 = Nord, 1 = Ost, 2 = SÜD, 3 = WEST. Die Zählung der Reihen und Spalten beginnt mit 0)



# Programmieren lernen mit dem Hamstermodell



## 2. Ausführen von Hamster-Befehlen

Zur gezielten Ansprache eines bestimmten Hamsters, setzen wir seinen Namen getrennt durch einen Punkt vor die Anweisung.

```
willi.vor( );           // Hamster willi geht einen Schritt vor

while (heidi.kornDa( ) )    // Hamster heidi nimmt alle Körner auf dem Feld
    heidi.nimm( );
```

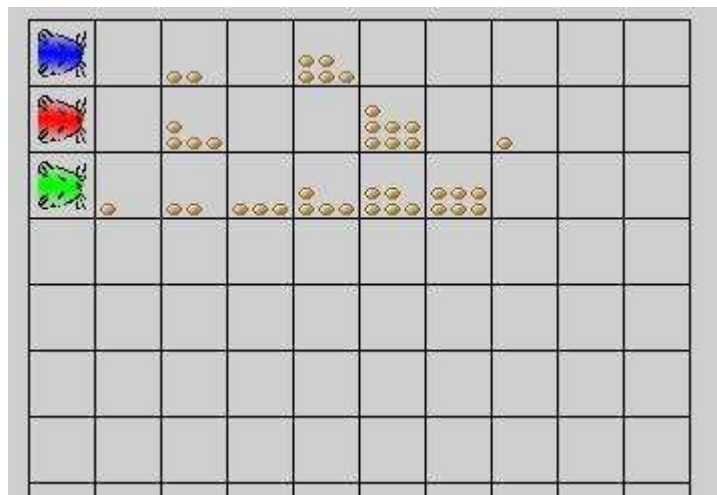
## 3. Neue Hamster-Befehle

Allen Hamster-Befehlen muss der Name des Hamsters vorangestellt werden, der den Befehl ausführen soll. In unseren Beispielen ist es Hamster *paul*

<b>paul.init(1,2,0,5)</b>	Initialisiert einen neuen Hamster mit den Attributen Reihe, Spalte, Blickrichtung, Anzahl Körner im Maul.
<b>paul.getReihe( )</b>	liefert einen int-Wert, der die Reihe angibt, auf der sich der H. befindet.
<b>paul.getSpalte( )</b>	liefert einen int-Wert, der die Spalte angibt, auf der sich der H. befindet.
<b>paul.getAnzahlKoerner( )</b>	liefert einen int-Wert, der angibt, wie viele Körner der H. im Maul hat.
<b>paul.getBlickrichtung( )</b>	liefert einen int-Wert, der die aktuelle Blickrichtung angibt.

## Aufgabe 15 ( 3 Punkte)

Erzeugen Sie den Standardhamster und geben Sie ihm den Namen *paul*, sowie zwei weitere Hamster namens *hans* und *tine* wie im Bild. Alle drei Hamster sollen bis zur gegenüberliegenden Wand laufen und dabei alle Körner fressen.







# Programmieren lernen mit dem Hamstermodell



## 4. Aufruf von Prozeduren

Wenn wir globale Variablen vermeiden wollen, müssen wir die erzeugten Hamster als Parameter an die Prozeduren übergeben, die der Hamster aufruft.

Bsp.: Wir haben den Standardhamster mit Namen *paul* sowie einen zur Laufzeit erzeugten Hamster mit Namen *heidi*. Es wird eine überladene Prozedur *vor()* (Kapitel 7, Teil A) geschrieben, die *heidi* *n* Schritte vor gehen lässt.

```
void main()
{
    Hamster paul=Hamster.getStandardHamster();
    Hamster heidi;
    heidi = new Hamster();
    heidi.init(5,2,3,0);
    vor(3,heidi);
}

void vor(int i, Hamster name)    //Übergabeparameter Anzahl Schritte und Datentyp Hamster
{
    for(int a=0;a<i;a++)
        name.vor();
}
```

### Aufgabe 16 ( 5 Punkte)

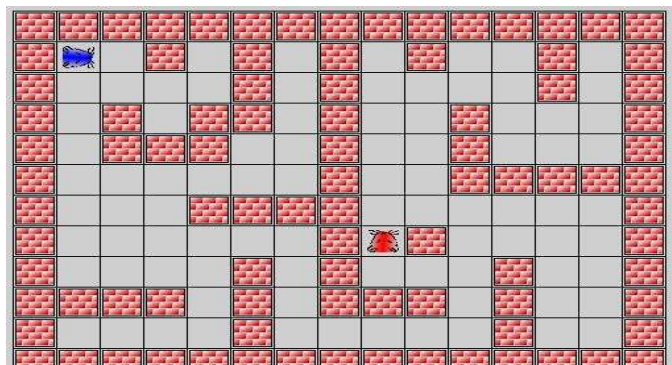
Ein neuer Hamster wird irgendwo in einem Hamster-Territorium ohne Mauern erzeugt. Seine Aufgabe ist es, den Standard-Hamster zu suchen und wenn er ihn gefunden hat, auf seinem Feld stehen zu bleiben. Der Standard-Hamster bewegt sich nicht von der Stelle.

Hinweis: Am einfachsten ist es, wenn der neue Hamster am Anfang nach Westen sieht und das Feld reihenweise nach oben absucht. Verwenden Sie dazu die neuen Befehle *getReihe()* und *getSpalte()*.

### Aufgabe 17 (7 Punkte)

Ein neuer Hamster hat sich irgendwo in einem Labyrinth versteckt. Der Standardhamster soll ihn suchen. Wenn er ihn gefunden hat, soll der neue Hamster dem Standardhamster bis zum Ausgangsfeld zurück folgen.

Der Standardhamster beginnt die Suche auf den Koordinaten 1,1 und orientiert sich an der rechten Mauer. Wenn er den neuen Hamster gefunden hat, macht er kehrt und läuft an der linken Mauer zurück bis zum Startfeld. Es ist darauf zu achten, dass der neue Hamster in der Richtung steht, in der er los laufen muss.







# Programmieren lernen mit dem Hamstermodell



## 5. Hamsterklassen

Unser Hamster hat einige Eigenschaften, wie Blickrichtung, Anzahl Koerner im Maul, Standort und einige Fähigkeiten, wie Körner sammeln, vor gehen, sich nach links drehen. Objekte, die durch Eigenschaften und Fähigkeiten gekennzeichnet sind, gehören zu einer **Klasse**. Die Eigenschaften einer Klasse nennt man **Attribute**, die Fähigkeiten nennt man **Methoden**. Eine Klasse definiert lediglich diese Attribute und Methoden. Die Objekte (hier Hamster), die zu einer Klasse gehören nennt man **Instanzen**.

Mit der Anweisung: `heidi = new Hamster( );` wird eine neue Instanz der Klasse Hamster erzeugt.

Schauen wir uns an, wie eine Klasse in der Programmiersprache Java angelegt wird:

```
public class Hamster
// Attribute
    private int reihe;           // r - Koordinate
    private int spalte;         // s - Koordinate
    private int blickrichtung;
    private int anzahlKoerner;

// Methoden
    public void vor( )
    {
        if (blickrichtung == Hamster.Nord)
            reihe = reihe - 1;
        if (blickrichtung == Hamster.SUED)
            reihe = reihe + 1;
        if (blickrichtung == Hamster.OST)
            spalte = spalte + 1;
        if (blickrichtung == Hamster.WEST)
            spalte = spalte - 1;
        // Hamster auf dem Bildschirm ein Feld vor bewegen
    }
}
```

Die Darstellung ist nicht vollständig. Wir erkennen aber das Prinzip:

Die Anweisung *public class Hamster* definiert eine Klasse, die die Bezeichnung Hamster hat. Zuerst werden alle Attribute definiert, die die Klasse besitzen soll, dann werden alle Methoden definiert, die die Klasse ausführen soll. Das Schlüsselwort *private* bedeutet, dass das Attribut nur von der Methode geändert werden kann, die dieses Attribut verwendet. *Public* bedeutet, dass die Methode von jedem benutzt werden kann, der Zugriff auf die Klasse hat.



# Programmieren lernen mit dem Hamstermodell



Wenn wir eine Methode aufrufen müssen wir den Namen des Hamsters, der die Methode ausführen soll voranstellen. Z.B.: paul.vor( ). Um dieses schon bei der Implementierung der Methoden kenntlich zu machen, kann man einen Platzhalter für das jeweilige Objekt verwenden, für das die Methode im Programm aufgerufen wird. Dieser Platzhalter ist das Schlüsselwort *this*. Für obige Methode vor( ) würde das dann so aussehen:

```
public class Hamster
{ ....
    public void vor( )
    {
        if(! this.vornFrei( ))
        {      //Programmabbruch }
        else {
            if (this.blickrichtung == Hamster.Nord)
                this.reihe = this.reihe - 1;
            if (this.blickrichtung == Hamster.SUED)
                this.reihe = this.reihe + 1;
            if (this.blickrichtung == Hamster.OST)
                this.spalte = this.spalte + 1;
            if (this.blickrichtung == Hamster.WEST)
                this.spalte = this.spalte - 1;
        }
    }
}
```

Zur Übung wollen wir noch mal die Implementierung von zwei weiteren Methoden zeigen.  
a) Die Methode *maulLeer* ist vom Typ boolean und sieht so aus:

```
public class Hamster
{.....
    public boolean maulLeer( )
    {
        return this.anzahlKoerner ==0;
    }
}
```



# Programmieren lernen mit dem Hamstermodell



b) die Methode *linksUm()* müsste so aussehen:

```
public class Hamster
{.....
    public void linksUm( )
    {
        if (this.blickrichtung==Hamster.NORD)
            this.blickrichtung==Hamster.WEST;
        if (this.blickrichtung==Hamster.WEST)
            this.blickrichtung==Hamster.SUED;
        if (this.blickrichtung==Hamster.SUED)
            this.blickrichtung==Hamster.OST;
        if (this.blickrichtung==Hamster.OST)
            this.blickrichtung==Hamster.NORD;
    }
}
```

## Aufgabe 18 ( 2 Punkte)

Schreiben Sie die Implementierung der Methode *getAnzahlKoerner*.  
Orientieren Sie sich an der Methode *maulLeer*.

## 6. Erweiterte Hamsterklassen

Wir können den Befehlsvorrat des Hamsters erweitern, indem wir Prozeduren schreiben. Unsere Hamsterobjekte (Instanzen) können diese neuen Befehle ausführen, wenn wir den Namen des Hamsters als Übergabeparameter an die Prozedur weitergeben.

Z.B. *kehrt(paul)*;

Eleganter wäre es aber, wenn wir neue Hamsterbefehle in der gleichen Weise verwenden könnten, wie die anderen auch, nämlich durch Voranstellen des Hamsternamens.

Z.B. *paul.kehrt()*;

Für diesen Zweck erweitern wir die Klasse Hamster. Mit einer erweiterten Hamster-Klasse können wir zusätzliche Attribute und Methoden für den Hamster definieren.

Folgendes Beispiel zeigt, wie eine erweiterte Hamsterklasse angelegt wird:



# Programmieren lernen mit dem Hamstermodell



```
class LaufHamster extends Hamster
{
    public void kehrt( )
    {
        this.linksUm( ); this.linksUm( );
    }

    public void vor( int s)
    {
        int schritte=0;
        while(schritte<s &&this.vornFrei())
        {
            this.vor();    i++;
        }
    }
}
```

Das Schlüsselwort *extends* zeigt an, dass es sich um eine erweiterte Hamsterklasse handelt. Die Klasse kennt die beiden zusätzlichen Methoden *kehrt( )* und die überladene Methode *vor(int)*. Diese Methoden können nun wie ein normaler Hamsterbefehl mit Voranstellen des Hamsternamens aufgerufen werden.

In der Funktion *main( )* soll der Hamster *willi* vom Typ *LaufHamster* erzeugt werden. Der Aufruf in der Funktion *main( )* sieht so aus:

```
void main( )
{
    LaufHamster willi = new LaufHamster( ); // Instanz erzeugen
    willi.vor(4);                          // Aufruf der überladenen Methode vor
}
```

Erweiterte Klassen sowie vollständig neue Klassen werden in der Regel separat als eigene Datei abgespeichert. Dazu wählt man im Editor das Menü Datei und Neu und wählt Klasse als neuen Dateityp. Die Klassen werden dann kompiliert und im selben Verzeichnis gespeichert, wie das Hauptprogramm. Der Vorteil besteht darin, dass diese Klasse dann in allen Programmen verwendet werden kann, ohne sie jedes Mal neu schreiben zu müssen.

## Aufgabe 19 (4 Punkte)

Verändern Sie Aufgabe 15 wie folgt: Der gesuchte Hamster ist eine Instanz der erweiterten Hamsterklasse *Sklavenhamster*. Diese Klasse erhält als zusätzliches Attribut den Integerwert *blickrichtung* sowie die Methode *setBlickrichtung*. Damit soll es möglich sein, dass sich der gesuchte Hamster in die richtige Laufrichtung dreht, wenn er gefunden wurde. Die richtige Initialisierung der Laufrichtung ist damit nicht mehr nötig.



# Programmieren lernen mit dem Hamstermodell



## Aufgabe 20 (6 Punkte)

Definieren Sie eine Klasse SuperHamster, die viele der häufig benötigten Methoden zur Verfügung stellt. Diese Klasse können Sie in den folgenden Aufgaben auch als Basisklasse für weitere erweiterten Hamsterklassen verwenden. Dieses Prinzip nennt man **Vererbung**. Erweiterte(abgeleitete) Klassen erben alle Attribute und Methoden der vorhergehenden Basisklassen. Dies gilt auch, wenn von einer erweiterten Klasse weitere erweiterte Klassen gebildet werden. Folgende Methoden sollen dem Superhamster zur Verfügung stehen:

kehrt( )	// 2 Linksdrehungen
rechtsUm( )	// 3 Linksdrehungen
int vor(int s)	// Hamster läuft s Schritte vor, wenn vorne frei und liefert die Anzahl der gelaufenen Schritte
int laufeZurWand( )	// Hamster läuft bis zur Wand und liefert die Anzahl der gelaufenen Schritte
gib(int anzahl)	// Hamster legt anzahl Körner ab, maximal jedoch so viele, wie er im Maul hat
nimm(int anzahl)	// Hamster frisst anzahl Körner, maximal jedoch so viele, wie auf der Kachel liegen
int gibAlle( )	// Hamster legt alle Körner ab, und liefert die Anzahl der abgelegten Körner
int nimmAlle( )	// Hamster frisst alle Körner auf der Kachel und liefert die Anzahl der gefressenen Körner
boolean linksFrei( )	// Hamster testet, ob die Kachel links von ihm frei ist
boolean rechtsFrei( )	// Hamster testet, ob die Kachel rechts von ihm frei ist.
boolean hintenFrei( )	// Hamster testet, ob die Kachel hinter ihm frei ist.

Testen Sie den Superhamster in dem Labyrinth zu Aufgabe 11.

## 7. Konstruktoren

Bei der Erzeugung der Instanz *willi* im Beispiel zu Kapitel 6 wurde vergessen, das neue Objekt mit der Anweisung *init* zu initialisieren. Falls dies passiert, werden alle Attribute automatisch mit dem Wert 0 initialisiert.

Es gibt eine besondere Methode mit der Objekte initialisiert werden können, die so genannten **Konstruktoren**. Sinn dieser Konstruktoren ist es, Objekte sofort bei der Erzeugung zu initialisieren.

### Definition von Konstruktoren

```
class LaufHamster extends Hamster {
    int schritte;
// Konstruktor
    LaufHamster(int r, int s, int b,int k, int schritte)
    {
        this.init(r,s,b,k,schritte);
    }
}
```



# Programmieren lernen mit dem Hamstermodell



Der Konstruktor ist also eine Methode ohne Angabe eines Rückgabedatentyps. Die Methode trägt den gleichen Namen, wie die Klasse, zu der er gehört.

Aufruf des Konstruktors in main

```
void main() {  
    Laufhamster willi = new Laufhamster(0,0,2,5,3);
```

## 8. Ein- und Ausgabe (Dialoge)

Im objektorientierten Hamstermodell hat der Benutzer nun auch die Möglichkeit in Dialog mit dem Hamster zu treten. D.h. das Programm kann Meldungen am Bildschirm anzeigen und der Benutzer kann Informationen über die Tastatur eingeben. Dadurch ergibt sich die Möglichkeit, den Programmablauf während der Laufzeit zu steuern. Für die Ausgabe von Informationen am Bildschirm gibt es die Methode **schreib**, bei der Eingabe über die Tastatur wird unterschieden, ob eine Zahl (**liesZahl**) oder eine Zeichenkette (**liesZeichenkette**) eingegeben wird. Diese Eingabe wird einer Variablen zugewiesen, die dann den eingegebenen Wert repräsentiert. Beispiele:

```
paul.schreib("Vielen Dank für das Spiel");  
zahl=paul.liesZahl("Gib eine Zahl ein: ");  
antwort=paul.liesZeichenkette("Noch einmal (ja/nein)");
```

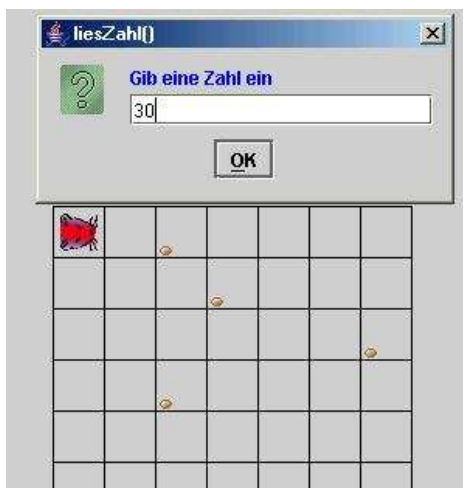
### Aufgabe 21 LottoHamster (7 Punkte)

Der StandardHamster möchte Ihnen helfen, Ihren Lottoschein auszufüllen. Er steht in einem Feld ohne Mauern mit sieben Reihen und sieben Spalten. Dieses Feld stellt den Lottoschein dar. Die erste Reihe markiert die Zahlen 1 bis 7, die zweite die Zahlen 8 bis 14 usw.

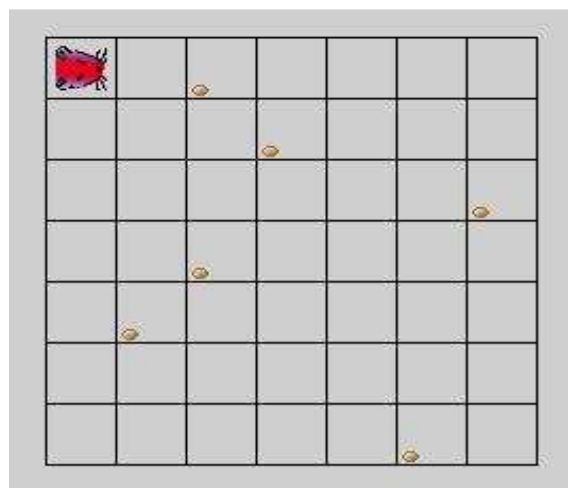
Der StandardHamster fragt nach einer Zahl und schickt dann einen Vertretungshamster los, um auf diesem Feld ein Korn abzulegen. Am Ende soll auf sechs Feldern ein Korn liegen

Definieren Sie eine erweiterte Klasse LottoHamster. Initialisieren Sie eine Instanz dieser Klasse mit 6 Körnern im Maul über einen Konstruktor und versehen sie diese Klasse mit einer Methode *tippeZahl*, die als Übergabeparameter die getippte Zahl hat und den Hamster ein Korn auf das entsprechende Feld ablegen lässt. Nach dem Ablegen läuft der LottoHamster mit Hilfe der Methode *zumStart* wieder auf seinen Ausgangspunkt (0,0) und wartet auf die nächste Eingabe.

so wird getippt



so sieht der Lottozettel aus (3, 10, 21, 24, 30, 48)





# Programmieren lernen mit dem Hamstermodell



## Aufgabe 22 Zufallshamster (10 Punkte)

In einem beliebig großen Territorium ohne Mauern leben zunächst zwei Hamster. Diese laufen zufallsgesteuert im Territorium herum. Jedes Mal, wenn sie auf dieselbe Kachel gelangen, produzieren sie Nachwuchs.

Lösungshinweis: Versuchen Sie zunächst einen einzigen Hamster der Klasse ZufallsHamster zu erzeugen und diesen zufallsgesteuert umher laufen zu lassen.  
Das Programm ist beendet, wenn es mehr als 8 Hamster gibt.

Hier ein Auszug aus der Klassendefinition für einen ZufallsHamster:

```
class ZHamster extends Hamster
{
    ZHamster ()
    {
        int r = this.erzeugeZufallsZahl(Territorium.getAnzahlReihen()-1);
        int s = this.erzeugeZufallsZahl(Territorium.getAnzahlSpalten()-1);
        int b = this.erzeugeZufallsZahl(3);
        this.init(r,s,b,0);
    }
    public int erzeugeZufallsZahl(int max)
    {
        return (int)(Math.random()*(max+1));
    }

    public void lauf()
    {
        int zahl=this.erzeugeZufallsZahl(3);
        if(zahl==0)
        {
            if (this.vornFrei()) this.vor();
        }
        if(zahl ==1)
        {
            this.linksUm();
            if (this.vornFrei()) this.vor();
        }
        if(zahl ==2)
        {
            this.kehrt();
            if (this.vornFrei()) this.vor();
        }
        if(zahl==3)
        {
            this.rechtsUm();
            if(this.vornFrei()) this.vor();
        }
    }

    public void kehrt()
    { this.linksUm(); this.linksUm(); }

    public void rechtsUm()
    { this.kehrt(); this.linksUm(); }
}
```

Versuchen Sie mal nachzuvollziehen, welche Zufallszahlen erzeugt werden. Die Funktion Math.random() erzeugt Zufallszahlen zwischen 0 und 1 (z.B. 0,3689...)



# *Programmieren lernen mit dem Hamstermodell*



## Verzeichnis aller Aufgaben

Aufgabe	Teil	Sollpunkte	Istpunkte
1	A	1	
2	A	1	
3	A	1	
4	A	3	
5	A	2	
6	A	4	
7	A	4	
8	A	5	
9	A	5	
10	A	8	
11	A	3	
12	A	4	
13	B	7	
14	B	10	
15	B	3	
16	B	5	
17	B	7	
18	B	2	
19	B	4	
20	B	6	
21	B	7	
22	B	10	





# **Programmieren mit Java und dem Java-Editor**

Skript zum Unterricht

## **Inhalt**

- 0. Wie funktioniert Java**
- 1. Das erste Programm**
- 2. Datentypen und Variablen**
- 3. Bildschirmausgabe**
- 4. Übungen**
- 5. Eingabe über die Tastatur**
- 6. Grundstrukturen**
- 7. Übungen**
- 8. Arrays**
- 9. Übungen**
- 10. Prozeduren und Methoden**
- 11. Übungen**
- 12. Packages**
- 13. Dateiverarbeitung**
- 14. Fehlerbehandlung**
- 15. Graphische Benutzeroberflächen (GUI)**
- 16. Ergänzungen**
- 17. Java Script und Java Applets**



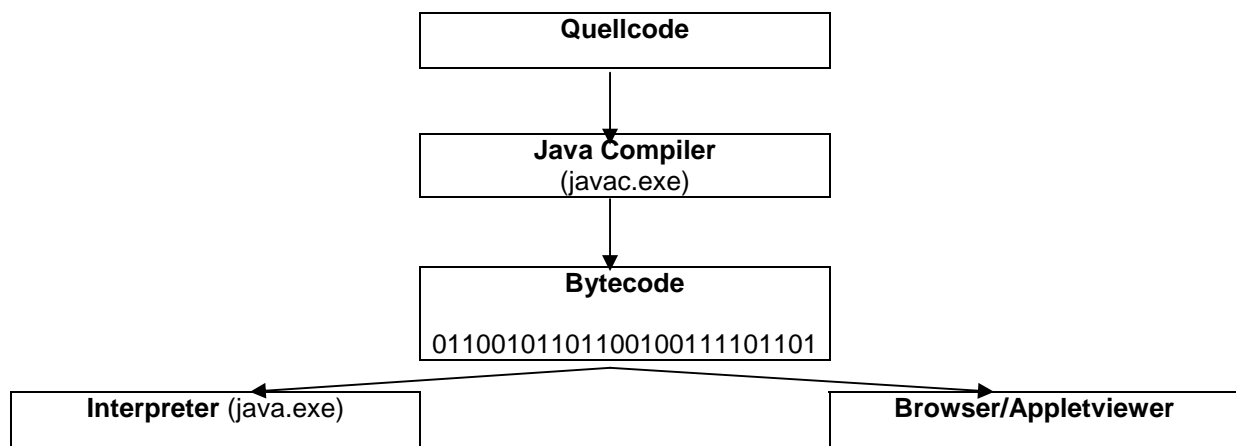
## 0. Wie funktioniert Java

Java ist eine objektorientierte Sprache, die konzipiert wurde, um zwei wichtige Eigenschaften zu erfüllen: Sie sollte **internetfähig** sein und sie sollte **plattformunabhängig** sein.

In der Sprache Java kann man Programme erstellen, die im Internet auf einem Browser laufen. Dieses können kleine Applets, wie z.B. eine Animation sein, aber auch umfangreiche Anwendungen, wie ein Shop bzw. eine Datenbankverwaltung.

Außerdem sollen Java-Programme auf allen Betriebssystemen und allen Computertypen laufen. Um dieses zu erreichen sind einige Vorbereitungen zu treffen, wenn ich mit Java arbeiten will.

Java Programme(Quellcode) werden von einem Java-Compiler in den sog. Bytecode übersetzt. Dieser Bytecode läuft auf jedem Rechner, der Java unterstützt. Der Bytecode wird anschließend von einem Interpreter oder einem Internetbrowser ausgeführt.



Um mit Java arbeiten zu können, benötigt man die Java-Entwicklungsumgebung, die kostenlos im Internet zu bekommen ist. Diese nennt man JDK (java development kit) und ist bei <http://java.sun.com/javase/> in verschiedenen Versionen herunterzuladen. Die aktuelle Version ist **jdk-6u21-windows-i586-p.exe** (ca. 72 MB). Dieses Programm installiert Java auf deinem Rechner. Je nachdem, wo Java installiert wurde gibt es nun einen Ordner mit der Bezeichnung **\jdk**. Im Unterordner **\bin** befindet sich der Compiler (javac.exe) und der Interpreter (java.exe). Damit Java Programme aus jeder Situation ausgeführt werden können, muss in der Systemsteuerung (System) ein Pfad auf diese beiden Dateien eingetragen werden. Dazu kann die bereits existierende Pfadvariable (path) z.B. um den folgenden Eintrag ergänzt werden: **...;c:\programme\jdk\bin**. Unter Windows 7 findet man diese Variable unter **Start → Systemsteuerung → System und Sicherheit → System → erweiterte Systemeinstellungen → Erweitert → Umgebungsvariablen**.

Zunächst arbeiten wir mit Konsolenanwendungen, die im DOS-Fenster laufen. Dazu schreiben wir ein Java-Programm mit einem beliebigen Editor und speichern es mit der Endung .java ab (z.B. prog1.java). Anschließend rufen wir auf der Kommandozeile den Compiler auf und lassen das Programm in Bytecode übersetzen (z.B. c:\daten\java\ javac prog1.java). Wenn dieses funktioniert, wird ein neues Programm mit der Endung .class erzeugt (prog1.class). Dieses können wir nun mit dem Interpreter ausführen (z.B. c:\daten\java\java prog1). Die Dateiendung muss hierbei nicht mit angegeben werden.

Komfortabler ist es natürlich einen Java-Editor zu verwenden, der uns die lästige Arbeit über die Kommandozeile abnimmt. Ich empfehle hier den Java-Editor von Gerhard Röhner.

<http://www.bildung.hessen.de/abereich/inform/skii/material/java/editor.htm>

Mit Java kann man verschieden Arten von Programmen erstellen. Dazu gehören graphische Benutzeroberflächen (sog. **GUIs**) oder Web-Applikationen (**Applets oder Servlets**). Der Java-Editor bietet die Möglichkeit, einfach GUIs mit graphischen Komponenten (awt oder swing) zu erstellen. Der Quellcode solcher Anwendungen ist allerdings wesentlich komplexer. Deswegen eignen sich solche Anwendungen nicht unbedingt für das Erlernen von Java-Grundlagen.



## 1. Das erste Programm

Wie immer, beginnen wir auch beim Erlernen von Java mit dem „Hallo Welt“ Programm. Hier ist es:

```
public class HalloWelt
{
    public static void main(String argv[ ]) {
        System.out.println("Hallo Welt !!!");
    }
}
```

In der ersten Zeile steht der Klassenname. Das Programm muss unter der Bezeichnung `Klassenname.java` abgespeichert werden. Es folgt dann die obligatorische Prozedur `main`. Auch hier ist immer auf die Blockklammern für jede Prozedur zu achten. Die Anweisung `System.out.println` ist die Anweisung für die Ausgabe auf der Konsole (Bildschirm). Wir werden diese häufig verwenden. Das Grundgerüst für jedes Programm sieht also wie folgt aus:

```
public class Klassenname
{
    public static void main(String argv[ ]) {

    }
}
```

Kommentare werden zeilenweise durch `//` eingeleitet oder über mehrere Zeilen mit `/*` begonnen und mit `*/` beendet. Verwenden Sie auf jeden Fall Kommentare am Beginn eines Programms!

### Zusammenfassung:

Das Programm kann mit einem beliebigen Editor geschrieben werden.

**HalloWelt.java** So lautet die Bezeichnung des java-Quellcodes. Diese Datei kann mit jedem beliebigen Editor erzeugt werden. Diese Datei wird durch den Compiler auf Syntax-Fehler überprüft. Bei erfolgreicher Übersetzung wird folgende Datei erzeugt:

**HallWelt.class** Dies ist der Java-Bytecode, der mit der Anweisung `java HalloWelt` ausgeführt werden kann.

Ein Java-Programm besteht immer mindestens aus zwei Teilen:

Klasse	Jedes Java-Programm wird in eine Klasse gepackt. In der ersten Zeile steht <b>public class Klassenname</b> . Darauf folgt ein geschweiftes Klammerpaar zwischen denen die eigentlichen Prozeduren stehen.
main	Dieser Programmteil wird zuerst ausgeführt und stellt das Hauptprogramm dar. Es beginnt mit der Zeile <b>public static void main(String [ ] args)</b> . Darauf folgt wieder ein paar geschweiften Klammern, zwischen denen die einzelnen Programmzeilen stehen.

Alle Java-Anweisungen stehen in irgendeiner Bibliothek. Diese sind Programmteile der Java-Laufzeitumgebung in denen Anweisungen oder Gruppen von Anweisungen programmiert sind. Die Standardbibliothek ist immer präsent. Wenn Befehle, die nicht zur Standardbibliothek gehören verwendet werden, muss die dazu gehörige Bibliothek in der allerersten Zeile importiert werden. Dies geschieht mit einer Anweisung der Art **import java.io.\***; Welche Bibliothek benötigt wird, erfährt man, wenn man die Hilfe zu der verwendeten Anweisung nachliest (siehe Kapitel 4)



## 2. Datentypen und Variablen

Bei der Deklaration einer Variablen muss ein bestimmter Datentyp angegeben werden. Der Datentyp ist abhängig von den Werten, die die Variable annehmen soll. Außerdem entscheidet der Datentyp darüber, wie viel Platz die Variable im Speicher belegt.

Die wichtigsten Datentypen sind:

**Ganzzahlen:** char  
short  
int  
long

**Gleitpunktzahlen:** float  
double

**Zeichen:** char einzelne Buchstaben oder sonst. Zeichen  
**Zeichenketten:** string Wörter, Texte  
**Boolesche** boolean true oder false

Typ	Größe	Bereich	Inhalt
boolean	1 Bit	true oder false	
char	16 Bit	0000-FFFF (hex)	Unicode Zeichen
byte	8 Bit	-128 bis +127	Zähler, sehr kleine Zahlen
short	16 Bit	-32.768 bis 32.767	Zähler, kleine Zahlen mit Vorzeichen
int	32 Bit	-2.147.483.648 bis 2.147.483.647	Große ganze Zahlen mit Vorzeichen
long	64 Bit	$-2^{63}$ bis $2^{63}-1$	Sehr große Zahlen
float	32 Bit	$1,18 \cdot 10^{-38}$ bis $3,40 \cdot 10^{38}$	7-stellige Genauigkeit
double	64 Bit	$2,23 \cdot 10^{-308}$ bis $1,79 \cdot 10^{308}$	15-stellige Genauigkeit

Eine Variable besteht immer aus einem **Namen** (Bezeichner), einem **Wert** und einem Platz im **Arbeitsspeicher** des Computers. Diesem Speicherplatz können nacheinander verschiedene Werte zugewiesen werden.

Eine Variable ist nur innerhalb der Prozedur ‚sichtbar‘, in der sie deklariert wurde und in allen untergeordneten Blöcken. Nicht aber in anderen Prozeduren (oder Methoden) auf der gleichen oder höheren Ebene. Man spricht von einer **lokalen** Variablen. Im Gegensatz dazu werden **globale** Variablen außerhalb einer Prozedur deklariert und sind somit innerhalb der ganzen Klasse sichtbar.

Im Gegensatz zu einer Variablen soll einer **Konstanten** nur einmal ein Wert zugewiesen werden, der dann nicht mehr verändert werden kann. Dies macht z.B. Sinn, wenn man den Wert für die Kreiszahl Pi mit 3.1415 als Konstante ablegen möchte. Konstante werden im Prinzip wie Variablen behandelt, bekommen aber das Schlüsselwort **final** vorangestellt. Der Name einer Konstanten sollte in Großbuchstaben geschrieben werden.

Beispiel: final float PI = 3.1415;

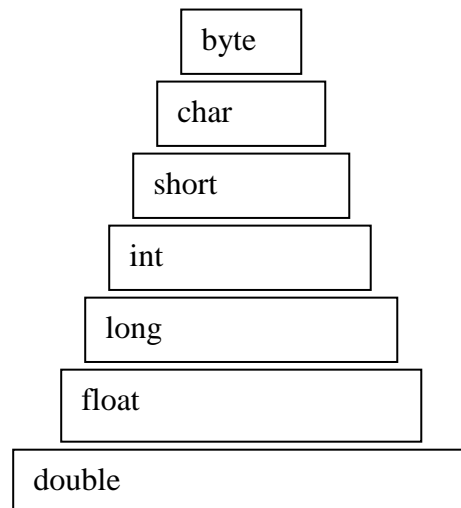


## 2.1 Verwendung unterschiedlicher Datentypen

Bei einer Zuweisung (z.B.  $x = a + 2$ ) wird der Wert des Ausdrucks rechts vom Gleichheitszeichen der links stehenden Variablen zugewiesen. Diese Zuweisung erfolgt allerdings nur dann korrekt, wenn die Datentypen auf beiden Seiten typkompatibel sind.

Wenn wir uns die Datentypen double, float, long, int, short, char und byte als Behälter vorstellen, so haben diese Behälter eine unterschiedliche Größe.

Es gilt, dass ein kleinerer Behälter in einen größeren gepackt werden kann, aber nicht umgekehrt. Die Größe der Behälter wird wie folgt veranschaulicht:



Alle Zuweisungen eines kleineren in einen größeren Datentyp werden akzeptiert und bewirken automatisch eine Typumwandlung des Ergebnisses in den größeren Datentyp. Außerdem ist bei Berechnungen auf den Wertebereich des Datentyps zu achten (siehe vorherige Seite). Zuweisungen von einem größeren in einen kleineren Datentyp werden entweder vom Compiler nicht akzeptiert oder es werden keine korrekten Ergebnisse erzeugt. (z.B.  $\text{int } x = 1 / 2$  bewirkt, dass der Wert von  $x = 0$  ist, da das Ergebnis  $1 / 2 = 0,5$  ist, dieses aber kein Integerwert ist und somit der Nachkommaanteil abgeschnitten wird).

## 2.3 Explizite Typumwandlung durch Casts

Durch sog. Type casts kann man einen Wert in einen anderen Datentyp umwandeln. Diesen Vorgang nennt man *Casten*. Das Casten erfolgt durch den **Castoperator**, der dem zu castenden Wert den neuen Datentyp in Klammern voranstellt.

Beispiel: `int n; short k; double d = 3.1415;`

`n = (int) d;` wandelt double nach int um (n = 3)  
`k = (short) 6.4567` wandelt float nach short um (k = 6)

## 2.4 Namenskonventionen

Sie sollten sich bei der Schreibweise und Namensvergabe von Klassen, Variablen und Methoden an folgende Namenskonventionen halten, die sich für Java durchgesetzt haben.

<b>Variablen und Methoden</b>	kleine Anfangsbuchstaben. Bei zusammengesetzten Wörtern wird der erste Buchstabe des folgenden Wortes groß geschrieben
<b>Klassen</b>	Großer Anfangsbuchstabe, dann klein fortsetzen, keine reservierten Wörter verwenden, keine Leerzeichen, Datumsangaben oder Sonderzeichen
<b>Packages</b>	immer klein
<b>Konstante</b>	vollständig in Großbuchstaben



## 3. Bildschirmausgabe

### 3.1 Bildschirmausgabe

Ein etwas umfangreicheres Beispiel:

```
1)  /* Prog3  Variablentest und Typumwandlung
2)  */
3)  import java.io.*;                //Package für input-output Klassen

4)  public class Prog3
5)  {
6)  public static void main(String argv[ ])
7)  {
8)      boolean x = true;
9)      int a = 7;
10)     int b = 2;
11)     double c = 1.4142;
12)     char d = 'a';
13)
14)     System.out.println(x);
15)     System.out.println(a);
16)     System.out.println(b);
17)     System.out.println(c);
18)     System.out.println(d);
19)     System.out.println(s);
20)     System.out.println("7 : 2 = "+(a/b));
21)     System.out.println("Das war falsch! Richtig ist: "+(double)a/b);
        //Typumwandlung
22) }
23) }
```

**Aufgabe:** Welche Ausgaben werden mit System.out.println erzeugt?

An diesem Beispiel soll Folgendes gezeigt werden:

- Der Einsatz von Variablen mit unterschiedlichem Datentyp
- Die Konsolenausgabe mit System.out und die Verwendung des Verknüpfungsoperators +
- Die Möglichkeit der Typumwandlung bei unterschiedlichem Zahlenformat
- Die Einbindung von Packages mit import



## **4. Übungen**

- 1) Schreiben Sie ein Programm, welches den Durchschnittsverbrauch eines Pkws auf 100 km ausgibt, wenn die gefahrenen Kilometer und die verbrauchte Kraftstoffmenge bekannt ist.
- 2) Schreiben Sie ein Programm, welches die Steigung einer Gerade berechnet. Dabei sind die x- und y-Koordinaten von zwei Punkten bekannt.
- 3) Schreiben Sie ein Programm, welches eine Kalkulation nach folgendem Schema durchführt:

Einkaufspreis:  
+ 40 % Handlungskostenzuschlag  
= Selbstkostenpreis  
+ 5 % Gewinnzuschlag  
= Nettoverkaufspreis  
+ 19 % Mehrwertsteuer  
= Bruttoverkaufspreis

- 4) Schreiben Sie ein Programm, welches den Body-Maß-Index (BMI) berechnet.



## 5. Eingabe über die Tastatur

Eingaben über die Tastatur sind in Java etwas umständlich zu realisieren. Das liegt daran, dass bei einer Tastatureingabe überprüft werden muss, ob die Eingabe fehlerhaft war. Außerdem erfolgt die Eingabe über einen sog. Puffer, dessen Zeichen je nach Datentyp unterschiedlich behandelt werden. Wir zeigen hier die Eingabe für eine Zeichenkette (String) eine Gleitkommazahl (double) und für eine Ganzzahl (Integer) **ohne Fehlerbehandlung**. Um uns dieses umständliche Procedere zu ersparen, werden wir uns danach eine eigene Eingabe- Ausgaberoutine schreiben, die wir als Bibliothek in unsere Programme einbinden. Wir brauchen dann nur noch die entsprechende Prozedur für unsere gewünschte Eingabe aufzurufen.

Im Folgenden wird der Quellcode für die Eingabe einer Zeichenkette (String) eines Floatwertes und eines Integerwertes über die Tastatur dargestellt. (Die für die Eingabe relevanten Anweisungen sind fett gedruckt).

// Eingabe einer Zeichenkette über die Tastatur

```
*/
import java.io.*;           //Package für input-output Klassen

public class s_in
{
    public static void main(String argv[]) throws IOException
    {
        String s="";
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Bitte eine Textzeile eingeben: ");
        s=in.readLine();

        System.out.println("Ausgabe: "+s);
    }
}
```

Für die Eingabe einer Zahl verwenden wir im Prinzip dieselbe Routine. Allerdings müssen wir den Eingabestring in eine Variable vom Typ float oder int umwandeln.

// Eingabe einer Gleitkommazahl(Double) über die Tastatur

```
import java.io.*;           //Package für input-output Klassen

public class double_in
{
    public static void main(String argv[]) throws IOException
    {
        String s="";
        double zahl=-1.0;
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Bitte eine Gleitkommazahl eingeben: ");
        s=in.readLine();
        zahl=Double.parseDouble(s); //oder Umwandeln in Float mit zahl=Float.parseFloat(s);
        System.out.println("Ausgabe: "+zahl);
    }
}
```

---





// Eingabe einer Ganzzahl (Integer) über die Tastatur

```
import java.io.*;           //Package für input-output Klassen
public class int_in
{
    public static void main(String argv[]) throws IOException
    {
        String s="";
        int zahl;
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Bitte eine Gleitkommazahl eingeben: ");
        s=in.readLine();
        zahl=Integer.parseInt(s);    // hier wird der Eingabestring umgewandelt
        System.out.println("Ausgabe: "+zahl);
        System.out.println(zahl+3);
    }
}
```

## Das Programm Console\_IO

Für Tastatureingabe verwenden wir ab jetzt das Programm Console\_IO.java. Dieses enthält acht Prozeduren, mit denen wir Daten einlesen können. Wir brauchen lediglich noch die entsprechende Prozedur aufzurufen und die Begriffe **throws IOException** hinter den Kopf der main-Methode zu schreiben. Zusätzlich muss die Bibliothek **java.io.\*** eingebunden werden.

Wichtig: Die Datei Console\_IO muss sich im selben Verzeichnis befinden, wie die Klasse, von der sie verwendet wird. Außerdem muss sie in kompilierter (\*.class) Form vorliegen.

Es gibt folgende Prozeduren:

<b>I_String</b>	Eingabe Zeichenkette
<b>I_int</b>	Eingabe Integerwert
<b>I_float</b>	Eingabe Floatwert
<b>I_double</b>	Eingabe Doublewert
<b>IO_String</b>	Ausgabertext, Eingabe Zeichenkette (Übergabewert String)
<b>IO_int</b>	Ausgabertext, Eingabe Integerwert (Übergabewert String)
<b>IO_float</b>	Ausgabertext, Eingabe Floatwert (Übergabewert String)
<b>IO_double</b>	Ausgabertext, Eingabe Doublewert (Übergabewert String)

Beispiel für die Eingabe eines Integerwertes mit vorhergehendem Eingabeaufforderungstext:

```
import java.io.*;
public class testprogramm
{
    public static void main(String argv[]) throws IOException
    {
        int x;
        x=Console_IO.IO_int("Bitte geben Sie eine ganze Zahl ein: ");
        System.out.println("Die eingegebene Zahl lautet: "+x);
    }
}
```

Der Aufruf der Prozedur erfolgt durch Angabe des Programmnamens und der Prozedurbezeichnung getrennt durch einen Punkt. Alle Prozeduren, die mit IO beginnen, haben als Übergabeparameter eine Zeichenkette, die den Aufforderungstext darstellt. Damit ersparen wir uns eine System.out-Anweisung im Programm.

Wenn sich das Programm Console\_IO im aktuellen Verzeichnis befindet, muss es nicht mit einer import-Anweisung am Beginn des Programms eingebunden werden.



## 6. Grundstrukturen

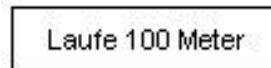
Jedes Programm kommt mit wenigen logischen Strukturen aus, mit denen man fast alle Probleme lösen kann. Diese Strukturen können graphisch durch Struktogramme (siehe Hamsterskript) dargestellt werden. Im Folgenden werden die Grundstrukturen an einem Beispiel noch mal vorgestellt.

Beispiel:: Tom ist Läufer. Manchmal läuft er auf der Bahn im Stadion, manchmal im Wald. Er steht unterschiedlichen Situation gegenüber.

### 1. Einfache Anweisung

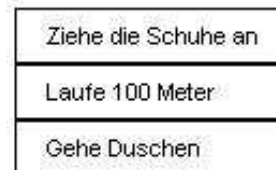
Tom will auf der Bahn 100 m laufen.

#### Struktogrammdarstellung



### 2. Folge von Anweisungen

Tom zieht die Schuhe an, läuft auf der Bahn 100 m und geht anschließend Duschen.



### 3. Verzweigung

Tom läuft im Wald. Als er an eine Weggabelung kommt entscheidet er sich für den linken Weg, wenn es trocken ist, andernfalls nimmt er den rechten Weg.



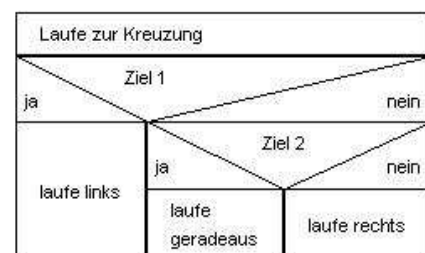
### 4. Mehrfachauswahl

Tom läuft im Wald. An einer Kreuzung hat er drei Möglichkeiten:  
 Wenn er nach Ziel 1 will, muss er den linken Weg nehmen.  
 Wenn er nach Ziel 2 will muss er geradeaus laufen.  
 In allen anderen Fällen muss er nach rechts laufen.



#### 4a. Alternativ: Verschachtelte Verzweigung

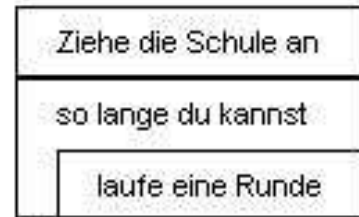
Eine Mehrfachauswahl kann auch mit Hilfe mehrerer Verzweigungen gelöst werden, die in einander verschachtelt werden.





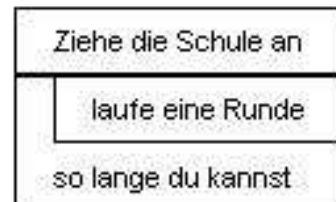
## 5a. Wiederholung, kopfgesteuerte Schleife

Tom läuft auf der Bahn. Nachdem er die Schuhe angezogen hat, überprüft er seine Kondition und läuft los, wenn er sich gut fühlt. Zu Beginn jeder Runde überlegt er wieder neu, ob seine Kondition noch für eine weitere Runde reicht.  
In diesem Fall kann Tom vorher entscheiden, ob er überhaupt eine Runde laufen will.



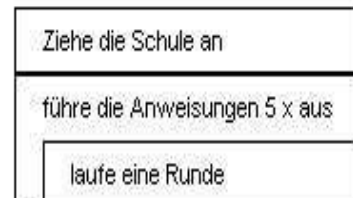
## 5b. Wiederholung, fußgesteuerte Schleife

Tom läuft auf der Bahn. Nachdem er die Schuhe angezogen hat, läuft er eine Runde und überprüft dann am Ende jeder Runde, ob seine Kondition noch für eine weitere Runde reicht.  
In diesem Fall läuft Tom auf jeden Fall mindestens eine Runde



## 5c. Wiederholung, zählergesteuerte Schleife

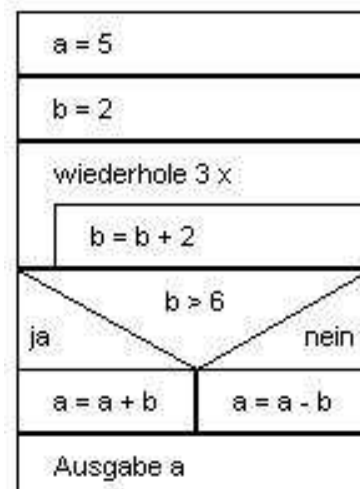
Tom läuft auf der Bahn. Er läuft genau 5 Runden.  
Die Anzahl der Schleifendurchläufe wird durch einen Zähler bestimmt, der zu Beginn der Schleife festgesetzt wird.



## Zusammenfassung

Alle Grundstrukturen können miteinander kombiniert werden. Aus der Struktogrammdarstellung ergibt sich der logische Ablauf eines Programms.

In nebenstehendem Beispiel soll am Ende der Wert der Variablen a ausgegeben werden.  
Nach Ausführung aller Anweisungen wird am Ende für a der Wert 13 ausgegeben.





## 6.1 Beispiele für logische Grundstrukturen

Anhand kurzer Beispiele werden die Grundstrukturen in Java-Syntax vorgestellt.

### a) Lineare Struktur

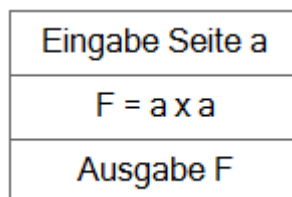
Beispiel:

Die Seitenlänge eines Quadrates wird eingegeben. Dann wird die Fläche berechnet und das Ergebnis der Berechnung am Bildschirm angezeigt.

Bei diesem Beispiel handelt es sich um die klassische Programmstruktur:

**Eingabe – Verarbeitung – Ausgabe.** Alle Anweisung werden hintereinander ausgeführt. Man spricht hier von einer linearen Struktur.

**Struktogramm:**



**Quellcode:**

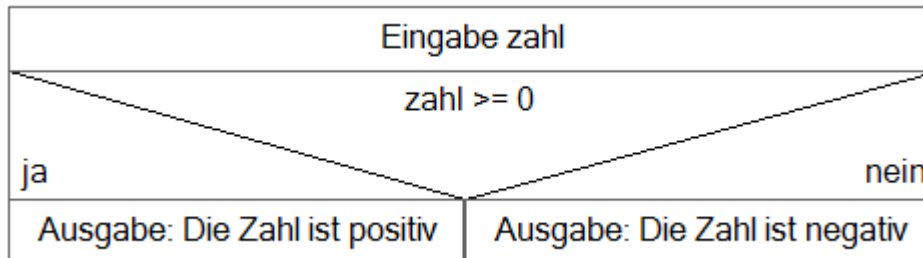
```
public class LinearTest {  
    public static void main(String argv [ ]) throws IOException  
    {  
        double a, F;  
        a = Console_IO.IO_double("Seitenlänge eingeben: ");  
        F = a * a;  
        System.out.println("Die Fläche beträgt: "+F);  
    }  
}
```



## b) Verzweigung (if..else)

### Beispiel:

Es wird eine beliebige Zahl eingegeben. Das Programm prüft, ob diese Zahl positiv ist. Je nach dem Ergebnis der Prüfung wird eine Antwort ausgegeben.



Durch Verzweigungen mit **if** kann der Programmverlauf in zwei unterschiedliche Richtungen gelenkt werden. Nach der Anweisung **if** muss eine Bedingung stehen, die entweder wahr (ja) oder falsch (nein) sein kann. Wenn es einen nein-Fall gibt, wird dieser mit dem Wort **else** eingeleitet. Der else-Zweig ist aber nicht zwingend erforderlich in einem if-Konstrukt.

```
public class IfTest {  
    public static void main(String argv[ ]) throws IOException  
    {  
        int zahl;  
        zahl = Console_IO.IO_int("Bitte eine Zahl eingeben: ");  
        if (zahl >= 0 )  
            System.out.println("Die Zahl ist positiv");  
        else  
            System.out.println("Die Zahl ist negativ");  
    }  
}
```



## c) Mehrfachauswahl (switch..case)

### Beispiel:

Es werden die Notenziffern einer Klassenarbeit eingegeben. Das Programm gibt je nach Notenwert die Bewertung als Text aus. Wenn eine ungültige Notenziffern eingegeben wird, erscheint eine Fehlermeldung.

Eingabe note						
1	2	3	4	5	6	note sonst
sehr gut	gut	befriedigend	ausreichend	mangelhaft	ungenügend	Fehlermeldung

Im Gegensatz zur einfachen Auswahl (ja/nein) mit if, können mit dieser Konstruktion beliebig viele Fälle abgefragt werden. Der **switch** (Schalter) bestimmt, welcher Fall eingetreten ist. Switch muss sich immer auf eine Variable vom Typ integer beziehen. Die einzelnen Fälle werden mit dem Wort **case** eingeleitet. Damit nicht jeder Einzelfall geprüft wird, sollte das switch..case-Konstrukt mit der Anweisung **break** wieder verlassen werden. Wenn keiner der Fälle zutrifft, wird der **default**-Zweig ausgeführt. Dieser kann aber auch entfallen.

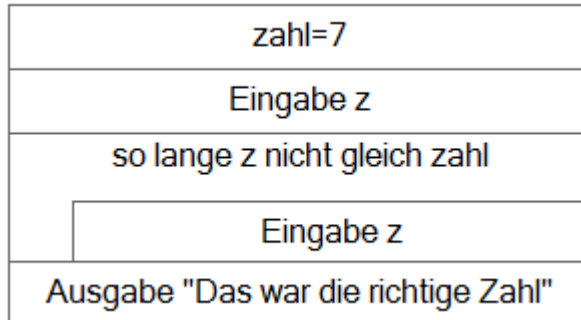
```
public class Switch {
    public static void main(String argv [ ]) {
        int zahl=5;
        switch(zahl) {
            case 1:
                System.out.println("sehr gut");
                break;
            case 2:
                System.out.println("gut");
                break;
            case 3:
                System.out.println("befriedigend");
                break;
            case 4:
                System.out.println("ausreichend");
                break;
            case 5:
                System.out.println("mangelhaft");
                break;
            case 6:
                System.out.println("ungenügend");
                break;
            default:
                System.out.println("irgendeine andere Zahl");
        }
    }
}
```



### d) Kopfgesteuerte Schleife (while)

#### Beispiel:

Zahlenraten: Es wird eine Zahl zwischen 1 und 10 im Quellcode definiert. Der Benutzer gibt eine Zahl zwischen 1 und 10 ein. Ist dies zufällig die definierte Zahl, ist das Spiel beendet, andernfalls wird die Abfrage wiederholt.



Bei der while-Schleife wird die Bedingung zu Beginn überprüft. Ist sie so schon zu Anfang erfüllt, wird die Schleife überhaupt nicht ausgeführt. Es ist zu beachten, dass das Kriterium für die Durchführung der Schleife (hier die Tastatureingabe) zweimal codiert wird: Einmal außerhalb der Schleife und einmal innerhalb der Schleife.

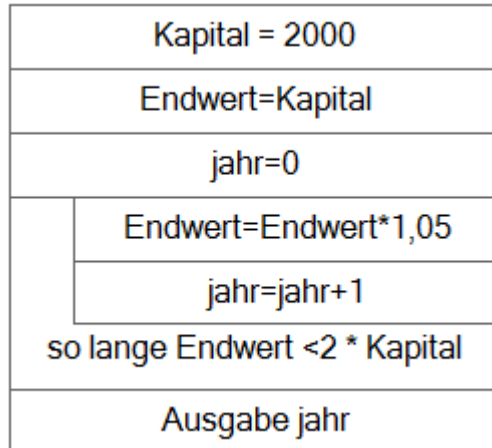
```
public class WhileTest {
    public static void main(String argv [ ]) throws IOException
    {
        int zahl=7;
        int z;
        z=Console_IO.IO_int("Bitte eine Zahl eingeben: ");
        while (z != zahl)
        {
            z=Console_IO.IO_int("Bitte eine Zahl eingeben: ");
        }
        System.out.println("Das war die richtige Zahl");
    }
}
```



### e) Fußgesteuerte Schleife (do ..while)

#### Beispiel:

Ein Anfangskapital wird mit 5 % pro Jahr verzinst. Die Zinsen bleiben stehen und werden dem Kapital zugeschlagen. Wenn sich das Anfangskapital mindestens verdoppelt hat, wird die Schleife verlassen und die Anzahl der Verzinsungsperioden wird ausgegeben.



Bei der do..while-Schleife wird erst am Ende eines Schleifendurchlauf die Bedingung überprüft. Das bedeutet, dass die Schleife auf jeden Fall ein Mal ausgeführt wird.

```
public class DoTest {  
    public static void main(String argv [ ]) {  
        int summe=0, i=1;  
        do  
        {    summe=summe+i;  
            i++;  
        }while(i<=100);  
        System.out.println("Summe der Zahlen 1 bis 100: „+summe);  
    }  
}
```

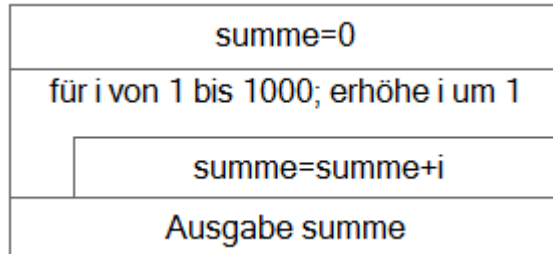




## f) Zählergesteuerte Schleife (for)

Beispiel:

Es soll die Summe aller Zahlen von 1 bis 1000 berechnet werden. Das Ergebnis wird am Ende ausgegeben.



Bei for-Schleifen steht die Anzahl der Schleifendurchläufe von Anfang an fest. Für den Zähler benötigt man eine Variable vom Typ integer. Der Schleifenkopf der for-Schleife besteht aus drei Teilen: Dem Anfangswert des Zählers, der Laufbedingung (bis zu welchem Wert des Zählers) und der Veränderung des Zählers nach jedem Durchlauf (meist i++, d.h. erhöhe den Zähler immer um 1).

```
public class ForTest {
    public static void main(String argv [ ])
    {
        int summe=0;
        for(int i=1; i <= 1000; i++)
        {
            summe=summe+i
        }
        System.out.println("Die Summe beträgt: "+summe);
    }
}
```

## g) break und continue

Mit der Anweisung **break** können Fallabfragen und Schleifen verlassen werden, ohne dass eine Bedingung überprüft wird. Das Programm wird dann an der Stelle hinter der letzten Schleifenanweisung fortgesetzt. Die **continue**-Anweisung sorgt dafür, dass der aktuelle Schleifendurchlauf wiederholt wird.

## 6.2 Bedingungs- und Vergleichsoperatoren

Für die Formulierung von Bedingungen innerhalb eines if-Konstruktes oder einer Schleife gibt es folgende Operatoren:

Beispiele:

==	gleich	if(a==b) ,	wenn a gleich b ist
< oder >	kleiner oder größer	if(a < b) ,	wenn a kleiner b ist
&&	und	while(a && b > 0) ,	so lange a und b größer als 0 sind
	oder	if(a    b > 10) ,	wenn a oder b größer als 10 ist
!	nicht	while(a != 1) ,	so lange a ungleich 1 ist.



## 7. Übungen

- 1) Ein Java-Programm fordert den Benutzer auf, eine Zahl zwischen 1 und 10 einzugeben. Wenn eine ungültige Zahl eingegeben wird, soll eine Fehlermeldung ausgegeben werden. Bei Eingabe einer gültigen Ziffer, wird die eingegebene Zahl wieder ausgegeben.
- 2) Schreiben Sie ein Programm, welches den Rechnungsbetrag anhand einer festgelegten (durch Eingabe oder Initialisierung) einer Auftragssumme ausrechnet: Es gilt:  
Bei einer Auftragssumme über 1000 € wird 10 % Rabatt gewährt, zwischen 100 und 1000 wird 5 % Rabatt gewährt und von 50 bis unter 100 € wird 3 % Rabatt gewährt.  
Zeichnen Sie ein Struktogramm zu dieser Aufgabe.

- 3) Ein Java-Programm soll die Zahlen von 1 bis 100 untereinander ausgeben.

- 4) Schreiben Sie ein Programm, welches in der ersten Zeile ein Sternchen (\*), in der zweiten Zeile 2 Sternchen usw. Die Anzahl der Zeilen soll variabel sein.

```
*  
**  
***  
****  
***** usw.
```

- 5) Schreiben Sie ein Programm *Quadrat*, welches nach Eingabe der Seitenlänge *a* ein Quadrat bestehend aus *a* Sternchen (\*) zeichnet.

Beispiel: *a* = 6

```
* * * * * *  
*           *  
*           *  
*           *  
*           *  
*           *  
* * * * * *
```

- 6) Es sollen alle ungeraden Zahlen zwischen zwei einzugebenden Werten am Bildschirm angezeigt werden.
- 7) Ein Kapital von 1000 Euro soll mit einem Zinssatz von 4 % verzinst werden.  
Wie viele Jahre dauert es, bis das Kapital einschließlich Zinsen  
a) sich verdoppelt hat  
b) auf 10 000 € angewachsen ist?
- 8) Nach Eingabe einer Monatsziffer (1 – 12) soll der passende Monat am Bildschirm ausgegeben werden (z.B. 3 = März).
- 9) Die Summe der Zahlen von 1 bis 5000 soll berechnet und ausgegeben werden.  
a) mit einer for-Schleife      b) mit einer while-Schleife



## 8. Arrays

Arrays werden auch Listen oder Tabellen genannt. Sie können mehrere Werte eines bestimmten Datentyps speichern. Je nachdem, ob die Tabelle aus einer Spalte oder mehreren besteht, unterscheidet man eindimensionale oder mehrdimensionale Arrays.

Beispiel: Wir wollen ein Array anlegen, das 10 Zahlen vom Datentyp Integer aufnehmen kann. Das Array soll den Bezeichner *feld* bekommen.

Zunächst deklarieren wir das Array mit folgender Anweisung:

```
int [ ] feld = new int [ 10] ;
```

Links vom Gleichheitszeichen wird das Array deklariert. Dieses erkennt man an den eckigen Klammern, die hinter dem Datentyp stehen. Rechts vom Gleichheitszeichen wird ein neues Objekt eines Arrays vom Datentyp integer erzeugt, welches 10 Arrayelemente enthält.

Die einzelnen Elemente des Arrays können mit dem Index angesprochen werden. Dabei hat das erste Element den Index 0. D.h. ein Array *x* mit 5 Elementen hat die Elemente *x*[0], *x*[1], *x*[2], *x*[3] und *x*[4];

Ein Array kann auch schon bei der Deklaration initialisiert (d.h. mit Werten versehen) werden. In diesem Fall muss es nicht mit *new* erzeugt werden.

```
int [ ] feld = {2,5,7,9,13,-4,6,3} ;
```

Die Größe des Arrays ergibt sich aus der Anzahl der Elemente, die durch Komma getrennt innerhalb der geschweiften Klammern stehen.

### 8.1 Verarbeiten eines Arrays

#### a) Ein- und Ausgabe von Elementen eines Arrays

Einzelne Elemente eines Arrays werden über den Index des Arrays angesprochen. Die Zuweisung eines Wertes in das 3. Element eines Integerarrays mit der Bezeichnung **zahlen** erfolgt mit der Anweisung: `zahlen[2] = 17;`

Die Anzeige dieses Elementes am Bildschirm erfolgt mit der Anweisung:

```
System.out.println(+zahlen[2]);
```

Sollen alle Elemente eines Arrays verarbeitet werden, so verwendet man eine *for..Schleife*. Der Zähler der *for..Schleife* ist gleichzeitig der Index der Elemente des Arrays. Siehe Beispiel oben.

#### b) Ausgabe aller Werte eines Arrayx

Wenn man alle Elemente eines Arrays verarbeiten möchte, indem man z.B. alle Werte ausgibt, einen Wert in einem Array sucht oder das Array sortieren möchte, so verwendet man ein Wiederholungskonstrukt. Wenn die Größe des Arrays bekannt ist, so verwendet man eine *for-Schleife*.

Beispiel: Ein Array enthält fünf Zahlen vom Typ Integer. Diese Zahlen sollen mit einer Ausgabeschleife am Bildschirm angezeigt werden.

```
public class Prog1 {  
    public static void main(String argv [ ])   
    {   int [ ] feld={5,7,0,15,234};  
        for(int i=0; i<5; i++)  
        {  
            System.out.println(feld[i]);  
        } } }
```



## c) Sortieren eines Arrays

Eine wichtige Aufgabe der Arrayverarbeitung ist das Sortieren eines Arrays in auf- oder absteigender Reihenfolge. Diese Aufgabe ist eine Standardaufgabe bei der Verarbeitung von großen Datenbeständen. (z.B. Sortiere alle Kunden nach Namen, Postleitzahl oder Alter)

In den meisten Programmiersprachen gibt es fertige Prozeduren, die uns die Aufgabe des Sortierens abnehmen. Dennoch dient es sehr dem Verständnis einer Programmiersprache, wenn wir uns selber Gedanken über die Logik des Sortierens machen. Es gibt unterschiedliche Sortierv Verfahren, die einfacher oder komplexer sind und unterschiedliche Zeit zum Sortieren eines Datenbestandes benötigen. Ein einfaches Verfahren wie der **Bubble-Sort** benötigt zum Sortieren eines Bestandes von 100 000 Daten 4 999 950 000 (ca. 5 Milliarden) Sortiervorgänge. Das braucht auch bei einem schnellen Prozessor eine ganze Weile. Schnellere Verfahren benötigen erheblich weniger Rechenvorgänge und sparen damit Rechenzeit.

Der **Bubble-Sort** und eine schnelleres Verfahren, wie der **Quick-Sort** sollten Gegenstand des Unterrichts sein.

## 8.2 Bestimmen der Länge eines Arrays

Bei Arrays, deren Länge nicht genau bekannt ist, kann die Länge mit der Variablen **arraybezeichnung.length** ermittelt werden.

Beispiel: 

```
int [] feld={6,5,4,3,2,1};
System.out.println("Länge des Arrays: "+feld.length);
```

Liefert die Ausgabe: Länge des Arrays: 6

## 8.3 Vordefinierte Funktionen zur Arrayverarbeitung

Für die Verarbeitung von Arrays bietet die Bibliothek **java.util.Arrays** eine Reihe von nützlichen Funktionen an. Für die Benutzung dieser Funktionen muss zuvor die Bibliothek **java.util.Arrays** mit **import** eingebunden werden. Zwei dieser Funktionen sollen hier vorgestellt werden.

### a) Ausgabe aller Werte des Arrays mit der **toString** - Methode

Die Methode **Arrays.toString(arraybezeichnung)** liefert den kompletten Inhalt des Arrays. Diese Methode kann in einer Ausgabeanweisung verwendet werden.

### b) Sortieren eines Arrays mit der **sort**-Methode.

Mit der Methode **Arrays.sort(arraybezeichnung)** werden die Elemente des Arrays aufsteigend sortiert.

Beispiel für die Anwendung der beschriebenen Funktionen:

```
import java.util.Arrays;
public class array1 {

    public static void main(String[] args)
    {
        int [] zahlen = {3,7,-4,9,1,5};
        System.out.println("Länge des Arrays "+zahlen.length);
        System.out.println("Unsortierte Ausgabe: "+Arrays.toString(zahlen));
        Arrays.sort(zahlen);
        System.out.println("Sortierte Ausgabe: "+Arrays.toString(zahlen));
    }
}
```

Das Programm liefert folgende Ausgabe:

Länge des Arrays: 6  
Unsortierte Ausgabe: [3, 7, -4, 9, 1, 5]  
Sortierte Ausgabe: [-4, 1, 3, 5, 7, 9]



## 8.4. Mehrdimensionale Arrays

Es ist möglich Arrays mit mehreren Dimensionen anzulegen. Am Beispiel eines zweidimensionalen Arrays soll dies gezeigt werden. Dargestellt werden soll folgende Tabelle:

5	12	6	19	-6
32	23	4	2	8
21	17	7	35	1

Die Tabelle besteht aus 3 Zeilen und 5 Spalten und kann 15 Werte aufnehmen.  
Das Array soll heißen: `z_tabelle`.

Die Deklaration des Arrays sieht so aus: `int z_tabelle [3] [5];`

Der Index läuft von 0 bis 2, bzw. von 0 bis 4:

Die Ausgabe der Zahl 35 würde dann folgendermaßen aussehen:

```
System.out.println(z_tabelle[2] [3]);
```

## 9. Übungen

- 1) Schreiben Sie ein Programm welches die Noten für eine Klassenarbeit über die Tastatur einliest und daraus anschließend einen Notenspiegel mit Durchschnittsnote erstellt.  
Es sollen mindestens 15 Noten in ein Integerarray eingelesen werden. Die Größe des Arrays wird zur Laufzeit festgelegt.
- 2) Schreiben Sie ein Programm, welches den größten Wert aus einem Array mit `n` Integerwerten ermittelt.
- 3) Gegeben ist das folgende Array: `int [ ] zahlen={ 17,3,64,23,1,9,4,90,55,41 }`
  - a) Die Werte des Arrays sollen unsortiert untereinander ausgegeben werden.
  - b) Die Werte des Arrays sollen unsortiert nebeneinander, getrennt durch ein Semikolon ausgegeben werden.
  - c) Das Array soll in aufsteigender Reihenfolge sortiert werden und anschließend wieder ausgegeben werden.
- 4) Legen Sie ein zweidimensionales Array mit 5 Zeilen und zwei Spalten vom Datentyp String an. In diesem Array werden Vorname und Nachname von Personen gespeichert.  
Lassen Sie alle Werte dieses Arrays so ausgeben, dass in einer Zeile jeweils der Vor- und Nachname durch ein Leerzeichen getrennt angezeigt werden.
- 5) Der Konzern CATON AG hat fünf Filialen in verschiedenen Städten. Die Filialorte sind in dem Array `FILIALEN` gespeichert:

Freiburg	Aachen	Passau	Schwerin	Düren
----------	--------	--------	----------	-------

Die Jahresumsätze (in Millionen) stehen im Array `JUMS` zur Verfügung  
(Die Reihenfolge entspricht der Reihenfolge der Filialen im Array `FILIALEN`)

289	144	89	321	128
-----	-----	----	-----	-----

Ermitteln Sie die Filiale mit dem höchsten Umsatz und geben Sie den Namen der Filiale und die Höhe des Umsatzes aus.

Erstellen Sie zunächst ein Struktogramm und realisieren Sie dann die Aufgabe in Java.



## 10. Prozeduren und Methoden

Anweisungen, die zusammen gehören und die gemeinsam eine Teilaufgabe des Programms lösen, werden zu Prozeduren oder Methoden zusammen gefasst. Wir nennen diese im Folgenden nur noch Prozedur. Jedes Java-Programm kennt mindestens eine Prozedur, die den Namen main trägt. In den ersten Programmen sind wir mit dieser einzigen Prozedur ausgekommen. Wenn die Programme aber komplexer werden, zerlegt man das Gesamtprogramm in mehrere Prozeduren. Diese Prozeduren können mehrfach verwendet werden und sie können sogar in einer Bibliothek gespeichert werden und damit auch anderen Programmen zur Verfügung stehen. Wir wollen das Prozedurkonzept anhand eines einfachen Beispiels darstellen:

Aufgabe:       Schreibe ein Programm 'Kreis', welches die Fläche Kreises berechnet, wenn ein Radius vorgegeben wird.

### a) Die Lösung des Problems innerhalb der Prozedur main könnte so aussehen:

```
public class Kreis1
{
    public static void main(String[] args)
    {
        final double pi=3.1415;
        double radius,flaeche;
        radius=10;
        flaeche=radius*radius*pi;
        System.out.println("Die Flaeche betraegt: "+flaeche);
    }
}
```

### b) Verwendung von Prozeduren

Eine Prozedur wird immer innerhalb einer Klasse definiert und hat allgemein folgendes Aussehen:

```
public static <Rückgabedatentyp> <Prozedurname> (Übergabeparameter)
{
    //hier steht der auszuführende Code
}
```

Man unterscheidet:

- a) Prozeduren ohne Parameter,
- b) Prozeduren mit Übergabeparameter
- c) Prozeduren mit Rückgabeparametern



## ba) Prozedur mit Übergabeparameter

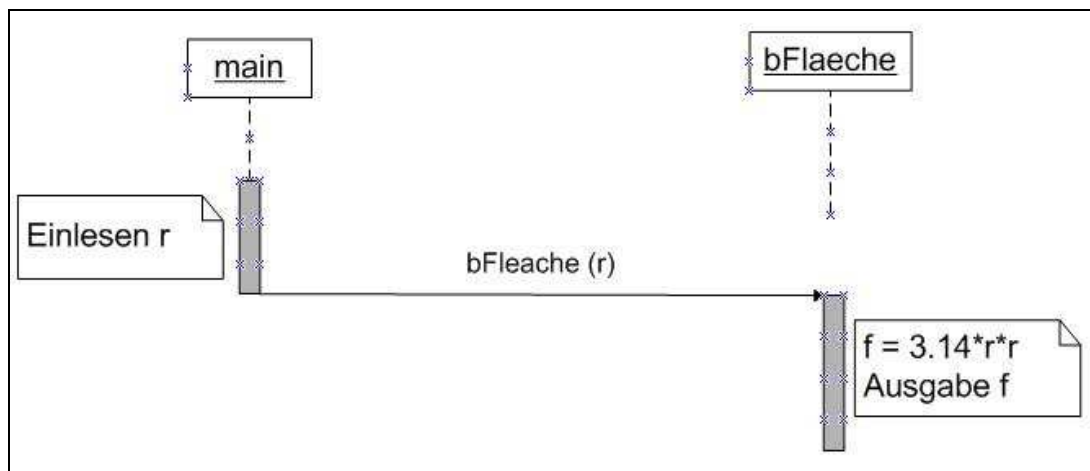
In der Prozedur main wollen wir lediglich den Radius einlesen. Das Berechnen der Fläche und die Ausgabe der Fläche wollen wir an eine Prozedur übergeben.

Dazu ein paar allgemeine Überlegungen: Wir verwenden für unser Programm die Variablen **radius** und **flaeche** sowie die Konstante **pi**.

In der Programmierung gilt, dass eine Variable nur innerhalb der Prozedur gültig ist, in der sie auch deklariert wurde. Eine Variable radius, die in der Prozedur main deklariert wurde, ist also in einer anderen Prozedur nicht bekannt. Wenn eine Variable auch in anderen Prozeduren gültig sein soll, so kann man sie außerhalb einer Prozedur deklarieren. Man nennt diese Variablen **globale Variablen**. Die allzu großzügige Verwendung globaler Variablen gilt aber als schlechter Programmierstil, da man so sehr leicht die Kontrolle über seine Variablen verliert, wenn man mit mehreren Prozeduren arbeitet. Deshalb muss der Wert einer Variablen an die Prozedur, die diesen Wert verwenden soll, übergeben werden. Man nennt dies auch eine **Botschaft**.

In unserem Fall heißt unsere Prozedur **bFlaeche**. Beim Aufruf der Prozedur müssen wir den Wert des Radius als Botschaft mitliefern. Dies geschieht, indem wir die Bezeichnung des Radius (hier r) in die Klammer hinter den Prozeduraufruf schreiben. Die beiden Klammern sind das Kennzeichen einer Prozedur.

Das Prinzip kann man gut in einem Sequenzdiagramm (siehe auch objektorientierter Teil) darstellen.



```
import java.io.*;
public class Kreis2
{
    final static double pi=3.1415;
    public static void main(String[] args) throws IOException
    {
        double r,f;
        r=Console_IO.IO_double("Bitte Radius eingeben: ");
        b_flaeche(r);
    }

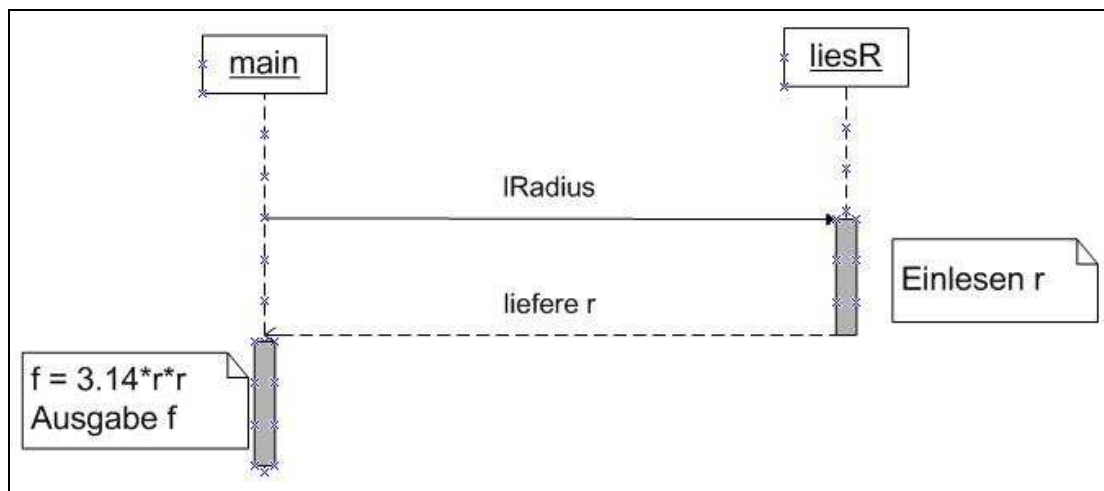
    public static void b_flaeche(double radius) //Dies ist die Prozedur
    {
        double flaeche;
        flaeche=radius*radius*pi;
        System.out.println("Die Flaeche betraegt: "+flaeche);
    }
}
```



Anmerkungen: Die Konstante pi wurde hier global definiert. Beim Aufruf der Prozedur wird lediglich ein **Wert** übergeben. Deshalb kann dieser Wert in der aufgerufenen Prozedur auch mit einer anderen Bezeichnung verwendet werden.

## bb) Prozedur mit Rückgabeparameter

In der Prozedur **liesR** soll lediglich der Radius eingelesen werden. Dieser soll dann an die Prozedur main zurückgeliefert werden.



```
import java.io.*;
public class Kreis1
{
    public static void main(String[] args) throws IOException
    {
        final double pi=3.1415;
        double f,r;
        r=liesR( );
        f=r*r*pi;
        System.out.println("Die Flaechе betraegt: "+f);
    }

    public static double liesR( ) throws IOException
    {
        double radius;
        radius = Console_IO.IO_double("Bitte Radius eingeben: ");
        return radius;
    }
}
```

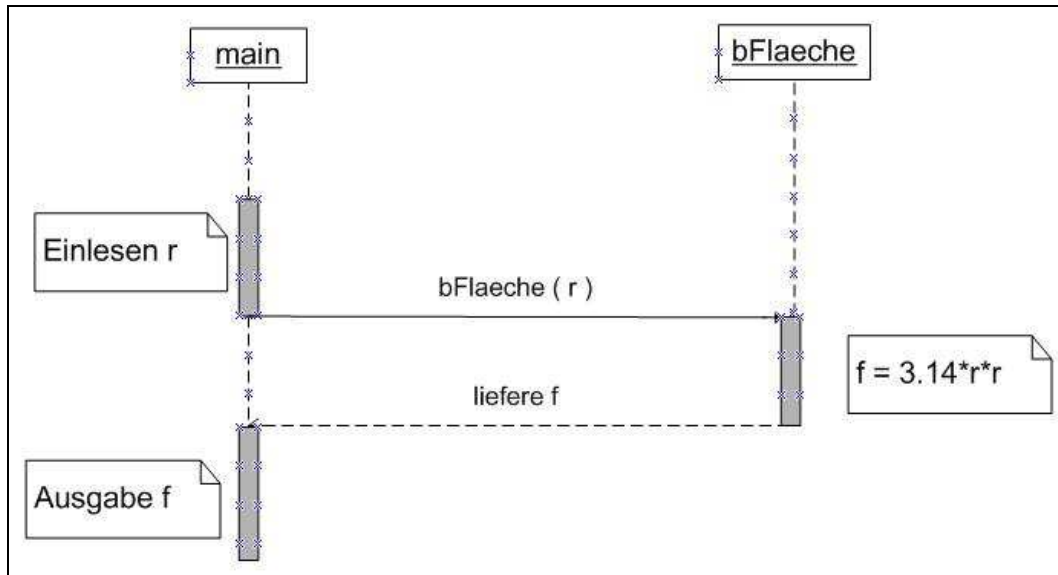
Anmerkungen: Die Rückgabe eines Wertes erfolgt mit dem Schlüsselwort **return**. Das zurückgelieferte Ergebnis muss in der beim Aufruf der Prozedur in Empfang genommen werden. Dies geschieht über die Wertzuweisung an die Variable r (r = liesR( ) ).





## bc) Prozeduren mit Übergabe und Rückgabewert

Man kann natürlich auch Prozeduren schreiben, die sowohl Übergabeparameter als auch einen Rückgabewert haben. **Eine Prozedur kann immer nur einen Wert zurückgeben.** Es können aber mehrere Werte übergeben werden. Hier soll die Prozedur **b\_flaeche** die Fläche des Kreise berechnen und zurückliefern.



```
import java.io.*;
public class Kreis2
{
    final static double pi=3.1415;
    public static void main(String[] args) throws IOException
    {
        double r,f;
        r=Console_IO.IO_double("Bitte Radius eingeben: ");
        f=b_flaeche(r);
        System.out.println("Die Flaeche betraegt: "+f);
    }

    public static double b_flaeche(double radius) //Dies ist die Prozedur
    {
        double flaeche;
        flaeche=radius*radius*pi;
        return flaeche;
    }
}
```



### **c) Verwendung von Arrays in Prozeduren**

Arrays können als Parameter und Rückgabewerte Prozeduren verwendet werden. Bei Rückgabe eines Arrays muss dessen Typ, gefolgt eckigen Klammern [ ] im Prozedurkopf genannt werden. Das gleiche gilt für Arrays als Übergabeparameter.

Beispiel: Der folgende Prozedurkopf erhält als Übergabeparameter ein Array und liefert ein Array zurück:

```
public int [ ] berechneArray( int array1 [ ] )
```

Bei der Rückgabe eines Arrays wird lediglich der Name des Arrays (ohne eckige Klammern) verwendet (call by reference)

z.B. return array1;

in der aufrufenden Prozedur muss das Array entgegengenommen werden, in dem es einer Arrayvariablen zugewiesen wird.

```
int array[ ] = berechneArray( a );
```



## 11. Übungen

- 1) Schreiben Sie das Programm welches zwei Zahlen a und b multipliziert. Die Multiplikation soll in einer Prozedur erfolgen, das Ergebnis wird an main zurückgegeben. Lösen Sie die Multiplikation durch eine iterierte Addition (ein Wiederholungskonstrukt, welches die Multiplikation durch eine Addition ersetzt).
- 2) Schreiben Sie ein Programm, welches nach Eingabe des Radius ein Auswahlmenü anbietet, in dem gewählt werden kann, ob die Fläche, der Umfang oder das Kugelvolumen berechnet werden soll. Die Auswahl erfolgt mit switch ...case. Für die einzelnen Berechnungen wird eine Prozedur aufgerufen.
- 3) Schreiben Sie ein Programm, welches nach Eingabe einer Codeziffer des ASCII-Codes (32 bis 128) das dazu gehörende Zeichen anzeigt. Die Ausgabe des Zeichens erfolgt in einer Prozedur, die die Nummer des Zeichens als Übergabeparameter erhält.
- 4) Ein Array mit Interwerten soll sortiert werden und nach dem Sortiervorgang wieder ausgegeben werden. Lösen Sie die Aufgabe mit Prozeduren.
  - a) Die Ausgabe des sortierten Arrays erfolgt in einer Prozedur
  - b) Der Sortiervorgang erfolgt in einer Prozedur

## 12. Packages

Jede in Java verwendete Klasse (d.h. auch jede Anweisung) wird in so genannten **Packages** (Paketen) verwaltet. Wir können auch eigene Klassen einem Package zuordnen. Ein Package ist ein Ordner bzw. eine Verzeichnisstruktur auf ihrem Rechner. Damit der Interpreter das Verzeichnis finden kann, in dem sich ein Package befindet, gibt es die CLASSPATH-Einstellung, in der das Standardverzeichnis für die Packages eingestellt wird.

Wenn eine Anweisung vom Compiler nicht erkannt wird, so kann es sein, dass diese sich in einem Package befindet, welches erst mit einer import-Anweisung eingebunden werden muss. Den Namen des Packages erfahren sie über die Hilfsfunktion für die entsprechende Anweisung.

Die Import Anweisung ist die erste Anweisung in einem Java-Quelltext (siehe Kap. 6, 1. Zeile im Beispiel). Die wichtigsten Packages sind:

<b>java.lang</b>	Standard-Funktionalitäten (wird immer automatisch eingebunden)
<b>java.io</b>	für Bildschirm- und Dateiein- und -ausgabe
<b>java.math</b>	für mathematische Methoden
<b>java.util</b>	Datenstrukturen, Tools- und Hilfsklassen
<b>java.text</b>	für Operationen mit Text und Zeichenketten

Wenn alle Klassen eines Packages eingebunden werden sollen, so verwendet man den \* - Operator  
z.B. `import java.io.*;`

Soll eine eigene Klasse einem Paket zugeordnet werden, so lautet die erste Anweisung in dieser Klasse : `package paketname;`

Wenn diese Package verwendet werden soll, so muss es mit einer import Anweisung wieder eingebunden werden.

**Pakete oder Klassen, die sich direkt im aktuellen Verzeichnis befinden, benötigen keine package-Anweisung und müssen nicht mit import eingebunden werden!**



## 13. Dateiverarbeitung

### a) Textdateien

Das folgende Listing zeigt, wie man eine Textdatei erzeugt und anschließend wieder einliest. Eine Textzeile wird über die Tastatur eingelesen und in der Datei line.txt gespeichert (Die Datei muss sich im selben Verzeichnis befinden, wie das aufrufende Programm). Anschließend wird die Datei wieder geöffnet und ihr Inhalt ausgegeben. Dateihandlings müssen mit *try* und *catch* auf Fehler überprüft werden (siehe Kapitel 14). Mit diesem Listing können Sie sich einen eigenen FileReader basteln, mit dem Sie den Inhalt jeder beliebigen Datei am Bildschirm anzeigen können.

```
import java.io.*;
public class FileWriteRead
{
    public static void main(String args[])
    {
        //Schreiben in eine Datei
        try
        {
            FileOutputStream schreib = new FileOutputStream("line.txt"); //Dateistream erzeugen
            int z=0;
            System.out.print("Gib eine Textzeile ein: ");
            String zeile=Console_IO.I_str();
            schreib.write(zeile.getBytes(),0,zeile.length()); //Schreiben der Textzeile in die Datei
            schreib.close(); //Datei wieder schließen
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Datei nicht gefunden!" + e.toString());
        }
        catch(IOException e)
        {
            System.out.println("Schreibfehler! " + e.toString());
        }
        //Lesen der Datei
        try
        {
            FileInputStream lies = new FileInputStream("line.txt");
            int z=0;
            while((z=lies.read())!=-1) //liest jeweils ein Zeichen aus der Datei, bis das letzte Zeichen
                                     erreicht ist (-1).
            {
                System.out.print((char)z);
            }
            lies.close() //Datei wieder schließen
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Datei nicht gefunden!" + e.toString());
        }
        catch(IOException e)
        {
            System.out.println("Lesefehler! " + e.toString());
        }
    }
}
```



## b) Datenbanken

Das folgende Beispiel zeigt, wie Datensätze in eine Kundendatei geschrieben werden. Damit die Attribute voneinander getrennt werden, ist in der Klassendefinition das Schlüsselwort **Serializable** zu verwenden. Die Anwendung ‚Kundendatenbank‘ schreibt Datensätze und liest diese Datensätze wieder aus. Selbstverständlich kann auch eine andere Applikation auf die Datensätze zugreifen, da diese ja gespeichert sind.

(Diese Darstellung gehört eigentlich schon in den objektorientierten Teil des Skripts)

zunächst die Applikation (Klassendefinition siehe nächste Seite):

```
public class Kundendatenbank
{
    public static void main(String[] args)throws Exception
    {
        String dateiname="Kunden.dat";
        FileOutputStream datAus=new FileOutputStream(dateiname);
        ObjectOutputStream oAus=new ObjectOutputStream(datAus);
        //Erzeugen von Objekten
        int anzahl = 3;
        Kunde a = new Kunde(101,"Meier","Herr",4711);
        Kunde b = new Kunde(102,"Wagner","Frau",1234);
        Kunde c = new Kunde(103,"Özdemir","Herr",7001);
        //Datensätze in Datei schreiben
        oAus.writeInt(anzahl);
        oAus.writeObject(a);
        oAus.writeObject(b);
        oAus.writeObject(c);
        oAus.close();
        System.out.println(anzahl+ " Datensätze in Datei "+dateiname+" geschrieben");
        System.out.println(a); //Aufruf der toString-Methode
        System.out.println(b);
        System.out.println(c);
        //Datensätze wieder ausgeben
        FileInputStream datEin=new FileInputStream(dateiname);
        ObjectInputStream oEin=new ObjectInputStream(datEin);
        int anz=oEin.readInt();
        for(int i=1; i<=anz;i++)
        {
            Kunde gelesen = (Kunde) oEin.readObject();
            //System.out.println(gelesen);
        }
        oEin.close();
    }
}
```

Für die Ausgabe der Datensätze wird hier die **toString** – Methode verwendet, die hier mit **oEin.readObject** aufgerufen wird.



## Klassendefinition

```
import java.io.*;

public class Kunde
implements Serializable           //damit die Attribute einzeln ausgegeben werden können
{
    public int knr;
    public String kname;
    public String kanrede;
    public int pin;

    //Konstruktor
    public Kunde(int nr,String kn,String ka,int p)
    {
        this.knr=nr;
        this.kname=kn;
        this.kanrede=ka;
        this.pin=p;
    }

    //toString-Methode für die Ausgabe eines Datensatzes
    public String toString()
    {
        String zustand= "Nr. "+knr+": "+kanrede+" "+kname+" Pin: "+pin;
        return zustand;
    }
}
```



## 14. Fehlerbehandlung (Exception Handling)

Beispiel: Wir wollen Eine Division der beiden Variablen a und b durchführen. Das folgende Programm soll die Aufgabe lösen und das Ergebnis am Bildschirm anzeigen:

```
public class Fehlertest
{
    public static void main(String argv[])
    {
        int a=4;
        int b=0;
        System.out.println("a : b = "+a/b);
    }
}
```

Da der Wert des Divisors 0 beträgt, kommt es zu einem Programmabsturz mit folgender Meldung:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Fehlertest.main(Fehlertest.java:12)
```

Eine solche Situation, die hier zu einem Programmabsturz führt, wird in Java als Ausnahmesituation (Exception) bezeichnet. Diese Situation müssen vom Programmierer vorher erkannt und entsprechend behandelt werden (Exception Handling). Normalerweise unterbricht Java in diesem Fall die Ausführung eines Befehls und löst eine zum Fehler passende Fehlermeldung aus. Das Erzeugen dieser Meldung wird als das **Werfen** einer Exception bezeichnet (**throw**). Wenn wir nun einen Programmabsturz verhindern wollen und unsere eigene Fehlerbehandlung programmieren, müssen wir den Fehler **Abfangen** (**catch**).

Zunächst probieren wir aber, ob eine bestimmte Anweisung einer Fehler erzeugt oder nicht. Dieser Versuch wird mit dem Schlüsselwort **try** bezeichnet. Beim Programmieren benötigen wir deshalb einen **try**- und einen **catch-Block**. Außerdem müssen wir wissen, um welche Art von Fehler es sich handelt. In unserem Fall handelt es sich um eine **ArithmeticException**, die auftritt, wenn bei sich bei Berechnungen Exceptions ergeben. Eine andere wichtige Exception-Klasse sind die **IOExceptions** für Fehler bei der Eingabe oder Ausgabe.

Unsere Lösung sieht nun so aus:

```
public class Fehlertest
{
    public static void main(String argv[])
    {
        int a=4;
        int b=0;
        try
        {
            System.out.println("a : b = "+a/b);
        }
        catch(ArithmeticException e)                //Exception Klasse und Art des Fehlers
        {
            System.out.println("Bei der Berechnung gab es ein Problem");
            System.out.println("Fehlerbeschreibung: "+e.getMessage());    //Ausgabe der Art
des Fehlers
        }
    }
}
```



Jetzt ergibt sich folgend Bildschirmausgabe und das Programm wird normal fortgesetzt.

```
Bei der Berechnung gab es ein Problem  
Fehlerbeschreibung: / by zero
```

Arithmetische Operationen, wie obige Division müssen in Java nicht zwingend abgefangen werden. Es gibt aber andere, die eine zwingende Fehlerbehandlung verlangen. So z.B. bei allen Eingabe- und Ausgabeoperationen von der Tastatur und mit Dateien. Programme, bei denen eine Exceptionbehandlung fehlt, werden erst gar nicht übersetzt. Da man aber nun nicht immer alle Fehlermöglichkeiten abfangen möchte, kann man die Fehlerbehandlung an das Java-System weiterleiten. Dies geschieht im Methodenkopf der main-Methode.

```
Public static void main(String [ ] args) throws IOException
```

Mit diesem Zusatz wird eine IOException an die übergeordnete Systemebene weitergeleitet und dort behandelt. Beim Auftritt eines Fehlers erhalten wir nun eine entsprechende Konsolenmeldung mit der die Ausführung des Programms abgebrochen wird.





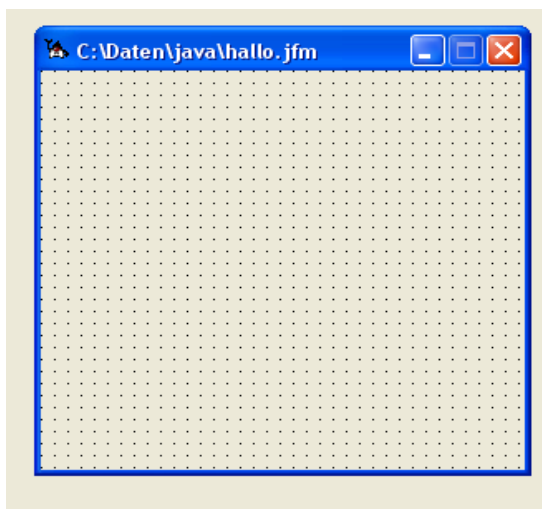
## 15. Graphische Benutzeroberflächen (GUI)

Auch wenn dieses Thema zu umfangreich ist, als dass man es in einem kurzen Skript darstellen kann, so soll hier kurz auf die Gestaltung von graphischen Oberflächen eingegangen werden.

Java stellt mit dem **AWT** (Abstract Window Toolkit) und **Swing** Bestandteile zur Verfügung mit deren Hilfe man graphische Elemente erzeugen kann. Die wichtigsten graphischen Elemente zur Gestaltung einer GUI sind Fenster (Frames), Labels, Eingabefelder, Schaltflächen (Buttons) und Menüs. Darüber hinaus gibt eine Reihe von zusätzlichen Komponenten.

Prinzipiell gibt es die Möglichkeit, den Quellcode für eine GUI selber zu schreiben, oder unsere Java-Entwicklungsumgebung stellt uns die graphischen Elemente zur Verfügung, so dass wir diese durch drag and drop auf dem Frame platzieren können. Dann wird der Quellcode für die graphischen Objekte und deren Eigenschaften automatisch erzeugt und wir müssen uns nur noch um die Programmierung der aktiven Elemente kümmern.

Diese zweite Möglichkeit erzeugt eine Menge Quellcode, der für den Anfänger zunächst noch sehr unübersichtlich und unverständlich ist. Deshalb wollen wir am Beispiel des ‚Hallo Welt‘ – Programms kurz die Funktionsweise darstellen.



Zunächst erzeugen wir mit dem Java-Editor ein sog. Frame als Ausgabefenster indem wir auf der rechten Seite der Menüleiste das Framesymbol anklicken und dieses unter der Bezeichnung ‚hallo‘ als Datei abspeichern. Damit wurde automatisch nebenstehender Quellcode erzeugt.

*Wenn wir die automatische Codegenerierung des Java-Editors nicht in Anspruch nehmen, dann können wir das Fenster auch nur über die main-Methode direkt programmieren:*

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

public class hallo1
{
    public static void main(String[] args) {
        Frame f1 = new Frame();
        f1.setTitle("Mein erstes Fenster");
        f1.setSize(250,250);
        f1.setVisible(true);
    }
}
```

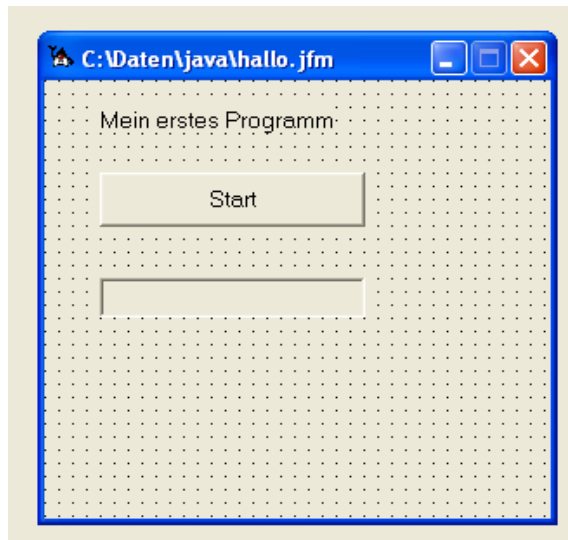
```
import java.awt.*;
import java.awt.event.*;

public class hallo extends Frame {
    // Anfang Variablen
    // Ende Variablen
    public hallo(String title) {
        // Frame-Initialisierung
        super("1.Beispiel: Hallo Welt");
        addWindowListener(new
        WindowAdapter() {
            public void
            windowClosing(WindowEvent evt) {
                System.exit(0); }
        });
        int frameWidth = 300;
        int frameHeight = 300;
        setSize(frameWidth, frameHeight);
        Dimension d =
        Toolkit.getDefaultToolkit().getScreenSize();
        int x = (d.width - getSize().width) / 2;
        int y = (d.height - getSize().height) / 2 ;
        setLocation(x, y);
        Panel cp = new Panel(null);
        add(cp);
        // Anfang Komponenten

        // Ende Komponenten

        setResizable(false);
        setVisible(true);
    }

    // Anfang Ereignisprozeduren
```



Die fertige Anwendung sieht aus wie oben:  
Wir haben jetzt zusätzlich zum Fenster drei Graphische Komponenten:

Ein Label mit der Bezeichnung **label1**  
und dem Text „Mein erstes Programm“.  
Der Text wird mit der Methode  
`label1.setText("Mein erstes Programm")` erzeugt.

Eine Schaltfläche mit der Bezeichnung **button1**  
Der Button trägt die Aufschrift Start  
`button1.setLabel("Start");`

Ein Textfeld mit der Bezeichnung **textField1**

Diese drei Objekte werden als Variable mit `new` erzeugt. Die genauen Eigenschaften eines Objekts (Größe, Lage, Farben, Schriftart, Label, Text usw.) werden dann im folgenden beschrieben.

Die aktive Komponente ist der Button. Hier soll ein Ereignis eintreten, wenn der Button gedrückt wird. Durch Drücken des Buttons soll im Textfeld ein Text erscheinen. Dieses Ereignis wird unter den Ereignisprozeduren für den `button1` programmiert. In unserem Beispiel ist die einzige Programmzeile, die wir selber schreiben müssen der Aufruf der Methode `setText` für das Textfeld:

```
textField1.setText("Hallo Welt");
```

Die Syntax zu dieser Anweisung kann man sich einige Zeilen weiter oben ‚abgucken‘.

Selbstverständlich kann hier auch komplexerer Programmcode programmiert werden, der unsere bekannten Strukturen wie Abfragen, Schleifen usw. enthält.

Wenn von dem fertigen Programm ein **-jar-Datei** erzeugt wird (Menü Start), kann es wie eine ausführbare Datei auf dem Desktop durch Anklicken ausgeführt werden. (Sofern Java auf dem Rechner installiert ist).

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

public class hallo extends Frame {
    // Anfang Variablen
    private Label label1 = new Label();
    private Button button1 = new Button();
    private TextField textField1 = new TextField();
    // Ende Variablen

    public hallo(String title) {
        // Frame-Initialisierung
        super(title);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent evt) { System.exit(0);
        }
    });
    int frameWidth = 300;
    int frameHeight = 300;
    setSize(frameWidth, frameHeight);
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    int x = (d.width - getSize().width) / 2;
    int y = (d.height - getSize().height) / 2;
    setLocation(x, y);
    Panel cp = new Panel(null);
    add(cp);
    // Anfang Komponenten

    label1.setBounds(32, 16, 135, 16);
    label1.setText("Mein erstes Programm");
    label1.setFont(new Font("MS Sans Serif", Font.PLAIN, 13));
    cp.add(label1);
    button1.setBounds(32, 56, 153, 33);
    button1.setLabel("Start");
    cp.add(button1);
    button1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            button1ActionPerformed(evt);
        }
    });

    textField1.setBounds(32, 120, 153, 24);
    textField1.setText("");
    cp.add(textField1);
    // Ende Komponenten

    setResizable(false);
    setVisible(true);
}

// Anfang Ereignisprozeduren
public void button1ActionPerformed(ActionEvent evt) {
    textField1.setText("Hallo Welt");
}

// Ende Ereignisprozeduren

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



## Verarbeitung von Zahlen in einem TextFeld:

Werte, die in ein Textfeld eingegeben werden, werden als Zeichenkette, d.h. als Text interpretiert. Oftmals sollen aber Zahlen eingegeben werden, mit denen im weiteren Programm gerechnet wird. Zu diesem Zweck müssen die Eingaben des Textfeldes einer Typumwandlung unterzogen werden. Wir kennen das bereits von der Tastatureingabe über die DOS-Konsole.

**Beispiel:** Wir wollen zwei Zahlen über zwei Textfelder einlesen und die Summe in einem dritten Textfeld wieder ausgeben. Die beiden Zahlen sollen den Datentyp **double** haben. Die Textfelder tragen die Bezeichnung **Zahl1**, **Zahl2** und **Summe**.

So soll das Ausgabefenster aussehen:

Wir programmieren den Button zum Berechnen und deklarieren zunächst drei Variablen `z1`, `z2` und `s` vom Datentyp `double`, in denen wir die umgewandelten Werte abspeichern.

Das Ergebnis wird im Textfeld `Summe` ausgegeben. Da ein Textfeld immer eine Zeichenkette als Ausgabe erwartet, stellen wir einen Leerstring unsere Summe voran.

```
public void button1_ActionPerformed(ActionEvent evt) {
    double z1, z2, s;
    z1= Double.parseDouble(Zahl1.getText());
    z2= Double.parseDouble(Zahl2.getText());
    s=z1+z2;
    Summe.setText(""+s);
}
```

Wollen wir die Ausgabe auf eine bestimmte Anzahl von Nachkommastellen formatieren, müssen wir die nachfolgenden Anweisungen einfügen (Hier für 1 NK-Stelle). Zusätzlich ist die Bibliothek `java.text.*` erforderlich. (siehe auch Kapitel 16).

```
public void button1_ActionPerformed(ActionEvent evt) {
    DecimalFormat d=new DecimalFormat("0.0");
    double z1, z2, s;
    z1= Double.parseDouble(Zahl1.getText());
    z2= Double.parseDouble(Zahl2.getText());
    s=z1+z2;
    Summe.setText(""+d.format(s));
}
```



## 16. Ergänzungen

In unsystematischer Reihenfolge werden hier kleine Probleme, die während unserer Arbeit auftauchen und die nicht eindeutig einem Kapitel zuzuordnen sind beschrieben.

### a) Formatierung von Ausgabewerten mit Dezimalzahlen in Konsolenanwendungen

Bsp.: Wir wollen eine Zahl mit 2 Nachkommastellen ausgeben:

Dazu verwenden wir die Klasse `DecimalFormat`, die in der Bibliothek `java.text` enthalten ist.

Wir benötigen folgende Anweisungen:

```
import java.text.* ;  
double z = 2.12345678                               //z ist die Zahl, die wir ausgeben wollen  
DecimalFormat d = new DecimalFormat("0.00");         //d ist ein beliebiger Bezeichner  
  
System.out.println("Ausgabe: "+d.format(z));
```

### b) Formatierung von Ausgabewerten mit dem Tabulatorzeichen

Durch Einfügen des Steuerungszeichens `\t` in einer `System.out`-Anweisung wird die nachfolgende Ausgabe um einen Tabulatorschritt nach rechts versetzt.

Bsp.: `System.out.print("Menge und Preis: "+m+"\t"+p);`

### c) Zeichenkettenverarbeitung (String handling)

Es kommt häufig vor, dass Eingaben über die Tastatur, die als Zeichenkette ankommen auf das Vorkommen bestimmter Zeichen untersucht werden sollen, oder dass bestimmte Zeichen oder Zeichenfolgen ersetzt werden sollen.

Beispielsweise ist es sinnvoll, dass bei der Eingabe einer Dezimalzahl die Eingabe des Punktes und des Kommas als Dezimaltrennzeichen möglich sein soll. Dazu ist es nötig, ein etwaiges Komma in der Zeichenkette durch einen Punkt zu ersetzen.

#### ba) Anzahl der Zeichen in einem String ermitteln ( `length()` )

```
Bsp.: String s1= "Max Weber Schule";  
int laenge;  
laenge = s1.length( );
```

#### bb) Bestimmte Zeichen in einer Zeichenkette finden und die Position anzeigen.

```
Bsp.: String muster = "x";  
int pos;  
pos = s1.indexOf(muster);  
if(pos ==-1)  
    System.out.println("Muster nicht gefunden ");  
else  
    System.out.println("Muster gefunden an Position: "+(pos+1));
```

#### bc) Ein Zeichen durch ein anderes Zeichen ersetzen

```
Bsp: Das e soll durch a ersetzt werden.  
char Ersatzzeichen='a';      Achtung: Hochkomma, keine Anführungszeichen!  
s1=s1.replace('e', Ersatzzeichen); //ersetzt alle e durch a
```

#### bd) Mehrere Zeichen oder Wörter mit `StringBuffer` ersetzen

```
Bsp: Ersetze das Wort Weber durch Maier  
String muster1 = "Weber";  
String muster2 = "Meier";  
StringBuffer s1neu = new StringBuffer(s1);  
int laenge1 = s1.length( );  
int laengemuster2.length( );  
pos = s1.indexOf(muster1);  
s1neu= s1neu.replace(pos;pos+laengemuster2,muster2);  
System.out.println("Neuer String: "+s1neu);
```



## 17. JavaScript und Java Applets

Ein Grund für den großen Erfolg von Java liegt in der Verwendung der Sprachelemente von Java für Webseiten. Durch die Verwendung von JavaScript, Java Applets und auch Java-Servlets lassen sich Webseiten z.B. interaktiv gestalten und Animationen ausführen. Die Unterschiede der drei genannten Möglichkeiten werden zunächst kurz beschrieben.

**JavaScript:** Der Java-Quellcode wird direkt in die HTML-Seite eingebunden. Der Browser benötigt einen Javascript-fähigen Interpreter, der aktiviert sein muss. JavaScript-Anweisungen haben wenig mit der Programmiersprache Java zu tun, die Grundstrukturen sind aber der Programmiersprache Java sehr ähnlich.

**Java-Applets:** Ein **Java-Applet** ist ein Computerprogramm in der Programmiersprache Java verfasst wurde, und normalerweise in einem Webbrowser ausgeführt wird. Java-Applets wurden eingeführt, um Programme in Webseiten ablaufen lassen zu können, die im Webbrowser (auf der Client-Seite) arbeiten und direkt mit dem Benutzer interagieren können, ohne Daten über die Leitung zum Server versenden zu müssen. Zunächst wird ein Java-Programm als Applet geschrieben, daraus eine `–class`-Datei erzeugt, die dann mittels eines HTML-Tags in die Webseite eingebunden wird. Voraussetzung dafür ist die Installation von Java auf dem PC oder innerhalb des Browsers.

**Java-Servlets:** Servlets sind ebenfalls Java-Programme, die allerdings auf dem Server ausgeführt werden. Der Inhalt der Antworten kann dabei *dynamisch*, also im Moment der Anfrage, erstellt werden und muss nicht bereits statisch in Form einer HTML-Seite für den Webserver verfügbar sein. Servlets stellen somit das Java-Pendant zu CGI-Skripten oder anderen Konzepten (wie PHP), mit denen dynamisch Web-Inhalte erstellt werden.

An einigen einfachen Beispielen soll die Funktionsweise von JavaScript und Java-Applets gezeigt werden.

### 17.1 Aufbau eines JavaScripts

Beispiel:

```
<html>
<head>
<title>JavaScript Testseite1</title>
<script language="JavaScript">
  <!--

  alert("Hello World")

  //-->

</script>
</head>
<body>
</body>
</html>
```

Diese HTML-Seite bewirkt die Ausgabe eines Fensters mit der Meldung Hello World. Die Einbindung des JavaScripts erfolgt hier im Kopf der HTML-Seite, gekennzeichnet durch das Tag `script` und `/script`. Die Zeichen `<!--` und `// ->` bewirken, dass der Quelltext von älteren Browsern, die JavaScript nicht unterstützen angezeigt wird. Der Script-Befehl **alert** ist die Ausgabe in Form eines Textfensters.



Eine andere Möglichkeit besteht darin, im Kopf eine Funktion zu definieren, die beim Eintritt eines bestimmten Ereignisses aufgerufen wird. Die Befehle für solche Ereignisse beginnen mit **on...** Im folgenden Beispiel haben wir eine Funktion *hello* geschrieben, die einen Text als Übergabeparameter erhält und diesen Text in einem Textfenster ausgibt. Der Übergabetext steht beim Aufruf der Funktion in einfachen Anführungszeichen. Der Aufruf der Funktion erfolgt, wenn das Dokument im Browser aufgerufen wird (onLoad).

```
<html>
<head>
<title>JavaScript Testseite1</title>
<script language="JavaScript">
<!--
function hello(ausgabetext)
{
    alert(ausgabetext)
}
//-->
</script>
</head>
<body onLoad="hello('Dies ist ein Text')">
</body>
</html>
```

Beispiel 3: Dieses Beispiel zeigt eine weitere Möglichkeit, Ereignisse auszulösen. Die Funktion *zeit* ermittelt sämtliche Daten des aktuellen Datums und der Uhrzeit. Wenn der Mauszeiger über den Text *Aktuelle Uhrzeit* wandert, soll die Uhrzeit mit Stunden und Minuten angezeigt werden. Diese Aktion wird mit dem Befehl **onMouseOver** ausgelöst. (Das Gegenstück zu diesem Befehl lautet **onMouseOut**, der ein weiteres Ereignis auslöst, wenn die Maus Text wieder verlässt.)

```
html>
<head>
<title>JavaScript Testseite1</title>

<script language="JavaScript">
<!--
function zeit( )
{
    uhr=new Date()
    document.write(uhr.getHours()+'.'+uhr.getMinutes())
}
//-->
</script>

</head>
<body >

<A href onMouseOver="zeit()">Aktuelle Uhrzeit</a>

</body>
</html>
```

Durch Experimentieren mit diesem Beispiel lassen sich bereits ansprechende Animationen erzeugen, wie z.B. wechselnde Bilder, die durch Mausbewegungen erzeugt werden.



## 17.2 Aufbau eines Java-Applets

Ein Java-Applet besteht aus mindestens zwei Dateien: Einer Datei im Java-Quellcode, die als Applet erzeugt wurde und in kompilierter Form vorliegt (\*.class) und einer html-Datei, die das Applet einbindet. Der folgende Code zeigt ein sehr einfaches Beispiel:

## 17.3 Einbinden eines Java-Applets in eine HTML-Seite

Wenn wir von einer HTML-Seite ein Java-Applet aufrufen wollen, gehen wir in drei Schritten vor:

1. Wir erstellen ein Java-Programm als Applet. Dazu erstellen wir eine Windows- Anwendung die die Klasse Applet erweitert. Z.B. mit der Klassendefinition: **public class Hallo extends Applet** (Unser Applet heißt also Hallo). Davon wird eine kompilierte .class Datei erzeugt.

```
//Hallo
import java.applet.*;
import java.awt.Graphics;
public class hallo extends Applet
{
    public void init()
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        g.drawString("Hi Welt",30,30);
    }
}
```

2. Wenn wir das Applet mit dem Java-Editor erzeugen wird automatisch eine HTML-Datei erzeugt, die das Applet einbindet. Dies geschieht mit dem <applet> -tag. Andernfalls wird sie selbst geschrieben. Sie trägt die Bezeichnung **Hallo.html** :

```
html>
<head>
<title>Hallo-Applet</title>
</head>
<body>
<h1></h1>
<applet code="Hallo.class" width="300" height="300">
</applet>
</body>
</html>
```

3. Jetzt wird die Datei Hallo.html als Link in die aufrufende Internetseite eingebunden:

```
<a href="Hallo.html">TestApplet</a>
```

# **Objektorientierte Programmierung mit Java und UML**

- 1. Grundlagen der objektorientierten Programmierung**
- 2. Die Klasse**
  - 2.1 Modellierung einer Klasse
  - 2.2 Realisierung
    - 2.2.1 Attribute
    - 2.2.2 Methoden
  - 2.3 Die Applikation (Arbeit mit einer Klasse)
  - 2.4 Übungen
  - 2.5 Get- und set-Methoden und Datenkapselung
  - 2.6 Übungen
- 3. Realisierung eines einfachen Anwendungsfalls (Fallbeispiel 1)**
  - 3.1 Beschreibung der Fallsituation
  - 3.2 Modellierung
    - 3.2.1 Use Cases
    - 3.2.2 Klassendiagramm
  - 3.3 Realisierung und Applikation
  - 3.4 Erweiterungen
    - 3.4.1 Der Konstruktor
    - 3.4.2 Die toString-Methode
  - 3.5 Endfassung
  - 3.6 Mehrere Objekte einer Klasse in einem Array verwalten
  - 3.7 Übungen
- 4. Realisierung einer Anwendung mit mehreren Klassen (Fallbeispiel 2)**
  - 4.1 Beschreibung der Fallsituation
  - 4.2 Analysephase
  - 4.3 Designphase
  - 4.4 Codierungsphase
  - 4.5 Erweiterung der Fallsituation – Vererbung
  - 4.6 Übungen
- 5. Arrays und Collections als Container für Beziehungen zwischen Klassen**
  - 5.1 Verweisattribute als Array
  - 5.2 Collections am Beispiel ArrayList
- 6. UML-Diagramme**
  - 6.1 Statische Sicht
    - 6.1.1 Das Anwendungsfalldiagramm (Use-case-Diagramm)
    - 6.1.2 Klassendiagramme und Erweiterungen
    - 6.1.3 Das Objektdiagramm
  - 6.2 Dynamischer Blick auf einen Anwendungsfall
    - 6.2.1 Das Aktivitätsdiagramm
    - 6.2.2 Das Sequenzdiagramm
  - 6.3 Weitere Diagramme
  - 6.4 Übungen
- 7. Das objektorientierte Vorgehensmodell**
- 8. Datenbankzugriff unter Java**
- 9. Wiederholung - Grundbegriffe der Objektorientierten Programmierung**



## 1. Grundlagen der objektorientierten Programmierung

Die Entwicklung komplexer moderner Softwaresysteme ist eine Aufgabe, die nur arbeitsteilig im Team vollzogen werden kann und die oft eine lange Zeit in Anspruch nimmt. Damit eine solche Aufgabe bewältigt werden kann, ist es notwendig, diese in mehrere Komponenten zu zerlegen. Man spricht von **Modularisierung**.

Modularisierte Programme erleichtern die Übersicht, lassen sich leichter warten und neuen Erfordernissen anpassen. Den Schlüssel dafür bietet die **objektorientierte Programmierung (OOP)**. Die Notwendigkeit für objektorientierte Programmierung ergibt sich auch dadurch, dass Programme heute auf verschiedenen Plattformen laufen sollen, dass von verschiedenen Programmen auf gemeinsame Daten zugegriffen wird, dass Veränderungen von Programmen mit möglichst geringem Aufwand vollzogen werden können.

Noch stärker als bei der prozeduralen und strukturierten Programmierung gilt für die OOP, dass das zu entwickelnde System vorher genau geplant werden muss. Bis das Programm entsteht, vergeht eine lange Entwicklungszeit, die für die Analyse, die Strukturierung und die Modellierung verwendet wird.

Die OOP ist einerseits einer Weiterentwicklung der prozeduralen Programmierung, indem sie mit dem Konzept der **Klasse** einen neuen umfassenden Datentyp zur Verfügung stellt, andererseits gibt es aber auch völlig neue Prinzipien, die die prozeduralen Programmierung nicht zur Verfügung stellt.

Die drei Basisprinzipien der OOP sind:

- **Kapselung**
- **Vererbung**
- **Polymorphie.**

Diese drei Prinzipien werden in den folgenden Kapiteln näher erläutert.

Bei der **prozeduralen** Programmierung stehen der logische Programmablauf im Vordergrund, bei der **objektorientierten** Programmierung stehen die **Daten** im Vordergrund.

**Objekte mit gleichen Eigenschaften (Attributen) und gleichen Fähigkeiten (Methoden) werden zu Klassen zusammengefasst.**

**Eine Klasse beinhaltet die Eigenschaften (= Daten) sowie die Methoden, die mit diesen Daten arbeiten.**

Beispiel: Ein Auto ist ein **Objekt**, das bestimmte **Eigenschaften und Methoden** besitzt. Diese werden zunächst in einer **Klassendefinition** genau beschrieben. Unser Auto soll zunächst einmal durch die **Eigenschaften**

- Kennzeichen
- Marke
- Kilometerstand
- Tankinhalt

beschrieben werden. Außerdem soll es die beiden **Methoden**

- Fahren
- Tanken

durchführen können

Dadurch, dass jetzt definiert wurde, welche Eigenschaften das Auto besitzt und welche Funktionen es ausführen kann, steht allerdings noch kein Auto vor der Tür. Dazu muss erst eine **Instanz** dieser Klasse erzeugt werden. **Eine Instanz ist dann das eigentliche Objekt**, von dem beliebig viele andere erschaffen werden können. Instanzen können z.B. jetzt *Annes Polo* und *Franks Golf* sein

## Attribute und Methoden

Die Eigenschaften einer Klasse werden auch deren **Attribute** genannt. Zu einer kompletten Klassendefinition gehören auch Angaben darüber, welche Werte diese Attribute annehmen können. Da Attribute nichts anderes sind als Variablen geschieht dies durch die Angabe des Datentyps (int, float, char ...)

Alle Instanzen besitzen die gleichen Attribute, allerdings in verschiedenen Ausprägungen (z.B. Farbe rot oder gelb).

Als **Methoden** werden die Funktionen bezeichnet, die in ein Objekt eingebunden sind. In den Methoden müssen die Funktionen der Klasse genau beschrieben werden. Für das Beschleunigen etwa müsste man alles vom Niederdrücken des Gaspedals, über das Öffnen der Benzinzufuhr bis zur Veränderung der Vergasereinstellung programmieren. Alle Instanzen der gleichen Klasse benutzen dieselben Funktionen

## Kapselung

Unter Kapselung versteht man, dass Attribute und Methoden in einem Objekt zusammengefasst werden und ihrem Umfeld nur als geschlossene Einheit zur Verfügung gestellt werden. Welches Attribut und welche Methode nach außen sichtbar sind, kann der Programmierer festlegen, indem er sie entweder als **private** oder **public** deklariert.

Alle Deklaration im private-Bereich sind **gekapselt**, d.h. sie können außerhalb der Klasse nicht angesprochen und verändert werden. Die Vereinbarungen im public-Bereich sind öffentlich, d.h. sie sind auch außerhalb der Klasse erreichbar. Die Datenstruktur soll ja von niemanden beeinflusst werden können, die Methoden aber sollen von unterschiedlichen Anwendungen benutzt werden können.

## Vererbung

Vererbung ist eines der wichtigsten Prinzipien der OOP. Man versteht darunter, dass von bereits bestehenden Klassen neue Klassen abgeleitet werden können, die zunächst die gleichen Attribute und Methoden besitzen wie ihre Vorgänger. Sie können aber mit zusätzlichen Elementen ausgestattet werden. Diejenige Klasse, von der eine neue **Unterklasse** abgeleitet wird, nennt man **Oberklasse** oder **Basisklasse**.

Von der Basisklasse Auto ließ sich beispielsweise eine Unterklasse **LKW** bilden, die als zusätzlichen Attribute die *Ladekapazität* sowie als neue Methoden *Beladen* und *Entladen* hätte.

## Polymorphie (Überladen von Funktionen)

Als Polymorphie bezeichnet man die Fähigkeit verschiedener Unterklassen ein und derselben Oberklasse, auf die gleiche Botschaft unterschiedlich zu reagieren. Das bedeutet, dass z. B. die Unterklasse LKW die gleiche Methode *Fahren* besitzt wie die Oberklasse Auto, die aber bei beiden unterschiedlich ausgeführt wird.

Dazu muss die vererbte Methode *Fahren* der Oberklasse durch eine veränderte Methode *Beschleunigen* überschrieben werden.

## Überladen

Unter dem Überladen einer Methode versteht man, dass es innerhalb einer Klasse mehrere Varianten der gleichen Methode geben kann. Diese haben den gleichen Namen, unterscheiden sich aber durch ihre Argumente (Übergabeparameter).

So wäre es beispielsweise denkbar, dass die Methode *Hupen* in zwei Varianten existiert. Wird sie ohne Übergabewert aufgerufen, ertönt ein gewöhnliches Hupsignal, werden dieser Methode aber die Variable *Dauer* übergeben, ertönt ein Hupsignal von unterschiedlicher Dauer.

## UML (Unified Modelling Language)

UML ist eine Modellierungssprache für objektorientierte Vorgehensmodelle. In der UML gibt es verschiedene Arten von Diagrammen, mit denen objektorientierte Programme in verschiedenen Phasen und aus unterschiedlichen Sichten modelliert werden.

Grundsätzlich kann man zwischen statischen und dynamischen UML-Darstellungen unterscheiden.

**Statische Diagramme** beschreiben den Zustand eines Systems. Das wichtigste Diagramm ist das **Klassendiagramm**.

Dynamische Diagramme beschreiben das zeitliche und logische Verhalten eines Systems oder die Aktivitäten innerhalb eines Systems. Wichtige Diagrammarten sind dafür das **Sequenzdiagramm** und das **Aktivitätsdiagramm**.

UML wird laufend weiterentwickelt und ist von der OMG standardisiert worden.

Die derzeit aktuelle Version ist die Version 2.1.2 (Nov. 07).

Da die UML mittlerweile einen sehr umfangreichen Notationsaufwand benötigt, werden wir im Rahmen des Unterrichts lediglich einige Diagramme der UML verwenden und den Notationsaufwand auf das Nötige begrenzen. In dieser eingeschränkten Form ist die UML ein sehr nützliches Hilfsmittel bei der Analyse und Modellierung objektorientierter Systeme.

Umfangreiche Möglichkeiten zur graphischen Darstellung von UML-Diagrammen bietet das Programm **Visio** von Microsoft. Einfache Klassendiagrammzeichner sind in vielen Java-Editoren integriert.

## Darstellung eines Klassendiagramms nach UML

Im UML-Klassendiagramm wird unser Auto-Beispiel wie folgt dargestellt:



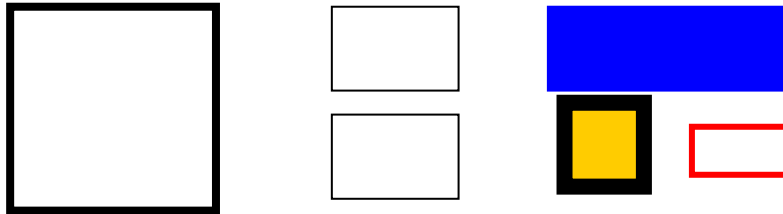
## Wiederholungsfragen

1. Nennen Sie Gründe, die zur Entwicklung der objektorientierten Programmierung geführt haben.
2. Durch welche Merkmale unterscheidet sie die OOP von der prozeduralen Programmierung?
3. Welche zwei Arten von Informationen gehören zur Definition einer Klasse?
4. Wie bezeichnet man die einzelnen Objekte, die von einer Klasse gebildet werden.
5. Erläutern Sie den Begriff Kapselung.
6. Was bedeutet Vererbung in der objektorientierten Programmierung?
7. Nennen Sie jeweils drei Attribute und Methoden einer fiktiven Klasse *Flugzeug* und zeichnen Sie die Klassendefinition in der UML-Darstellung.

## 2. Die Klasse

Objekte mit gleichen Eigenschaften werden zu Klassen zusammengefasst.  
Eine Klasse ist der Plan nach dem konkrete Objekte konstruiert werden.

Diesen Zusammenhang wollen wir uns am Beispiel einer Klasse mit der Bezeichnung **Rechteck** verdeutlichen.



In obigem Schaubild sehen wir eine Anzahl von Rechtecken, die alle die gleichen Attribute, jedoch unterschiedliche Attributwerte aufweisen.

Alle Rechtecke haben Gemeinsamkeiten. Sie haben alle vier Seiten, von denen jeweils zwei gegenüberliegende Seiten parallel und gleich lang sind. Jeder Winkel im Rechteck beträgt 90 Grad. Die Attribute (Seitenlängen, Farbe, Linienstärke, Linienfarbe) sind aber bei jedem Rechteck anders ausgeprägt.

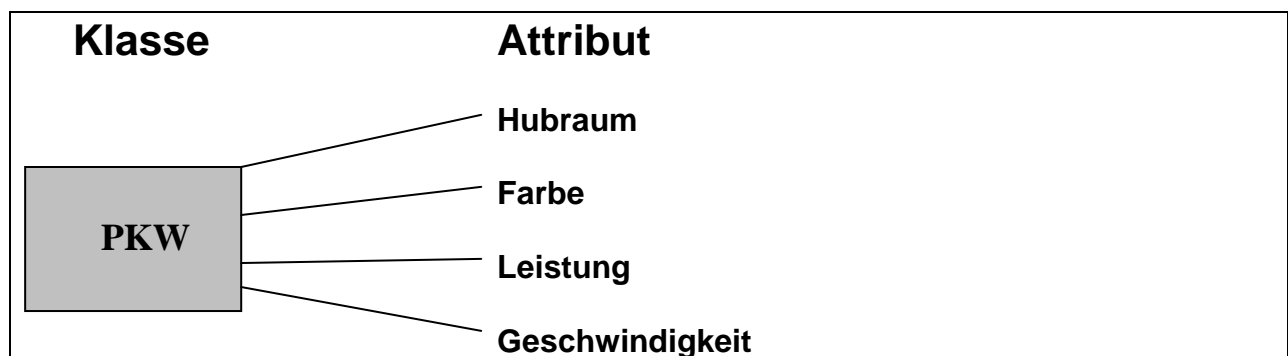
Die einzelnen, oben abgebildeten Rechtecke sind **Objekte** mit unterschiedlichen Attributausprägungen, die alle zur gemeinsamen Klasse **Rechteck** gehören.

Wir sagen auch: Die Rechtecke sind **Instanzen** oder **Objekte** der Klasse Rechteck.

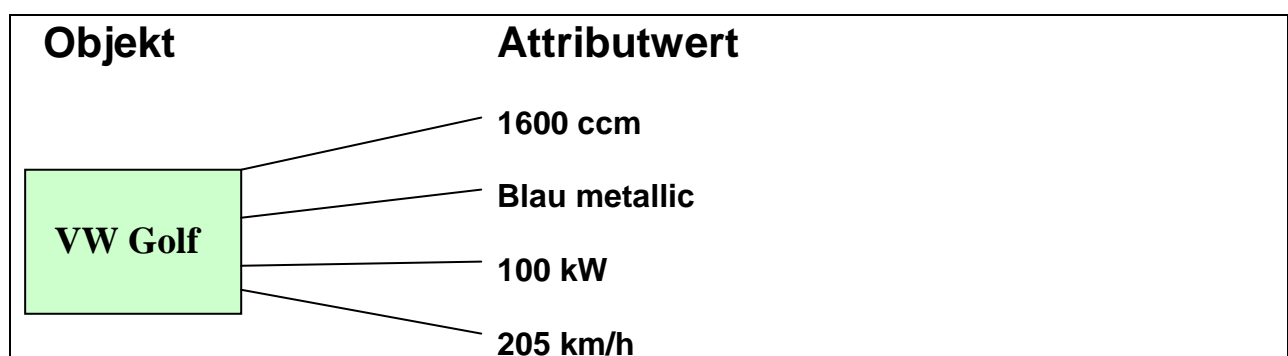
Für andere geometrische Formen wie Dreiecke, Kreise usw. lassen sich ähnliche Beziehungen bilden.

### Beispiel für Objekte und Klassen:

Nehmen wir an, alle PKWs bilden eine Klasse mit folgenden Attributen:



Eine **Instanz (Objekt)** der Klasse PKW könnte folgendermaßen aussehen:



Auf diese Art lassen sich also beliebig viele Objekte der Klasse PKW erzeugen.

## 2.1 Modellierung einer Klasse

Anhand des Auto-Beispiels wollen wir lernen, wie man das Design einer Klasse entwirft, in einem UML-Klassendiagramm darstellt und wie man das Diagramm in Java-Quellcode umsetzt. Dann werden wir Objekte dieser Klasse erzeugen und mit einer Applikation auf die Klasse zugreifen.

Als erstes legen wir fest, welche Eigenschaften (**Attribute**) ein Auto haben soll: Wir beschränken uns dabei auf die Attribute **kennzeichen**, **marke**, **kilometerstand** und **tankinhalt**. Für jedes Attribut legen wir den Datentyp fest und erhalten folgende Übersicht:

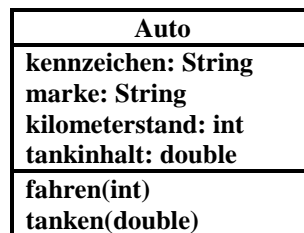
Datentyp	Bezeichnung
String	kennzeichen
String	marke
int	kilometerstand
double	tankinhalt

Jetzt legen wir fest, welche Fähigkeiten (Methoden) unser Auto haben soll. Wir beschränken uns auf die beiden Methoden **fahren** und **tanken**

Fahren bedeutet, dass sich der Kilometerstand des Fahrzeuges erhöht, Tanken bedeutet, dass der Tankinhalt des Autos erhöht wird.

(Es ist Java-Konvention, alle Attributsbezeichner und Methodennamen mit kleinen Buchstaben zu beginnen)

Nachdem wir uns über Aufbau und Funktion der Klasse einig sind, zeichnen wir das **Klassendiagramm**.



Das Klassendiagramm ist die wichtigste Diagrammart in der UML. Es beschreibt den Aufbau einer Klasse mit Attributen und Methoden. Es stellt sozusagen den Bauplan dar für die Objekte, die dann von dieser Klasse erzeugt werden. Folgende Regeln sollten für die Erstellung eines Klassendiagramms beachtet werden:

Das Klassendiagramm besteht aus einem Rechteck, welches in drei Bereiche unterteilt ist.

In der **oberen Zeile** steht der **Klassenname** zentriert und fett.

In der **zweiten Zeile** werden die **Attribute** aufgelistet. Das Minuszeichen vor der Attributsbezeichnung bedeutet, dass das Attribut gekapselt ist und mit dem Schlüsselwort **private** versehen ist (siehe Kapitel Kapselung). Wir werden das in Zukunft immer so handhaben.

Außerdem ist es möglich, den Datentyp des Attributes hinzuzufügen. Z.B. *-kilometerstand ; int*

In der **dritten Zeile** stehen die **Methoden**. Diese sind mit dem Schlüsselwort **public** versehen, was an dem Pluszeichen zu erkennen ist. Methoden können mit Übergabe- oder Rückgabewert noch näher beschrieben werden. Zur Darstellung von Klassendiagrammen mit mehreren Klassen und deren Beziehungen zueinander, können Attribute und Methoden weggelassen werden.

## 2.2 Realisierung

Nachdem der Aufbau der Klasse durch das Klassendiagramm modelliert wurde geht es daran, die Klasse Auto in Java umzusetzen. Dazu erzeugen wir eine neue Java-Klasse mit der Bezeichnung Auto.

### 2.2.1 Attribute

Der **Java-Quellcode** für unser Auto sieht zunächst folgendermaßen aus:

```
public class Auto {  
    //Attribute  
    String kennzeichen;  
    String marke;  
    int kilometerstand;  
    double tankinhalt;  
}
```

Nach dem Klassenkopf mit dem Namen der Klasse werden die Attribute mit ihrem Datentyp einzeln aufgelistet. (Klassenbezeichnungen beginnen mit Großbuchstaben, Attribute mit Kleinbuchstaben). Damit ist die Klasse zunächst funktionsfähig und kann nun erfolgreich kompiliert werden.

### 2.2.2 Methoden

Unser Auto verfügt jetzt über Attribute aber noch nicht über Methoden. Methoden werden in die Klassendefinition eingebaut und sehen aus wie Prozeduren. Nach Realisierung der Methoden **fahren** und **tanken** sieht unsere Klasse wie folgt aus:

```
public class Auto {  
    //Attribute  
    String kennzeichen;  
    String marke;  
    int kilometerstand;  
    double tankinhalt;  
    //Methoden  
    public void fahren(int km)  
    {  
        this.kilometerstand=this.kilometerstand+km;  
    }  
    public void tanken(int l)  
    {  
        this.tankinhalt=this.tankinhalt+l;  
    }  
}
```

Zum Aufbau einer Methode siehe Kapitel 11 im strukturierten Teil des Skripts. Die Methode **fahren** funktioniert nun wie folgt:

Der Methodenkopf zeigt an, dass der Methode ein Integerwert übergeben wird, der die gefahrenen Kilometer repräsentiert. Im Rumpf der Methode wird nun das Attribut kilometerstand um den übergebenen Wert erhöht. Das Schlüsselwort **this** welches durch einen Punkt mit dem Attribut verbunden ist, verweist darauf, dass der Wert des Attributes für jedes Auto unterschiedlich sein kann.

Die Klassendatei wird als eigene Datei unter der gleichen Bezeichnung wie der Klassenname gespeichert und kompiliert. Sie enthält keine main-Prozedur, da sie nicht ausführbar ist.

Mit dem Java-Editor kann man die Klasse über die Option **UML – Neue Klasse** interaktiv erstellen, ohne dass wir den Quellcode selbst schreiben müssen. Lediglich die beiden Methoden **fahren** und **tanken** müssen wir im Quellcode ergänzen.

Die Klassendatei **auto.java** wird nun kompiliert aber nicht ausgeführt, da eine Klasse keine ausführbare Datei ist (keine main-Methode).

### 2.3 Die Applikation (Arbeit mit einer Klasse)

Wir haben jetzt zwar den Plan, wie ein Auto aussehen soll, es existiert aber noch kein konkretes Auto. Im nächsten Schritt erstellen wir eine Applikation **autofahren**, die ein Auto-Objekt erzeugt und alle Attribute besetzt.

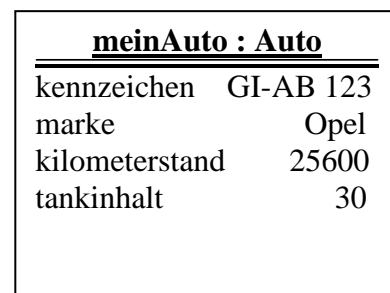
Zunächst überlegen wir uns, welche Eigenschaften unser Auto haben soll, d.h. welche Ausprägungen die Attribute der Klasse haben sollen.

Zur Veranschaulichung dient hier ein weiteres UML-Diagramm, das **Objektdiagramm**.

Das Objektdiagramm ähnelt dem Klassendiagramm, es werden allerdings die konkreten Ausprägungen der Attribute für ein Objekt benannt.

#### Objektdiagramm

In der ersten Zeile steht die Objektbezeichnung gefolgt von der Klassenbezeichnung fett und unterstrichen. In der zweiten Zeile folgen die Attributbezeichnungen und die konkreten Ausprägungen der Attribute.



Um nun von unserer Klasse reale Objekte zu erzeugen, müssen wir eine **Java-Applikation** schreiben.

Die entscheidende Zeile zur Erzeugung eines Objektes lautet: **auto meinAuto = new auto( );** Das neue Objekt wird nun unter der Bezeichnung meinAuto angesprochen. Diese Bezeichnung ist beliebig.

```
// autofahren
public class autofahren
{
    public static void main(String argv [ ])
    {
        auto meinAuto = new auto();
        meinAuto.kennzeichen="GI-AB 123";
        meinAuto.marke="Opel";
        meinAuto.kilometerstand=25600;
        meinAuto.tankinhalt=30;
    }
}
```

Mit den Anweisungen die der Objekterzeugung folgen, werden die Attribute mit den entsprechenden Werten initialisiert.

Wenn wir für ein Objekt eine Methode aufrufen wollen, so muss die Objektbezeichnung immer der Methodenbezeichnung durch einen Punkt getrennt vorangestellt werden.

Je nachdem, ob die Methode Übergabeparameter enthält oder Rückgabewerte liefert, müssen Werte in die Klammern des Methodenaufrufs geschrieben werden oder der Rückgabewert des Methodenaufrufes in der Applikation aufgefangen werden. Die Methode **tanken** rufen wir wie folgt auf:

```
meinAuto.tanken(25);           // Wir tanken 25 Liter
```

entsprechend die Methode **fahren**

```
meinAuto.fahren(170);         // Wir fahren 170 km
```

Wir können nun die geänderten Werte für den tankinhalt und den kilometerstand zur Kontrolle wieder am Bildschirm ausgeben. Dazu folgende Anweisungen:

```
System.out.println("Neuer Tankinhalt: "+meinAuto.tankinhalt);  
System.out.println("Neuer Kilometerstand: "+meinAuto.kilometerstand);
```

Es sollte folgende Bildschirmausgabe erscheinen:

```
Neuer Tankinhalt: 55.0  
Neuer Kilometerstand: 25770
```

### 2.4 Übungen

1. Wir benötigen eine Klasse **Person** mit den Attributen **name**, **vorname** und **alter**.  
Spezielle Methoden werden nicht benötigt. Erzeugen Sie diese Klasse und dazu eine Applikation.  
Erzeugen Sie zwei unterschiedliche Objekte und lassen sie deren Attributwerte am Bildschirm ausgeben.
2. Wir benötigen eine Klasse **Rechteck** mit den Attributen **seiteA** und **seiteB** und einer Methode **showFlaeche**, mit der die Fläche des Rechtecks berechnet werden soll.  
Zeichnen Sie das Klassendiagramm, codieren Sie die Klasse und erstellen Sie eine Applikation in der die Fläche eines Objekts der Klasse Rechteck am Bildschirm angezeigt wird.
3. Codieren Sie anhand des Klassendiagrammes die Klasse und eine Applikation, die die Ergebnisse der beiden Methoden am Bildschirm ausgibt.

Kreis
radius; double
berFlaeche: double
berUmfang: double
4. Es soll eine Klasse **Artikel** zur Erfassung der Artikeldaten eines Unternehmens erstellt werden. Es werden die Attribute, **artikelnr** – int, **artbez** – String, **preis** – double, **bestand** – int benötigt. Außerdem eine Methode **berWert**, die aus preis und bestand den Gesamtwert berechnet. Codieren Sie die Klasse, erzeugen Sie mindestens 2 Instanzen dieser Klasse und lassen sie den Gesamtwert aller Artikel anzeigen.



## 2.5 Get- und set-Methoden und Datenkapselung

Wie bereits gesagt wurde, ist die Datenkapselung ein wesentliches Merkmal der objektorientierten Programmierung. Das bedeutet, dass Attribute und Methoden einer Klasse eine Einheit darstellen und das Attribute vor Manipulationen geschützt sein sollen. Diese wird über das Schlüsselwort **private** erreicht. Attribute, die gekapselt werden sollen, werden mit dem Schlüsselwort **private** versehen. Damit sind sie vor jeglichem Zugriff geschützt. Eine Veränderung solcher Attribute ist nur noch mit einer speziellen Methode möglich, die das Schlüsselwort **public** besitzen muss. Für jedes Attribut, welches den Status **private** besitzt sollte es je eine Methoden geben, die das Attribut mit einem Wert besetzen kann und eine Methode, die den aktuellen Wert eines Attributes zurückliefert. Diese Methoden werden get- und set-Methode genannt.

Wir werden in weiteren Verlauf alle Attribute einer Klasse kapseln. Die get- und set-Methoden für jedes Attribut müssen nicht im Klassendiagramm dargestellt werden, da dieses sonst zu unübersichtlich werden würde. Die meisten Case-Tools zum Design und zur Codierung einer Klasse erzeugen get- und set-Methoden automatisch.

Nach Einführung der Datenkapselung sieht unsere Klasse nun wie folgt aus:

```
public class Auto {
    //Attribute
    private String kennzeichen;
    private String marke;
    private int kilometerstand;
    private double tankinhalt;
    //Methoden
    public void fahren(int km)
    {
        this.kilometerstand=this.kilometerstand+km;
    }
    public void tanken(int l)
    {
        this.tankinhalt=this.tankinhalt+l;
    }
    public void setKennzeichen(String k)
    { this.kennzeichen = k;
    }
    public String getKennzeichen( )
    { return this.kennzeichen;
    }
    public void setMarke(String m)
    { this.marke = m;
    }
    public String getMarke( )
    { return this.marke;
    }
    public void setKilometerstand(int k)
    { this.kilometerstand = k;
    }
    public int getKilometerstand( )
    { return this.kilometerstand;
    }
    public void setTankinhalt(double l)
    { this.tankinhalt = l;
    }
    public double getTankinhalt( )
    { return this.tankinhalt;
    }
}
```

Im Klassendiagramm werden Attribute die private sind mit einem – gekennzeichnet und public-Methoden mit einem +



## 2.6 Übungen

1. Verwenden Sie die Klasse **Artikel** aus Aufgabe 4 von Kapitel 2.4. Versuchen Sie alle Attribute mit dem Schlüsselwort **private**. Welche Fehlermeldung erhalten Sie, wenn Sie das Attribut **artnr** des Objektes **a1** mit der Anweisung `System.out.println("Artikelnr.: "+a1.artnr);` ausgeben wollen?

Wie können Sie das Problem lösen?

2. Welche Änderungen ergeben sich in nebenstehendem Klassendiagramm, wenn die Eigenschaften **private** und **public** mit dargestellt werden sollen?

Kreis
radius; double
berFlaeche: double
berUmfang: double

3. Codieren Sie die Klasse **Person** (Aufg. 1, Kapitel 2.4) neu, indem alle Attribute gekapselt werden.

Wie sieht der Aufruf in der Applikation aus, mit der das Attribut **vorname** des Objektes **p1** am Bildschirm angezeigt wird?

Zur Wiederholung

### a) Wie werden Attribute in einer Klasse angelegt:

Zum Anlegen eines Attributes in einer Klasse werden üblicherweise 3 Schlüsselwörter verwendet:

#### **Zugriffsrecht, Datentyp und Attributsbezeichnung**

**Beispiel:**     **private int breite;**

Zugriffsrecht: Für Attribute gibt es die Zugriffsrechte `privat`, `public` und `protected`.

- **private:**  
( - )     Wegen der Datenkapselung sollten Attribute grundsätzlich als `private` gekennzeichnet werden. Sie sind dann außerhalb der Klasse nicht mehr direkt ansprechbar, sondern müssen über `get-` und `set-` Methoden angesprochen werden
- **public**  
( + )     Mit `public` gekennzeichnete Attribute können überall genutzt werden. Dieses Zugriffsrecht wird weniger für Attribute als für Methoden oder Klassen verwendet.
- **protected**  
( # )     Auf Attribute und Methoden, die mit `protected` gekennzeichnet sind, kann innerhalb desselben Paketes zugegriffen werden. Derartige Attribute und Methoden werden an alle Subklassen weitervererbt und sind dort zugänglich. Attribute einer Basisklasse, die in einer Methode einer abgeleiteten Klasse verwendet werden, müssen `protected` sein.

Datentyp: Die gängigen Datentypen sind `String`, `int`, `double`, `char` oder Klassenbezeichner

Attributsbezeichnungen: Die Bezeichnung eines Attributes ist frei wählbar. Java-Konvention ist es, generell mit kleinen Buchstaben zu beginnen und bei zusammengesetzten Wörtern die Anfangsbuchstaben des folgenden Wortes groß zu schreiben: Bsp.: **zimmerBreite**.

## b) Wie ist eine Methode aufgebaut:

Jede Methode besteht aus einem Methodenkopf, einem Paar geschweifter Klammern und dem Methodenrumpf.

**Allgemein:**     *Zugriffsrecht Rückgabedatentyp Methodenbezeichnung (Übergabeparameterliste)*  
                  {  
                                  *Methodenrumpf (Quellcode)*  
                  }

**Beispiel:**     `public int berFläche(int a, int b)`  
                  {  
                                  `int f;`                             oder kürzer  
                                  `f = a * b;`                         `return a*b;`  
                                  `return f;`  
                  }

Zugriffsrecht:     Methoden sollten in der Regel mit **public** gekennzeichnet sein.

Rückgabedatentyp:     Wenn eine Methode mit **return** ein Ergebnis zurückliefert, so ist hier der Datentyp des zurückgelieferten Wertes anzugeben. Wenn es kein return gibt, so ist der Rückgabedatentyp **void**. Eine Methode kann immer nur einen Wert zurückgeben.

Methodenbezeichnung: frei wählbar, Java-Konventionen beachten

Übergabeparameter:     Es können beliebig viele Parameter übergeben werden, die mit Datentyp und Bezeichnung angegeben werden müssen. Die Bezeichnung ist innerhalb der Methode frei wählbar und muss nicht mit der Variablenbezeichnung beim Methodenaufruf übereinstimmen (call by value). Die richtige Reihenfolge muss aber unbedingt eingehalten werden.

Beispiel: obiges Beispiel kann in der Applikation wie folgt aufgerufen werden:

`r1.berFläche(zahl1, zahl2)`     oder auch     `r1.berFlächet(5, 7);`

(r1 steht für die Bezeichnung des aufrufenden Objekts)

## 3. Realisierung eines einfachen Fallbeispiels

Nachdem wir nun die Grundlagen des Klassenkonzepts kennen gelernt haben wollen wir anhand von zwei Fallbeispielen im Kapitel 3 und 4 zeigen, wie man einfache objektorientierte Anwendungssysteme entwickelt. Dabei wollen wir auf objektorientierte Vorgehensmodelle eingehen und weitere Anwendungsmöglichkeiten der UML kennen lernen

### 3.1 Beschreibung der Fallsituation

Auslöser einer objektorientierten Anwendungsentwicklung ist häufig eine Fallsituation, die in Form eines Textdokumentes oder eines Befragungsergebnisses vorliegt. Dieses liefert die Ausgangslage für die Modellierung des Anwendungssystems

Peter Merz hat sich eine neue Wohnung gemietet. Vor dem Einzug ist noch viel zu tun. Der Fußboden muss neu verlegt werden und die Wände müssen gestrichen werden. Für die Berechnung der notwendigen Materialmengen muss Peter Merz die Grundfläche jedes einzelnen Zimmers sowie die Wandfläche jedes Zimmers berechnen. Wir wollen diese Aufgabe mit Hilfe eines objektorientierten Java-Programms lösen

Anhand der vorliegenden Aussagen werden die zu realisierenden Anwendungsfälle sowie die benötigten Klassen herausgefunden. Dieses gehört zur **Analysephase** der Softwareentwicklung.

### 3.2 Modellierung

Anhand der Ergebnisse der Analysephase wird versucht, das künftige System mit Hilfe der UML-Diagramm Use-case-Diagramm und Klassendiagramm zu entwerfen. Dieses gehört zur **Designphase** der Softwareentwicklung.

#### 3.2.1 Use Cases

Auslöser unserer Aufgabenstellung ist die Absicht zwei **Anwendungsfälle** zu realisieren:

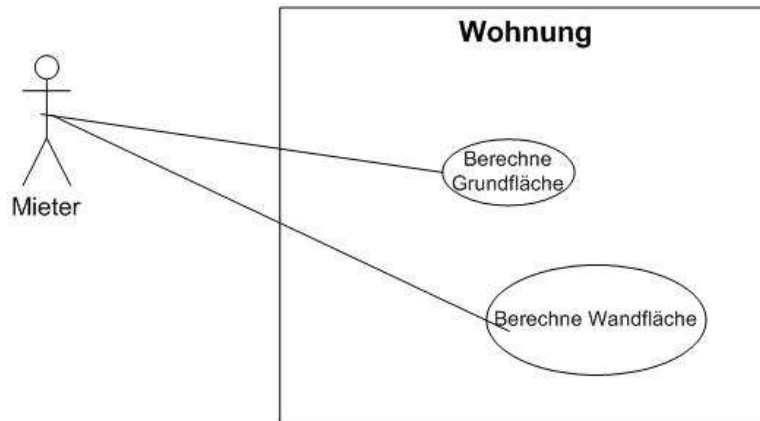
1. Berechnen der Grundfläche eines Zimmers
2. Berechnen der Wandfläche eines Zimmers

In der UML gibt es für die Darstellung der Anwendungsfälle einen speziellen Diagrammtyp, das sog. **Anwendungsfalldiagramm** oder auch **Use-Case-Diagramm**.

Anwendungsfalldiagramme werden zur Vereinfachung der Kommunikation zwischen Entwickler und zukünftigen Nutzer bzw. Kunde erstellt. Sie sind vor allem bei der Festlegung der benötigten Kriterien des zukünftigen Systems hilfreich. Somit treffen Anwendungsfalldiagramme eine Aussage, was zu tun ist, aber nicht *wie* das erreicht wird.

Anwendungsfälle werden durch **Ellipsen**, die den Namen des Anwendungsfalles tragen und einer Menge von beteiligten Objekten (**Akteuren**) dargestellt. Zu jedem Anwendungsfall gibt es eine Beschreibung in Textform. Die entsprechenden Anwendungsfälle und Akteure sind durch **Linien** miteinander verbunden. Akteure können durch Strichmännchen dargestellt werden. Die **Systemgrenze** wird durch einen Rahmen um die Anwendungsfälle symbolisiert.

Für unser Fallbeispiel könnte das Anwendungsfalldiagramm so aussehen:



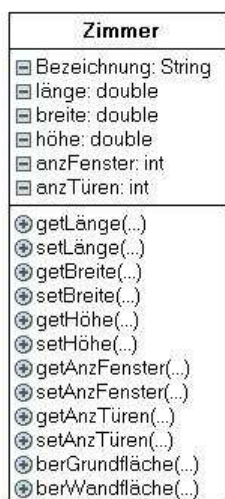
Das Anwendungsfalldiagramm ist sehr einfach und anschaulich und stellt den Ausgangspunkt für die Realisierung des Systems dar.

(Zum Use-case-Diagramm siehe auch Kapitel 9.3)

## 3.2.1 Das Klassendiagramm

Aus der Beschreibung der Fallsituation ergibt sich, dass wir mit einer einzigen Klasse auskommen können, da Peters Wohnung aus den Zimmern Wohnzimmer, Schlafzimmer, Küche, Bad und Flur besteht. Alle Zimmer haben eine rechteckige Grundfläche und haben eine unterschiedliche Zahl an Fenstern und Türen. Es liegt daher nahe, eine Klasse **Zimmer** zu modellieren.

Wir wollen folgende Attribute verwenden: **Länge**, **Breite** und **Höhe** des Zimmers, die **Anzahl Türen** und die **Anzahl Fenster**. Länge, Breite und Höhe haben den Datentyp double, die Zahl der Türen und Fenster sind Integerwerte. Außerdem benötigen wir ein Attribut **Bezeichnung** als String-Wert, welches uns anzeigt, um welches Zimmer es sich handelt. Es ergibt sich folgendes Klassendiagramm:



Die wichtigen beiden Methoden der Klasse sind die Methoden **berGrundfläche** und **berWandfläche**, mit der die Grundfläche und die Wandflächen berechnet werden sollen

Außerdem sind hier im Klassendiagramm alle get- und set-Methoden mit aufgeführt. Dies ist normalerweise nicht üblich bzw. notwendig, da sich bei einer großen Anzahl von Attributen das Klassendiagramm nur unnötig aufblähen würde. Für die Attribute sind die Datentypen ersichtlich, bei den Methoden sind in dieser Darstellung die Übergabe und Rückgabeparameter nicht ersichtlich. Die Ausführlichkeit des Klassendiagramms hängt von dem jeweiligen case-Tool ab, welches zum Design der Klasse verwendet wird.

## 3.3 Realisierung und Applikation

Der Quellcode für die **Klasse** sieht folgendermaßen aus:

```
public class Zimmer {  
  
    // Anfang Variablen  
    private String Bezeichnung;  
    private double länge;  
    private double breite;  
    private double höhe;  
    private int anzFenster;  
    private int anzTüren;  
    // Ende Variablen  
  
    // Anfang Ereignisprozeduren  
  
    public double getLänge() {  
        return länge;  
    }  
  
    public void setLänge(double länge) {  
        this.länge = länge;  
    }  
  
    public double berGrundfläche()  
    {  
        return this.breite*this.länge;  
    }  
    public double berWandfläche()  
    {  
        double w,w1,w2,tf,ff;  
        w1=this.länge*this.höhe*2;  
        w2=this.breite*this.höhe*2;  
        tf=this.anzTüren*1*2;  
        ff=this.anzFenster*1*0.8;  
        w=w1+w2-(tf+ff);  
        return w;  
    }  
    public String getBezeichnung() {  
        return Bezeichnung;  
    }  
  
    public void setBezeichnung(String Bezeichnung) {  
        this.Bezeichnung = Bezeichnung;  
    }  
  
    // Ende Ereignisprozeduren  
}
```

Zur Verkürzung wird hier lediglich die get- und set-Methode für das Attribut **länge** dargestellt.

Die Methode **berGrundfläche** soll die Grundfläche eines Zimmer berechnen. Sie benötigt dazu die beiden Attribute **länge** und **breite**. Die Methode enthält keine Übergabeparameter, da die Werte für die Länge und die Breite schon bei der Erzeugung des Objekts mit einer set-Methode festgelegt wurden. Das Ergebnis der Berechnung **länge \* breite** wird mit **return** zurückgeliefert.

Die Methode **beWandfläche** soll die Fläche aller Zimmerwände berechnen. Sie benötigt dazu die Attribute **länge**, **breite** und **höhe**. Außerdem sind die Flächen für die Türen und die Fenster abzuziehen. Aus Vereinfachungsgründen wollen wir eine Tür mit den Maßen 1 m x 2 m und ein Fenster mit den Maßen 1 m \* 0,8 m ansetzen. Die Methode enthält ebenfalls keine Übergabeparameter.

Wir nennen unser **Applikation** ,**wohnen**' und erzeugen zunächst ein Objekt z1, welches das Wohnzimmer sein soll. Das folgende Objektdiagramm zeigt die Ausgangsdaten:

<u><b>z1 : Zimmer</b></u>	
Bezeichnung	Wohnzimmer
länge	5
breite	4
höhe	2,4
anzFenster	3
anzTüren	2

```
public class wohnen
{
    public static void main(String argv[])
    {
        Zimmer z1= new Zimmer( );
        z1.setLänge(5);
        z1.setBreite(4);
        z1.setHöhe(2.4);
        z1.setAnzFenster(3);
        z1.setAnzTüren(2);
```

Der Quellcode links erzeugt ein Objekt mit der internen Bezeichnung z1 und initialisiert alle Attribute mit einer set-Methode.

```
//wir rufen die Methoden zur Berechnung der Grundfläche und der Wandfläche auf und lassen den
//Rückgabewert am Bildschirm anzeigen
```

```
    System.out.println("Grundfläche: "+z1.berGrundfläche());
    System.out.println("Wandfläche: "+z1.berWandfläche());
}
}
```

Auf die gleiche Weise müssen wir nun die übrigen Objekte für die anderen Zimmer erzeugen und initialisieren.

## 3.4 Erweiterungen

### 3.4.1 Der Konstruktor

Attribute für alle Objekte mit einer set-Methode zu initialisieren ist mitunter recht mühsam. Einfacher wäre es, wenn es möglich wäre, direkt bei der Erzeugung eines neuen Objektes alle Attribute mit einer einzigen Anweisung zu initialisieren. Dieses ermöglicht ein **Konstruktor**.

Eine Konstruktor ist eine Methode, die ein neues Objekt einer Klasse erzeugt. Der Aufruf `Zimmer z1 = new Zimmer( );` ist ein solcher Konstruktor. Wir erkennen an dem Klammerpaar, dass es sich um eine Methode handelt. Ein Konstruktor ist eine Methode, die die gleiche Bezeichnung trägt wie die Klasse. Ein Konstruktor, der keine Übergabeparameter aufweist ist der sog. **Standardkonstruktor**. Dieser muss nicht explizit programmiert werden, sondern wird vom Compiler beim Erzeugen eines Objekts als Standardmethode aufgerufen.

Wenn wir einen Konstruktor verwenden möchten, der etwas mehr tut, als ein Objekt lediglich zu erzeugen, so müssen wir uns einen eigenen Konstruktor programmieren und diesen in der Klasse implementieren.

Konstruktor als Methode der Klasse Zimmer:

```
public Zimmer(String z,double l, double b, double h, int f, int t)
{
    this.Bezeichnung=z;
    this.länge=l;
    this.breite=b;
    this.höhe=h;
    this.anzFenster=f;
    this.anzTüren=t;
}
```

**this:** Der Wert eines Attributes gehört immer zu einem bestimmten Objekt der Klasse. In der Applikation wird dies immer durch die Objektbezeichnung gefolgt von einem Punkt kenntlich gemacht. Da wir in der Klasse das konkrete Objekt nicht kennen, verwendet man den Parameter **this** als Platzhalter für das konkrete Objekt, welches in der Applikation gemeint ist. **This** kann innerhalb der Klasse vor einem Attribut oder einer Methode stehen.

So wird der Konstruktor in der Applikation aufgerufen:

```
Zimmer z1= new Zimmer("Wohnzimmer",5,4,2.40,3,2);
```

Es ist darauf zu achten, dass Reihenfolge und Datentypen der Übergabeparameter in der Applikation und in der Klasse genau übereinstimmen.

Achtung: Man kann mehrere unterschiedliche Konstruktoren definieren. Wenn ein eigener Konstruktor geschrieben wurde, wird der Standardkonstruktor dadurch überschrieben. Er existiert dann nicht mehr. Wenn man jetzt ein Objekt erzeugen möchte ohne Attribute zu initialisieren, so muss der **Standardkonstruktor** erst wieder explizit codiert werden. Das sieht einfach so aus:

```
public Zimmer() {
}
```

### 3.4.2. Die toString-Methode

Im Rahmen der Dateiverarbeitung im vorhergehenden Teil des Skripts wurde bereits die toString-Methode als Möglichkeit, Daten in eine Datei zu schreiben, erwähnt. Die toString-Methode kann dazu verwendet werden, die Attributwerte eines Objektes wieder auszugeben, ohne die jeweilige get-Methode für die Attribute verwenden zu müssen. Dafür bilden wir einen Ausgabestring, der unsere gewünschte Bildschirmausgabe enthält. Der Aufruf der toString-Methode ist dann denkbar einfach: Es wird lediglich die Objektinstanz innerhalb der Ausgabeanweisung übergeben. Zuvor muss in der Klasse die toString-Methode in der Klasse implementiert werden. Für unser Beispiel wie folgt:

```
public String toString()
{
    String ausgabe = "Zimmer: "+this.Bezeichnung+", Länge: "+this.länge+", Breite: "+this.breite;
    return ausgabe;
}
```

**Der Aufruf in der Applikation erfolgt dann so**

```
System.out.println(z1);
```

Und erzeugt folgende Ausgabe:  
**Zimmer: Wohnzimmer, Länge: 5, Breite: 4**



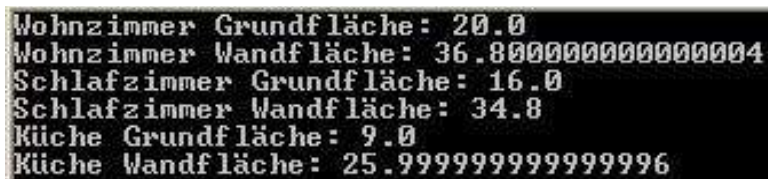
## 3.5 Endfassung

Die entgeltige Applikation mit Ausgabe der Daten über die to-String-Methode und Ausgabe aller Flächen sieht so aus:

```
public class wohnen
{
    public static void main(String argv[])
    {
        Zimmer z1= new Zimmer("Wohnzimmer",5,4,2.40,3,2);
        Zimmer z2= new Zimmer("Schlafzimmer",4,4,2.40,2,1);
        Zimmer z3= new Zimmer("Küche",3,3,2.40,1,1);

        System.out.println(""+z1.getBezeichnung()+" Grundfläche: "+z1.berGrundfläche());
        System.out.println(""+z1.getBezeichnung()+" Wandfläche: "+z1.berWandfläche());
        System.out.println(""+z2.getBezeichnung()+" Grundfläche: "+z2.berGrundfläche());
        System.out.println(""+z2.getBezeichnung()+" Wandfläche: "+z2.berWandfläche());
        System.out.println(""+z3.getBezeichnung()+" Grundfläche: "+z3.berGrundfläche());
        System.out.println(""+z3.getBezeichnung()+" Wandfläche: "+z3.berWandfläche());
    }
}
```

erzeugt folgende Ausgabe:



```
Wohnzimmer Grundfläche: 20.0
Wohnzimmer Wandfläche: 36.800000000000004
Schlafzimmer Grundfläche: 16.0
Schlafzimmer Wandfläche: 34.8
Küche Grundfläche: 9.0
Küche Wandfläche: 25.999999999999996
```

(Um die unregelmäßige Anzahl Nachkommastellen zu vermeiden, müssten wir eine Ausgabeformatierung vornehmen, wie in Kapitel 16 des Teils zur strukturierten Programmierung beschrieben).

## 3.6 Mehrere Objekte einer Klasse in einem Array verwalten

Normalerweise existiert von einer Klasse eine Vielzahl von Objekten. Sie wie unsere Wohnung mehrere Zimmer hat, so hat eine Klasse Kunden möglicherweise mehrere Hundert Objekte. Eine Auswertung aller Objekte in der obigen Form ist nun nicht mehr zu leisten. Wir können nicht Hunderte von System.out-Anweisungen schreiben, um alle Attributwerte auszugeben. Dieses wird durch Wiederholungskonstrukte, wie for-, while- oder do while-Schleifen erledigt. Dazu ist es aber nötig, dass man die einzelnen Objekte der Klasse über einen **Index** ansprechen kann.

Die einfachste Lösung ist es, die Objekte in einem Array zu speichern.

**Beispiel:** Wir wollen alle Objekte der Klasse Zimmer in einem Array speichern:

1. Wir erzeugen ein Array vom Datentyp der Klasse, die wir verarbeiten wollen

`Zimmer [] z = new Zimmer[5];` Dieses Array kann 5 Zimmerobjekte aufnehmen.

2. Wir erzeugen Objekte und speichern diese im Array

```
z[0] = new Zimmer("Wohnzimmer",5,4,2.40,3,2);  
z[1] = new Zimmer("Schlafzimmer",4,4,2.40,2,1); usw.
```

3. Die Verarbeitung, Auswertung usw. geschieht über eine Schleife.

In Kapitel 5.2 werden wir noch sehen, dass es auch noch komfortablere und flexiblere Methoden gibt, eine große Zahl von Objekten zu verwalten.

Wenn die Objekte einer Klasse dauerhaft gespeichert werden sollen, müssen die Daten in einer Datenbank abgelegt werden. Eine Darstellung zur Speicherung in einer Datenbank liefert das Kapitel 12 (Dateiverarbeitung) des vorhergehenden Skripts.

## 3.7 Übungen

1. Benötigt wird eine Klasse **Kunde** mit den Attributen **kundennummer**(int), **anrede**(String) und **name**(String).  
Schreiben Sie diese Klasse mit get- und set-Methoden, einem Konstruktor und einer to-String-Methode.  
Erzeugen Sie zwei Kundenobjekte in einer Applikation, initialisieren Sie diese mit dem Konstruktor und lassen alle Werte über die to-String-Methode wieder ausgeben.  
Verändern Sie einige Werte (set-Methode) und lassen sie sich die veränderten Werte wieder anzeigen (get-Methode).
2. Benötigt wird eine Klasse **Artikel** mit den Attributen **artnr**(int), **artbez**(String), **bestand**(int) und **preis**(double).  
Schreiben Sie diese Klasse mit get- und set-Methoden, einem Konstruktor und einer to-String-Methode.  
Speichern Sie alle erzeugten Objekte in einem Array und lassen Sie alle Werte über die to-String-Methode wieder ausgeben. Berechnen Sie den Gesamtwert der Artikel mit einem Wiederholungskonstrukt

## 4. Realisierung einer Anwendung mit mehreren Klassen (Fallbeispiel 2)

In praktischen Anwendungsfällen der OOP hat man es normalerweise nicht nur mit einer einzigen Klasse zu tun, sondern mit mehreren Klassen zwischen denen Beziehungen bestehen.

In der folgenden Situation realisieren wir einen Anwendungsfall, der aus mehreren Klassen besteht. Wir wollen uns an die Vorgehensweise in der objektorientierten Softwareentwicklung halten und die Phasen **Analyse – Design – Codierung** durchlaufen.

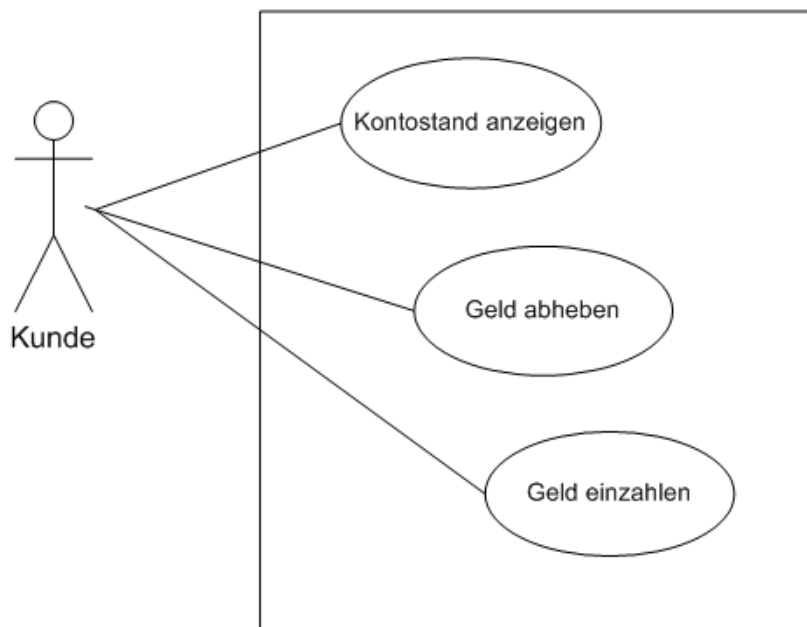
### 4.1 Beschreibung der Fallsituation

#### Fallsituation:

Eine Bank hat Kunden und Konten. Jeder Kunde hat genau ein Konto. Die Kunden wollen ihren Kontostand abfragen, Geld abheben und Geld einzahlen können.

### 4.2 Analysephase

Die zu realisierenden Anwendungsfälle sind: **Kontostand abfragen**, **Geld abheben** und **Geld einzahlen**. Wir stellen dies in einem **Use-case-Diagramm** dar:



### 4.3 Die Designphase

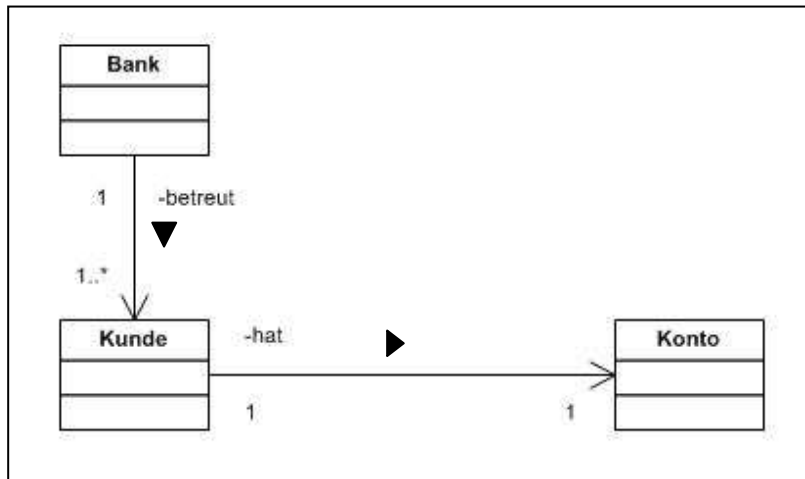
Anhand der kurzen Fallbeschreibung identifizieren wir folgende Klassen: **Bank**, **Kunde** und **Konto**. Wir stellen dies in einem **Klassendiagramm** dar:

Zwischen den genannten Klassen **Bank**, **Kunde** und **Konto** bestehen logische und mengenmäßige Beziehungen:

**Assoziationen:** Bezeichnung der Beziehung und der Beziehungsrichtung: z.B. Bank betreut Kunden oder Kunde hat Konto

**Kardinalität:** Mengenmäßige Beziehung zwischen Klassen: z.B. Eine Bank betreut eine unbekannte Anzahl (aber mindestens 1) Kunden.

Daraus ergibt sich folgendes Klassendiagramm:



Zur besseren Übersichtlichkeit wurden hier keine Attribute und Methoden in der Klasse dargestellt.

Die Pfeilspitze der Assoziationslinien gibt die **Richtung** der Assoziation an. In diesem Fall spricht man von einer **gerichteten** Assoziation. Wenn die Assoziationslinie keinen Pfeil hat, ist die Assoziation **ungerichtet**, d.h. die Assoziationsrichtung geht in beide Richtungen. Die Art der Assoziation wird durch den **Assoziationsnamen** ausgedrückt ( betreut, hat).

(In der korrekten UML-Syntax sollte die Leserichtung durch eine ausgefüllte Pfeilspitze hinter dem Assoziationsnamen dargestellt werden.)

Für die Klasse Kunde werden die beiden Attribute **kdnr** und **name** benötigt.

Für die Klasse Konto die Attribute **ktonr**, **ktoStand** und **pin**.

Die Methode **kontostandAbfragen** kann durch eine einfache get-Methode realisiert werden.

Bei den Methoden **einzahlen** und **auszahlen** wird jeweils der aktuelle Wert des Attributes **ktoStand** erhöht bzw. vermindert. Evtl. ist darauf zu achten, dass durch die Auszahlung ein bestimmtes Dispositionslimit nicht überschritten werden darf. Dieses wollen wir aber zunächst außer acht lassen.

## 4.4 Die Codierungsphase

Hier die Klassen **Kunde** und **Konto**. Zur Vereinfachung verzichten wir auf die Klasse Bank

```
public class Kunde {
    // Anfang Variablen

    private int kdnr;
    private String name;
    private Konto meinKonto; //Verweisattribut

    // Ende Variablen

    public Kunde(int k, String n)
    {
        this.kdnr=k;
        this.name=n;
    }

    // Anfang Methoden
    public void setKdnr(int k)
    {
        this.kdnr=k;
    }
    public int getKdnr() {
        return kdnr;
    }
    public void setName(String n)
    {
        this.name=n;
    }
    public String getName() {
        return name;
    }
    public Konto getMeinKonto() {
        return meinKonto;
    }
    public void setMeinKonto(Konto meinKonto) {
        this.meinKonto = meinKonto;
    }
    // Ende Methoden
}
```

```
public class Konto {

    // Anfang Variablen
    private int ktonr;
    private double ktoStand;

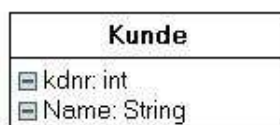
    public Konto(int k, double s)
    {
        this.ktonr=k;
        this.ktoStand=s;
    }
    public double getKtoStand() {
        return ktoStand;
    }

    public void setKtoStand(double ktoStand) {
        this.ktoStand = ktoStand;
    }

    public int getKtonr() {
        return ktonr;
    }

    public void setKtonr(int ktonr) {
        this.ktonr = ktonr;
    }
    public void auszahlen(double b)
    {
        this.ktoStand=this.ktoStand-b;
    }
    public void einzahlen(double b)
    {
        this.ktoStand=this.ktoStand+b;
    }
    // Ende Methoden
}
```

Im Klassendiagramm stehen diese beiden Klassen zunächst noch ohne Beziehung nebeneinander.



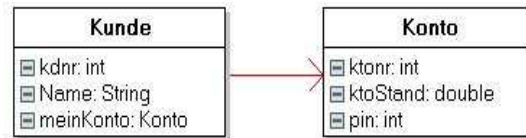
## 4.4.1 Assoziationen zwischen Klassen herstellen

Die Assoziation, dass ein Kunde ein Konto hat wird durch Einfügen eines **Verweisattributes** in der Klasse Kunde realisiert. Diese Attribut muss auf das dem Kunden zugehörige Kontoobjekt zeigen und muss daher den Datentyp **Konto** haben. Dieses Attribut kann folgendermaßen in der Klasse **Kunde** codiert werden:

```
private Konto meinKonto;
```

(Die Attributsbezeichnung meinKonto ist beliebig)

Jetzt wird die Assoziation im Klassendiagramm sichtbar.



In der Applikation wird das Attribut **meinKonto** erst dann initialisiert, wenn das dazu gehörige Kontoobjekt existiert.

```
public class bank
{
    public static void main(String argv[])
    {
        Kunde k1=new Kunde(123,"Meier");
        Konto kt1 = new Konto(4711,3500.00,111);
        k1.setMeinKonto(kt1);
    }
}
```

Wir verwenden hier einen Konstruktor, um die Objekte zu erzeugen. Die Anweisung **k1.setMeinKonto(kt1);** weist dem Kunden sein Konto zu und erzeugt damit die Assoziation. Für das Verweisattribut muss es eine get- und set-Methode geben.

## 4.4.2 Zugriff auf Klassenmethoden über das Verweisattribut:

Um die genannten Anwendungsfälle **kontoStandAbfrage**, **einzahlen** und **auszahlen** zu realisieren müssen wir folgende Überlegungen anstellen: Auslöser für einen dieser Anwendungsfälle ist jeweils der Kunde (siehe use-case-Diagramm). D.h. die Klasse Konto kann nicht selbst einzahlen oder auszahlen. Dies bedeutet, dass der Kunde über das Verweisattribut auf sein Konto zugreifen muss. Da das Verweisattribut gekapselt ist, geschieht dies über die get-Methode.

Mit dem Verweisattribut kann nun über den Kunden direkt auf sein Konto zugegriffen werden, ohne die Klasse Konto direkt anzusprechen. Wir wollen den Kontostand des Kunden Meier am Bildschirm anzeigen:

```
System.out.println("Kunde "+k1.getName()+" Kontostand: "+k1.getMeinKonto().getKtoStand());
```

Im zweiten Aufruf benötigen wir zwei Punkte, da wir dem Methodenaufruf zwei Objekte voranstellen:

- **k1** ist das Kundenobjekt
- **getMeinKonto()** ist die Methode, die das dem Kunden zugehörige Kontoobjekt aufruft.
- **getKtoStand()** ist eine Methode der Klasse Konto.

## 4.4.2 Die Applikation

Die drei Anwendungsfälle werden nachfolgend über eine kleine Menüstruktur, die über ein switch-case-Konstrukt gelöst wird realisiert. Selbstverständlich ließen sich auch die Ein- und Auszahlungen über eine Tastatureingabe komfortabler regeln:

```
import java.io.*;
public class bank
{
    public static void main(String argv[]) throws IOException
    {
        Kunde k1=new Kunde(123,"Meier");
        Konto kt1 = new Konto(4711,3500.00);
        k1.setMeinKonto(kt1);
        int eingabe;
        do
        {
            System.out.println("\n1 - Kontostand anzeigen");
            System.out.println("2 - Einzahlen");
            System.out.println("3 - Auszahlen");
            System.out.println("0 - Beenden\n");
            eingabe=Console_IO.IO_int("Eingabe: ");
            switch(eingabe)
            {
                case 1: System.out.println("Kunde "+k1.getName()+" Kontostand: "+k1.getMeinKonto().getKtoStand());
                        break;
                case 2: k1.getMeinKonto().einzahlen(150); break;
                case 3: k1.getMeinKonto().auszahlen(900.95); break;
                case 0: break;
            }
        }
        while(eingabe!=0);
    }
}
```

## 4.5 Erweiterung der Fallsituation - Vererbung

Wir verändern unser Fallbeispiel derart, dass wir jetzt zwischen Girokonten und Sparkonten unterscheiden wollen. Jeder Kunde kann ein **Girokonto** und ein **Sparkonto** haben.

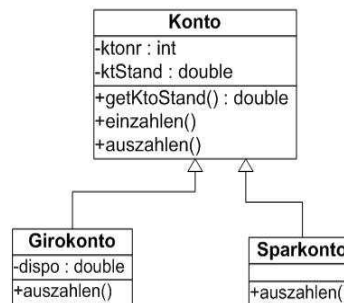
Girokonten und Sparkonten haben nun aber mehr Gemeinsamkeiten als Unterschiede. So haben beide eine Kontonummer und einen Kontostand. Der Unterschied könnte darin bestehen, dass das Girokonto einen Dispositionskredit aufweist und das Sparkonto eine höhere Guthabenverzinsung.

Hier bietet sich das Prinzip der **Vererbung** an. Bei der Vererbung erbt eine abgeleitete Klasse (Unterklasse) sämtliche Eigenschaften einer **Oberklasse** und kann zusätzlich eigene Merkmale aufweisen.

Vererbung wird in der UML durch eine nicht ausgefüllte Pfeilspitze dargestellt. Die folgende Abbildung zeigt die korrekte Darstellung für das Klassendiagramm:

Klassendiagramm in korrekter UML-Darstellung

Die Vererbungsbeziehung wird durch eine nicht ausgefüllte Pfeilspitze dargestellt, die von der abgeleiteten Klasse auf die Oberklasse zeigt



Innerhalb einer Klassenhierarchie kann es gleiche Methoden mit gleichem Aufbau geben. Die Methode **auszahlen** erscheint hier in allen drei Klassen. Es wird jeweils die Methode aufgerufen, die zu der Klasse gehört, dessen Objekt diese Methode aufruft.

Man bezeichnet, die Möglichkeit, gleiche Methodenbezeichnungen innerhalb einer Vererbungshierarchie zu verwenden als **Polymorphie**.

Methoden mit gleicher Bezeichnung können andere Methoden **überladen** oder **überschreiben**. Als **Überladen** bezeichnet man es, wenn Methoden gleichen Namens eine unterschiedliche Parameterliste haben. Der Compiler erkennt dann anhand der Parameter, welche Methode angesprochen werden soll. Als **Überschreiben** bezeichnet man es, wenn aus einer Klassenhierarchie ersichtlich ist, welche Methode angesprochen wird, obwohl sowohl Methodenbezeichnung als auch Parameterliste identisch sind.

Der Vorteil des Überschreibens von Methoden liegt darin, dass man keine unterschiedlichen Bezeichnungen für gleiche Funktionalitäten benötigt. Die Klasse **Girokonto** ist von der Basisklasse **Konto** abgeleitet. Beide Klassen haben eine Methode **auszahlen**, die aber in der Subklasse **Girokonto** eine andere Funktionalität besitzt, wie in der Klasse **Konto**.

### 4.5.1 Realisierung der Vererbungsbeziehung

Eine Vererbungsbeziehung wird im Kopf der Klasse durch das Schlüsselwort **extends** angezeigt.

```
public class Girokonto extends Konto {

    // Anfang Variablen
    private double dispo;
```



## 4.5.2 Zugriff auf Attribute der Basisklasse

Attribute der Basisklasse, die mit `private` deklariert wurden sind in den abgeleiteten Klassen nur über ihre `get`- und `set`-Methoden ansprechbar. Dieses ist innerhalb einer Klassenhierarchie nicht unbedingt sinnvoll, da es häufig vorkommt, dass Methoden der abgeleiteten Klassen Attribute der Basisklasse verwenden. Aus diesem Grund ist es sinnvoll, die Attribute in den Basisklassen mit dem Schlüsselwort **protected** zu deklarieren. Das Symbol dafür ist das hash-Zeichen (`#`). Damit sind die Attribute innerhalb der Klassenhierarchie verfügbar.

## 4.5.3 Konstruktoren und Vererbung

Ein Konstruktor einer Basisklasse kann nicht vererbt werden. Die abgeleitete Klasse muss einen eigenen Konstruktor implementieren. Attribute der Basisklasse, die im Konstruktor der abgeleiteten Klasse verwendet werden, müssen in der Basisklasse als **protected** deklariert sein.

Wenn die abgeleitete Klasse über einen eigenen Konstruktor verfügt, muss in der Basisklasse ein Standardkonstruktor vorhanden sein. Ohne diesen Standardkonstruktor kann kein Objekt der abgeleiteten Klasse erzeugt werden.

Hat die Basisklasse einen eigenen Konstruktor implementiert (wie im Quellcodeauszug unten), so wird dadurch der Standardkonstruktor überschrieben. Dieser muss dann explizit wieder definiert werden.

## 4.5.4 Arbeiten mit `super()`

Mit dem Aufruf von **`super()`** kann man von einer abgeleiteten Klasse aus auf eine überschriebene Methode der Basisklasse zugreifen. Dieses wird sehr häufig beim der Verwendung von Konstruktoren verwendet. Wenn ein neues Objekt einer abgeleiteten Klasse mit einem Konstruktor erzeugt werden soll, so muss der Konstruktor in der abgeleiteten Klasse komplett neu geschrieben werden. Da er aber meist so ähnlich aufgebaut ist, wie der Konstruktor der Basisklasse, wäre es sinnvoll, wenn der Konstruktor der Basisklasse mit verwendet werden könnte. Dies ist mit dem Aufruf von **`super()`** möglich.

### Basisklasse

```
public class Konto {  
  
    protected int ktnr;  
    protected double ktStand;  
  
    public Konto() {}  
  
    public Konto(int n, double b)  
    {  
        this.ktnr=n;  
        this.ktStand=b;  
    }  
}
```

### abgeleitete Klasse

```
public class Girokonto extends Konto {  
  
    private double dispo;  
  
    public Girokonto(int a, double b, double d)  
    {  
        super(a,b);        //Aufruf Konstruktor  
        this.dispo=d;  
    }  
}
```

## 4.6 Übungen

1. Eine Schule besteht aus Schülern und Lehrern. Fassen Sie wichtige Eigenschaften in einer Oberklasse Person zusammen und bilden Sie durch Vererbung zwei Unterklassen Schüler und Lehrer. Von den Schülern werden wieder zwei Klassen Vollzeitschüler und Teilzeitschüler abgeleitet. Zeichnen Sie ein Klassendiagramm
2. Eine Schule beschäftigt genau 25 Lehrer. Jeder Lehrer unterrichtet in bis zu 5 Klassen. In einer Klasse sind maximal 25 Schüler. Erstellen Sie ein Klassendiagramm und bilden Sie geeignete Assoziationen und Kardinalitäten für die genannten Klassen jeweils in zwei Richtungen.
4. Eine Automobilfirma hat zwei Werke. In jedem Werk wird ein bestimmter Fahrzeugtyp hergestellt. Für die Fahrzeuge werden die Attribute *Produktname*, *Preis* und *Verkaufsmenge* erfasst. Für die Werke werden die Attribute *Standort* und *Werkumsatz* sowie die Methode *Werkumsatz\_ermitteln* erfasst. Die Firma hat einen *Namen* und verfügt über die Methode *Gesamtumsatz\_ermitteln*.
  - a) Erstellen Sie ein Klassendiagramm mit den Assoziationen.
  - b) Der Firmeninhaber möchte den Gesamtumsatz ermitteln. Erstellen Sie ein Sequenzdiagramm für diese Anwendung.
5. Ein Zug besteht aus Waggons. Jeder Waggon hat genau sechs Abteile. In einem Abteil können bis zu sechs Fahrgäste sitzen.

Stellen Sie in einem Klassendiagramm die Assoziationen und die Kardinalitäten dar.

## 6. Kursverwaltung

Ein Fortbildungsinstitut möchte seine Software zur Kursverwaltung auf die objektorientierte Programmierung umstellen. Zu diesem Zweck soll zunächst folgender Sachverhalt als Klassendiagramm modelliert werden:

Für die Teilnehmer eines Kurses werden der Name, die Anschrift und der Status (beschäftigt, Schüler/Student bzw. arbeitslos) gespeichert. Jeder Teilnehmer kann sich für ein oder mehrere Kurse anmelden. Für jeden Kurs werden dessen Nummer, die Bezeichnung, das Datum sowie die Kursgebühr gespeichert. An einem Kurs können nicht mehr als 20 Teilnehmer teilnehmen. Jeder Kurs wird von einem Kursleiter angeboten. Ein Kursleiter kann mehrere Kurse anbieten. Für den Kursleiter werden Name und Firma gespeichert. Ein Teilnehmer kann nicht gleichzeitig Kursleiter sein. Jeder Teilnehmer hat genau ein Konto. Im Konto werden Kontonummer, und die bezahlte Kursgebühr gespeichert.

Erstellen Sie das Klassendiagramm mit den Klassenbezeichnungen, den Attributen und den Assoziationen. Get- und set-Methoden für die Attribute müssen nicht dargestellt werden.

## 5. Arrays und Collections als Container

### 5.1 Verweisattribute als Array

Wenn ein Kunde mehrere Konten haben kann, dann gibt es mehrere Verweisattribute, die in einem Array gespeichert werden können.

Die Syntax für das Anlegen dieses Arrays lautet für unser Beispiel:

```
public class Kunde
{
    private int kdnr;
    private String Name;
    private Konto meineKonten[ ];

    public Kunde(int k, String n)           //Konstruktor
    {
        this.kdnr=k;
        this.Name=n;
        meineKonten = new Konto[3];    //damit können Verweise auf 3 Konten erzeugt werden.
    }
    public void setMeineKonten(Konto k,int i)
    {
        this.meineKonten[i]=k;
    }
    public Konto getMeineKonten(int i)
    {
        return this.meineKonten[i];
    }
}
```

In der Applikation sieht das so aus:

```
public class bank
{
    public static void main(String argv[])
    {
        Kunde k1=new Kunde(123,"Meier");
        Konto kt1 = new Konto(4711,3500.00,111);
        Konto kt2 = new Konto(815,500.00,222);
        k1.setMeinKonto(kt1);
        k1.setMeineKonten(kt1,0);
        k2.setMeineKonten(kt2,1);
        //Ausgabe von Kontendaten über den Kunden
        System.out.println("Kunde "+k1.getName()+" Kontostand: "+k1.getMeineKonten(0).getKtoStand()); }}

```

Es erscheint folgende Ausgabe:

Kunde Meier Kontostand: 3500.00
---------------------------------

## 5.2 Collections am Beispiel ArrayList

Wenn wir größere Datenmengen des gleichen Typs benötigen, haben wir sie bisher in einem Array gespeichert. Auf diese Art konnten wir auch bequem mehrere Objekte einer Klasse verwalten. Der Nachteil der Arbeit mit Arrays ist nur, dass man schon bei der Erzeugung wissen muss, wie viele Elemente man benötigt.

Nehmen wir an, wir wollen eine Kundendatei erstellen und die Objekte der Klasse Kunden in einem Array speichern. Dazu müssen wir schon am Anfang wissen, wie viele Kunden wir haben. Wenn unser Unternehmen sich noch im Aufbau befindet, kann aber die Zahl der Kunden sehr schnell wachsen.

Zur Lösung dieses Problems bietet Java die so genannten **Collection-Klassen** an. Diese sind eine Sammlung verschiedener Klassen, die eine beliebige Anzahl von Objekten speichern können. Wir werden hier ein Beispiel für die Klasse **ArrayList** vorstellen.

Beispiel: Wir wollen eine ArrayList anlegen, in der wir beliebig viele Objekte der Klasse Kunden speichern können. Die Klasse Kunde besteht aus den Attributen kdnr (Kundennummer) und kdName (Kundenname). Neben den get- und set-Methoden für die Attribute gibt es eine toString-Methode für die Datenausgabe sowie einen Konstruktor.

Für die Verwendung der Klasse ArrayList müssen wir die Bibliothek **java.util** einbinden. Für die Arbeit mit der ArrayList stehen uns die folgenden Grundfunktionen zur Verfügung:

Anweisung	Bedeutung
<code>Import java.util.ArrayList;</code>	Importiert die Klasse ArrayList
<code>ArrayList kliste = new ArrayList( );</code>	Anlegen einer neuen ArrayList mit dem Namen kliste
<code>ArrayList&lt;Kunde&gt; kliste = new ArrayList&lt;Kunde&gt;( )</code>	Anlegen einer neuen ArrayList mit dem Namen kliste, die nur Daten vom Typ Kunde aufnehmen kann.
<code>Kunde k = new Kunde(100, "Meier"); kliste.add(k);</code>	Erzeugt ein Objekt der Klasse Kunde Fügt das Objekt der Liste zu.
<code>kliste.remove(2)</code>	Entfernt den dritten Eintrag des Arrays. Beachte, dass das Array ab dem Index 0 gezählt wird.
<code>kliste.get(1)</code>	Liest das Element an der Stelle 2
<code>kliste.size( )</code>	Liefert die Anzahl der Elemente des Arrays
<code>kliste.contains("Zeichenkette")</code>	Es wird geprüft, ob eine bestimmte Zeichenkette in der Liste enthalten ist.

Das folgende Beispiel zeigt die Verwendung einer ArrayList. Zunächst wird die Klasse Kunde dargestellt, anschließend werden in einer Applikation einige der obigen Anweisungen angewendet.

```
public class Kunde {  
  
    private int kdnr;  
    private String name;  
  
    public Kunde(int p,String n)  
    {  
        this.kdnr=p;  
        this.name=n;  
    }  
    public int getKdnr() {  
        return kdnr;  
    }  
  
    public void setKdnr(int kdn) {  
        this.kdnr = kdn;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String toString()  
    {  
        String ausgabe="" +this.kdsnr+" "+this.name;  
        return ausgabe;  
    }  
}
```

```
1  import java.util.ArrayList;

2  public class Kundenverwaltung {

3  public static void main(String[] args)
4  {
5      ArrayList<Kunde> kListe = new ArrayList<Kunde>(); //generische Liste nur für
                                                    //Kundenobjekte

6      Kunde k1 = new Kunde(100,"Meier");
7      kliste.add(k1);
8      Kunde k2 = new Kunde(101,"Berger");
9      kliste.add(k2);
10     System.out.println("Größe der Liste: "+kliste.size());
11     System.out.println("Eintrag an der Stelle 2: "+kliste.get(1));
12     System.out.println("Ein Attribut ausgeben: "+kliste.get(0).getName());
13     //Ein Attribut ändern
14     kliste.get(1).setName("Neu");
15     //Alle Werte mit for ausgeben
16     for(int k=0; k<kliste.size(); k++)
17     {
18         System.out.println(kliste.get(k));
19     }
20     //noch mehr Objekte mit for erzeugen und alle wieder ausgeben
21     for(int m=2; m<5; m++)
22     {
23         Kunde k = new Kunde(199,"Test");
24         kliste.add(k);
25     }
26     for(int k=0; k < kliste.size(); k++)
27     {
28         System.out.println(kliste.get(k));
29     }
30     // Objekt löschen und alles wieder ausgeben
31     kliste.remove(3);
32     System.out.println("Element 4 gelöscht");
33     for(int k=0; k<kliste.size(); k++)
34     {
35         System.out.println(kliste.get(k));
36     }
37 }
38 }
```

### Erläuterungen:

Zeile 1: Import für Java-Collections-Klasse ArrayList

Zeile 5: Erzeugen einer ArrayList mit dem Namen kliste, die Daten vom Typ Kunde aufnimmt

Zeile 7: Der Liste wird das Objekt k1 der Klasse Kunde hinzugefügt.

Zeile 10: Die Anzahl der Elemente, die die Liste enthält wird ausgegeben.

Zeile 11: Das Element mit dem Index 1 (Position 2) wird ausgelesen und mit der toString-Methode angezeigt.

Zeile 12: Ein bestimmtes Attribut des 1. Elementes der Liste wird angezeigt.

Zeile 31: Das Element mit dem Index 3 (Position 4) wird gelöscht.

## 5.2.1 Der Iterator

Der Iterator ermöglicht es, alle Elemente einer Liste kontrolliert zu durchlaufen.  
Der Iterator liefert folgende Methoden, die ebenfalls im Paket `java.util` verfügbar sind:

<b>hasNext( )</b>	liefert <code>true</code> , wenn noch mindestens ein Element in der Liste steht
<b>next( )</b>	liefert das jeweils nächste Element der Liste
<b>remove( )</b>	entfernt das zuletzt mit <code>next</code> angesprochene Element aus der Liste

In obigen Beispiel wird die Liste in den Zeilen 16 bis 19 mit einer `for`-Schleife abgearbeitet. Mit dem Iterator-Konzept ist der Zugriff auf eine Liste noch effizienter.  
Wir können obige `for`-Schleife durch folgende Anweisungen ersetzen:

```
Iterator it = kliste.iterator( );           //erzeugt einen Iterator mit der Bezeichnung it für die Liste
while(it.hasNext( ) )                      //so lange noch weitere Elemente vorkommen
{
    System.out.println(it.next( ) );        //Ausgabe des zuletzt aufgerufenen Objekts
}
```

Mit `it.remove( )` wird das zuletzt mit `it.next( )` aufgerufene Objekt gelöscht.

## 6. UML-Diagramme

Bei den Diagrammarten unterscheiden wir zwischen statischen Diagrammen, die den Zustand eines Systems beschreiben und dynamischen Diagrammen, die Abläufe in ihrer logischen und zeitlichen Reihenfolge darstellen.

### 6.1 Statische Sicht

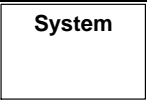



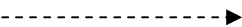
#### 6.1.1 Das Anwendungsfalldiagramm (Use- case - Diagram)

Das Anwendungsfalldiagramm ist ein *Verhaltensdiagramm*. Es zeigt eine bestimmte Sicht auf das *erwartete* Verhalten eines Systems und wird deshalb für die Spezifikation der Anforderungen an ein System eingesetzt. Anwendungsfalldiagramme beschreiben die Beziehungen zwischen einer Gruppe von Anwendungsfällen und den teilnehmenden Akteuren.

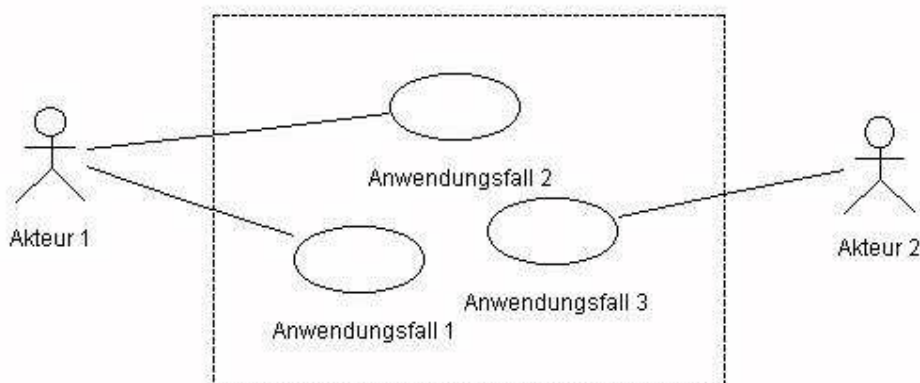
Dabei ist zu beachten, dass ein Anwendungsfalldiagramm nicht das Systemdesign widerspiegelt und damit keine Aussage über die Systeminterna trifft. Anwendungsfalldiagramme werden zur Vereinfachung der Kommunikation zwischen Entwickler und zukünftigen Nutzer bzw. Kunde erstellt. Sie sind vor allem bei der Festlegung der benötigten Kriterien des zukünftigen Systems hilfreich. Somit treffen Anwendungsfalldiagramme eine Aussage, *was* zu tun ist, aber nicht *wie* das erreicht wird.

Anwendungsfälle werden durch **Ellipsen** die den Namen des Anwendungsfalles tragen und einer Menge von beteiligten Objekten (**Akteuren**) dargestellt. Zu jedem Anwendungsfall gibt es eine Beschreibung in Textform. Die entsprechenden Anwendungsfälle und Akteure sind durch **Linien** miteinander verbunden. Akteure können durch Strichmännchen dargestellt werden. Die **Systemgrenze** wird durch einen Rahmen um die Anwendungsfälle symbolisiert. **Include-Beziehungen** bestehen zwischen Anwendungsfällen, die einen anderen Anwendungsfall beinhalten. Diese werden durch gestrichelte Linien dargestellt.

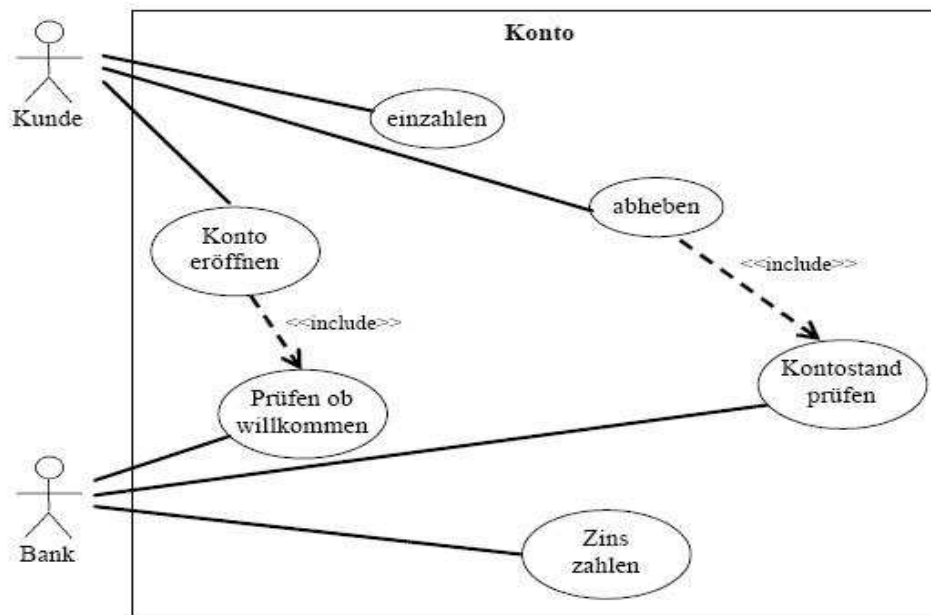
Das use case Diagramm beinhaltet folgende Elemente:

	Der Rahmen stellt das Anwendungssystem dar, innerhalb dessen der Anwendungsfall realisiert wird.
	Ein Anwendungsfall ist ein Teilproblem, welches innerhalb eines Anwendungssystems gelöst werden soll. Der Anwendungsfall wird immer von einem oder mehreren Akteuren oder anderen Anwendungsfällen ausgelöst
	Der Akteur kann eine Person oder ein anderes Anwendungssystem sein, welches einen Anwendungsfall auslöst
	Die Assoziation besteht verbindet Akteure und Anwendungsfälle
	include – oder extend-Beziehung. Eine include-Beziehung besteht zwischen einem Anwendungsfall, der einen anderen Anwendungsfall beinhaltet. Eine extends-Beziehung besteht, wenn eine Anwendungsfall durch einen anderen Anwendungsfall erweitert wird, wenn eine bestimmte Bedingung eintritt.

Allgemeines Beispiel

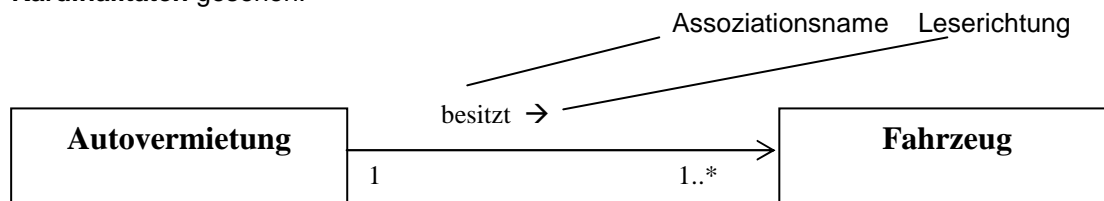


Beispiel für Anwendungsfälle in einer Bank



## 6.1.2 Klassendiagramm und Erweiterungen

Im Kapitel 4.3 haben wir schon den Aufbau eines Klassendiagramms mit **Assoziationen** und **Kardinalitäten** gesehen.



Kardinalität sagt aus, wie viele Objekte einer Klasse existieren können.

Kardinalität	Bedeutung
1	Genau 1
0..1	1 oder kein
*	Beliebig viele
1..*	1 bis beliebig viele
5..14	5 bis 14
6,7	6 oder 7

**Rollename:** Die Beziehungen zwischen zwei Klassen können unterschiedlich sein





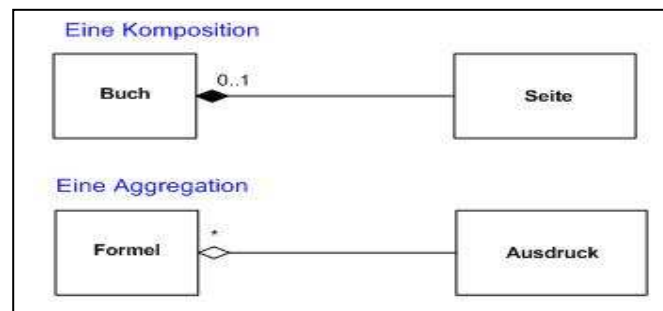
Neben der Vererbungsbeziehung, die durch die nicht ausgefüllten Pfeilspitzen dargestellt wird (siehe Kapitel 4.5), gibt es noch die Beziehung **Aggregation und Komposition**

Eine Aggregation beschreibt eine Ganzes – Teil Beziehung, wobei die Teile auch eigenständig existieren können.

Die Aggregation wird durch einen Pfeil mit einer nicht ausgefüllten Raute dargestellt.

Eine Komposition ist eine strenge Ganzes – Teil Beziehung. Die Lebensdauer des Teilobjektes richtet sich nach der Lebensdauer des Aggregationsobjektes

Die Komposition wird durch einen Pfeil mit einer ausgefüllten Raute dargestellt.



weitere Beispiele:

**Aggregation:** Kurs



Teilnehmer

Teilnehmer sind zwar Teil eines Kurses, können aber auch unabhängig vom Kurs existieren.

**Komposition:** Firma



Abteilung

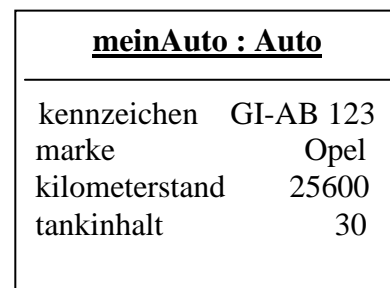
Die Lebensdauer der Abteilung ist abhängig von der Lebensdauer der Firma

## 6.1.3 Objektdiagramme

Das Objektdiagramm dient dazu, konkrete Klassenausprägungen (Objekte oder Instanzen) darzustellen. Dabei werden die Objektbezeichnung und die Attributwerte genannt. Methoden werden nicht dargestellt. Das Objektdiagramm dient eher dazu, anhand eines Objektes beispielhaft dessen Aufbau konkret darzustellen.

### Objektdiagramm

In der ersten Zeile steht die Objektbezeichnung gefolgt von der Klassenbezeichnung **fett** und **unterstrichen**. In der zweiten Zeile folgen die Attributbezeichnungen und die konkreten Ausprägungen der Attribute



## 6.2 Dynamischer Blick auf einen Anwendungsfall





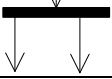
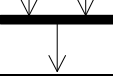
Use-case-Diagramme, Klassendiagramme und Objektdiagramme beschreiben den Zustand (**statische Sicht**) eines Systems. Der eigentliche Ablauf in zeitlicher und logischer Sicht ist nicht erkennbar.

Aus der strukturierten Programmierung kennen wir das **Struktogramm** als eine Darstellungsform, welche uns den logischen (und zeitlichen) Ablauf einer Anwendung vor Augen führt. Solche Darstellungen stellen die **dynamische Sicht** auf ein System dar.

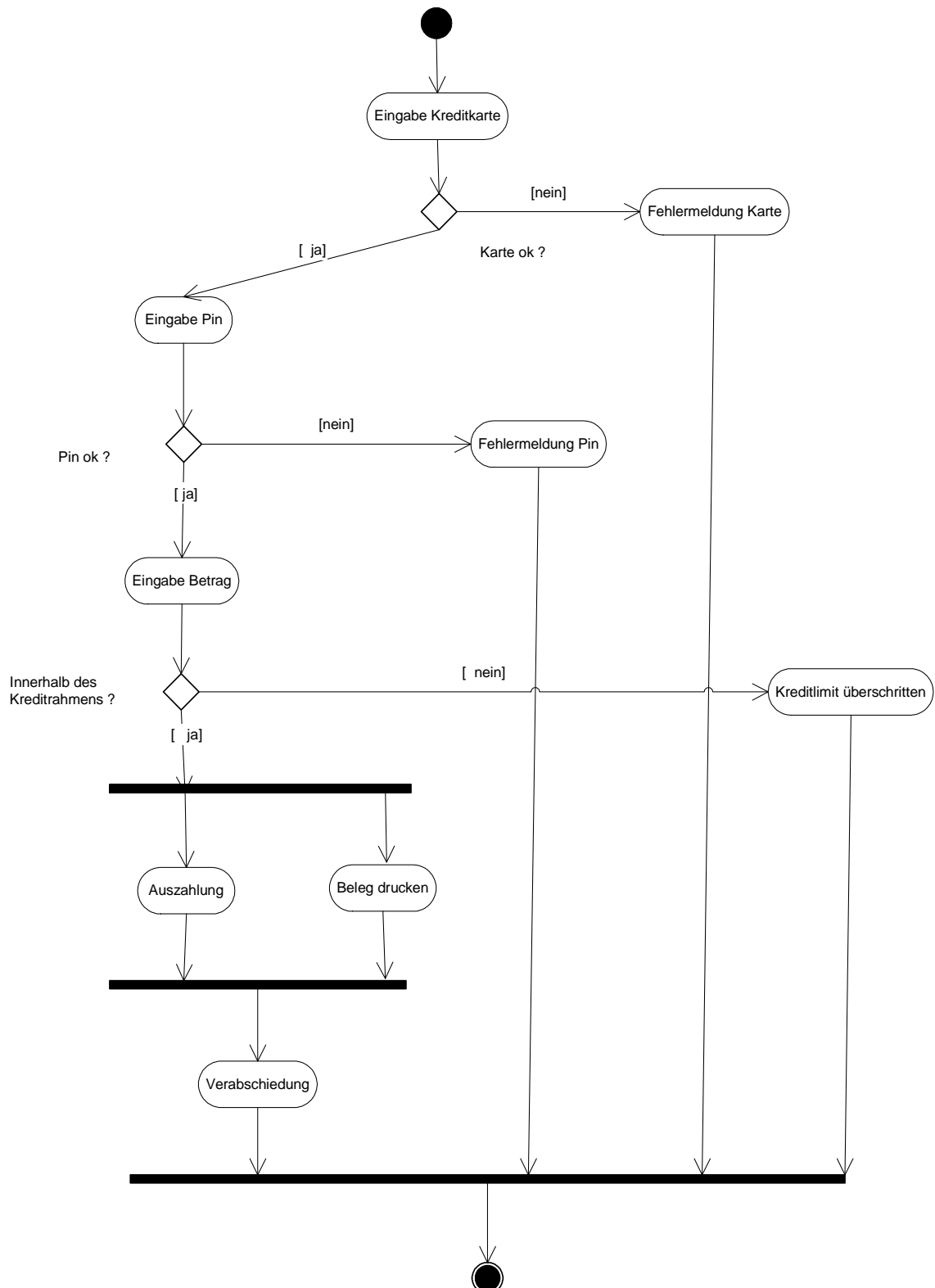
In der UML gibt es verschiedene Diagrammarten für diese Sichtweise. Hier sollen das **Aktivitätsdiagramm** und das **Sequenzdiagramm** vorgestellt werden.

### 6.2.1 Das Aktivitätsdiagramm

Das Aktivitätsdiagramm stellt den Ablauf des zu entwickelnden Programms dar. Es entspricht etwa der Programmdarstellung durch ein Struktogramm oder einen Programmablaufplan in der strukturierten Programmierung. Ein Aktivitätsdiagramm enthält sechs unterschiedliche Elemente:


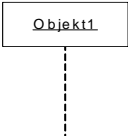
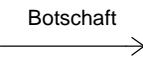
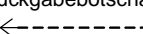

	Startpunkt einer Aktivität
	Endpunkt einer Aktivität
	Beschreibt die Aktion, die vom Programm auszuführen ist
	Entscheidung. In Abhängigkeit von einer Bedingung können verschiedene Wege beschritten werden
	Gabelung. Eine Aktivität gabelt sich in zwei weitere Aktivitäten auf.
	Zusammenführung. Zwei Aktivitäten werden zu einer Aktivität zusammengeführt.

Beispiel: Ein Bankkunde möchte an einem Geldautomat Geld abheben. Zunächst wird die Gültigkeit seiner Kreditkarte geprüft. Danach gibt er seine Pin ein. Ist die Pin gültig, kann er einen Betrag eingeben. Falls sein Kontostand innerhalb des Kreditrahmens liegt, wird der Betrag ausgezahlt und gleichzeitig ein Beleg gedruckt, andernfalls wird die Auszahlung verweigert.

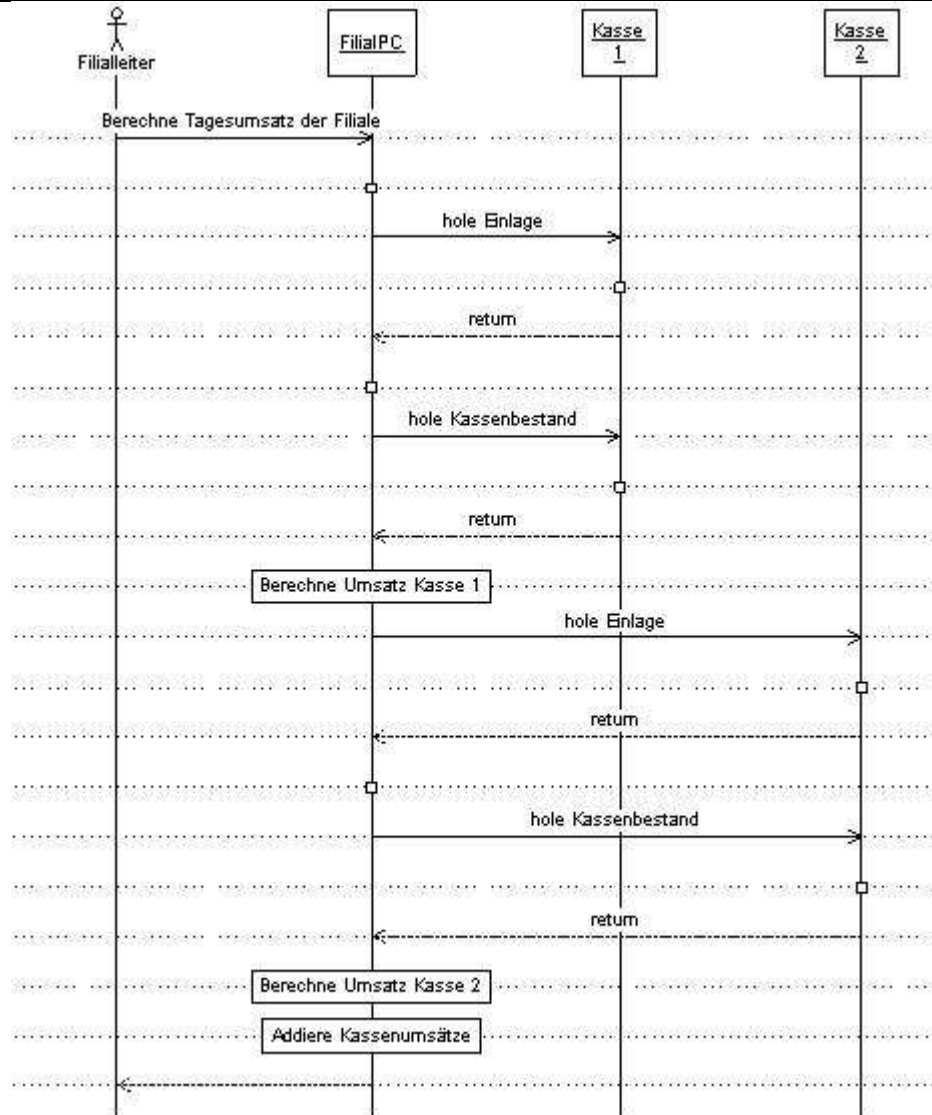


## 6.2.2 Das Sequenzdiagramm

Im Sequenzdiagramm wird der Nachrichtenaustausch (**Botschaften**) zwischen den Objekten in zeitlicher Reihenfolge dargestellt. In den Spalten werden die angesprochenen Objekte dargestellt, in den Zeilen die Reihenfolge der Aktivitäten, die sich hier als Nachricht darstellen. Rechtecke bezeichnen Rechenzeiten des Computers und können näher erläutert werden. Im Gegensatz zum Aktivitätsdiagramm bezieht das Sequenzdiagramm die beteiligten Objekte mit in das Diagramm ein. Ein Sequenzdiagramm besteht aus folgenden Grundelementen:

 Auslösendes Objekt	Das auslösende Objekt, der Akteur, der den Anwendungsfall auslöst
	Ein beteiligtes Objekt, welches eine Botschaft übermittelt oder eine Botschaft erhält. Die gestrichelte Linie ist die Lebenslinie des Objekts.
	Eine Botschaft, die mittels eines Methodenaufrufes übermittelt wird. Es kann die Methodenbezeichnung sowie die Übergabeparameter genannt werden.
	Der Antwort auf eine Botschaft. Die Antwort kann auch als Text auf der Linie stehen.
	Stellt den Zeitraum dar, der zwischen der Botschaft und der Rückmeldung liegt. Symbolisiert die Rechnerzeit in der die Daten verarbeitet werden. Innerhalb des Kästchens kann auch eine kurze Beschreibung des Vorgangs stehen.

Beispiel: Der Filialleiter eines Supermarktes ruft abends von seinem PC aus die Tagesumsätze der beiden Kassen ab und ermittelt den täglichen Filialumsatz. Das Objekt FilialPC gehört zur Klasse Filiale und besitzt die Methode **BerecheTagesumatzFiliale**. Diese Methode ruft die Methoden **getEinlage** und **getKassenbestand** bei den einzelnen Kassenobjekten auf, die jeweils die Einlage bzw. den Kassenbestand als Rückgabeparameter liefern. Die Methode **BerecheTagesumatzFiliale** berechnet die Kassenumsätze, addiert diese auf und liefert dann das Ergebnis zurück.



## Umsetzung des Sequenzdiagramms in Quellcode

Das Sequenzdiagramm zeigt dem Programmierer, welche Objekte beteiligt sind und welche Methoden in welcher Reihenfolge aufgerufen werden. Damit sollte er in der Lage sein, den Anwendungsfall zu programmieren. Es sind die Klassen **Filiale** und **Kasse** beteiligt. Die Klasse **Filiale** hat eine Methode **BerechneTagesumsatz**, die Klasse **Kasse** hat die Methoden **getEinlage** und **getKassenbestand**.

In der Applikation wird die Methode **BerechneTagesumsatz** für das Objekt **FilialPC** der Klasse **Filiale** aufgerufen:

```
public class filialeAbrechnen
{
    public static void main(String argv[])
    {
        ...
        System.out.println("Umsatz der Filiale: "+FilialPC.BerechneTagesumsatz( ));
    }
}
```

### Methode der Klasse Filiale:

```
public class filiale
{
    ..
    ..
    public double BerechneTagesumsatz( )
    {
        double e,k,fumsatz ;
        e=Kasse1.getEinlage( );
        k=Kasse2.getKassenbestand( );
        fumsatz=k-e;
        e=Kasse2.getEinlage( );
        k=Kasse2.getKassenbestand( );
        fumsatz=fumsatz+(k-e);
        return fumsatz;
    }
}
```

### Anmerkung:

Sinnvoller wäre es hier gewesen, in der Klasse Filiale Verweisattribute auf die zugehörigen Kassen anzulegen. Da eine Filiale mehrere Kassen haben kann, würde man hier ein Array des Datentyps kasse anlegen. (siehe Kapitel 5.3)

z.B.:

```
private kasse[ ] meineKassen = new kasse [10];
```

dann weiter in der Methode:

```
e=meineKassen[0].getEinlage( );
k=meineKassen[0].getKassenbestand( );
usw.
```

### Methoden der Klasse Kasse:

```
public class filiale
{
    private double einlage;
    private double kassenbestand ;
    public double getEinlage( )
    {
        return einlage;
    }
    public double getKassenbestand( )
    {
        return kassenbestand;
    }
}
```

## 6.3 Weitere Diagrammarten

Auf die folgenden der insgesamt 13 Diagrammarten wird in diesem Skript nicht näher eingegangen:

- Zustandsdiagramme
- Kollaborationsdiagramme (Kommunikationsdiagramm)
- Komponentendiagramme
- Verteilungsdiagramme
- Paketdiagramme

## 6.4 Übungen

1. Eine Autovermietung vermietet Fahrzeuge an Kunden. Eine Vermietung hat jeweils eine Buchung zur Folge.



Aufgabe: Zeichnen Sie Assoziationen und Kardinalitäten ein.

2. Eine Supermarktkette verwaltet von der Zentrale (Z1) aus 2 Filialen (F1 und F2). Die Filiale 1 hat 3 Kassen (k1, k2 und k3), die Filiale 2 hat 2 Kassen (k4, k5). Über den Zentrale sollen die Tagesumsätze aller Filialen ermittelt werden. Die Kassen speichern die Kassenbezeichnung, die morgendliche Bargeldeinlage und den aktuellen Kassenbestand. Die Filialen werden mit Name und Tageseinnahme erfasst.
- Erstellen Sie das Klassendiagramm
  - Erstellen Sie ein Sequenzdiagramm für die Ermittlung der Tagesumsätze durch die Zentrale.

3. Eine Studentin möchte ein Buch aus der Bibliothek ausleihen
- Sie gibt bei der Ausleihe Lesernummer, Autor und Titel des Buches an.
  - Die Ausleihe prüft über den Buchbestand, ob das Buch in der Bibliothek geführt wird.
  - Wenn das Buch existiert, wird im zweiten Schritt versucht, dieses auszuleihen.
  - Es wird jetzt geprüft, ob ein Exemplar vorhanden ist.
  - Sollte in Exemplar vorliegen, wird es mit der Lesernummer der Studentin ausgeliehen.
  - Die Studentin erhält von der Ausleihe das Buch.

- a) Erstellen Sie ein Klassendiagramm  
b) Erstellen Sie ein Aktivitätsdiagramm

4. In einem Unternehmen soll die Auftragsbearbeitung mit einem neuen Programm erledigt werden. In einem Klassendiagramm soll zunächst die Struktur des Systems modelliert werden. Die Grundlage dazu liefern folgende Informationen:

Wenn ein Kunde einen Auftrag erteilt, werden die einzelnen Auftragspositionen erfasst. Bei jeder Auftragsposition wird anhand der Produktdaten die Lieferfähigkeit überprüft. Nach Abschluss der Auftragserfassung wird eine Rechnung generiert.

Aufträge werden mit Auftragsnummer, Datum und Auftragssumme erfasst.  
Für Kunden werden die Kundennummer, Name und Anschrift gespeichert. Auftragspositionen beinhalten die Positionsnummer, Artikelnummer und Menge.  
Die Produktdaten speichern Artikelnummer, Artikelbezeichnung, Bestand und Preis.  
Rechnungen beinhalten die Rechnungsnummer und das Rechnungsdatum.

- a) Erstellen Sie zunächst das **Klassendiagramm**.  
b) Wie werden die **Assoziationen** zwischen den Klassen realisiert?  
c) Stellen Sie den Anwendungsfall „Auftragsstatistik erstellen“ in einem **Sequenzdiagramm** dar. Als Ergebnis der Auftragsstatistik soll täglich eine Liste mit den Aufträgen des Tages erstellt werden nach folgendem Muster:

Auftragsnummer	Kundennummer	Kundenname	Auftragssumme
080703001	2007	Meier KG	559,50
080703002	2003	Rinn und Keil	1300,00
080703003	2013	Loose GmbH	712,50
080703004	2003	Rinn und Keil	79,95



## 7. Das objektorientierte Vorgehensmodell

Für Anwendungssysteme, die objektorientiert gelöst werden und deren Entwicklungsphasen durch UML-Diagramme unterstützt werden sollen, bietet sich das **Phasenmodell** als Vorgehensweise an. Dabei können folgende Entwicklungsphasen unterschieden werden:

### 1. Problemstellung (Kundenauftrag)

### 2. Analyse

Analyse des Geschäftsprozesses Darstellung mit **EPK** und/oder **Anwendungsfalldiagramm**.

### 3. Design (Entwurf)

Entwurf der Klassen mit Attributen und Methoden  
dynamisches Verhalten der Klassen, Ablauflogik

**Klassendiagramm**  
**Sequenz- und**  
**Aktivitätsdiagramme**  
**auch Struktogramme u. a.**

Häufig wird das sog. Prototyping oder RAD (Rapid-Analyse-Design) benutzt, um dem Auftraggeber schon in einem frühen Stadium ein grobes Modell der Problemlösung vorzustellen. Anhand des Prototyps können evt. Abweichende Vorstellung noch rechtzeitig korrigiert werden bevor die Systementwicklung schon zu weit fortgeschritten ist.

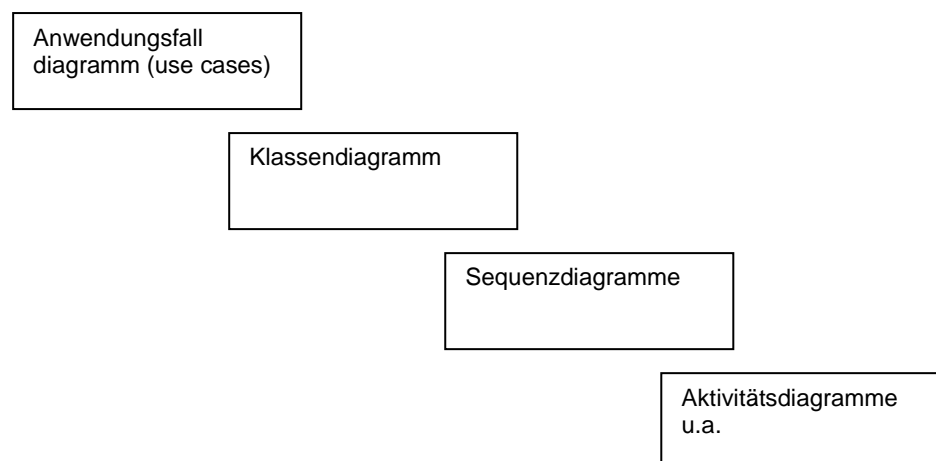
### 4. Codierung

### 5. Test

### 6. Implementierung

#### Modellierung mit UML

In den einzelnen Phasen des Vorgehensmodells können UML-Diagramme eingesetzt werden, wie sie im Skript dargestellt wurden. Zunächst wird das zu entwickelnde System durch eine Sammlung informeller Aussagen beschrieben. Dies kann durch verwendete Formulare ergänzt werden. Daraus können folgende Diagramme abgeleitet werden:



Eine detaillierter Darstellung der unterschiedlichen Vorgehensmodelle bei der Systementwicklung finden Sie in der Powerpoint-Präsentation **ae2007.ppt**.

## 8. Datenbankzugriff unter Java

Im folgenden Kapitel wird in Kürze dargestellt, wie man mit Java auf SQL-Datenbanken zugreifen kann. Einfache Anwendungen lassen sich unter DOS realisieren, komfortablere Verarbeitung sollte man mit einer GUI programmieren.

Wenn Java auf eine Datenbank zugreifen will, wird das Paket JDBC benötigt. Darin ist der notwendige JDBC-ODBC Treiber enthalten, mit dem man auf die meisten Datenbanken zugreifen kann. (Es gibt insgesamt 4 Klassen für Datenbanktreiber (Class 1-4) ). Der Treiber-Manager im JDBC regelt den Zugriff von Java auf die Datenbank. Die Art der Datenbank spielt dabei für den Java-Programmierer keine Rolle

Damit für die Kommunikation mit einer Access-Datenbank Access nicht mehr benötigt wird, muss die Datenbank im ODBC-Manager angemeldet werden.

### 1. ODBC-Treiber registrieren

Dazu aufrufen: **Systemsteuerung – Verwaltung – ODBC-Datenquellen**

**Benutzer-DSN – Hinzufügen** . Dann Microsoft-Access-Treiber auswählen.

Der Datenquelle einen beliebigen Namen geben, unter dem sie später in Java angesprochen wird. Z.B. **mydb**. Dann **Auswählen** und die entsprechende Datenbank auswählen. Alles mit OK bestätigen.

### 2. Das Java-Programm schreiben:

```
import java.sql.*;
```

**Laden des Datenbanktreibers:**

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Exception mit try und catch abfangen

### 3. Verbindung zur Datenbank herstellen.

Dazu wir ein Object der Klasse Connection benötigt.

```
Connection conn;
```

```
conn=DriverManager.getConnection("jdbc:odbc:mydb");
```

allgemein: DriverManager.getConnection(url, user, password)

url ist immer wie folgt aufgebaut: jdbc:odbc:Datenquelle

### 4. Eine SQL-Anfrage absetzen

Um Anfragen zu stellen, muss ein Objekt der Klasse **Statement** erzeugt werden.

```
Statement stm = conn.createStatement();
```

Das Ergebnis der Anfrage wird einem Objekt des Interface ResultSet zugewiesen:

Die wichtigsten Methoden des Statement-Objekts sind die Methoden **executeQuery** und **execute Update**. Die Methode executeQuery gibt ein Objekt vom Typ **ResultSet** zurück.

```
ResultSet rs=stm.executeQuery("Select * from Artikel"); oder
```

```
ResultSet rs=stm.executeQuery("Select Artnr, Artbez from Artikel WHERE Artnr >100");
```

Artikel sei hier eine Tabelle aus der Datenbank, die unter mydb angesprochen wurde.

## 5. Das Ergebnis der SQL-Anfrage auswerten.

Die Ergebnistabelle wurde in dem resultSet-Objekt **rs** gespeichert und kann nun ausgewertet werden.

Zum Auswerten kann der Datenbankzeiger verwendet werden. Es gibt:

rs.next( );      nächster Datensatz  
rs.previous( )    vorheriger Datensatz  
rs.first( )       erster Datensatz  
**rs.last( )**       letzter Datensatz

**Ausgabe:** Gesamte Tabelle mit der **getString-Methode**:

```
while(rs.next( ) )  
    System.out.println(rs.getString(Artnr)+" "+rs.getString(Artbez));
```

Parameter der getString-Methode ist entweder die Spaltenüberschrift oder der Spaltenindex, z.B. (rs.getString(1)).

Bei der get-String-Methode wird das Ergebnis in einen String umgewandelt:

Für andere Datentypen gibt es andere Methoden wie:

getInt( ); getDouble( ), getDate( ), getBoolean( ) usw.

## 6. Update einer Datenbank mit der Methode executeUpdate

```
int us = stm.executeUpdate("Update arikel set Preis=59.50 where Artnr=104")
```

Diese Anweisung gibt kein Ergebnis zurück, welches direkt abgefragt werden kann.

### Beispiel:

```
import java.sql.*; //SQL-Treiber laden  
public class jdbc100  
{  
    public static void main(String[] args)  
    {  
        try {  
            //Installation des odbc-Treibers  
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
            System.out.println("Treiber geladen");  
        }  
        catch (ClassNotFoundException e)  
        {  
            System.out.println("Treiber nicht gefunden");  
        }  
        try {  
            Connection conn;  
            conn = DriverManager.getConnection("jdbc:odbc:mydb");  
            System.out.println("Datenbank gefunden");  
            //eine Abfrage auf die tabelle KUNDEN erzeugen  
            Statement stm=conn.createStatement();  
            ResultSet rs = stm.executeQuery("Select * From Artikel"); //eine Tabelle aus der Datenbank  
            while(rs.next())  
            {  
                System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));  
            }  
        }  
        catch (SQLException ex)  
        {  
            System.out.println("Fehler");  
        }  
    }  
}
```

## 9. Wiederholung - Grundbegriffe der Objektorientierten Programmierung

### **Generalisierung**

Die Kunst, gemeinsame Strukturen von Objekten zu erkennen und diese in so genannten Superklassen zu verallgemeinern

### **Vererbung**

Das Prinzip, nach dem sich Eigenschaften von der Superklasse automatisch auf die Subklassen übertragen

### **Objekt oder Instanz**

Ist die konkrete Ausprägung einer Klassendefinition

### **Methoden**

Die Fähigkeiten, die die Objekte einer Klasse besitzen, um z.B. die Attributsausprägungen der Klasse zu verändern.

### **Assoziation**

Bezeichnet die Beziehung, die zwischen zwei Klassen besteht.

### **Kapselung**

Das Verbinden von Variablen (Attributen) und Methoden in einem Objekt. Der interne Aufbau eines Objekts wird vor den Benutzern versteckt, um dessen Programme unabhängig vor Änderungen im internen Klassenaufbau zu machen und unbefugten Zugriff auf Attribute zu verhindern.

### **Polymorphie**

Ein Mechanismus, der das individuelle Gestalten von Methoden innerhalb einer Klassenhierarchie erlaubt, wobei die Methoden den gleichen Namen tragen.

### **Klasse**

Darin werden Objekte des gleichen Typs zusammengefasst.

### **Komposition**

Aus mehreren Objekten wird eine neue Gesamtheit gebildet, wobei die Objekte unabhängig voneinander betrachtet keine sinnvolle Funktion haben.

### **Attribut**

Ein Merkmal, das zur Beschreibung einer Klasse gehört.

### **Aggregation**

Objekte werden miteinander zu einem sinnvollen Ganzen verbunden. Die einzelnen Objekte können aber auch unabhängig voneinander existieren.

### **Kardinalität**

Beschreibt die mengenmäßige Beziehung zwischen zwei Klassen

### **Botschaft**

Eine Nachricht, die einer Methode übergeben wird oder die eine Methode zurückliefert.

### **Konstruktor**

Bereits beim Erzeugen eines neuen Objekts werden Attribute initialisiert.

## Anhang

### Software

- Java-Hamster-Modell: [www.java-hamster-modell.de](http://www.java-hamster-modell.de)
- Javaeditor von G. Röhner <http://lernen.bildung.hessen.de/informatik/javaeditor/>
- NetBeans Java Entwicklung <http://www.netbeans.org/>
- Java-Development Kit <http://java.sun.com/javase>
- Struktogrammeditor [www.strukted.de](http://www.strukted.de)
- Microsoft Visio Für alle Arten von Diagrammen. Testversion bei <http://office.microsoft.com/de-de/visio/>
- Mind Maps erstellen <http://schule.bildung.hessen.de/sponsor/mind-manager/hinweise>

### Literatur

Deck, Neuendorf: Java-Grundkurs für Wirtschaftsinformatiker, Vieweg 2007  
Rau, Karl-Heinz, Objektorientierte Systementwicklung, Vieweg 2007

### Links

Auf meiner Seite **[www.pellatz.de](http://www.pellatz.de)** gibt es dieses Skript, aktuelle Informationen zum Stand des Unterrichts sowie weitere nützliche Links.