

# **Programmieren mit Java**

Eine Einführung für Anfänger ohne Vorkenntnisse

**Hubert Partl**  
**Zentraler Informatikdienst ZID**  
**Universität für Bodenkultur Wien**  
**Version Jan. 2007**

# Inhalt

	Seite
<b>1 Grundlagen</b> .....	<b>3</b>
1.1 Programmieren .....	3
1.2 Java 6	
<b>2 Sprachelemente</b> .....	<b>9</b>
2.1 Namen .....	9
2.2 Daten 9	
2.3 Anweisungen .....	14
2.4 Übungen .....	19
2.5 Fehlersuche und Fehlerbehebung .....	19
2.6 Programmierung - Konzepte .....	23
<b>3 Objekt-orientierte Programmierung</b> .....	<b>24</b>
3.1 Objekte und Klassen.....	24
3.2 Vererbung .....	30
3.3 Objekt-orientierte Analyse und Design .....	32
3.4 Kapselung.....	34
<b>4 Die Java Klassenbibliothek</b> .....	<b>38</b>
4.1 Überblick über die wichtigsten Pakete .....	38
4.2 Graphische User-Interfaces (GUI) .....	38
4.3 Datenbanken .....	49
<b>5 Weitere Informationen</b> .....	<b>64</b>
5.1 Online-Dokumentation .....	64
5.2 Dokumentation der eigenen Programme .....	65
5.3 Web-Sites und Bücher über Java .....	66
<b>6 Musterlösungen zu den Übungen</b> .....	<b>67</b>
6.1 Musterlösung EinfRech.....	68
6.2 Musterlösung EinfVergl.....	69
6.3 Musterlösung Quadrat .....	70
6.4 Musterlösung einfache Person .....	71
6.5 Musterlösung einfacher Student .....	72
6.6 Musterlösung Analyse und Design Konto .....	73

6.7	Musterlösung einfaches Konto.....	74
6.8	Musterlösung Person als Java Bean .....	76
6.9	Musterlösung Student als Java Bean .....	78
6.10	Musterlösung Konto als Java Bean .....	80
6.11	Musterlösung Geburtsjahr.....	82
6.12	Musterlösung SuchFrame.....	83

## Copyright

Sowohl die gedruckte als auch die über das Internet unter der Adresse

- <http://www.boku.ac.at/javaeinf/>

verfügbare Version dieser Kursunterlage und der Übungsbeispiele sind urheberrechtlich geschützt.

Die nicht kommerzielle Nutzung durch Universitäten, Schulen und Privatpersonen für den eigenen Gebrauch ist kostenlos erlaubt, wenn der Inhalt (einschließlich Autoren- und Copyright-Angabe) und die Form unverändert bleiben.

Bearbeitungen (einschließlich Kürzungen, Ergänzungen, Übersetzungen) sowie alle Arten von kommerziellen Nutzungen (z.B. in Schulungen oder Beratungen) oder Weitergaben (z.B. auf Papier, in elektronischer Form, auf CD-ROM oder anderen Datenträgern) sind **nur** nach Rücksprache mit dem Autor erlaubt:

- [partl@boku.ac.at](mailto:partl@boku.ac.at)

Der Autor kann keine Gewähr für die Aktualität und Richtigkeit der Dokumentation und der Beispiele übernehmen.

---

# 1 Grundlagen

## 1.1 Programmieren

### 1.1.1 *Wie sage ich dem Computer, was er tun soll?*

Beispiel: Ein Computerprogramm soll Umfang und Fläche eines Kreises berechnen, also z.B. die folgende Ausgabe auf den Bildschirm schreiben:

```
Radius   =    5.00000
Umfang   =   31.41593
Flaeche  =   78.53982
```

Ein Programm, das diese Ausgabe bewirkt, könnte etwa so aussehen (in Fortran, das ist eine Sprache, die sich besonders für mathematische Formeln eignet):

```
program kreis
pi = 3.14159265
r = 5.0
umfang = 2 * r * pi
flaeche = r * r * pi
write (*,1) 'Radius ', r
write (*,1) 'Umfang ', umfang
write (*,1) 'Flaeche', flaeche
1  format (a7, ' = ', f10.5)
end
```

Die Details werden wir später lernen, aber hier sehen wir schon ein paar typische Eigenschaften:

- Das Programm enthält englische Wörter, die eine bestimmte Bedeutung haben: program, write, format, end
- Das Programm enthält Sonderzeichen, die eine bestimmte Bedeutung haben: = \* ( ) , ' und Zeilenwechsel

- 
- Das Programm enthält Namen, die der Programmierer selbst wählen kann: Kreis, pi, r, Umfang, Flaeche
  - Das Programm enthält Zahlenangaben wie z.B. 3.14 und 5.0 und mathematische Formeln wie z.B.  $2 * r * pi$ .
  - Das Programm führt zuerst ein paar Berechnungen aus und schreibt dann die Ergebnisse auf den Bildschirm.

Ähnlich ist es auch bei Programmen in den anderen Programmiersprachen, von Cobol bis Java.

### **1.1.2 Programmiersprachen**

Welche Programmiersprachen gibt es, und warum?

- commercial & business-oriented language: COBOL, PL/1, RPG, SQL
- formula translator: FORTRAN, Basic
- algorithmic language: Algol, Pascal, Ada, C
- objekt-orientiert: Modula-2, Simula, Delphi, Smalltalk, Eiffel, C++, VisualBasic, Java
- andere: APL, Forth, Snobol, Logo, LISP, Prolog, Perl, PHP, JavaScript, ...

### **1.1.3 Programm-Elemente**

Hier sehen Sie zunächst eine kurze Zusammenstellung der wichtigsten Elemente, die in Computer-Programmen vorkommen, und der dafür üblichen Fachausdrücke. In den folgenden Abschnitten lernen Sie dann, wie Sie diese Elemente in der Programmiersprache Java formulieren können.

- Variable = Datenfelder
- Konstante = Datenwerte
- Dateien (Files) = Ein- und Ausgabe von Daten
  
- Anweisungen (Statements) = Aktionen
  - eine nach der anderen ;
  - Abzweigungen (if, else)
  - Schleifen (for, while)

- 
- strukturierte Programmierung:
    - Blöcke von Statements { }
    - Unterprogramme (Methoden)
    - Hauptprogramm (main)

Die Anweisungen werden in einer bestimmten Reihenfolge ausgeführt, dabei gibt es auch Abzweigungen (Entscheidungen, z.B. mit if) und Wiederholungen (Schleifen, z.B. mit for oder mit while).

Anweisungen, die gemeinsam ausgeführt werden sollen, werden zu sogenannten Blöcken von Anweisungen zusammengefasst, die wiederum geschachtelt werden können.

Das gesamte Programm besteht meistens aus mehreren Einheiten, die als Unterprogramme (Subroutinen, Prozeduren, Funktionen, Methoden) bezeichnet werden. Die Einheit, mit der die Verarbeitung beginnt, wird als Hauptprogramm (main-Methode) bezeichnet.

### **1.1.4 Beispiele: Einfaches HelloWorld-Programm**

Es ist üblich, einen ersten Eindruck für eine neue Programmiersprache zu geben, indem man ein extrem einfaches Programm zeigt, das den freundlichen Text "Hello World!" auf die Standard-Ausgabe schreibt.

#### **Java:**

```
public class HelloWorld {  
    public static void main (String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Dieses Java-Source-Programm muss in einem File mit dem Namen HelloWorld.java liegen. Die Übersetzung und Ausführung erfolgt dann mit

```
javac HelloWorld.java  
java HelloWorld
```

Ein als Muster für objekt-orientierte Java-Programme besser geeignetes Beispiel für eine HelloWorld-Applikation finden Sie im Kapitel über Objekte und Klassen.

---

## 1.2 Java

### 1.2.1 Insel, Kaffee oder Programmiersprache

Java wurde von der Firma Sun entwickelt und 1995 als neue, objekt-orientierte, einfache und plattformunabhängige Programmiersprache vorgestellt. Sun besitzt das Schutzrecht auf den Namen Java und stellt damit sicher, daß nur "100 % richtiges Java" diesen Namen tragen darf. Die Sprache ist aber für alle Computersysteme verfügbar (im Allgemeinen kostenlos).

Java geht auf die Sprache Oak zurück, die 1991 von Bill Joy, James Gosling und Mike Sheridan im Green-Projekt entwickelt wurde, mit dem Ziel, eine einfache und plattformunabhängige Programmiersprache zu schaffen, mit der nicht nur normale Computer wie Unix-Workstations, PCs und Apple programmiert werden können, sondern auch die in Haushalts- oder Industrieräten eingebauten Micro-Computer, wie z.B. in Waschmaschinen und Videorekordern, Autos und Verkehrsampeln, Kreditkarten und Sicherheitssystemen und vor allem auch in TV-Settop-Boxes für "intelligente" Fernsehapparate.

Allgemein anerkannt wurde Java aber erst seit 1996 in Verbindung mit Web-Browsern und Internet-Anwendungen. Inzwischen wird Java aber weniger für Applets in Web-Pages, sondern mehr für selbständige Anwendungs-Programme sowie für Server-Applikationen verwendet.

Der Name wurde nicht direkt von der indonesischen Insel Java übernommen sondern von einer bei amerikanischen Programmierern populären Bezeichnung für Kaffee.

### 1.2.2 Eigenschaften

Die wichtigsten Eigenschaften der Programmiersprache Java sind:

- plattformunabhängig
- Objekt-orientiert
- Syntax ähnlich wie bei C und C++
- umfangreiche Klassenbibliothek
- Sicherheit von Internet-Anwendungen

### 1.2.3 Das Java Development Kit (JDK)

Das Java Development Kit (JDK) umfasst die für die Erstellung und das Testen von Java-Applikationen und Applets notwendige Software, die Klassenbibliothek mit den zur Grundausstattung gehörenden Java-Klassen, und die Online-Dokumentation.

---

Zur Software gehören der Java-Compiler, das Java Runtime Environment (die Java Virtual Machine) für die Ausführung von Applikationen, der Appletviewer für die Ausführung von Applets, ein Java-Debugger und verschiedene Hilfsprogramme.

Die Online-Dokumentation umfasst eine Beschreibung aller Sprachelemente und aller Klassen der Klassenbibliothek (das so genannte "Application Programming Interface" API).

Das JDK für ein bestimmtes System erhält man meist kostenlos (z.B. zum Download über das Internet oder auf CD-ROM) vom jeweiligen Hersteller, also die Solaris-Version von Sun, die HP-Version von HP, die IBM-Version von IBM. Versionen für Windows-PC, Macintosh und Linux kann man von Sun oder IBM bekommen.

### 1.2.4 Vorgangsweise

Das Schreiben, Übersetzen und Ausführen eines Computer-Programmes erfolgt immer mit den folgenden drei Schritten:

1. Editor => Source-Programm  
`notepad "XXXXX.java"`
2. Compiler, Übersetzer => Binärprogramm, Bytecode  
`javac XXXXX.java`
3. Ausführung => Ergebnisse  
`java XXXXX`

### 1.2.5 Übung: HelloWorld-Programm

Schreiben Sie das oben angeführte HelloWorld-Programm (oder eine Variante davon mit einem anderen Text) in ein Java-Source-File und übersetzen Sie es und führen Sie es aus.

Vorgangsweise:

- schreiben und korrigieren:  
`notepad "HelloWorld.java"`
- übersetzen:  
`javac HelloWorld.java`
- ausführen:  
`java HelloWorld`

---

## 1.2.6 Typische Anfänger-Fehler

**Newbie:** Ich habe das HelloWorld-Programm aus meinem Java-Buch abgeschrieben, aber es funktioniert nicht. :-)

**Oldie:** Das ist schon richtig :-) so, das HelloWorld-Beispiel dient dazu, dass Du die typischen Anfänger-Fehler kennen lernst und in Zukunft vermeiden kannst. Der erste Fehler war schon: Wenn Du uns nicht den genauen Wortlaut der Fehlermeldung, die Version Deiner Java-Software (JDK, IDE) und die relevanten Teile Deines Source-Programms dazu sagst, können wir den Fehler nicht sehen und Dir nicht helfen. In diesem Fall kann ich nur raten. Du hast wahrscheinlich einen der folgenden typischen Newbie-Fehler gemacht:

- Du hast das Programm nicht genau genug abgeschrieben (Tippfehler, Groß-Kleinschreibung, Sonderzeichen, Leerstellen), lies doch die Fehlermeldungen und Korrekturhinweise, die der Compiler Dir gibt.
- Du hast das Programm nicht unter dem richtigen Filenamen abgespeichert. Wenn die Klasse HelloWorld heißt, muss das File HelloWorld.java heißen, nicht helloworld.java und auch nicht HelloWorld.java.txt, im letzteren Fall versuch es mit  
notepad "HelloWorld.java"
- Du hast beim Compiler nicht den kompletten Filenamen mit der Extension angegeben (wieder mit der richtigen Groß-Kleinschreibung):  
javac HelloWorld.java
- Du hast bei der Ausführung nicht den Klassennamen ohne die Extension angegeben (wieder mit der richtigen Groß-Kleinschreibung):  
java HelloWorld
- In der Umgebungsvariable PATH ist das Directory, in dem sich die JDK-Software befindet, nicht neben den anderen Software-Directories enthalten, versuch  
set PATH=%PATH%;C:\jdk1.2\bin  
oder wie immer das auf Deinem Rechner heißen muss.
- Die Umgebungsvariable CLASSPATH ist (auf einen falschen Wert) gesetzt. Diese Variable sollte überhaupt nicht gesetzt sein, nur in seltenen Spezialfällen und dann so, dass sie sowohl die Stellen enthält, wo die Java-Klassenbibliotheken liegen, als auch den Punkt für das jeweils aktuelle Directory.
- Du hast den Compiler nicht in dem Directory bzw. Folder aufgerufen, in dem Du das Java-File gespeichert hast.

---

## 2 Sprachelemente

### 2.1 Namen

#### 2.1.1 Namenskonventionen

Es wird dringend empfohlen, die folgenden Konventionen bei der Wahl von Namen strikt einzuhalten und ausführliche, gut verständliche Namen zu verwenden:

Namen von **Klassen** und Interfaces beginnen mit einem **Großbuchstaben** und bestehen aus Groß- und Kleinbuchstaben und Ziffern. Beispiel: HelloWorld, Button2.

Namen von **Datenfeldern**, **Methoden** etc. beginnen mit einem **Kleinbuchstaben** und bestehen aus Groß- und Kleinbuchstaben und Ziffern. Beispiele: openFileButton, button2, addActionListener, setSize, getSize.

Groß- und Kleinbuchstaben haben in jedem Fall verschiedene Bedeutung (case-sensitive).

#### 2.1.2 Reservierte Wörter

Die folgenden Namen sind reserviert und dürfen nicht neu deklariert werden:

abstract, boolean, break, byte, case, catch, char, class, continue, default, do, double, else, extends, false, final, finally, float, for, generic, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while.

### 2.2 Daten

#### 2.2.1 Datentypen

Die wichtigsten Datentypen sind:

- **int** (von integer = ganz) für ganze Zahlen
- **double** (von doppelter Genauigkeit) für Gleitkomma-Zahlen
- **boolean** (von Boole'scher Logik) für Entscheidungen, ja oder nein
- **String** (von Kette) für Zeichenketten, Texte

---

## 2.2.2 Konstante Werte

Normale ganze Zahlen haben den Typ `int` und werden dezimal interpretiert.

Normale Zahlen mit Dezimalpunkt (oder mit E für den Exponent) haben den Typ `double`.

Logische Konstanten sind die Wörter `true` und `false`.

String-Konstanten werden zwischen Double-Quotes geschrieben, Beispiel: `"Hubert Partl"`. Double-Quotes innerhalb des Strings müssen mit einem Backslash maskiert werden. Sonderzeichen können mit Backslash und u und dem Unicode-Wert angegeben werden, also z.B. `"\u20ac"` für das Euro-Zeichen.

```
int i;      i = -1234;
double x;   x = -123.45;   x = 1.0E23
boolean b;  b = true;     b = false;
String s;   s = "Abc";    s = null;
```

## 2.2.3 Deklarationen und Anfangswerte von Datenfeldern (Variablen)

Deklaration eines Datenfeldes:

```
typ name;
```

Zuweisung eines Wertes:

```
name = wert;
```

Deklaration mit Zuweisung eines Anfangswertes:

```
typ name = wert;
```

Achtung! Die Deklaration gilt immer nur für den Bereich (Klasse, Methode, Block innerhalb einer Methode), innerhalb dessen die Deklaration steht. Achten Sie darauf, dass Sie ein Datenfeld nicht irrtümlich zwei Mal deklarieren!

---

## 2.2.4 Deklarationen von Objekten oder Strings

Deklaration einer Variablen, die eine **Referenz** auf Objekte einer Klasse enthalten kann:

```
ClassName name;  
ClassName name = null;
```

Anlegen eines **Objekts** dieser Klasse und Zuweisung der Referenz auf dieses Objekt an die deklarierte Variable:

```
name = new ClassName();
```

wobei `ClassName()` für einen Konstruktor (constructor) der Klasse steht.

Deklaration, Anlegen und Zuweisen in einem:

```
ClassName name = new ClassName();
```

Deklaration und Anlegen von Strings:

```
String s;  
String s = null;  
s = "Hubert Partl";  
String myName = "Hubert Partl";
```

## 2.2.5 Konstruktoren

Konstruktoren (vom englischen Wort für Baumeister) dienen zum Anlegen von Objekten der Klasse mit dem `new`-Operator. Sie legen den für das Objekt benötigten Speicherplatz an und initialisieren das Objekt, d.h. sie setzen es in einen gültigen Anfangszustand. Mehr darüber finden Sie im Kapitel über Klassen und Objekte.

## 2.2.6 Garbage Collector

Objekte werden mit dem Operator `new` und dem Konstruktor **explizit** angelegt und dabei der Speicherplatz für das Objekt reserviert und mit den Anfangswerten belegt.

Sobald es keine Referenz mehr auf das Objekt gibt, wird der vom Objekt belegte Speicherplatz **automatisch** zurückgegeben und für neuerliche Verwendung frei gemacht. Der Teil des Java-Systems, der diese Arbeit erledigt, wird Garbage Collector genannt, das ist der englische Ausdruck für die Müllabfuhr.

---

## 2.2.7 Ausdrücke und Operatoren

Die Datenfelder können nicht nur konstante Werte zugewiesen bekommen, sondern es können auch mathematische Berechnungen oder logische Vergleiche durchgeführt werden.

### mathematische Operatoren

Addition:

```
c = a + b;
```

Subtraktion:

```
c = a - b;
```

Multiplikation:

```
c = a * b;
```

Division (double oder int):

```
c = a / b;
```

Divisionsrest (int):

```
crest = a % b;
```

Umgekehrtes Vorzeichen:

```
u = -x
```

### Zuweisungen

```
variable = ausdruck ;
```

Zuerst wird der Ausdruck auf der rechten Seite berechnet, dann wird das Ergebnis in der Variablen gespeichert, deren Name auf der linken Seite steht. Beispiele:

```
y = ( a + b ) * c;
```

```
i = i + 1;
```

---

Kurzschreibweisen:

`i++ ;`

ist eine Abkürzung für die Zuweisung

`i = i + 1 ;`

`x-- ;`

ist eine Abkürzung für die Zuweisung

`x = x - 1 ;`

## Logische Operatoren

Zahlenvergleiche:

`<` für kleiner als,

`>` für größer als,

`<=` für kleiner oder gleich,

`>=` für größer oder gleich,

`==` für gleich,

`!=` für ungleich

Logische Ausdrücke:

`!` für nicht (Gegenteil),

`&&` für und (beides muss wahr sein),

`||` für oder (mindestens eines muss wahr sein)

## Klammern

für kompliziertere Ausdrücke

Beispiele:

`y = ( a + b ) * c ;`

`if ( ( a >= 10 ) && ( a <= 99 ) )`

---

## Text-Operationen (String)

+ für das Zusammenfügen von Textteilen

Beispiel:

```
String name = vorName + " " + zuName;
```

## mathematische Funktionen

in der Klasse Math

Beispiel:

```
y = Math.sqrt(x);
```

## 2.3 Anweisungen

für die Aktionen im Programm

### 2.3.1 Statements und Blöcke

Statements werden mit ; (Strichpunkt, Semicolon) beendet.

```
Statement ;
```

Blöcke von Statements werden zwischen { und } (geschwungene Klammern, braces) eingeschlossen und können leer sein oder ein oder mehrere Statements enthalten, die jeweils mit ; beendet werden.

```
{  
    Statement ;  
    Statement ;  
}
```

---

## 2.3.2 Kommentare

Alles, was zwischen `/*` und `*/` oder zwischen `//` und dem Zeilenende steht, wird vom Java-Compiler ignoriert und gilt als Kommentar.

Beispiele:

```
... // Kommentar bis Zeilenende
```

```
... /* Kommentar */ ...
```

## 2.3.3 Beispiel für ein einfaches Programm

```
public class ProgExpr {  
  
    /* Beispiel fuer eine einfache Berechnung */  
  
    public static void main (String[] args) {  
        int guthaben = 1000;  
        System.out.println("Guthaben    = " + guthaben);  
        int einzahlung = 500;  
        System.out.println("Einzahlung = " + einzahlung);  
        guthaben = guthaben + einzahlung;  
        System.out.println("Guthaben    = " + guthaben);  
    }  
}
```

**Ausgabe:**

```
Guthaben    = 1000  
Einzahlung  = 500  
Guthaben    = 1500
```

---

## 2.3.4 Steuer-Strukturen

Entscheidungen (Abzweigungen) und Wiederholungen (Schleifen)

## 2.3.5 Entscheidungen mit *if* und *else*

einfache Abzweigung (Entscheidung der Form wenn - dann):

```
if ( logischer Ausdruck ) {  
    Statements;  
}
```

doppelte Verzweigung (wenn - dann - sonst):

```
if ( logischer Ausdruck ) {  
    Statements;  
}  
else {  
    Statements;  
}
```

**Beispiel:**

```
public class ProgIf {  
    public static void main (String[] args) {  
        int guthaben = 2000000;  
        System.out.println("Guthaben    = " + guthaben);  
        if ( guthaben >= 1000000 ) {  
            System.out.println("Du bist Millionaer!");  
        }  
    }  
}
```

**Ausgabe:**

```
Guthaben    = 2000000  
Du bist Millionaer!
```

---

## 2.3.6 Wiederholungen mit while

Wiederholung (Schleife) mit beliebiger Bedingung:

```
while ( logischer Ausdruck ) {  
    Statements;  
}
```

### Beispiel:

```
public class ProgWhile {  
    public static void main (String[] args) {  
        int guthaben = 1000;  
        int sparziel = 8000;  
        int einzahlung = 600;  
        System.out.println ("Guthaben    = " + guthaben);  
        System.out.println ("Sparziel    = " + sparziel);  
        while ( guthaben < sparziel ) {  
            guthaben = guthaben + einzahlung;  
            System.out.println ("neues Guthaben = " + guthaben);  
        }  
        System.out.println( "Sparziel erreicht.");  
    }  
}
```

### Ausgabe:

```
Guthaben    = 1000  
Sparziel    = 8000  
neues Guthaben = 1600  
neues Guthaben = 2200  
neues Guthaben = 2800  
neues Guthaben = 3400  
neues Guthaben = 4000  
neues Guthaben = 4600  
neues Guthaben = 5200  
neues Guthaben = 5800  
neues Guthaben = 6400  
neues Guthaben = 7000  
neues Guthaben = 7600  
neues Guthaben = 8200  
Sparziel erreicht.
```

---

## 2.3.7 Wiederholungen mit for

Wiederholung (Schleife) mit bestimmter Anzahl:

```
for (int name=Anfangswert ;
     logischer Ausdruck ;
     Wertänderung) {
    Statements;
}
```

**Beispiel:**

```
public class ProgFor {
    public static void main (String[] args) {
        int guthaben = 1000;
        int einzahlung = 500;
        System.out.println("Guthaben   = " + guthaben);
        System.out.println("Einzahlung = " + einzahlung
            + " pro Monat");
        for ( int monat=1; monat<=6; monat = monat + 1 ) {
            guthaben = guthaben + einzahlung;
            System.out.println(monat + ". Monat:");
            System.out.println("   Guthaben = " + guthaben);
        }
    }
}
```

**Ausgabe:**

```
Guthaben   = 1000
Einzahlung = 500 pro Monat
1. Monat:
   Guthaben = 1500
2. Monat:
   Guthaben = 2000
3. Monat:
   Guthaben = 2500
4. Monat:
   Guthaben = 3000
5. Monat:
   Guthaben = 3500
6. Monat:
   Guthaben = 4000
```

---

## 2.4 Übungen

### 2.4.1 eine einfache Rechnung

a = 6

b = 7

Wie viel ist a mal b?

### 2.4.2 ein einfacher Vergleich

a = 6

b = 7

Ist a größer als b?

### 2.4.3 Rechenbeispiel Quadratzahlen

Quadratzahlen berechnen und ausgeben:

1 \* 1 = 1

2 \* 2 = 4

3 \* 3 = 9

usw. bis

20 \* 20 = 400

## 2.5 Fehlersuche und Fehlerbehebung

**Computer** sind unglaublich dumme Geräte,  
die unglaublich intelligente Sachen können.  
**Programmierer** sind unglaublich intelligente Leute,  
die unglaublich dumme Sachen produzieren.

("Die Presse", 30.8.1999)

Programmfehler (auf englisch bugs = Wanzen, Ungeziefer genannt) gehören zum täglichen Brot jedes Programmierers. Nur wer nichts arbeitet, macht keine Fehler. Der gute

---

Programmierer ist nicht der, der keine Fehler macht, sondern der, der seine Fehler rasch findet und behebt.

Es gibt eigene Software-Tools, die bei der Fehlersuche helfen können. In den meisten Fällen genügt es aber, an allen wichtigen Stellen im Programm mit `System.out.println` oder `System.err.println` Informationen über das Erreichen dieser Programmstelle und über den Inhalt von wichtigen Variablen auszugeben. Damit kann man dann meistens sehr rasch finden, wo, wann und warum ein Programmfehler aufgetreten ist und was man tun muss, um ihn zu vermeiden.

### **2.5.1 Beispielskizze**

Ein Programm hat den folgenden Aufbau:

```
// Anfangswerte eingeben
anfangswert = ...
// erster Teil der Berechnungen
zwischenenergebnis = ...
// zweiter Teil der Berechnungen
ergebnis = ...
// Ausgabe des Ergebnisses:
System.out.println ("Ergebnis = " + ergebnis);
```

Sie führen das Programm aus und - das Ergebnis ist falsch.

### **2.5.2 Wo liegt der Fehler?**

Wo kann der Fehler liegen? Die Anfangswerte können falsch sein, im ersten Schritt der Berechnungen kann ein Fehler liegen, im zweiten Schritt der Berechnungen kann ein Fehler liegen.

### **2.5.3 Wie kann ich finden, wo der Fehler liegt?**

Das einfachste ist es, mit `System.out.println` zusätzliche Ausgaben in das Programm einzubauen:

```
System.out.println ("Programm beginnt.");
// Anfangswerte eingeben
anfangswert = ...
System.out.println ("Anfangswert = " + anfangswert);
```

---

```
System.out.println ("Erster Teil beginnt.");
// erster Teil der Berechnungen
zwischenenergebnis = ...
System.out.println ("Zwischenergebnis = " +
    zwischenenergebnis);
System.out.println ("Zweiter Teil beginnt.");
// zweiter Teil der Berechnungen
ergebnis = ...
// Ausgabe des Ergebnisses:
System.out.println ("Ergebnis = " + ergebnis);
```

Eventuell kann man auch andere Test-Anweisungen einbauen, zum Beispiel mit if-Statements überprüfen, ob gewisse Annahmen oder Einschränkungen (assertions, constraints) für die einzelnen Daten auch tatsächlich zutreffen.

Wenn Sie das so erweiterte Programm ausführen, sehen Sie Schritt für Schritt, was das Programm getan hat.

- Wenn schon der Anfangswert falsch ist, liegt der Fehler bei der Eingabe der Anfangswerte.
- Wenn der Anfangswert richtig, aber das Zwischenergebnis falsch ist, liegt der Fehler im ersten Teil der Berechnung.
- Wenn Anfangswert und Zwischenergebnis richtig sind, liegt der Fehler im zweiten Teil der Berechnung.

Den Teil, der den Fehler enthält, können Sie dann nach dem gleichen Prinzip noch genauer untersuchen, bis Sie sehen, welche Zahl falsch ist - und dann bemerken Sie meistens auch gleich, welchen Fehler Sie bei der Berechnung dieser Zahl gemacht haben und was Sie ändern müssen, damit Sie das richtige Ergebnis erhalten.

### ***2.5.4 Wie kann ich das Programm wieder sauber machen?***

Wenn der Fehler behoben ist, soll das Programm wieder nur das von Benutzer erwartete Endergebnis ausgeben und nicht die Zwischenschritte - die waren nur für die Fehlersuche durch den Programmierer gedacht, nicht für die Verwendung durch den Enduser. Im einfachsten Fall entfernen Sie also alle diese zusätzlichen Anweisungen wieder.

Aber irgendwann später wird das Programm verbessert, und dann brauchen Sie die Test-Anweisungen wieder, um auch dann wieder alle Fehler zu finden und zu korrigieren. Deshalb ist es besser, die Test-Anweisungen nicht zu entfernen, sondern mit Hilfe einer logischen Variablen und if-Statements ein- und auszuschalten:

```
static final boolean TEST = true;
...
if (TEST) {
    System.out.println ("Programm beginnt.");
}
```

---

```
}
// Anfangswerte eingeben
anfangswert = ...
if (TEST) {
    System.out.println ("Anfangswert = " + anfangswert);
    System.out.println ("Erster Teil beginnt.");
}
// erster Teil der Berechnungen
zwischenenergebnis = ...
if (TEST) {
    System.out.println ("Zwischenergebnis = " +
        zwischenenergebnis);
    System.out.println ("Zweiter Teil beginnt.");
}
// zweiter Teil der Berechnungen
ergebnis = ...
// Ausgabe des Ergebnisses:
System.out.println ("Ergebnis = " + ergebnis);
```

Jetzt müssen Sie nur den Wert der logischen Variablen TEST ändern:

- Wenn sie true ist, haben Sie die Test-Version des Programms, bei der die Test-Anweisungen zusätzlich ausgeführt werden.
- Wenn sie false ist, haben Sie die Produktionsversion des Programms, bei der die Test-Anweisungen nicht ausgeführt werden.

### **2.5.5 Was hilft noch?**

Außerdem ist es für das Verstehen und die (eventuell erst Jahre später erfolgende) Wartung der Programme wichtig, mit Kommentaren innerhalb der Programme möglichst ausführliche Hinweise auf den Zweck und die Funktionsweise des Programmes und der einzelnen Programmabschnitte sowie Erklärungen zu allen programmtechnischen Tricks anzugeben.

---

## 2.6 Programmierung - Konzepte

Beim Design eines komplexen Programmsystems können je nach Zweckmäßigkeit einige der folgenden Konzepte eingesetzt werden:

- **strukturierte Programmierung:**  
Komplexe Aufgaben werden in möglichst kleine, gut überschaubare Einheiten zerlegt.
- **objekt-orientierte Programmierung:**  
Die Daten und die mit ihnen ausgeführten Aktionen werden zusammengefasst.
- **top-down:**  
Zuerst überlegt man sich das Grundgerüst, dann die Details.
- **bottom-up:**  
Zuerst überlegt man sich die Einzelteile, dann erst, wie man sie zum Gesamtsystem zusammenfügt.
- **model-view-controller:**  
Bei Graphischen User-Interfaces muss man jeweils die folgenden drei Aspekte berücksichtigen:  
die Daten (Datenmodell, model),  
die Darstellung der Daten (Ansicht, view) und  
die Ablaufsteuerung (Änderung der Daten, controller)

Die entsprechenden Sprachelemente (Objekte, Klassen, Methoden etc.) werden wir in den folgenden Kapiteln kennen lernen.

---

## 3 Objekt-orientierte Programmierung

Bei der (älteren) prozeduralen Programmierung erfolgt die Definition der Datenstrukturen und der mit den Daten ausgeführten Prozeduren (Aktionen) unabhängig voneinander. Das Wissen um die Bedeutung der Daten und die zwischen ihnen bestehenden Beziehungen ist nicht einfach bei den Daten sondern mehrfach in allen Programmen, die auf diese Daten zugreifen, gespeichert.

Bei der (modernen) objekt-orientierten Programmierung (OOP) werden Objekte ganzheitlich beschrieben, d.h. die Festlegung der Datenstrukturen und der mit den Daten ausgeführten Aktionen erfolgt in einem.

Die wichtigsten Vorteile der objekt-orientierten Programmierung sind:

- Aufspaltung von komplexen Software-Systemen in kleine, einfache, in sich geschlossene Einzelteile,
- einfache und klar verständliche Schnittstellen zwischen den einzelnen Komponenten,
- weitgehende Vermeidung von Programmierfehlern beim Zusammenspiel zwischen den Komponenten,
- geringer Programmieraufwand durch die Wiederverwendung von Elementen (Reusability).

Je kleiner und einfacher die Objekte und Klassen und die Schnittstellen zwischen ihnen gewählt werden, desto besser werden diese Ziele erreicht. Mehr darüber folgt später in den Abschnitten über Kapselung und Vererbung sowie über Analyse und Design.

### 3.1 Objekte und Klassen

Als **Klasse** (class) bezeichnet man die Definition einer Idee, eines Konzepts, einer Art von Objekten.

Als **Objekt** (object), Exemplar oder Instanz (instance) bezeichnet man eine konkrete Ausprägung eines Objekts, also ein Stück aus der Menge der Objekte dieser Klasse.

Beispiele: Hubert Partl und Monika Kleiber sind Objekte der Klasse Mitarbeiter. Die Universität für Bodenkultur ist ein Objekt der Klasse Universität.

Je einfacher die Klassen gewählt werden, desto besser. Komplexe Objekte können aus einfacheren Objekten zusammengesetzt werden (z.B. ein Bücherregal enthält Bücher, ein Buch enthält Seiten, ein Atlas enthält Landkarten). Spezielle komplizierte Eigenschaften können auf grundlegende einfache Eigenschaften zurückgeführt werden (Vererbung, z.B. ein Atlas ist eine spezielle Art von Buch).

---

### 3.1.1 Definition von Klassen - Aufbau von Java-Programmen

Java-Programme sind grundsätzlich Definitionen von **Klassen**. Sie haben typisch den folgenden Aufbau:

```
public class ClassName {  
    // Definition von Datenfeldern  
    // Definition von Methoden  
}
```

In Spezialfällen kann eine Klasse auch weitere Elemente enthalten, z.B. die Definition von Konstruktoren.

### 3.1.2 Verwendung von Klassen - Anlegen von Objekten

**Objekte** werden in Java-Programmen mit dem Operator `new` und einem Konstruktor der Klasse angelegt. Beispiel:

```
House myNewHome = new House();
```

Innerhalb der Klasse kann man das aktuelle Objekt dieser Klasse mit dem symbolischen Namen `this` ansprechen.

### 3.1.3 Beispiel: objekt-orientierte HelloWorld-Applikation

```
public class HelloText {  
  
    public String messageText = "Hello World!";  
  
    public void printText() {  
        System.out.println (messageText);  
    }  
}
```

---

```
public static void main (String[] args) {
    HelloText h = new HelloText();
    h.printText();
}
}
```

Im Gegensatz zum statischen Hello-World-Programm lässt sich das objekt-orientierte Programm leicht auf die Verwendung von mehreren Objekten erweitern. Beispiel:

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

        HelloText engl = new HelloText();
        HelloText germ = new HelloText();
        germ.messageText = "Hallo, liebe Leute!";
        HelloText cat = new HelloText();
        cat.messageText = "Miau!";

        engl.printText();
        germ.printText();
        engl.printText();
        cat.printText();
        engl.printText();
    }
}
```

Hier werden drei Objekte der Klasse HelloText erzeugt, mit verschiedenen Texten, und dann erhalten diese Objekte die "Aufträge", ihren Text auszugeben ("Delegation"). Die Ausgabe sieht dann so aus:

```
Hello World!
Hallo, liebe Leute!
Hello World!
Miau!
Hello World!
```

Anmerkung: Eine weitere Verbesserung dieses HelloWorld-Programmes folgt im Kapitel Kapselung.

---

### 3.1.4 Datenfelder

Datenfelder der Klasse, also die Eigenschaften (Attribute) der Objekte, werden typisch in der folgenden Form deklariert:

```
public typ name;  
public typ name = anfangswert;
```

Nachdem ein Objekt dieser Klasse angelegt wurde, können seine Datenfelder in der Form

```
object.name
```

angesprochen werden. Wenn `private` statt `public` angegeben wurde, können sie jedoch von anderen Klassen nicht direkt angesprochen werden (siehe Kapselung).

Datenfelder der Klasse sind eine Art von "globalen Variablen" für alle Methoden der Klasse und für eventuelle innere Klassen.

Innerhalb der Klasse kann man die Datenfelder einfach mit

```
name
```

ansprechen, oder in der Form

```
this.name
```

### 3.1.5 Methoden

In den Methoden werden (wie in den Funktionen oder Prozeduren in konventionellen Programmiersprachen) die Aktionen definiert, die von oder an einem Objekt ausgeführt werden können. Die Deklaration erfolgt in der folgenden Form:

Methode mit Rückgabewert (Funktion):

```
public typ name () {  
    Statements;  
    return wert;  
}
```

Methode ohne Rückgabewert (Prozedur):

```
public void name () {  
    Statements;  
}
```

---

Methode mit Parametern:

```
public typ name (typ name, typ name, typ name) {  
    Statements;  
    return wert;  
}
```

Mit return; bzw. return wert; innerhalb der Statements kann man die Methode vorzeitig verlassen (beenden).

Nachdem ein Objekt der Klasse angelegt wurde, können seine Methoden in einer der folgenden Formen ausgeführt werden:

```
x = object.name();  
object.name();  
x = object.name (a, b, c);  
object.name (a, b, c);
```

Man spricht in diesem Fall vom Aufruf der Methode für das Objekt oder vom Senden einer Botschaft an das Objekt. Wenn private statt public angegeben wurde, kann die Methode von anderen Klassen nicht direkt angesprochen werden (siehe Kapselung).

Innerhalb der Klasse kann man die Methoden mit

```
name (parameterliste)
```

oder

```
this.name (parameterliste)
```

aufrufen.

Lokale Variable, die nur innerhalb einer Methode deklariert sind, können weder von den anderen Methoden dieser Klasse noch von anderen Klassen aus angesprochen werden. Bei der Deklaration von lokalen Variablen darf daher (im Gegensatz zu den Datenfeldern der Klasse) weder public noch private angegeben werden.

### **3.1.6 Statische Methoden**

Wenn man vor dem Typ einer Methode das Wort static angibt, dann kann diese Methode aufgerufen werden, ohne dass vorher ein Objekt der Klasse erzeugt werden muss. Der Aufruf erfolgt dann in der Form

```
ClassName.name();
```

Statische Methoden können freilich nicht auf Datenfelder oder nicht-statische Methoden der Klasse oder auf die Objektreferenz "this" zugreifen. Innerhalb von statischen Methoden können aber Objekte mit new angelegt und dann deren Datenfelder und Methoden angesprochen werden.

---

Die beiden wichtigsten Anwendungsfälle für statische Methoden sind

- die main-Methode, mit der die Ausführung einer Java-Applikation beginnt, und
- mathematische Funktionen, die nicht mit Objekten sondern mit primitiven Datentypen arbeiten.

### 3.1.7 Die main-Methode

Jede Java-Applikation muss eine statische Methode mit der folgenden Definition enthalten:

```
public static void main (String[] args)
```

Diese main-Methode wird beim Starten der Applikation mit dem Befehl java aufgerufen und kann ein oder mehrere Objekte der Klasse oder auch von anderen Klassen anlegen und verwenden. Der Parameter args enthält die Menge der beim Aufruf eventuell angegebenen Parameter.

### 3.1.8 Konstruktoren

Konstruktoren dienen zum Anlegen von Objekten der Klasse mit dem new-Operator. Sie legen den für das Objekt benötigten Speicherplatz an und initialisieren das Objekt, d.h. sie setzen es in seinen Anfangszustand, indem sie allen Datenfeldern die jeweiligen Anfangswerte zuweisen.

Laut *Tov Are Jacobsen* erfüllt der Konstruktor die Funktion eines Baumeisters, der ein Objekt mit den im Plan festgelegten Eigenschaften errichtet:

When you write a class you describe the behaviour of potential objects. Much like designing a house on paper: you can't live in it unless you construct it first. And to do that you say "I want a new house, so I'll call the constructor":

```
House myNewHome;  
myNewHome = new House();
```

(*Tov Are Jacobsen*, in der Newsgruppe comp.lang.java.help, 1998)

Im einfachsten Fall enthält die Definition der Klasse **keine** explizite Definition von Konstruktoren. Dann wird vom Java-Compiler ein Default-Konstruktor mit leerer Parameterliste erzeugt, der in der Form

```
ClassName object = new ClassName();
```

aufgerufen werden kann und nur den Speicherplatz anlegt und alle Datenfelder auf ihre Anfangswerte setzt.

---

### 3.1.9 Übung: Person

Schreiben Sie eine einfache Klasse "Person" mit den folgenden Datenfeldern:

- Vorname (Typ String)
- Zuname (Typ String)
- eventuell auch Geburtsjahr (Typ int) u.a.

und mit einer Methode

- `public String toString()`

die einen "schönen Text" liefert, nämlich den Vornamen, eine Leerstelle und den Zunamen.

Fügen Sie eine main-Methode an, in der Sie diese Klasse kurz testen, indem Sie ein paar Personen-Objekte anlegen und deren Eigenschaften setzen und auf den Bildschirm ausgeben.

Probieren Sie auch aus, was passiert, wenn Sie als Parameter von `println()` einfach nur das Objekt angeben, also z.B.

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

## 3.2 Vererbung

Wie kann ich ein spezielles Programm auf ein anderes, bereits existierendes, allgemeineres Programm zurückführen?

Wie kann ich erreichen, dass meine Programme möglichst allgemein brauchbar und wieder-verwertbar sind (re-usability)?

### 3.2.1 Ober- und Unterklassen

Eine der wichtigsten Eigenschaften von objekt-orientierten Systemen stellt die sogenannte "Vererbung" von Oberklassen (Superklassen) auf Unterklassen (Subklassen) dar. Dies stellt eine wesentliche Vereinfachung der Programmierung dar und ist immer dann möglich, wenn eine Beziehung der Form "A ist ein B" besteht.

Beispiele: Ein Auto ist ein Fahrzeug (hier ist Fahrzeug der Oberbegriff, also die Superklasse, und Auto ist die spezielle Subklasse). Ein Bilderbuch ist ein Buch. Ein Mitarbeiter ist eine Person...

Keine Vererbung ist in den folgenden Fällen gegeben: Ein Bücherregal enthält Bücher. Ein Bilderbuch enthält Bilder. Ein Mitarbeiter liest ein Buch.

Der Vorteil der Vererbung besteht darin, dass man die selben Konzepte nicht mehrfach programmieren muss, sondern auf einfache Grundbegriffe zurückführen und wieder verwenden kann.

Dazu wird zunächst die Oberklasse, die den allgemeinen Grundbegriff möglichst einfach beschreibt, mit den für alle Fälle gemeinsam geltenden Datenfeldern und Methoden definiert.

---

Dann gibt man bei der Definition der Unterklasse mit `extends` an, dass es sich um einen Spezialfall der Oberklasse handelt, und braucht jetzt nur die Datenfelder und Methoden zu definieren, die zusätzlich zu denen der Oberklasse notwendig sind. Alle anderen Definitionen werden automatisch von der Oberklasse übernommen.

Beispiel:

```
public class Person {
    public String name;
    public Date geburtsDatum;
    ...
}

public class Beamter extends Person {
    public int dienstKlasse;
    public Date eintrittsDatum;
    ...
}
```

Dann enthalten Objekte vom Typ `Person` die beiden Felder `name` und `geburtsDatum`, und Objekte vom Typ `Beamter` enthalten die vier Felder `name`, `geburtsDatum`, `dienstKlasse` und `eintrittsDatum`. Das analoge gilt für die hier nicht gezeigten Methoden.

Man kann auch Methoden, die bereits in der Oberklasse definiert sind, in der Unterklasse neu definieren und damit die alte Bedeutung für Objekte des neuen Typs überschreiben (`override`).

Referenzen auf ein Objekt der Unterklasse können sowohl in Variablen vom Typ der Unterklasse als auch vom Typ der Oberklasse abgespeichert werden. Das gilt auch bei der Verwendung in Parameterlisten. Erlaubt sind also z.B. die folgenden Zuweisungen:

```
Person mutter;
Person vater;
mutter = new Person();
vater = new Beamter();
```

Allerdings können in diesem Fall auch für das Objekt `vater` nur die Datenfelder und Methoden aufgerufen werden, die für Personen definiert sind, andernfalls erhält man einen Compiler-Fehler. Dies ist ja gerade der Zweck der Vererbung: Uns interessiert in diesem Programm nur, dass es sich um Personen handelt, und es ist völlig egal, ob es ein Beamter mit Dienstklasse oder ein Student mit Studienrichtungen oder ein Angestellter mit Arbeitgeber und Gehalt ist.

---

### 3.2.2 Übung: Person und Student

Wir haben bereits eine einfache Klasse "Person" mit den Datenfeldern vorname und zuname und einer Methode toString(), die einen String liefert, der den Vor- und Zunamen enthält.

Schreiben Sie nun eine einfache Klasse "Student" als Unterklasse von Person, mit dem zusätzlichen Datenfeld uni und mit einer entsprechend erweiterten Methode toString(), die einen String liefert, der den Vor - und Zunamen und die Universität enthält.

Erweitern Sie die in der Übung "Person" erstellte main-Methode so, dass darin nicht nur Personen-Objekte, sondern auch Studenten-Objekte angelegt werden, und deren Eigenschaften jeweils gesetzt und auf den Bildschirm ausgegeben werden. Probieren Sie aus, was bei Personen und bei Studenten passiert, wenn Sie als Parameter von println() nur das Objekt angeben.

## 3.3 Objekt-orientierte Analyse und Design

Frisch geplant ist halb gewonnen!

### 3.3.1 Vorgangsweise

Die Erstellung von objekt-orientierten Programmen besteht aus folgenden Schritten

1. **Objekt-orientierte Analyse (OOA)**  
= die Überlegung, aus welchen Objekten und Klassen eine Aufgabenstellung besteht, und welche Eigenschaften und Aktionen diese Objekte haben - vorerst noch unabhängig von der verwendeten Programmiersprache,
2. **Objekt-orientiertes Design (OOD)**  
das Konzept, wie das Ergebnis der Objekt-orientierten Analyse am besten in einer bestimmten Programmiersprache realisiert werden kann,
3. **Objekt-orientierte Programmierung (OOP)**  
das Schreiben der Programme, in denen die Eigenschaften und Aktionen der Objekte bzw. der Klassen von Objekten festgelegt werden.

Die Analyse erfolgt zunächst unabhängig von der verwendeten Programmiersprache und soll die logisch richtige Sicht des Problems und damit die Grundlage für das Design liefern.

Das Design, also das Konzept für die Programmierung, berücksichtigt dann die Eigenschaften der Programmiersprache (z.B. Einfach- oder Mehrfachvererbung) und die verfügbaren Klassenbibliotheken (z.B. Schnittstellen zu Graphischen User-Interfaces oder zu Datenbanken).

Um komplexe Systeme für die objekt-orientierte Programmierung in den Griff zu bekommen, müssen sie also analysiert werden, aus welchen Objekten sie bestehen und welche Beziehungen zwischen den Objekten bestehen, welche Eigenschaften die Objekte haben und welche Aktionen mit ihnen ablaufen.

---

Die wichtigsten Design-Regeln für das Zerlegen von komplexen Systemen in einzelne Objekte sind:

- Die Objekte sollen möglichst **einfach** sein, d.h. ein kompliziertes Objekt soll in mehrere einfache Objekte zerlegt werden.
- Die Objekte sollen möglichst **abgeschlossen** sein, d.h. jedes Objekt soll möglichst wenig über die anderen Objekte wissen müssen.

Die Analyse und das Design ergeben sich am einfachsten, indem man versucht, das System bzw. die Aufgabenstellung mit einfachen deutschen Sätzen zu beschreiben:

- Hauptwörter (Nomen) bedeuten Objekte oder primitive Datenfelder.
- Eigenschaftswörter (Adjektiv) bedeuten Datenfelder oder get-Methoden.
- Zeitwörter (Verb) bedeuten Methoden.
- Sätze der Form "A hat ein B" bedeuten Datenfelder (Eigenschaften).
- Sätze der Form "A ist ein B" bedeuten Oberklassen (Vererbung).

### **3.3.2 Übung: Analyse und Design Bankkonto**

Überlegen Sie die Analyse und das Design für ein einfaches Konto-System:

In welchen Beziehungen stehen die im folgenden in alphabetischer Reihenfolge angeführten Begriffe (Analyse in Form von einfachen deutschen Sätzen)?

Wie können sie durch Klassen, Datenfelder und Methoden realisiert werden (Design, d.h. Entwurf bzw. Konzept für die Programmierung)?

- abheben
- einzahlen
- Guthaben
- Konto
- Person
- Student
- Vorname
- Zuname

---

### 3.3.3 Programmierübung Person und Konto

Schreiben Sie eine Klasse Konto, die auch die in der früheren Übung erstellte Klasse Person verwendet.

Ein Konto soll folgende Eigenschaften haben:

- inhaber (Typ: Person)
- guthaben (Typ: double)

und die folgenden Methoden:

- public void einzahlen (double betrag)
- public void abheben (double betrag)
- public String toString()

Die Methoden einzahlen und abheben sollen das Guthaben um den angegebenen Betrag vergrößern oder verkleinern, und die toString-Methode soll einen String liefern, der den Inhaber und das momentane Guthaben enthält.

Zum Testen schreiben Sie eine main-Methode, in der Sie zuerst eine oder mehrere Personen anlegen, dann jeder Person ein Konto geben, und im Konto mehrere Einzahlungen und Abhebungen durchführen.

Überlegen Sie schließlich, ob und wie Sie die Klasse Konto ändern müssen, damit auch Studenten oder andere Unterklassen von Person ein Konto besitzen können, und probieren Sie das in der main-Methode aus.

## 3.4 Kapselung

### 3.4.1 Die Sichtbarkeit (public oder private)

Bisher haben wir alle unsere Datenfelder und Methoden jeweils als public deklariert. Es sind jedoch statt public auch andere Angaben für die so genannte Sichtbarkeit möglich. Die beiden wichtigsten Möglichkeiten sind:

- **public**  
das Element kann überall angesprochen werden (öffentlich).
- **private**  
das Element kann nur innerhalb dieser Klasse angesprochen werden (privat).

Es wird dringend empfohlen, bei jeder Klasse genau zu überlegen, welche Datenfelder und Methoden "von außen" direkt angesprochen werden müssen und welche nur innerhalb der Klasse benötigt werden. Alles, was nicht für die Verwendung von außen, sondern nur für die Programmierung im Inneren notwendig ist, sollte im Inneren "verborgen" werden. Dies wird als Kapselung (Encapsulation, Data Hiding) bezeichnet und hat vor allem die folgenden Vorteile:

- 
- einfache und klare Schnittstelle für die Verwendung dieser Klasse bzw. ihrer Objekte,
  - Sicherung, dass die Datenfelder stets gültige Werte haben,
  - weitgehende Unabhängigkeit der internen Programmierung dieser Klasse von der Programmierung anderer Klassen,
  - weitgehende Vermeidung von Programmierfehlern beim Zusammenspiel zwischen den verschiedenen Klassen und Programmen.

### 3.4.2 Die Java-Beans-Konventionen

Unter Java-Beans versteht man Java-Klassen, die nach bestimmten Konventionen geschrieben wurden und deshalb besonders gut von anderen Java-Klassen verwendet werden können. Dies ist also so eine Art "Gütesiegel" für Java-Programme. Der Name bedeutet eigentlich Kaffeebohne und meint hier den Grundstoff, aus dem gute Java-Systeme erzeugt werden.

Um ungültige Datenwerte in den Datenfeldern zu vermeiden, ist es empfehlenswert, alle Datenfelder als `private` zu definieren und eigene `public` Methoden für das Setzen und Abfragen der Werte vorzusehen, die schon bei der Speicherung der Daten alle Fehler verhindern.

Nach der für "Java-Beans" eingeführten Konvention sollen diese Methoden Namen der Form `setXxxx` und `getXxxx` haben und nach folgendem Schema definiert werden:

```
public class ClassName {  
  
    private typ xxxx = anfangswert;  
  
    public void setXxxx (typ xxxx) {  
        // Gültigkeit des Wertes kontrollieren  
        this.xxxx = xxxx;  
    }  
  
    public typ getXxxx() {  
        return this.xxxx;  
    }  
}
```

---

### 3.4.3 Beispiel: HelloWorld-Bean

Das HelloWorld-Programm wird nach diesen Bean-Konventionen also so abgeändert, dass das Datenfeld `messageText` `private` deklariert wird und für den Zugriff die `public` Methoden `setMessageText` und `getMessageText` vorgesehen werden:

```
public class HelloBean {

    private String messageText = "Hello World!";

    public void setMessageText(String newText) {
        messageText = newText;
    }

    public String getMessageText() {
        return messageText;
    }

    public void printText() {
        System.out.println (messageText);
    }

    public static void main (String[] args) {
        HelloBean engl = new HelloBean();
        HelloBean germ = new HelloBean();
        germ.setMessageText("Hallo, liebe Leute!");
        HelloBean cat = new HelloBean();
        cat.setMessageText("Miau!");

        engl.printText();
        germ.printText();
        engl.printText();
        cat.printText();
        engl.printText();
    }
}
```

---

### 3.4.4 Beispiel: schlechte Datums-Klasse

Eine sehr ungünstige Möglichkeit, eine Klasse für Datum-Objekte zu definieren, bestünde darin, einfach nur die 3 int-Felder als public zu definieren und dann alles Weitere den Anwendungsprogrammierern zu überlassen:

```
public class BadDate {
    public int year;
    public int month;
    public int day;
}
```

In der main-Methode können dann Objekte dieser Klasse angelegt und verwendet werden. Dabei kann es passieren, dass ungültige Werte erzeugt werden (frei nach Erich Kästner):

```
public static void main (String[] args) {
    BadDate birthDay = new BadDate();
    birthDay.day = 35;
    birthDay.month = 5;
    birthDay.year = 95; // statt 1995
    ...
}
```

und das würde dann zu verwirrenden Folgefehlern in allen Programmen führen, die dieses BadDate-Objekt verwenden. Deswegen soll man es nicht so machen, sondern die drei Datenfelder als private definieren und public Methoden vorsehen, mit denen diese Werte richtig gesetzt, verändert und verwendet werden.

In Wirklichkeit ist es natürlich nicht notwendig, eine eigene Klasse für Datumsangaben zu schreiben, die ist bereits in der Klassenbibliothek vorhanden: die Klasse Date im Paket java.util.

### 3.4.5 Übungen

Ändern Sie die Klassen Person, Student und Konto so ab, dass sie den Beans-Konventionen entsprechen.

---

## 4 Die Java Klassenbibliothek

Einer der großen Vorteile von Java ist die umfangreiche Klassenbibliothek, die zum Java-Laufzeitsystem dazu gehört. Sie enthält weit mehr als tausend nützliche Klassen und Programme, die jeder Java-Programmierer in seinen eigenen Programmen verwenden kann. Damit können auch komplexe Aufgaben wie Graphische User-Interfaces, Client-Server-Systeme oder Datenbank-Abfragen mit sehr wenig eigenem Aufwand rasch und bequem, sicher und robust programmiert werden.

### 4.1 Überblick über die wichtigsten Pakete

Die Klassenbibliothek ist in so genannte Pakete aufgeteilt, das sind jeweils Gruppen von Klassen, die zusammengehören, ähnlich wie Dateien in Directories.

Die wichtigsten Pakete für Anwendungs-Programmierer sind:

- **java.lang** (language):  
wichtige grundlegende Klassen, z.B. System, String, Math
- **java.awt** und **java.awt.event** (abstract windowing toolkit):  
für graphische User-Interfaces
- **java.util** und **java.text** (utilities):  
verschiedene nützliche Hilfsprogramme wie Date, DecimalFormat u.a.
- **java.io** (input, output):  
für das Lesen und Schreiben von Dateien und Datenströmen
- **java.net** (networking):  
für die Datenübertragung über das Internet
- **java.sql** (structured query language):  
für den Zugriff auf Datenbanken

### 4.2 Graphische User-Interfaces (GUI)

Als Beispiel für die Verwendung von Klassen aus der Java-Klassenbibliothek wollen wir ein graphisches User-Interface für unsere Konto-Klasse erstellen. Die dazu benötigten Klassen finden wir in den Paketen java.awt und java.awt.event, Außerdem verwenden wir auch ein Hilfsprogramm aus dem Paket java.text.

Sie werden sehen, dass wir nur sehr wenig selbst programmieren müssen, das Meiste können wir relativ einfach durch eine geeignete Kombination von Objekten aus diesen Paketen und durch Aufrufen der in ihnen enthaltenen Methoden erledigen.

---

### 4.2.1 Erster Schritt: Das Datenmodell (model)

Als erstes wollen wir uns den generellen Programmaufbau und das so genannte Datenmodell (englisch: *model*) überlegen.

Zunächst skizzieren wir die main-Methode:

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

        Person hans = new Person();
        hans.setVorname("Hans");
        hans.setZuname("Reich");

        Konto hansKonto = new Konto();
        hansKonto.setInhaber(hans);

        KontoFrame fenster = new KontoFrame();
        fenster.setKonto(hansKonto);
        fenster.setVisible(true);

    }
}
```

Wir haben drei Objekte:

- ein Objekt vom Typ `Person`, mit Vorname und Zuname
- ein Objekt vom Typ `Konto`. Mit der `setInhaber`-Methode geben wir an, welcher Person dieses Konto gehört.
- ein Objekt vom Typ `KontoFrame` für das Fenster im Windowing-System. Mit der `setKonto`-Methode geben wir an, welches Konto in diesem Fenster bearbeitet werden soll. Mit der `setVisible`-Methode machen wir das Fenster am Bildschirm sichtbar.

Nun müssen wir die Klasse `KontoFrame` schreiben. Zunächst müssen wir mit `import`-Anweisungen angeben, aus welchen Java-Paketen wir Klassen verwenden:

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

Das Paket `java.awt` enthält Klassen für die graphischen Komponenten wie z.B. `Frame` für ein Fenster mit Rahmen, `Label` für eine Textanzeige, `TextField` für eine Texteingabe, `Button` für

---

eine Schaltfläche, Menubar für einen Menüleiste etc, sowie ein paar Hilfsprogramme wie z.B. die Layout-Manager-Programme. Ein paar dieser Klassen benötigen wir im ersten und zweiten Schritt.

Das Paket `java.awt.event` enthält die Klassen für das Event-Handling, die wir erst im dritten Schritt benötigen.

Das Paket `java.text` enthält unter anderem die Klasse `DecimalFormat`, die wir im zweiten Schritt für die Formatierung des Geldbetrages verwenden werden.

Die Klasse `Frame` im Paket `java.awt` beschreibt Bildschirm-Fenster mit Rahmen, Titelleiste, Schließen-Knopf etc. Unser Konto-Fenster ist ein Spezialfall eines solchen Fensters, daher werden wir unsere Klasse als Unterklasse von `Frame` definieren. Damit "erben" wir alle Eigenschaften von der Klasse `Frame`, wie z.B. die Methode `setVisible`, mit der das Fenster sichtbar gemacht werden kann.

```
public class KontoFrame extends Frame {  
    ...  
}
```

Unser Programm muss "wissen", welches Konto darin bearbeitet werden soll. Daher brauchen wir eine Variable vom Typ `Konto`, mit entsprechenden `get`- und `set`-Methoden:

```
private Konto meinKonto = null;  
  
public void setKonto (Konto k) {  
    this.meinKonto = k;  
}  
  
public Konto getKonto() {  
    return meinKonto;  
}
```

Wenn wir das jetzt übersetzen und ausführen, dann wird das schon funktionieren. Alle Daten sind richtig eingesetzt. Allerdings sehen wir die Daten des Kontos noch nicht am Bildschirm, sondern es erscheint nur ein sehr kleines, leeres Fenster. Wir haben ja die Größe und den Inhalt des Fensters noch nicht festgelegt. Dies werden wir im zweiten Schritt tun. Außerdem funktioniert auch der Schließen-Knopf des Fensters noch nicht, da wir die Ablaufsteuerung, das so genannte Event-Handling, erst im dritten Schritt festlegen werden. Bis dahin können wir unser Programm nur dadurch beenden, dass wir es mit `Strg-C` im DOS-Fenster oder mit dem Task-Manager abbrechen.

---

## 4.2.2 Zweiter Schritt: Die Ansicht (view)

Nun wollen wir festlegen, wie das Fenster aussehen soll, also welche Komponenten es enthalten soll. Da dies relativ umfangreich wird, werden wir dafür eine eigene Methode schreiben, der wir den Namen `init` geben, weil sie den Anfangszustand des Fensters festlegt.

So soll das Fenster aussehen:



Wir haben innerhalb unseres Fensters also 8 Komponenten: 2 Anzeigefelder (für Inhaber und Kontostand), 3 fixe Beschriftungen, 1 Eingabefeld (für den Betrag) und 2 Buttons (einzahlen und abheben). Diese Komponenten deklarieren wir nicht als lokale Variable innerhalb der `init`-Methode, sondern auf der obersten Ebene der Klasse, als `private` Eigenschaften unseres Fensterobjektes. Dies ist notwendig, damit wir die Komponenten in allen Methoden der Klasse ansprechen können, im speziellen Fall in der `init`-Methode, die wir jetzt gleich schreiben, und in der `actionPerformed`-Methode, die wir im dritten Schritt hinzufügen werden.

```
private Label inhaberText = new Label("Inhaber");
private Label inhaberAnzeige = new Label();
private Label guthabenText = new Label("Kontostand");
private Label guthabenAnzeige = new Label();
private Label betragText = new Label("Betrag eingeben");
private TextField betragEingabe = new TextField("100");
private Button einzahlenButton = new Button("einzahlen");
private Button abhebenButton = new Button("abheben");
```

Die Klassen `Label` und `Button` haben Konstruktoren, in denen man gleich den Text angeben kann, der in der Textanzeige bzw. auf der Schaltfläche stehen soll. Bei den beiden Anzeigefeldern geben wir vorerst noch keinen Text an. Dort werden dann später die entsprechenden Daten des Kontos mit der `setText`-Methode eingefügt.

Außerdem deklarieren wir zwei Hilfsobjekte, die wir benötigen werden.

Für die Festlegung, wie, in welcher Größe und an welcher Stelle die Komponenten im Fenster angeordnet werden, wird ein Objekt des entsprechenden `Layout-Manager`-Programms

---

angegeben. Wir wählen für unser Layout ein GridLayout, bei dem die Komponenten wie in einem Gitter oder in einer Tabelle angeordnet werden. Im Konstruktor von GridLayout wird angegeben, aus wievielen Zeilen und Spalten das Layout besteht. In unserem Fall werden wir die 8 Komponenten in 4 Zeilen mit 2 Spalten anordnen:

```
private GridLayout fensterLayout = new GridLayout (4, 2);
```

Außerdem verwenden wir das Hilfsprogramm DecimalFormat für die Formatierung des Geldbetrages. Im Konstruktor der Klasse DecimalFormat geben wir an, nach welchen Regeln dieses Hilfsobjekt Zahlen formatieren soll, mit einem Muster (englisch *pattern*), wie es auch in MS-Excel verwendet wird:

```
private DecimalFormat geldFormat =  
    new DecimalFormat ("#,##0.00");
```

Darin bedeutet die Null, dass diese Stellen jedenfalls angezeigt werden, auch wenn sie null sind, und das Nummernzeichen, dass diese Stellen nur dann angezeigt werden, wenn es keine führenden Nullen sind. Der Punkt gibt (wie in Amerika üblich) den Dezimalpunkt an und das Komma den Tausendertrenner. Wir wollen also die Geldbeträge immer mit 2 Nachkommastellen angeben und, wenn sie groß genug sind, mit einem Tausendertrenner.

Nun schreiben wir die Methode, in der der gesamte Fensterinhalt festgelegt wird. Wir nennen diese Methode `init`, weil sie den Anfangszustand des Fensters festlegt:

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

In dieser Methode setzen wir zunächst den Kontoinhaber in das entsprechende Anzeigefeld. Dazu holen wir zunächst den Inhaber mit der `getInhaber`-Methode aus dem Konto-Objekt. Dieses Ergebnis ist vom Typ `Person`. Wir brauchen aber einen Text, den wir anzeigen können, daher wandeln wir die `Person` mit Hilfe der `toString`-Methode in einen `String` um. Diesen `String` setzen wir schließlich mit der `setText`-Methode in das Anzeigefeld:

```
Person inhaber = meinKonto.getInhaber();  
String inhString = inhaber.toString();  
inhaberAnzeige.setText(inhString);
```

Ähnlich gehen wir auch mit dem Guthaben vor: Mit der `getGuthaben`-Methode bekommen wir den Zahlenwert, und mit der `format`-Methode unseres `DecimalFormat`-Objektes wandeln wir diese Zahl in einen schön formatierten `String` um, den wir schließlich mit `setText` in das Anzeigefeld einsetzen:

```
double guthaben = meinKonto.getGuthaben();  
String gutString = geldFormat.format(guthaben);  
guthabenAnzeige.setText(gutString);
```

Nun können wir das Aussehen und den Inhalt unseres Fensters festlegen, also des Objektes unserer `KontoFrame`-Klasse, das durch das Schlüsselwort `this` angegeben wird. Zunächst geben wir mit der Methode `setSize` die Fenstergröße an. Der erste Parameter gibt die Breite in Pixeln an, der zweite die Höhe:

---

```
    this.setSize(300,150);
```

Dann geben wir an, welcher Text in der Titelleiste des Fensters stehen soll:

```
    this.setTitle("Konto-Fenster");
```

Bevor wir die einzelnen Komponenten im Fenster anordnen, geben wir mit `setLayout` den `LayoutManager` an, also die Regeln, nach denen die Komponenten von den folgenden `add`-Methoden angeordnet werden sollen:

```
    this.setLayout(fensterLayout);
```

Nun fügen wir die 8 Komponenten mit `add`-Befehlen zum Fenster hinzu, und zwar in der richtigen Reihenfolge, wie sie in der Gitterstruktur erscheinen sollen:

```
    this.add(inhaberText);
    this.add(inhaberAnzeige);
    this.add(guthabenText);
    this.add(guthabenAnzeige);
    this.add(betragText);
    this.add(einzahlenButton);
    this.add(betragEingabe);
    this.add(abhebenButton);
```

Damit ist unsere `init`-Methode komplett. Zu guter Letzt müssen wir noch die `main`-Methode so erweitern, dass die `init`-Methode aufgerufen wird:

```
    ...
    KontoFrame fenster = new KontoFrame();
    fenster.setKonto(hansKonto);
    fenster.init();
    fenster.setVisible(true);
```

Hier ist die richtige Reihenfolge wichtig: Zuerst müssen wir mit der `setKonto`-Methode angeben, welches Konto in unserem Fenster bearbeitet werden soll. Dann kann mit der `init`-Methode der Inhalt des Fensters festgelegt werden. Schließlich wird das Fenster mit seinem richtigen Inhalt mit der `setVisible`-Methode sichtbar gemacht.

Wenn wir das nun übersetzen und ausführen, bekommen wir ein Fenster, das schon richtig aussieht. Allerdings funktionieren die Buttons noch nicht, d.h. wir können das Guthaben noch nicht mit den Buttons einzahlen und abheben verändern, und wir können das Fenster auch noch nicht mit dem vorgesehenen Schließen-Knopf schließen, sondern müssen immer noch `Strg-C` im DOS-Fenster oder den Task-Manager verwenden, um unser Programm abzu-brechen. Uns fehlt noch das Event-Handling, das wir im dritten Schritt in unser Programm einbauen werden.

---

### 4.2.3 Dritter Schritt: Die Ablaufsteuerung (Event-Handling, controller)

Nun wollen wir die Ablaufsteuerung realisieren, die so genannte Ereignis-Behandlung (englisch *event handling*) bzw. die Steuerung unseres Programms (englisch *controller*).

Ein graphisches User-Interface bietet dem Anwender verschiedene Eingabemöglichkeiten, Buttons etc. an und wartet dann, dass der Anwender etwas mit seiner Maus oder mit seiner Tastatur tut. Wenn der Anwender so etwas tut, dann wird das als Ereignis (englisch *event*) bezeichnet, und es gibt einen eigenen Programmteil, der auf ein solches Ereignis wartet, das wird auf englisch als *Listener* bezeichnet.

Wir erweitern unsere Klassendefinition zunächst so, dass wir angeben, welche *Listener* für welche Events wir in unserer Klasse realisieren:

```
public class KontoFrame extends Frame
    implements ActionListener, WindowListener {
    ...
}
```

Das Schlüsselwort `implements` wird ähnlich wie das Schlüsselwort `extends` verwendet. Nach `implements` gibt man eine oder mehrere so genannte Schnittstellen (englisch *interfaces*) an. Ein Interface gibt an, welche Methoden in einer Klasse implementiert, also programmiert werden müssen.

Das Interface `ActionListener` enthält die Methode `actionPerformed`, die beim Anklicken eines Buttons oder eines Menüpunktes oder bei der Eingabe in einem Textfeld aufgerufen wird. Das werden wir für die beiden Buttons einzahlen und abheben brauchen.

Das Interface `WindowListener` enthält unter anderem die Methode `windowClosing`, die dann aufgerufen wird, wenn der User auf den Schließen-Knopf des Fensters klickt oder das Fenster mit Alt-F4 schließen will.

Diese Methoden werden wir also in unserer Klasse schreiben. Wir müssen aber auch angeben, wann diese Methoden aufgerufen werden sollen. Mit anderen Worten, wir müssen unsere Komponenten "aktiv" machen, d.h. wir müssen bei den Buttons und beim Fenster den "Listener" einschalten, der auf die entsprechenden Ereignisse wartet.

Zunächst geben wir innerhalb der `init`-Methode bei den beiden Buttons mit der Methode `addActionListener` an, in welchem Programm sich die `actionPerformed`-Methode befindet, die aufgerufen werden soll, wenn der Anwender diesen Button anklickt. In unserem Fall ist das unser eigenes Programm, das durch das Schlüsselwort `this` angegeben wird:

```
    einzahlenButton.addActionListener(this);
    abhebenButton.addActionListener(this);
```

Dann geben wir auch noch bei unserem Fenster mit der Methode `addWindowListener` an, in welchem Programm sich die `windowClosing`-Methode befindet, die dann aufgerufen wird, wenn der Anwender das Fenster schließen will. Dieses Fenster ist ein Objekt unserer eigenen Klasse `KontoFrame`, es wird also mit dem Schlüsselwort `this` angesprochen, und die Methode

---

liegt ebenfalls in unserem eigenen Programm, der Parameter ist also auch wieder das Schlüsselwort `this`:

```
this.addWindowListener(this);
```

Mit diesen Ergänzungen ist die `init`-Methode komplett.

Nun müssen wir noch die Methoden schreiben, die in den Listener-Interfaces vorgesehen sind, d.h. wir müssen angeben, was passieren soll, wenn der Anwender einen der Buttons anklickt oder das Fenster schließt.

Zunächst schreiben wir die Methode `actionPerformed`, die immer dann aufgerufen wird, wenn einer der beiden Buttons angeklickt wird. Diese Methode hat einen Parameter von Typ `ActionEvent`, der alle Informationen über das eingetretene Ereignis enthält.

```
public void actionPerformed (ActionEvent e) {  
    ...  
}
```

Zunächst müssen wir wissen, welcher Geldbetrag eingezahlt oder abgehoben werden soll. Dazu holen wir uns mit der Methode `getText` den Text, der im Eingabefeld steht, und entfernen mit der Methode `trim` eventuelle Leerzeichen, die vor oder nach den Ziffern stehen. Mit der Methode `parseDouble` wandeln wir diesen Text (also die Ziffernfolge) in den entsprechenden Zahlenwert um:

```
String betragString = betragEingabe.getText().trim();  
double betrag = Double.parseDouble( betragString );
```

Als nächstes müssen wir feststellen, welcher der beiden Buttons angeklickt wurde. Zu diesem Zweck können wir die Methode `getSource` des `ActionEvent` verwenden:

```
Object angeklickt = e.getSource();
```

Wenn es der `einzahlenButton` war, dann rufen wir die Methode `einzahlen` unseres Kontos auf, wenn es der `abhebenButton` war, dann die Methode `abheben`:

```
if (angeklickt == einzahlenButton ) {  
    meinKonto.einzahlen(betrag);  
}  
if (angeklickt == abhebenButton ) {  
    meinKonto.abheben(betrag);  
}
```

Damit haben wir den richtigen neuen Wert in unserem Datenmodell gespeichert. Wir müssen aber auch noch dafür sorgen, dass auch in unserem Fenster (in unserer Ansicht) der richtige neue Wert des Guthabens angezeigt wird, und zwar in beiden Fällen, also nach den beiden `if`-Blöcken. Dazu verwenden wir wieder die gleiche Formatierung wie in der `init`-Methode:

```
double guthaben = meinKonto.getGuthaben();  
String gutString = geldFormat.format(guthaben);  
guthabenAnzeige.setText(gutString);
```

---

Damit ist die actionPerformed-Methode fertig.

Nun schreiben wir noch die Methode windowClosing, die dann aufgerufen wird, wenn der Anwender das Fenster schließen will. Diese Methode soll unser Fenster (Schlüsselwort this) tatsächlich schließen, also mit der setVisible-Methode unsichtbar machen, und dann das ganze Programm mit der exit-Methode beenden:

```
public void windowClosing (WindowEvent e) {  
    this.setVisible(false);  
    System.exit(0);  
}
```

Der WindowListener sieht auch noch ein paar weitere Methoden vor, die bei anderen Ereignissen unseres Fensters aufgerufen werden, etwa beim Auf- und Zuklappen. In allen diesen Fällen soll in unserem Programm gar nichts passieren, wir versehen diese Methoden also alle mit einem leeren Inhalt:

```
public void windowClosed (WindowEvent e) { }  
public void windowOpened (WindowEvent e) { }  
public void windowIconified (WindowEvent e) { }  
public void windowDeiconified (WindowEvent e) { }  
public void windowActivated (WindowEvent e) { }  
public void windowDeactivated (WindowEvent e) { }
```

Nun ist unser GUI-Programm fertig, und wir können es übersetzen und ausführen, und es wird so funktionieren, wie wir es uns vorgestellt haben.

#### **4.2.4 Das komplette Programm**

Hier zur Übersicht noch das komplette Programm, wie wir es schrittweise zusammengesetzt haben:

Zunächst die komplette main-Methode:

```
public class FensterTest {  
    public static void main (String[] args) {  
  
        Person hans = new Person();  
        hans.setVorname("Hans");  
        hans.setZuname("Reich");  
  
        Konto hansKonto = new Konto();  
        hansKonto.setInhaber(hans);  
  
        KontoFrame fenster = new KontoFrame();  
        fenster.setKonto(hansKonto);  
    }  
}
```

---

```
        fenster.init();
        fenster.setVisible(true);
    }
}
```

Und nun die komplette Klasse KontoFrame:

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

public class KontoFrame extends Frame
    implements ActionListener, WindowListener {

    private Konto meinKonto = null;

    private Label inhaberText = new Label("Inhaber");
    private Label inhaberAnzeige = new Label();
    private Label guthabenText = new Label("Kontostand");
    private Label guthabenAnzeige = new Label();
    private Button einzahlenButton = new Button("einzahlen");
    private Button abhebenButton = new Button("abheben");

    private GridLayout fensterLayout = new GridLayout (3, 2);
    private DecimalFormat geldFormat =
        new DecimalFormat("#,##0.00");

    public void setKonto (Konto k) {
        this.meinKonto = k;
    }

    public Konto getKonto() {
        return meinKonto;
    }

    public void init() {

        Person inhaber = meinKonto.getInhaber();
        String inhString = inhaber.toString();
        inhaberAnzeige.setText(inhString);

        double guthaben = meinKonto.getGuthaben();
        String gutString = geldFormat.format(guthaben);
        guthabenAnzeige.setText(gutString);

        this.setSize(300,150);
        this.setTitle("Konto-Fenster");
    }
}
```

---

```

        this.setLayout(fensterLayout);
        this.add(inhaberText);
        this.add(inhaberAnzeige);
        this.add(guthabenText);
        this.add(guthabenAnzeige);
        this.add(einzahlenButton);
        this.add(abhebenButton);

        einzahlenButton.addActionListener(this);
        abhebenButton.addActionListener(this);
        this.addWindowListener(this);
    }

    public void actionPerformed (ActionEvent e) {
        Object angeklickt = e.getSource();
        if (angeklickt == einzahlenButton ) {
            meinKonto.einzahlen(betrag);
        }
        if (angeklickt == abhebenButton ) {
            meinKonto.abheben(betrag);
        }
        double guthaben = meinKonto.getGuthaben();
        String gutString = geldFormat.format(guthaben);
        guthabenAnzeige.setText(gutString);
    }

    public void windowClosing (WindowEvent e) {
        this.setVisible(false);
        System.exit(0);
    }

    public void windowClosed (WindowEvent e) { }
    public void windowOpened (WindowEvent e) { }
    public void windowIconified (WindowEvent e) { }
    public void windowDeiconified (WindowEvent e) { }
    public void windowActivated (WindowEvent e) { }
    public void windowDeactivated (WindowEvent e) { }
}

```

---

## 4.3 Datenbanken

Zu den wichtigsten Anwendungsgebieten von Java zählen User-Interfaces zu Datenbanksystemen. Das Java-Programm kann dabei am selben Rechner wie die Datenbank laufen oder auch auf einem anderen Rechner und von dort über das Internet oder ein Intranet auf die Datenbank zugreifen. Auf diese Weise kann man die Vorteile von Java, die vor allem bei der Gestaltung von (graphischen und plattformunabhängigen) User-Interfaces und von Netzverbindungen liegen, mit der Mächtigkeit von Datenbanksystemen verbinden.

Die "Java Database Connectivity" (JDBC) ist im Sinne der Plattformunabhängigkeit von Java so aufgebaut, dass das Java-Programm von der Hard- und Software des Datenbanksystems unabhängig ist und somit für alle Datenbanksysteme (MS-Access, MySQL, Oracle, DB2 etc.) funktioniert.

Hier finden Sie zunächst eine kurze allgemein Einführung, was Datenbanken sind und wie man Daten in Datenbanken mit SQL-Befehlen abfragen oder verändern kann. Anschließend finden Sie dann, wie Sie das innerhalb von Java-Programmen realisieren können.

### 4.3.1 Relationale Datenbanken

Unter einem **Datenbanksystem** versteht man ein Software-Paket zur Verwaltung und Verarbeitung von Datenbanken. Beispiele: MS-Access, MySQL, Oracle, DB2.

Unter einer **Datenbank** versteht man eine Menge von zusammen gehörenden Daten. Beispiele: die Personaldatenbank, die Forschungsdatenbank, der Bibliothekskatalog, der Veranstaltungskalender, die Kursdatenbank.

Relationale Datenbanken bestehen aus **Tabellen** (Relationen). Die Tabellen entsprechen in etwa den **Klassen** in der Objektorientierten Programmierung. Beispiele: Eine Personaldatenbank enthält Tabellen für Mitarbeiter, Abteilungen, Projekte. Eine Literaturdatenbank enthält Tabellen für Bücher, Zeitschriften, Autoren, Verlage.

Diese Tabellen können in Beziehungen zueinander stehen (daher der Name "Relation"). Beispiele: Ein Mitarbeiter gehört zu einer Abteilung und eventuell zu einem oder mehreren Projekten. Jede Abteilung und jedes Projekt wird von einem Mitarbeiter geleitet. Ein Buch ist in einem Verlag erschienen und hat einen oder mehrere Autoren.

---

Beispielskizze für eine Tabelle "**Mitarbeiter**":

Abteilung	Vorname	Zuname	Geburtsjahr	Gehalt
EDV-Zentrum	Hans	Fleißig	1972	2400.00
EDV-Zentrum	Grete	Tüchtig	1949	3200.00
Personalstelle	Peter	Gscheitl	1968	1600.00

Jede **Zeile** der Tabelle (row, Tupel, Record) enthält die Eigenschaften eines Elementes dieser Menge, entspricht also einem **Objekt**. In den obigen Beispielen also jeweils ein bestimmter Mitarbeiter, eine Abteilung, ein Projekt, ein Buch, ein Autor, ein Verlag. Jede Zeile muss eindeutig sein, d.h. verschiedene Mitarbeiter müssen sich durch mindestens ein Datenfeld (eine Eigenschaft) unterscheiden.

Jede **Spalte** der Tabelle (column, field, entity) enthält die gleichen Eigenschaften der verschiedenen Objekte, entspricht also einem **Datenfeld**. Beispiele: Vorname, Zuname, Geburtsjahr und Abteilung eines Mitarbeiters, oder Titel, Umfang, Verlag und Erscheinungsjahr eines Buches.

Ein Datenfeld (z.B. die Sozialversicherungsnummer eines Mitarbeiters oder die ISBN eines Buches) oder eine Gruppe von Datenfeldern (z.B. Vorname, Zuname und Geburtsdatum einer Person) muss eindeutig sein, sie ist dann der **Schlüssel** (key) zum Zugriff auf die Zeilen (Records) in dieser Tabelle. Eventuell muss man dafür eigene Schlüsselfelder einrichten, z.B. eine eindeutige Projektnummer, falls es mehrere Projekte mit dem gleichen Namen gibt.

Die **Beziehungen** zwischen den Tabellen können auch durch weitere Tabellen (Relationen) dargestellt werden, z.B. eine Buch-Autor-Relation, wobei sowohl ein bestimmtes Buch als auch ein bestimmter Autor eventuell in mehreren Zeilen dieser Tabelle vorkommen kann, denn ein Buch kann mehrere Autoren haben und ein Autor kann mehrere Bücher geschrieben haben. Das Gleiche gilt für die Relation Mitarbeiter-Projekt.

Das Konzept (Design) einer Datenbank ist eine komplexe Aufgabe, es geht dabei darum, alle relevanten Daten (Elemente, Entities) und alle Beziehungen zwischen ihnen festzustellen und dann die logische Struktur der Datenbank entsprechend festzulegen. Die logisch richtige Aufteilung der Daten in die einzelnen Tabellen (Relationen) wird als Normalisierung der Datenbank bezeichnet, es gibt dafür verschiedene Regeln und Grundsätze, ein Beispiel ist die sogenannte dritte Normalform.

Fast alle Datenbanksysteme seit Ende der 70er- oder Beginn der 80er-Jahre sind relationale Datenbanksysteme und haben damit die in den 60er- und 70er-Jahren verwendeten, bloß hierarchischen Datenbanksysteme abgelöst. Eine zukunftsweisende Weiterentwicklung der Relationalen Datenbanken sind die "Objektrelationalen Datenbanken", bei denen - vereinfacht gesprochen - nicht nur primitive Datentypen sondern auch komplexe Objekte als Datenfelder möglich sind, ähnlich wie in der objektorientierten Programmierung.

---

## Datenschutz und Datensicherheit

Datenbanken enthalten meist umfangreiche und wichtige, wertvolle Informationen. Daher muss bei Datenbanksystemen besonderer Wert auf den Datenschutz und die Datensicherheit gelegt werden.

Die Datenbank-Benutzer werden in zwei Gruppen aufgeteilt: den Datenbankadministrator und die Datenbankanwender.

Der **Datenbankadministrator** legt mit der Data Definition Language (DDL) die logischen Struktur der Datenbank fest und ist für den Datenschutz, die Vergabe der Berechtigungen an die Datenbankanwender und für die Datensicherung verantwortlich.

Die **Datenbankanwender** können mit einer Data Manipulation Language (DML) oder Query Language die Daten in der Datenbank speichern, abfragen oder verändern.

Mit der Hilfe von Usernames und Passwörtern werden die einzelnen Benutzer identifiziert, und der Datenbankadministrator kann und muss sehr detailliert festlegen, welcher Benutzer welche Aktionen (lesen, hinzufügen, löschen, verändern) mit welchen Teilen der Datenbank (Tabellen, Spalten, Zeilen) ausführen darf.

## Datenkonsistenz

Die in einer Datenbank gespeicherten Informationen stehen meistens in Beziehungen zueinander, bestimmte Informationen hängen in eventuell recht komplexer Weise von anderen, ebenfalls in der Datenbank gespeicherten Informationen ab. Es muss sichergestellt werden, dass die gesamte Datenbank immer in einem gültigen Zustand ist, dass also niemals ungültige oder einander widersprechende Informationen darin gespeichert werden.

Dies wird mittels **Transaktionen** erreicht: Unter einer Transaktion versteht man eine Folge von logisch zusammengehörenden Aktionen, die nur entweder alle vollständig oder überhaupt nicht ausgeführt werden dürfen.

Beispiel: Wenn zwei Mitarbeiter den Arbeitsplatz tauschen, muss sowohl beim Mitarbeiter 1 der Arbeitsplatz von A auf B als auch beim Mitarbeiter 2 der Arbeitsplatz von B auf A geändert werden. Würde bei einer dieser beiden Aktionen ein Fehler auftreten, die andere aber trotzdem ausgeführt werden, dann hätten wir plötzlich 2 Mitarbeiter auf dem einen Arbeitsplatz und gar keinen auf dem anderen.

Ein anderes Beispiel: Wenn in der Datenbank nicht nur die Gehälter der einzelnen Mitarbeiter sondern auch die Summe aller Personalkosten (innerhalb des Budgets) gespeichert ist, dann müssen bei jeder Gehaltserhöhung beide Felder um den gleichen Betrag erhöht werden, sonst stimmen Budgetplanung und Gehaltsauszahlung nicht überein.

---

Um solche Inkonsistenzen zu vermeiden, müssen zusammengehörende Aktionen jeweils zu einer **Transaktion** zusammengefasst werden:

- Wenn alle Aktionen erfolgreich abgelaufen sind, wird die Transaktion beendet (**commit**) und die Datenbank ist in einem neuen gültigen Zustand.
- Falls während der Transaktion irgendein Fehler auftritt, werden alle seit Beginn der Transaktion ausgeführten unvollständigen Änderungen rückgängig gemacht (**rollback**), und die Datenbank ist wieder in dem selben alten, aber gültigen Zustand wie vor Beginn der versuchten Transaktion.

### 4.3.2 Structured Query Language (SQL)

SQL hat sich seit den 80er-Jahren als die von allen Datenbanksystemen (wenn auch eventuell mit kleinen Unterschieden) unterstützte Abfragesprache durchgesetzt, und die Version SQL2 ist seit 1992 auch offiziell genormt.

SQL umfasst alle 3 Bereiche der Datenbankbenutzung:

- die Definition der Datenbankstruktur,
- die Speicherung, Löschung und Veränderung von Daten und
- die Abfrage von Daten.

Für eine komplette Beschreibung von SQL wird auf die Fachliteratur verwiesen, hier sollen nur ein paar typische Beispiele gezeigt werden.

#### Datentypen

SQL kennt unter anderem die folgenden Datentypen:

- `CHAR (n)` = ein Textstring mit einer Länge von genau `n` Zeichen (ähnlich wie String, wird mit Leerstellen aufgefüllt)
- `VARCHAR (n)` = ein Textstring mit einer variablen Länge von maximal `n` Zeichen (`n < 255`)
- `LONGVARCHAR (n)` = ein Textstring mit einer variablen Länge von maximal `n` Zeichen (`n > 254`)
- `DECIMAL` oder `NUMERIC` = eine als String von Ziffern gespeicherte Zahl (ähnlich wie `BigInteger`)
- `INTEGER` = eine ganze Zahl (4 Bytes, wie `int`)
- `SMALLINT` = eine ganze Zahl (2 Bytes, wie `short`)

- 
- BIT = wahr oder falsch (wie boolean)
  - REAL oder FLOAT = eine Fließkommazahl (wie float)
  - DOUBLE = eine doppelt genaue Fließkommazahl (wie double)
  - DATE = ein Datum (Tag)
  - TIME = eine Uhrzeit
  - TIMESTAMP = ein Zeitpunkt (Datum und Uhrzeit, ähnlich wie Java-Date)

### **Definition der Datenbankstruktur (DDL)**

CREATE = Einrichten einer neuen Tabelle.

Beispiel:

```
CREATE TABLE Employees (  
    INT      EmployeeNumber ,  
    CHAR(30) FirstName ,  
    CHAR(30) LastName ,  
    INT      BirthYear ,  
    FLOAT    Salary  
)
```

definiert eine Tabelle "Employees" (Mitarbeiter) mit den angegebenen Datenfeldern. Die Erlaubnis dazu hat meistens nur der Datenbankadministrator.

ALTER = Ändern einer Tabellendefinition.

DROP = Löschen einer Tabellendefinition.

---

## Änderungen an den Daten (Updates)

INSERT = Speichern eines Records in einer Tabelle.

Beispiel:

```
INSERT INTO Employees
    (EmployeeNumber, FirstName, LastName, BirthYear, Salary)
VALUES ( 4710, 'Hans', 'Fleißig', 1972, 2400.0 )
INSERT INTO Employees
    (EmployeeNumber, FirstName, LastName, BirthYear, Salary)
VALUES ( 4711, 'Grete', 'Tüchtig', 1949, 3200.0 )
```

speichert zwei Mitarbeiter-Records mit den angegebenen Daten.

UPDATE = Verändern von Datenfeldern in einem oder mehreren Records.

Beispiel:

```
UPDATE Employees
    SET Salary = 3600.0 WHERE LastName = 'Tüchtig'
```

setzt das Gehalt bei allen Mitarbeitern, die Tüchtig heißen, auf 3600.

DELETE = Löschen eines oder mehrerer Records

Beispiel:

```
DELETE FROM Employees WHERE EmployeeNumber = 4710
```

löscht den Mitarbeiter mit der Nummer 4710 aus der Datenbank.

## Abfrage von Daten

SELECT = Abfragen von gespeicherten Daten.

Beispiele:

```
SELECT * FROM Employees
```

liefert alle Datenfelder von allen Records der Tabelle Employees.

```
SELECT LastName, FirstName, Salary FROM Employees
```

liefert die angegebenen Datenfelder von allen Records der Tabelle Employees.

---

```
SELECT LastName, Salary
FROM Employees WHERE BirthYear <= 1970
ORDER BY Salary DESC
```

liefert den Zunamen und das Gehalt von allen Mitarbeitern, die 1970 oder früher geboren sind, sortiert nach dem Gehalt in der umgekehrten Reihenfolge (höchstes zuerst).

```
SELECT * FROM Employees
WHERE LastName = 'Fleißig' AND FirstName LIKE 'H%'
```

liefert alle Daten derjenigen Mitarbeiter, deren Zuname Fleißig ist und deren Vorname mit H beginnt.

### **4.3.3 Datenbank-Zugriffe in Java (JDBC)**

Mit Hilfe der "Java Database Connectivity" (JDBC) kann man innerhalb von Java-Programmen auf Datenbanken zugreifen und Daten abfragen, speichern oder verändern, wenn das Datenbanksystem die "Standard Query Language" SQL verwendet, was bei allen wesentlichen Datenbanksystemen seit den 80er-Jahren der Fall ist.

Im Java-Programm werden nur die logischen Eigenschaften der Datenbank und der Daten angesprochen (also nur die Namen und Typen der Relationen und Datenfelder), und die Datenbank-Operationen werden in der genormten Abfragesprache SQL formuliert.

Das Java-Programm ist somit von der Hard- und Software des Datenbanksystems unabhängig. Erreicht wird dies durch einen sogenannten "Treiber" (Driver), der zur Laufzeit die Verbindung zwischen dem Java-Programm und dem Datenbanksystem herstellt - ähnlich wie ein Drucker-Treiber die Verbindung zwischen einem Textverarbeitungsprogramm und dem Drucker oder zwischen einem Graphikprogramm und dem Plotter herstellt. Falls die Datenbank auf ein anderes System umgestellt ist, braucht nur der Driver ausgetauscht werden, und die Java-Programme können ansonsten unverändert weiter verwendet werden.

Der JDBC-Driver kann auch über das Internet bzw. Intranet vom Java-Client direkt auf den Datenbank-Server zugreifen, ohne dass man eine eigene Server-Applikation (CGI oder Servlet) schreiben muss.

Das JDBC ist bereits im Java Development Kit JDK enthalten (ab 1.1), und zwar im Package `java.sql`.

Die Software für die Server-Seite und die JDBC-Driver auf der Client-Seite müssen vom jeweiligen Software-Hersteller des Datenbanksystems erworben werden (also z.B. von der Firma Oracle). Dafür gibt es dann eigene Packages wie z.B. `sun.jdbc` oder `com.firma.produkt`. Das JDK enthält auch eine sogenannte JDBC-ODBC-Brücke für den Zugriff auf ODBC-Datenbanken (Open Database Connectivity, z.B. MS-Access).

---

### 4.3.4 Die wichtigsten Programmelemente

Die Klassen und Interfaces für JDBC liegen im **Paket** `java.sql`, dieses Paket muss daher importiert werden:

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

Damit die Verbindung zur Datenbank hergestellt werden kann, muss zur Laufzeit ein entsprechender **JDBC-Treiber** zum Java-Laufzeitsystem dazu geladen werden. Dies erfolgt mit der Methode `forName` in der Klasse `Class`:

```
Class.forName("drivername");
```

Eine Verbindung zur Datenbank (**Connection**) wird mit der Methode `getConnection` in der Klasse `DriverManager` erzeugt. Als ersten Parameter gibt man an, auf welche Datenbank über welche Methode zugegriffen werden soll. Wenn eine User-Verwaltung in der Datenbank eingerichtet ist, gibt man als zweiten und dritten Parameter den Username und das Passwort an, die festlegen, auf welche Daten der Zugriff erlaubt ist:

```
Connection con = DriverManager.getConnection  
("methode:datenbankname", "username", "passwort");
```

Innerhalb der `Connection` richtet man ein oder mehrere sogenannte **Statements** ein, die festlegen, wie die SQL-Befehle an die Datenbank gesendet werden. Meistens ist dies ein einfaches Statement, das man mit der Methode `createStatement` erzeugt:

```
Statement stmt = con.createStatement();
```

In Spezialfällen (z.B. wenn eine große Anzahl von Datenänderungen sehr effizient erfolgen soll) verwendet man ein `PreparedStatement` oder ein `CallableStatement` anstelle des einfachen `Statement`.

Innerhalb dieses `Statement` kann man dann **SQL-Befehle** zur Datenbank senden. Wenn es sich um eine Abfrage mit `SELECT` handelt, dann mit

```
ResultSet rs = stmt.executeQuery("sql-befehl");
```

wobei das **ResultSet** das Ergebnis der Abfrage in Form einer Tabelle enthält; bei allen anderen SQL-Befehlen mit

```
int n = stmt.executeUpdate("sql-befehl");
```

wobei der Rückgabewert bloß die **Anzahl** angibt, wie viele Änderungen in der Datenbank durchgeführt wurden. Der SQL-Befehl wird in beiden Fällen als String angegeben bzw. - je nach den Eingaben des Benutzers - aus einzelnen Teilen zusammengesetzt.

Diese Programmelemente werden im Folgenden an Hand von Beispielen genauer beschrieben.

---

### 4.3.5 Programmaufbau und Fehlerbehandlung

Die Klassen und Methoden für die Verwendung von Datenbanken liegen im Paket `java.sql`. Ein Datenbank-Programm hat im einfachsten Fall den folgenden Aufbau:

```
import java.sql.*;
public class Xxxx {
    public static void main (String[] args) {
        ...
    }
}
```

Bei allen Datenbank-Operationen gibt es zwei Möglichkeiten, die man im Programm behandeln muss:

- entweder der Methodenaufruf liefert ein **Ergebnis**, und die Verarbeitung kann fortgesetzt werden,
- oder die Methode liefert eine **Ausnahme-Situation** (englisch *exception*), die extra behandelt werden muss, also zum Beispiel "Keine Verbindung zur Datenbank" oder "Falsches Passwort" oder "Fehler im SQL-Statement" oder "Zugriff auf diese Daten nicht erlaubt".

Dementsprechend muss der Programmablauf in zwei Teile geteilt werden:

- einen **try**-Block (englisch für "versuchen") für den normalen Programm-Ablauf
- und einen **catch**-Block (englisch für "Fehler abfangen") für die Behandlung der Ausnahme-Situationen.

Im einfachsten Fall wird im `catch`-Block nur die Fehlermeldung auf den Bildschirm geschrieben und die Verarbeitung mit der `exit`-Methode abgebrochen:

```
try {
    ...
    // der normale Ablauf, Details siehe unten ...
    ...
} catch (Exception e) {
    System.out.println("*** Fehler: " + e);
    System.exit(1);
}
```

---

### 4.3.6 Beispiel für Datenabfragen

Innerhalb des try-Blockes müssen die folgenden Schritte ausgeführt werden:

Bevor wir irgendetwas mit der Datenbank tun können, muss der für diese Datenbank benötigte **Datenbank-Treiber** zum Java-Laufzeitsystem dazu geladen werden:

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

Nun können wir die Verbindung zur Datenbank aufbauen und bekommen damit ein Objekt vom Typ **Connection**:

```
String username="admin";
String password="geheim";
Connection con = DriverManager.getConnection
    ("jdbc:odbc:databasename", username, password);
```

Die Verbindung zur Datenbank wird dabei durch eine Art URL (Uniform Resource Locator) angegeben, der zunächst die Java-Methode JDBC, dann die Art des Datenbank-Treibers (in unserem Fall ODBC) und schließlich den Namen der Datenbank enthält (im Fall von ODBC den so genannten Datenquellennamen, unter dem die Datenbank im ODBC-Manager registriert wurde).

Falls die Datenbank keinen Datenschutz über Usernames und Passwörter hat, gibt man in der Methode getConnection nur einen Parameter an:

```
Connection con = DriverManager.getConnection
    ("jdbc:odbc:databasename");
```

Standardmäßig sind in einer Datenbankverbindung alle Operationen erlaubt, also sowohl Abfragen als auch Änderungen an der Datenbank. Falls man nur Daten abfragt und keine Datenänderungen durchführen will, ist es günstig, die Datenbankverbindung in den Lese-Modus zu setzen:

```
con.setReadOnly(true);
```

Als nächstes müssen wir angeben, wie wir die SQL-Befehle an die Datenbank senden wollen. Dafür gibt es drei Möglichkeiten: ein einfaches Statement, ein PreparedStatement oder ein CallableStatement. Die beiden letztgenannten sind nur dann notwendig, wenn man spezielle Performance-Optimierungen braucht, in unserem Fall genügt ein einfaches **Statement**:

```
Statement stmt = con.createStatement();
```

Innerhalb dieses Statement können nun **SQL-Befehle** an die Datenbank gesendet werden, einer nach dem anderen. Wenn es sich um ein SELECT-Statement handelt, muss dafür die Methode executeQuery verwendet werden, die uns die Ergebnisse der Datenbank-Abfrage als ResultSet liefert:

```
ResultSet rs = stmt.executeQuery
    ("SELECT LastName, Salary, Age FROM Employees");
```

Das Ergebnis der Abfrage ist eine Tabelle, die genau die gewünschten Daten enthält. Diese Tabelle wird im Java-Programm als ein Objekt vom Typ **ResultSet** angesprochen.

---

Für jeden in der Datenbank gefundenen Mitarbeiter enthält unser ResultSet eine Zeile, die für diesen Mitarbeiter die im SELECT-Statement gewünschten drei Datenfelder enthält.

Mit der Methode `next` können wir dieses Ergebnis Zeile für Zeile durchgehen, d.h. wir können der Reihe nach auf den ersten, zweiten usw. Mitarbeiter zugreifen. Die Methode `next` liefert `true`, wenn es noch eine Zeile im Ergebnis gibt, und `false`, wenn es keine weitere Zeile mehr gibt, also wenn wir bereits alle Mitarbeiter verarbeitet haben. Dies eignet sich für eine `while`-Schleife:

```
while (rs.next()) {  
    ...  
}
```

Innerhalb der Schleife sind wir jeweils auf eine Zeile des Ergebnisses positioniert (also auf einen Mitarbeiter) und können nun mit `get`-Methoden die einzelnen Datenfelder für diesen Mitarbeiter ins Java-Programm herein holen und dann verarbeiten.

Mit `getString(1)` bekommen wir das erste im SELECT-Statement angeführte Datenfeld (in unserem Fall `LastName`) als Text vom Typ `String`, mit `getDouble(2)` bekommen wir das zweite Feld als Zahl vom Typ `double`, und so weiter:

```
while (rs.next()) {  
    String name = rs.getString(1);  
    double salary = rs.getDouble(2);  
    int age = rs.getInt(3);  
    ...  
    // Daten verarbeiten  
    ...  
}
```

In unserem Fall müssen wir die Daten gar nicht in Variablen zwischenspeichern, sondern geben sie gleich mit `print`- und `println`-Befehlen auf den Bildschirm aus:

```
System.out.println("List of all employees:");  
while (rs.next()) {  
    System.out.print(" name=" + rs.getString(1) );  
    System.out.print(" salary=" + rs.getDouble(2) );  
    System.out.print(" age=" + rs.getInt(3) );  
    System.out.println();  
}
```

Nun ist die Verarbeitung fertig und wir **schließen** das `ResultSet`, das `Statement` und die `Connection` und beenden damit unsere Verbindung zur Datenbank, damit sie wieder für andere Anwendungen frei ist:

---

```
rs.close();
stmt.close();
con.close();
```

Das **komplette Abfrage-Programm** sieht so aus:

```
import java.sql.*;

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

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String username="admin";
            String password="geheim";
            Connection con = DriverManager.getConnection
                ("jdbc:odbc:databasename", username, password);
            con.setReadOnly(true);
            Statement stmt = con.createStatement();

            ResultSet rs = stmt.executeQuery
                ("SELECT LastName, Salary, Age FROM Employees");
            System.out.println("List of all employees:");
            while (rs.next()) {
                System.out.print(" name=" + rs.getString(1) );
                System.out.print(" salary=" + rs.getDouble(2) );
                System.out.print(" age=" + rs.getInt(3) );
                System.out.println();
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (Exception e) {
            System.out.println("*** Fehler: " + e);
            System.exit(1);
        }
    }
}
```

---

### 4.3.7 Beispiel für Datenänderungen

Der Anfang des Programms ist gleich wie bei den Datenabfragen, nur dass wir diesmal nicht in den Lese-Modus setzen. Dadurch werden auch Veränderungen an der Datenbank zugelassen:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String username="admin";
String password="geheim";
Connection con = DriverManager.getConnection
    ("jdbc:odbc:databasename", username, password);
Statement stmt = con.createStatement();
```

Für alle **SQL-Befehle**, die etwas an den Daten in der Datenbank ändern, also für alle Befehle außer SELECT, muss die Methode `executeUpdate` verwendet werden. Sie führt die Änderungen aus und liefert uns zur Kontrolle die Anzahl, wie viele Änderungen durchgeführt wurden:

```
int rowCount = stmt.executeUpdate
    ("UPDATE Employees " +
     "SET Salary = 50000.0 WHERE LastName = 'Partl' ");
System.out.println(
    rowCount + "Gehaltserhöhungen durchgeführt.");
```

Zuletzt schließen wir wieder das Statement und die Connection:

```
stmt.close();
con.close();
```

Das **komplette Update-Programm** sieht so aus:

```
import java.sql.*;

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

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String username="admin";
            String password="geheim";
            Connection con = DriverManager.getConnection
                ("jdbc:odbc:databasename", username, password);
            Statement stmt = con.createStatement();
```

---

```

int rowCount = stmt.executeUpdate
    ("UPDATE Employees " +
     "SET Salary = 50000.0 WHERE LastName = 'Part1' ");
System.out.println(
    rowCount + "Gehaltserhöhungen durchgeführt.");

stmt.close();
con.close();
} catch (Exception e) {
    System.out.println("*** Fehler: " + e);
    System.exit(1);
}
}
}

```

### 4.3.8 Übungsbeispiele

#### Vorbereitung

Legen Sie - zum Beispiel mit MS-Access - eine Datenbank an, die eine Tabelle "Mitarbeiter" mit den folgenden Datenfeldern enthält: Abteilung (Text, String), Vorname (Text, String), Zuname (Text, String), Geburtsjahr (Zahl, int), Gehalt (Zahl, double).

Füllen Sie diese Tabelle mit ein paar Datenrecords, etwa wie im Beispiel im Kapitel über relationale Datenbanksysteme.

Registrieren Sie diese Datenbank auf Ihrem Rechner, zum Beispiel mit dem ODBC-Manager der MS-Windows Systemsteuerung als DSN (Datenquellennamen). Dann können Sie den einfachen JDBC-ODBC-Driver verwenden, der im JDK enthalten ist (aber manchmal unerklärliche Fehler liefert).

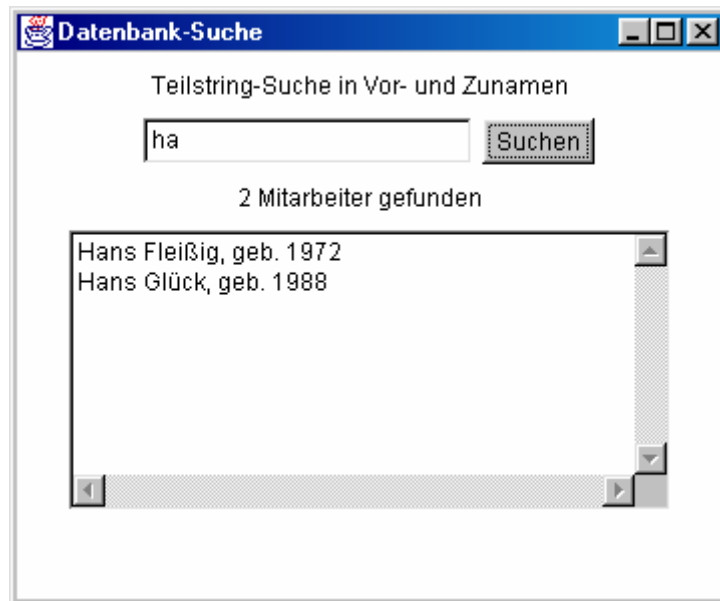
Bei der Verwendung von anderen Datenbanksystemen müssen Sie sicherstellen, dass ein entsprechender JDBC-Driver verfügbar ist, und die Datenbank dem entsprechend anlegen und registrieren.

#### Liste der Geburtsjahre

Schreiben Sie eine einfache Java-Applikation, die eine Liste aller Mitarbeiter mit Vorname und Geburtsjahr auf den Bildschirm ausgibt.

#### Graphisches User-Interface zur Suche in der Datenbank

Schreiben Sie ein graphisches User-Interface, mit dem man nach Mitarbeitern suchen kann. Der User soll einen Teil des Vornamens oder Zunamens eingeben und dann eine Liste von allen zutreffenden Mitarbeitern erhalten, jeweils mit Vorname, Zuname und Geburtsjahr, sowie die Anzahl der gefundenen Mitarbeiter.



Das Graphische User-Interface besteht aus einem Fenster (Klasse Frame), das die folgenden Komponenten enthält:

- eine Textzeile (Klasse Label) für die Überschrift,
- ein einzeliges Eingabefeld (Klasse TextField) für die Eingabe des Suchstrings,
- einen Such-Button (Klasse Button),
- eine Textzeile (Klasse Label) für die Anzeige der Anzahl und
- ein mehrzeiliges Ausgabefeld (Klasse TextArea) für die Anzeige der Mitarbeiter.

Die Aktion der Suche startet, wenn der User entweder im Eingabefeld die Enter-Taste drückt oder mit der Maus auf den Such-Button klickt. Um die Datenbank nicht zu lange zu belegen, soll die Datenbank-Verbindung nach jeder Abfrage sofort wieder geschlossen werden. Etwaige Fehlermeldungen sollen in das Ausgabefeld geschrieben werden.

Das SQL-Statemet hat den folgenden Aufbau, wobei für xxx die User-Eingabe eingesetzt werden muss:

```
SELECT Vorname, Zuname, Geburtsjahr FROM Mitarbeiter
WHERE Vorname LIKE '%xxx%' OR Zuname LIKE '%xxx%'
ORDER BY Vorname, Zuname
```

Hinweis: Wenn der User einen Leerstring als Suchmuster eingibt, erhält er die komplette Liste aller Mitarbeiter.

---

## 5 Weitere Informationen

### 5.1 Online-Dokumentation

Die Online-Dokumentation der Klassenbibliothek kann entweder direkt am Java-Server der Firma Sun gelesen werden (siehe Web-Sites) oder von dort heruntergeladen und lokal am eigenen Rechner installiert oder auf einem Web-Server gespeichert werden. Die Klassenbibliothek wird in diesem Zusammenhang als das so genannte Application Programming Interface API bezeichnet.

Die Dokumentation der Klassenbibliothek liegt in der Form von Web-Pages (HTML-Files mit Hypertext-Links) vor und kann mit jedem beliebigen Web-Browser gelesen werden, z.B. mit Netscape oder Internet-Explorer.

Die wichtigsten Files innerhalb dieser Dokumentation sind:

- **Overview** für einen Überblick über alle Pakete
- **Tree** (Class Hierarchy) für die Suche nach einer bestimmten Klasse und deren Beschreibung mit allen ihren Konstruktoren, Datenfeldern und Methoden.
- **Index** (All Names, Index of Fields and Methods) für die Suche nach Datenfeldern und Methoden, wenn man nicht weiß, in welcher Klasse sie definiert sind.

In jedem dieser Files kann man mit dem "Find-in-current" Befehl des Web-Browsers (je nach Browser z.B. mit Strg-F oder im Edit-Menü) nach Namen oder Substrings **suchen**.

Das Index-File ist sehr groß und deshalb in 26 Einzelfiles nach den Anfangsbuchstaben aufgeteilt. Meist kommt man schneller zum Ziel, wenn man die Suche im Tree der Klassen-Hierarchie beginnt.

Wenn Sie einen **Klassennamen** anklicken, erhalten Sie die komplette Beschreibung dieser Klasse und ihrer Konstruktoren, Datenfelder und Methoden sowie eine Angabe ihrer Oberklassen. Falls Sie eine Methode in der Dokumentation einer Unterklasse nicht finden, dann sehen Sie in ihren Oberklassen nach (siehe Vererbung).

Wenn Sie den Namen einer **Methode** oder eines **Datenfeldes** anklicken, erhalten Sie sofort die Beschreibung dieser Methode bzw. des Datenfeldes. Dabei müssen Sie aber beachten, dass es mehrere gleichnamige Methoden sowohl innerhalb einer Klasse als auch in verschiedenen Klassen geben kann.

Zusätzlich zur Dokumentation der Klassenbibliothek (API) gibt es am Java-Server von Sun auch die Dokumentation der Sprachdefinition (language specification) zum Lesen oder Herunterladen, und Sie finden dort auch zahlreiche weitere Online-Informationen und Tutorials (siehe Web-Sites und Bücher).

---

## 5.2 Dokumentation der eigenen Programme

Mit dem Hilfsprogramm **javadoc**, das Teil des JDK ist, kann man eine Dokumentation der eigenen Java-Programme im HTML-Format erzeugen, im gleichen Stil und Aussehen wie die Online-Dokumentation (API) der Klassenbibliothek.

Der Aufruf von

```
javadoc *.java
```

erzeugt die komplette Dokumentation für alle Java-Programme im aktuellen Directory. Man erhält für jedes Source-Programm `Xxxx.java` ein Dokumentations-File `Xxxx.html` sowie alle Übersichts-Files für den Überblick (Overview), die Klassenhierarchie (Tree), den Index, die Liste der Pakete, die Liste der Klassen usw. und eine Datei `index.html` für den Einstieg in die Dokumentation.

Vorsicht, bereits existierende gleichnamige Dateien werden dabei überschrieben, und der Vorgang funktioniert nur, wenn alle Source-Programme fehlerfrei sind.

Javadoc extrahiert automatisch alle public Deklarationen von Klassen, Datenfeldern, Konstruktoren und Methoden.

Zusätzliche Erklärungen kann man in Form von speziellen Kommentaren hinzufügen, die jeweils in `/**` und `*/` eingeschlossen werden, unmittelbar vor der jeweiligen Deklaration stehen müssen und einfachen HTML-Text enthalten können. Dieses Prinzip, dass der Text der Programm-Dokumentation als Kommentar-Zeilen in die Source-Programme integriert ist, wird auch als "Literate Programming" bezeichnet und hat den Vorteil, dass man bei Änderungen an den Programmen gleich auch die Programmbeschreibung entsprechend ändern kann.

Die erzeugte Programm-Dokumentation kann man mit dem Web-Browser lesen, unter MS-Windows z.B. durch Doppelklick auf den Dateinamen `index.html` im Windows-Explorer oder mit dem Befehl

```
start index.html
```

---

## 5.3 Web-Sites und Bücher über Java

Englischsprachige Web-Sites:

- Online-Dokumentation (API) des JDK, lokal oder auf <http://java.sun.com/docs/>
- **<http://java.sun.com/>** und <http://www.javasoft.com/>
- <http://java.sun.com/docs/books/tutorial/index.html>
- <http://sunsite.unc.edu/javafaq/javafaq.html>
- <http://mindprod.com/gloss.html>
- <http://www.bruceeckel.com/>

Deutschsprachige Web-Sites:

- <http://www.dclj.de/>
- **<http://www.boku.ac.at/javaeinf/>**
- <http://www.gkrueger.com/>
- <http://www.tutego.com/javabuch/>
- <http://www.selfjava.de/>

Englischsprachige Newsgruppen:

- `comp.lang.java.*`

Deutschsprachige Newsgruppe:

- `de.comp.lang.java`

Bücher:

- Guido Krüger: "Handbuch der Java-Programmierung", Addison-Wesley (vorherige Titel: "Java 1.1 lernen" und "Go to Java 2")
- Chrisitan Ullenbohm: "Java ist auch eine Insel", Galileo
- "Java ... in a Nutshell", O'Reilly (mehrere Bände!)
- und viele andere ...

---

## 6 Musterlösungen zu den Übungen

Die Musterlösungen zu den praktischen Übungen erhalten Sie **nach** der Durchführung der Übungen am Ende des Kurses.

---

## 6.1 Musterlösung EinfRech

```
public class EinfRech {  
    public static void main (String[] args) {  
        int a = 6;  
        int b = 7;  
        int c = a * b;  
        System.out.println(c);  
    }  
}
```

---

## 6.2 Musterlösung EinfVergl

```
public class EinfVergl {
    public static void main (String[] args) {
        int a = 6;
        int b = 7;
        if ( a > b ) {
            System.out.println("ja, a ist größer.");
        }
        else {
            System.out.println("nein, a ist nicht größer.");
        }
    }
}
```

---

## 6.3 Musterlösung Quadrat

```
public class Quadrat {
    public static void main (String[] args) {
        System.out.println(" Quadratzahlen:");
        for ( int zahl=1; zahl<=20; zahl=zahl+1 ) {
            int quad = zahl * zahl;
            System.out.println (zahl + " * " + zahl +
                " = " + quad );
        }
    }
}
```

---

## 6.4 Musterlösung einfache Person

### 6.4.1 Klasse Person

```
public class Person {  
  
    public String vorname;  
    public String zuname;  
  
    public String toString() {  
        return this.vorname + " " + this.zuname;  
    }  
}
```

### 6.4.2 main-Methode

```
public class PersTest {  
    public static void main (String[] args) {  
  
        Person anna = new Person();  
        anna.vorname = "Anna";  
        anna.zuname = "Schlosser";  
  
        Person georg = new Person();  
        georg.vorname = "Georg";  
        georg.zuname = "Fischer";  
  
        System.out.println( anna.vorname );  
        System.out.println( georg.vorname );  
        System.out.println( anna );  
        System.out.println( georg );  
    }  
}
```

---

## 6.5 Musterlösung einfacher Student

### 6.5.1 Klasse Student

```
public class Student extends Person {  
  
    public String uni;  
  
    public String toString() {  
        String s;  
        s = vorname + " studiert an " + uni;  
        return s;  
    }  
}
```

### 6.5.2 Änderungen in der main-Methode

```
public class PersTest {  
    public static void main (String[] args) {  
        ...  
        Student georg = new Student();  
        georg.vorname = "Georg";  
        georg.zuname = "Fischer";  
        georg.uni = "BOKU";  
        ...  
    }  
}
```

---

## 6.6 Musterlösung Analyse und Design Konto

### 6.6.1 Analyse

- Jede Person hat einen Vornamen.
- Jede Person hat einen Zunamen.
- Manche Personen haben ein oder mehrere Konten, manche haben kein Konto.
- Jeder Student ist eine Person.
- Jedes Konto gehört einer Person (Inhaber).
- Jedes Konto hat einen Kontostand (Guthaben).
- Auf ein Konto kann man einen Geldbetrag einzahlen.
- Von einem Konto kann man einen Geldbetrag abheben.

### 6.6.2 Design

- Klasse Person
  - Eigenschaften: Vorname, Zuname, evtl. Liste der Konten
- Klasse Student
  - Unterklasse von Person
- Klasse Konto
  - Eigenschaften: Inhaber (Person), Guthaben
  - Methoden: einzahlen, abheben

---

## 6.7 Musterlösung einfaches Konto

### 6.7.1 Klasse Konto

```
public class Konto {  
  
    public Person inhaber;  
    public double guthaben = 0.0;  
  
    public void einzahlen (double betrag) {  
        guthaben = guthaben + betrag ;  
    }  
  
    public void abheben (double betrag) {  
        guthaben = guthaben - betrag ;  
    }  
  
    public String toString() {  
        return inhaber + ": " + guthaben + " Euro";  
    }  
  
}
```

---

## 6.7.2 main-Methode

```
public class KontoTest {

    public static void main (String[] args) {

        Person hans = new Person();
        hans.vorname = "Hans";
        hans.zuname = "Reich";

        Konto hansKonto = new Konto();
        hansKonto.inhaber = hans;

        System.out.println(hansKonto);
        hansKonto.einzahlen(1000.0);
        System.out.println(hansKonto);
        hansKonto.abheben(200.0);
        System.out.println(hansKonto);

        Student anna = new Student();
        anna.vorname = "Anna";
        anna.zuname = "Fleißig";
        anna.uni = "TU Wien";

        Konto annasKonto = new Konto();
        annasKonto.inhaber = anna;

        System.out.println(annasKonto);
        annasKonto.einzahlen(600.0);
        System.out.println(annasKonto);

    }
}
```

---

## 6.8 Musterlösung Person als Java Bean

### 6.8.1 Klasse Person

```
public class Person {

    private String vorname;
    private String zuname;

    public void setZuname (String s) {
        this.zuname = s;
    }
    public String getZuname() {
        return this.zuname;
    }

    public void setVorname (String s) {
        this.vorname = s;
    }
    public String getVorname() {
        return this.vorname;
    }

    public String toString() {
        return this.vorname + " " + this.zuname;
    }

}
```

---

## 6.8.2 main-Methode

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

        Person anna = new Person ();
        anna.setVorname("Anna");
        anna.setZuname("Schlosser");

        Person georg = new Person ();
        georg.setVorname("Georg");
        georg.setZuname("Fischer");

        System.out.println();
        System.out.println("Vornamen:");
        System.out.println( anna.getVorname() );
        System.out.println( georg.getVorname() );
        System.out.println();
        System.out.println("Komplette Informationen:");
        System.out.println( anna );
        System.out.println( georg );
        System.out.println();
    }
}
```

---

## 6.9 Musterlösung Student als Java Bean

### 6.9.1 Klasse Student

```
public class Student extends Person {

    private String uni;

    public void setUni (String s) {
        this.uni = s;
    }
    public String getUni() {
        return this.uni;
    }

    public String toString() {
        String s;
        s = this.getVorname() + " studiert an " + this.uni;
        return s;
    }
}
```

---

## 6.9.2 main-Methode

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

        Person anna = new Person ();
        anna.setVorname("Anna");
        anna.setZuname("Schlosser");

        Student georg = new Student ();
        georg.setVorname("Georg");
        georg.setZuname("Fischer");
        georg.setUni("BOKU");

        System.out.println();
        System.out.println("Vornamen:");
        System.out.println( anna.getVorname() );
        System.out.println( georg.getVorname() );
        System.out.println();
        System.out.println("Komplette Informationen:");
        System.out.println( anna );
        System.out.println( georg );
        System.out.println();
    }
}
```

---

## 6.10 Musterlösung Konto als Java Bean

### 6.10.1 Klasse Konto

```
public class Konto {

    private Person inhaber;
    private double guthaben = 0.0;

    public void setInhaber (Person neuerInhaber) {
        this.inhaber = neuerInhaber;
    }

    public Person getInhaber() {
        return this.inhaber;
    }

    private void setGuthaben (double neuerBetrag) {
        if (neuerBetrag >= 0.0) {
            this.guthaben = neuerBetrag;        }
    }
    public double getGuthaben() {
        return this.guthaben;
    }

    public void einzahlen (double betrag) {
        this.setGuthaben( this.guthaben + betrag );
    }

    public void abheben (double betrag) {
        this.setGuthaben( this.guthaben - betrag );
    }

    public String toString() {
        return this.inhaber + ": " + this.guthaben + " Euro";
    }

}
```

---

## 6.10.2 main-Methode

```
public class KontoTest {

    public static void main (String[] args) {

        Person hans = new Person();
        hans.setVorname("Hans");
        hans.setZuname("Reich");

        Konto hansKonto = new Konto();
        hansKonto.setInhaber(hans);
        System.out.println(hansKonto);
        hansKonto.einzahlen(1000.0);
        System.out.println(hansKonto);
        hansKonto.abheben(200.0);
        System.out.println(hansKonto);

        Student anna = new Student();
        anna.setVorname("Anna");
        anna.setZuname("Fleißig");
        anna.setUni("TU Wien");

        Konto annasKonto = new Konto ();
        annasKonto.setInhaber(anna);
        System.out.println(annasKonto);
        annasKonto.einzahlen(600.0);
        System.out.println(annasKonto);

    }
}
```

---

## 6.11 Musterlösung Geburtsjahr

```
import java.sql.*;

public class GebJahr {
    public static void main (String[] args) {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:odbc:TestDB1");
            con.setReadOnly(true);
            Statement stmt = con.createStatement();

            ResultSet rs = stmt.executeQuery
                ("SELECT Vorname, Geburtsjahr FROM Mitarbeiter");
            while (rs.next()) {
                System.out.println( rs.getString(1) + " "
                    + rs.getInt(2) );
            }

            rs.close();
            stmt.close();
            con.close();
        } catch (Exception e) {
            System.out.println("*** Fehler: " + e);
        }
    }
}
```

---

## 6.12 Musterlösung SuchFrame

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

public class SuchFrame extends Frame
    implements ActionListener, WindowListener {

    private String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    private String database = "jdbc:odbc:TestDB1";

    private Label ueberschrift = new Label
        ("Teilstring-Suche in Vor- und Zunamen");
    private TextField eingabe = new TextField(20);
    private Button suchButton = new Button ("Suchen");
    private Label ergebnis = new Label
        ("0 Mitarbeiter gefunden");
    private TextArea ausgabe = new TextArea(8,40);
    private FlowLayout layout = new FlowLayout();

    public void init() {
        try {
            Class.forName( driver );
        } catch (Exception e) {
            ausgabe.setText("Fehler: " + e);
        }
        this.setTitle("Datenbank-Suche");
        this.setLayout(layout);
        this.setSize(360, 300);
        this.add(ueberschrift);
        this.add(eingabe);
        this.add(suchButton);
        this.add(ergebnis);
        this.add(ausgabe);
        eingabe.addActionListener(this);
        suchButton.addActionListener(this);
        this.addWindowListener(this);
    }

    public void actionPerformed (ActionEvent event) {
        // Enter im TextField oder Klick auf den Button
        String suchString = eingabe.getText().trim();
        suchen (suchString);
    }
}
```

---

```

private void suchen (String x) {
    int count = 0; // bisher nichts gefunden
    ausgabe.setText(""); // alte Ausgabe löschen
    String sql =
        "SELECT Vorname, Zuname, Geburtsjahr FROM Mitarbeiter "
        + " WHERE Vorname LIKE '%" + x + "%' "
        + " OR Zuname LIKE '%" + x + "%' "
        + " ORDER BY Vorname, Zuname ";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        con = DriverManager.getConnection ( database );
        stmt = con.createStatement();
        rs = stmt.executeQuery ( sql );
        while (rs.next()) {
            String zeile = rs.getString(1) + " "
                + rs.getString(2) + ", geb. " + rs.getInt(3);
            ausgabe.append( zeile + "\n" );
            count = count + 1;
        }
    } catch (Exception e) {
        ausgabe.append( "Fehler: " + e + "\n");
    }
    ergebnis.setText( count + " Mitarbeiter gefunden");
    // alles schließen, was eventuell geöffnet wurde
    try {rs.close(); } catch (Exception ex) {}
    try {stmt.close(); } catch (Exception ex) {}
    try {con.close(); } catch (Exception ex) {}
}

public void windowClosing (WindowEvent e) {
    this.setVisible(false);
    System.exit(0);
}

public void windowClosed (WindowEvent e) { }
public void windowOpened (WindowEvent e) { }
public void windowIconified (WindowEvent e) { }
public void windowDeiconified (WindowEvent e) { }
public void windowActivated (WindowEvent e) { }
public void windowDeactivated (WindowEvent e) { }

public static void main (String[] args) {
    SuchFrame fenster = new SuchFrame();
    fenster.init();
    fenster.setVisible(true);
}
}

```