1 Kostenlose Literatur

0)
Online-Buch
http://www.tigerjython.ch

1) Die Java Spezifikation
The Java Language Specification kann man downloaden unter:
http://java.sun.com/docs/books/jls/

2) Java ist auch eine Insel
Im Browser als Stichwort "Java Insel Galileo" eingeben

3) Buch von Guido Krüger
Im Browser als Stichwort "Guido Krüger Download" eingeben

4) Hubert Partl
http://www.boku.ac.at/javaeinf/

5)
http://www.dpunkt.de/java/

6) Buch von Bruce Eckel
"Bruce Eckel Thinking in Java"
Im Browser als Stichwort "Bruce Eckel Thinking in Java" eingeben

Hilfe bei Problemen:
www.java-forum.org
http://www.netbeans-forum.de/
www.stackoverflow.com

Links zu UML:
http://www.jeckle.de/umllinks.htm

2 Begriffe (Java intern)

1) Java-Compiler
Der Java-Compiler (Compiler = javac.exe) übersetzt den Quellcode (normalerweise in einer Datei mit der Endung "java" gespeichert) in den sogenannten Bytecode (normalerweise in einer Datei mit der Endung class gespeichert).

2) Java-Interpreter
Dieser Bytecode wird vom Java-Interpreter, auch JVM (Java Virtual Machine) genannt, (Interpreter = java.exe) ausgeführt. Theoretisch ist ein und derselbe Bytecode auf den verschiedensten Plattformen lauffähig.
Praktisch gibt es Ausnahmen:
1. Beispiel:
Unter Windows wird "\" als Verzeichnistrenner (in einem Pfad) benutzt.
Unter Windows darf deshalb "\" in Dateinamen nicht vorkommen, unter Linux ist es dagegen erlaubt.

2. Beispiel:
In einem Java-Programm ist es unter Windows es möglich unten Rechts auf dem Desktop in der sogenannten Taskbar ein Symbol zu erzeugen.
Unter Linux kann man sich aber nicht darauf verlassen dass dort der Windowmanager so eine Taskbar überhaupt besitzt.

3) JRE
JRE (Java Runtime Enviroment) ist JVM plus die Klassen.

4) JDK
JDK (Java Developement Kit) ist im Prinzip ein JRE plus Compiler (Compiler = javac.exe).
Zum Programmieren braucht man also das JDK, zum Ausführen reicht das JRE.
Der JIT-Compiler ist in die VM integriert, also im Prinzip unabhängig von JRE und JDK.

3 Variable und Initialisierungen

Es gibt in Java zwei verschiedene Arten von Variablen:
- Variable, die Werte von primitiven Datentypen speichern
- Variable, die Referenzen (Verweise) auf Objekte speichern

3.1 Primitive Datentypen und ihre Werte

Datentyp	Bereich	Speicher-grösse
boolean	true, false	?
char	\u0000 bis \uffff (dezimal: 0 bis 65535) Beipiele: 'a', \u41, \uff	2 Byte
byte	-128 bis 127	1 Byte
short	-32768 bis 32767	2 Byte
int	-2147483648 bis 2147483647	4 Byte
long	-9223372036854775808 bis 9223372036854775807	8 Byte
float	Zahlen mit 32 Bit Genauigkeit Beipiel: 13.4f (Angabe von f ist notwendig)	4 Byte
double	Zahlen mit 64 Bit Genauigkeit Beipiel: 13.5 oder 13.5d	8 Byte

Bemerkung:
Die JLS (Java Language Specification) definiert nicht wie viele Byte zum Speichern von Werten der primitiven Datentypen benötigt werden. Es wird lediglich definiert welche Wertebereiche diese haben sollen, und so folgt z.B. dass man wenigstens 32-Bit also vier Byte braucht um alle möglichen Werte eines Integers abbilden zu können.
Speziell wird auch keine Angabe darüber gemacht, wie viele Bytes für eine Adresse (Referenzwert) benötigt werden.

Beispiele:
```
double d1;          // Deklaration
int i1, i2;         // Deklaration
long l1 = -12345;   // Deklaration und Initialisierung
```

3.2 Referenzvariablen

Der Wert (= Referenzwert) einer Referenzvariablen ist ein Verweis (Referenz) auf ein Objekt.
Technisch gesehen wird dieser Verweis durch eine Adresse realisiert.
Der Wert einer Referenzvariablen ist also technisch gesehen eine Adresse.
Wurde noch kein Objekt erzeugt und der Referenzvariablen zugewiesen, bekommt die Referenzvariable den Wert null.
Wurde ein Objekt erzeugt und der Referenzvariablen zugewiesen, bekommt die Referenzvariable als Wert die Adresse dieses Objekts.

Beispiel:

1) Folgende Deklaration erzeugt die Variable k, die initialisiert wird.
Es wird also ein Objekt erzeugt (mit dem Schlüsselwort new und der Angabe des Klassennamens Kuh mit der Parameterliste wird der Konstruktor aufgerufen) und der Verweis auf dieses Objekt wird in der Variable k gespeichert.

```
Kuh k=new Kuh();
```

3.3 Initialisierungen

Eine Initialisierung ist die erste Zuweisung für eine Variable.

1) Jede lokale Variable (innerhalb von Methoden) hat (wenn ihr kein Wert zugewiesen wurde) einen unbestimmten, dem Programmierer nicht bekannten Wert.
Deshalb muss sie vom Programmierer vor einer lesenden Verwendung initialisiert werden, da es sonst einen Fehler beim Kompilieren gibt.

2) Wenn dagegen mit new ein Objekt einer Klasse erzeugt wird, werden automatisch (d.h. ohne Zutun des Programmierers) alle Attribute (bzw. wenn das Objekt ein Array ist: alle Komponenten = Elemente = Zellen dieses Arrays) dieses gerade erzeugten Objekts wie folgt initialisiert:

Datentyp	standardmäßige Vorbelegung
Boolean	false
Char	\u0000
Byte	0
Short	0
Int	0
Long	0L
Float	0.0f
Double	0.0
Referenzvariable	Null

3.3.1 Beispiele

3.3.1.1 Beispiel

```
public class MainArray9{
  public static void main(String argv[]){
    Hund h1;
    // Fehler beim Kompilieren: h1 nicht initialisiert
    System.out.println(h1);
    Hund h2 = new Hund();
    // Kein Fehler beim Kompilieren: das Attribut name
    // wurde mit null initialisiert.
    h2.druckeName();
  }
}

class Hund{
  private String name;

  public void druckeName(){
    // Kein Fehler beim Kompilieren: name hat Wert null
    System.out.println(name);
  }
}
```

3.3.1.2 Beispiel

```
public class MainArray10{
  static Hund hh;

  public static void main(String argv[]){
    Hund h;
    // Kein Fehler beim Kompilieren: hh hat Wert null
    System.out.println(hh);
    // Fehler beim Kompilieren: h nicht initialisiert
    System.out.println(h);
  }
}

class Hund{
  private String name;

  public void druckeName(){
    // Kein Fehler beim Kompilieren: name hat Wert null
    System.out.println(name);
  }
}
```

3.4 Namenskonventionen für Variable

Klassen mit Grossbuchstaben beginnen.
Methoden mit Kleinbuchstaben beginnen
Variable mit Kleinbuchstaben beginnen
Konstante komplett mit Grossbuchstaben schreiben.

4 Klassen, Objekte, Konstruktoren

4.1 Beispiel

```
public class TiereTest{
  public static void main(String[] args){
    int kuhAlter;
    int hennenAlter;
    Kuh q1 = new Kuh("Elfriede", 3);
    Kuh q2 = new Kuh();
    // auch möglich:
    Kuh q3;
    q3 = new Kuh("Frida", 4);
    Henne h = new Henne();
    q1.setAlter(7);
    h.setAlter(2);
    kuhAlter = q1.getAlter();
    hennenAlter = h.getAlter();
    System.out.println("Kuhalter q1= "+kuhAlter);
    System.out.println("Kuhalter q2= "+q2.getAlter());
    System.out.println("Kuhalter q3= "+q3.getAlter());
    System.out.println("Hennenalter h= "+hennenAlter);    }
}

class Henne{
  private String name;
  private int alter;
  // Legeleistung: Eieranzahl/Zeiteinheit
  private int legeleistung;

  public void setName(String pname){
    name = pname;
  }

  public String getName(){
    return(name);
  }

  public void setAlter(int palter){
    alter = palter;
  }

  public int getAlter(){
    return(alter);
  }

  public void setLegeleistung(int plegeleistung){
    legeleistung = plegeleistung;
  }

  public int getLegeleistung(){
    return(legeleistung);
  }
}
```

```
class Kuh{
  private String name;
  private int alter;
  // Milchleistung: Liter pro Zeiteinheit
  private double milchleistung;

  public Kuh(String pname, int palter){
      name = pname;
      alter = palter;
  }

  public Kuh(){
    name = "Mustermann";
    alter = 0;
  }

  public void setName(String pname){
    name = pname;
  }

  public String getName(){
    return(name);
  }

  public void setAlter(int palter){
    alter = palter;
  }

  public int getAlter(){
    return(alter);
  }

  public void setMilchleistung(int pmilchleistung){
    milchleistung = pmilchleistung;
  }

  public double getMilchleistung(){
    return(milchleistung);
  }
}
```

4.2 Beschreibung des Beispiels

4.2.1 Klassen, Methoden, Attribute

Eine Klasse enthält Daten (in der Klasse Kuh: name, alter) und Funktionen (in der Klasse Kuh: setAlter, gibAlter).
Die Daten heißen **Attribute** (Membervariable), die Funktionen heissen **Methoden**.
Der Sammelbegriff für Daten und Funktionen ist **Member** (Mitglieder).
In der objektorientierten Programmierung will man nur mit Methoden auf die Attribute zugreifen (lesen bzw. ändern der Attribute).
Damit will man verhindern, dass ein Programmierer direkt auf Attribute zugreift und diese auf unerwünschte Werte setzt. (Im Supermarkt greift man auch nicht direkt auf das Geld (Attribut) in der Kasse zu, sondern macht dies über den Kassierer (Methode)).
Mit Hilfe der Methoden soll man auf die Attribute zugreifen (lesen bzw. ändern der Attribute).

Bemerkung:
Um Attribute der Klasse und formale Parameter einer Methode zu unterscheiden, kann man den formalen Parameter mit dem Buchstaben p (wie Parameter) beginnen, wie z.B:

```
public void setName(String pname){
  name = pname;
}
```

4.2.2 Variable und Initialisierungen

4.2.2.1 Variable, die Werte von primitiven Datentypen speichern

4.2.2.1.1 Erste Möglichkeit

Deklarieren und Nichtinitialisieren der Variablen

```
int kuhAlter;
```

Dadurch wird die Variable kuhAlter angelegt. Da diese vom Programmierer nicht initialisiert wurde und den Datentyp double hat, wird ihr automatisch (ohne Zutun des Programmierers) standardmäßig der Wert 0 zugewiesen (siehe oben "Initialisierungen").

Technisch gesehen:

Variable	Wert
kuhAlter	0

4.2.2.1.2 Zweite Möglichkeit

Deklarieren und Initialisieren der Variablen

```
int kuhAlter = 13;
```

Dadurch wird die Variable kuhAlter angelegt, die zusätzlich noch vom Programmierer initialisiert wurde.

Technisch gesehen:

Variable	Wert
kuhAlter	13

4.2.2.2 Variable, die Referenzen (Verweise) auf Objekte speichern

4.2.2.3 Erste Möglichkeit

1) Deklarieren und Nichtinitialisieren der Variablen:

```
Kuh q;
```

Dadurch wird die Referenzvariable q angelegt. Da diese vom Programmierer nicht initialisiert wurde und eine Referenzvariable ist, wird ihr automatisch (ohne Zutun des Programmierers) standardmäßig der Wert null zugewiesen, was bedeutet, dass sie noch auf kein Objekt zeigt (siehe oben "Initialisierungen").

4.2.2.4 Technisch gesehen

Variable	Wert
q	null

2) Objekt erzeugen und der Variablen zuweisen:
Jetzt will man noch ein Objekt der Klasse Kuh erzeugt werden. Dazu muss der Konstruktor Kuh mit den zwei Parametern aufgerufen werden, die den Namen und das Alter der Kuh festlegen.
Dies kann man z.B. wie folgt machen:

```
q = new Kuh("Frida", 4);
```

4.2.2.5 Technisch gesehen

Variable	Wert
Q	04711

Adresse	Wert
04711	"Frida"
	4

4.2.2.6 Zweite Möglichkeit

Die zwei folgenden Schritte

```
Kuh q;
q = new Kuh("Frida", 4);
```

können auch zu einem zusammengefasst werden:

```
Kuh q = new Kuh("Frida", 4);
```

D.h. die Variable (Exemplarvariable) wird deklariert und initialisiert.

Bemerkung:
Die Variable q wird dynamisch, d.h. während der Laufzeit erzeugt und automatisch durch den sogenannten Garbage Collector wieder entfernt, wenn sie nicht mehr benötigt wird.

4.2.2.7 Konstruktor

Zusätzlich wird automatisch beim Erzeugen der Instanz z.B. der zweiparametrige Konstruktor Kuh("Elfriede", 3) und der parameterlose Konstruktor Henne() aufgerufen.

Ein Konstruktor ist eine spezielle Methode in der Klasse mit dem gleichen Namen wie die Klasse, aber ohne einen Rückgabewert (auch nicht void).
Ein Konstruktor wird automatisch beim Anlegen und Initialisieren einer Objektvariable aufgerufen.

4.2.2.7.1 Standardkonstruktor

Von besonderer Bedeutung ist der parameterlose Konstruktor, der sogenannte Standardkonstruktor:
Implementiert der Programmierer für eine Klasse überhaupt keinen Konstruktor (wie z.B. in der Klasse Henne), dann erzeugt das Java-System automatisch diesen parameterlosen Konstruktor. Dieser Konstruktor hat als einzige Anweisung: **super(),** die den Konstrukor der Oberklasse aufruft.
Außerdem werden sämtliche Attribute (Membervariable) der Klasse mit den Standardwerten initialisiert (siehe Initialisierung).
Falls aber ein nicht parameterloser Konstruktor (d.h. mit mindestens einem Parameter) implementiert wird und ein Objekt einer Klasse angelegt wird, das einen parameterlosen Konstruktor benötigt, muss dieser implementiert werden. Der Standardkonstruktor wird in diesem Fall dann nicht mehr vom Compiler erzeugt !

Um zu klären, was das Überladen von Konstruktoren bedeutet, muss folgendes definiert werden:

4.2.2.7.2 Signatur

Zwei Methoden haben die gleiche Signatur, wenn:
a) die Namen der Methoden sind gleich UND
b) die Reihenfolge (und die Anzahl) der Parameter (mit zugehörigen Typen) sind gleich
Wichtig:
Der Return-Typ (der Datentyp der durch Return zurückgegeben wird) muss nicht gleich sein.

4.2.2.7.3 Überladen von Konstruktoren

Es darf mehrere Konstruktoren (mit dem gleichen Namen) in der gleichen Klasse geben.
Dies nennt man auch das Überladen von Konstruktoren.
Man darf Methoden (und Konstruktoren) überladen, wenn sie den gleichen Namen, aber verschieden Signaturen haben.

Beispiel dreier Konstrukoren in derselben Klasse:
a)
```
public Kuh(String pname, int palter){
  // ...
}
```

b)
```
public Kuh(int palter , String pname){
  // ...
}
```

c)
```
public Kuh(){
  // ...
}
```

Erklärung:
a) unterscheidet sich von b) durch die Reihenfolge und von c) durch die Anzahl der Parameter.

Bemerkung:
Überschreiben (siehe Vererbung) ist bei gleicher Signatur und gleichem Return-Type möglich.

4.2.2.8 Dekonstruktor

Die in Java eingebaute Speicherbereinigung (garbage collection) - umgangssprachlich auch Müllabfuhr genannt - erleichtert die Programmierung ungemein. Dieses Konzept, das z.B. in der funktionalen Programmierung schon längst verwendet wird (nach dem Programmaufruf wird der auf dem Stack reservierte Speicher automatisch wieder freigegeben), nimmt dem Programmierer die Speicherverwaltung des Programms ab. Es ist also im Gegensatz zu C++ nicht mehr nötig, Dekonstruktoren zu den Objekten selbst anzugeben. Da nicht mehr überlegt werden muß, wann ein Dekonstruktor aufzurufen ist, werden hierbei eventuelle Programmierfehler vermieden.
Kurz: In Java gibt es keinen Dekonstruktor

4.2.2.9 Aufgabe

Gegeben sind die Klassen Gerade2D und Punkt2D, die Geraden und Punkte im zweidimensionalen Raum modellieren.
Die Klasse Punkt besteht aus den Attributen x und y, die die Koordinaten im zweidimensionalen Raum beschreiben. Außerdem besteht die Klasse Punkt aus den üblichen get- und set-Methdoen.

Die Klasse Gerade besteht aus den Attributen p1 und p2 (der Klasse Punkt2D), die zwei Punkte auf der Geraden beschreiben. Durch diese zwei Punkte wird die Gerade festgelegt.

Der folgende Konstruktor der Geraden Gerade2D setzt die x- und y-Korrdinaten der 2 Punkte p1 und p2 der Geraden auf bestimmte Werte.

```
class Gerade{
     Punkt2D p1;
     Punkt2D p2;

     Gerade(Punkt2D pp1, Punkt2D pp2){
          p1.setPunkt(pp1.getX(), pp1.getY());
          p2.setPunkt(pp2.getX(), pp2.getY());
     }
     ...
}
```

Frage:
Warum erzeugt die JVM während der Laufzeit des Programms einen Fehler (NullPointerException)?

Lösung:
Wenn auf p1 mit setPunkt(...) zugegriffen wird, zeigt p1 auf einen nicht reservierten Speicherbereich.

4.2.2.10 Aufgabe

Ändern Sie den Quellcode so ab, so dass es keinen Laufzeitfehler mehr gibt.

Lösung:

```
class Gerade{
      Punkt2D p1;
      Punkt2D p2;

      Gerade(Punkt2D pp1, Punkt2D pp2){
            p1=pp1;
            p2=pp2;
      }
      ...
}
```

4.2.2.11 Die Methode main

Jede Klasse kann die Methode main enthalten, die beim Programmstart automatisch aufgerufen wird. Welche main-Methode wird aber aufgerufen, wenn es mehrere Klassen mit dieser Methode in der gleichen Datei gibt ?
Wenn man mit dem Java-Interpreter das Programm startet, nimmt dieser die public-Klasse, die mit dem Dateiname übereinstimmt (in der Datei TiereTest.java wird also main von der Klasse TiereTest verwendet). Also:
Der Klassenname, in der die Methode main vorkommt und der Dateiname ("Vorname") der Java-Datei müssen den **gleichen** Namen haben!!
In dem Beispiel oben, in dem main in der Klasse TiereTest als Methode vorkommt, heißt die Datei, in der dieser Quellcode geschrieben ist: TiereTest.java

Will man die main-Methode einer anderen Klasse starten, muß man im DOS-Fenster die entsprechende Klasse mit angeben (mit korrekt eingestelltem classpath).

Beispiele:
Starten von main der Klasse TiereTest mit:
  java TiereTest
Starten von main der Klasse Kuh mit:
  java Kuh

4.2.2.12 Bemerkung

1) Es ist viel besser, jede Klasse in eine eigene Datei zu schreiben. Dann kann man auch in TiereTest und Kuh die jeweils dort enthaltene main-Methode mit dem JavaEditor starten (nur zu Testzwecken während man die Klassen TiereTest und Kuh entwickelt). Wenn sie fertig sind wird man diese main-Methoden zum Kommentar umfunktionieren, dann sind die class-Dateien zur Auslieferung kleiner. Siehe packages (unten).

2) Im Gegensatz zu der Programmiersprache C++ gibt es in Java keine Funktionen (d.h. Unterprogramme, die zu keiner Klasse gehören). Jede Methode muss zu einer Klasse gehören. Damit muss auch die Methode **main** zu einer Klasse gehören !!

4.2.2.13 Zugriffschutz

Mit **private** wird angezeigt, dass ein Member privat ist, d.h. dass darauf nur innerhalb einer sich in einer Klasse befindlichen Methode darauf zugegriffen werden kann (in der Klasse Kuh wird z.B. auf alter in der Methode setAlter zugegriffen).
Mit **public** wird angezeigt, dass ein Member öffentlich ist, d.h. dass darauf wie bei einem private-Member und zusätzlich auch noch von außerhalb einer Klasse zugegriffen werden kann. Außerhalb bedeutet, dass von einem Objekt, durch einen Punkt getrennt, die Methode angegeben wird, mit der man auf das private-Member zugreift (in main wird z.B. auf alter durch k.setAlter(4) zugegriffen).

4.2.3 Nicht referenzierbare Objekte

4.2.3.1 Beispiel (siehe oben)

```
public class TiereTest{
  public static void main(String[] args){
    Kuh q1 = new Kuh("Elfriede", 3);
    q1.setAlter(7);
    // Alternative Möglichkeit:
    new Kuh("Elfriede", 3). setAlter(7);
}

class Kuh{
  // wie oben
}
```

Statt auf das Objekt über die lokale Variable q1 zuzugreifen, kann man dies auch über ein **nicht referenzierbares** Objekt realisieren:
`new Kuh("Elfriede", 3). setAlter(7);`
Hier wird ein Objekt erzeugt und sofort auf die entsprechende Methode dieses Objekts zugegriffen.
Allerdings kann danach - mangels einer fehlenden Bezeichnung - nicht mehr auf dieses Objekt zugegriffen werden.
Da man keine Referenz auf das Objekt hat, bedeutet dies für den Garbage Collector (Java Müllabfuhr), dass es aufgeräumt werden kann.

4.3 Aufgaben

1) Ergänzen Sie die noch fehlenden Konstruktoren und Methoden in dem obigen Beispiel. Fügen Sie neben den schon im obigen Beispiel bestehenden Attributen (wie z.B. name) neue Attribute bzw. die dazu zugehörigen Methoden Ihrer Wahl hinzu.

2) Fügen Sie dem obigen Programm neue Klassen Ihrer Wahl hinzu.

4.4 Eingaben in Java

Bemerkung:
Viele Begriffe, die jetzt in diesem Kapitel verwendet werden, werden erst weiter hinten genau erklärt und müssen jetzt einfach so hingenommen werden (der Programmierer soll jetzt einfach schnell die Möglichkeit erhalten, Eingaben über die Tastatur zu realisieren).

In Java werden Ein- und Ausgabeoperationen mittels sogenannter Datenströme realisiert. Der Datenstrom kann von einer beliebeigen Quelle herkommen (Internet, Festplatte, Tastatur, usw.).

4.4.1 Byteorientierte Datenströme (Lesen / Schreiben einzelner Bytes)

Die abstrakten Klassen InputStream (System.in ist vom Typ InputStream) und OutputStream kapseln die wichtigsten Eigenschaften eines Eingabe- bzw. Ausgabe-Datenstroms. Da die Klassen abstrakt sind, können von diesen Klassen keine Objekte gebildet werden, aber von davon abgeleiteten Klassen wie z.B. BufferedInputStream und BufferedOutputStream.

4.4.2 Zeilenorientierte Datenströme (Lesen / Schreiben einzelner Zeilen)

Die abstrakten Klassen Reader und Writer kapseln die wichtigsten Eigenschaften eines Eingabe- bzw. Ausgabe-Datenstroms. Da die Klassen abstrakt sind, können von diesen Klassen keine Objekte gebildet werden, aber von davon abgeleiteten Klassen wie z.B. BufferedReader und BufferedWriter.

4.4.2.1 Beispiel

```
import java.io.*;

public class MainEingabe{
  public static void main(String[] args) throws Throwable {
    int i;
    double d;
    String myString;

    BufferedReader myEingabeString = new
             BufferedReader(new InputStreamReader(System.in));
    myString = myEingabeString.readLine();
    /* Ausgabe des Strings */
    System.out.println("myString= "+myString);
    /* wandelt String in double um */
    d = Double.parseDouble(myString);
    /* wandelt String in int um */
    i = Integer.parseInt(myString);
    System.out.println("i= "+i);
    System.out.println("d= "+d);
    }
}
```

Bemerkung:
1) System ist eine "final class", d.h. wie folgt definiert.
public final class System
Der Modifizierer final bedeutet, dass eine Klasse keine Unterklassen bilden darf .Dadurch kann vermieden werden, dass Unterklassen Eigenschaften nachträglich verändern können. Ein Versuch, von einer finalen Klasse zu erben, führt zu einem Compilerfehler

2) in ist eine Klassenvariable in der Klasse System und wurde wie folgt definiert:
static InputStream in

4.5 Dokumentation in Java

1)
Mit Java kann man seine Methoden wunderbar dokumentieren. Dies geht mit dem Tool javadoc.
javadoc erstellt aus den Kommentaren eine HTKM-Site.
javadoc berücksichtigt nur:
Kommentare, die mit
Schrägstrich **/**** beginnen und mit ***/** enden.
Diese Kommentare (hier javadoc-Kommentare genannt) werden nur berücksichtigt vor:
- Klassen bzw. Interface Deklarationen
- Methodendeklarationen
- Attributdeklarationen

2)
Aufruf in Eclipse
File --> Export --> Java--> Javadoc -->
Bei "Javadoc command" den Pfad des javadoc-tools javadoc.exe angeben. Dieses Tool befindet sich im Verzeichnis bin des JDK

Bei Destination muss noch der Pfad angegeben werden, wo die erzeugte HTML-Site abgespeichert werden soll.

5 Packages (Pakete)

5.1 Beispiel

Der Quellcode des folgenden Programms befindet sich in nur **einer** Java-Datei.

Inhalt der Datei C:\JEP\TiereTest.java:

```
public class TiereTest{
  public static void main(String[] args){
    // wie üblich
    //...    int kuhAlter;
   }
}

class Kuh{
    // wie üblich
    //...    int kuhAlter;
}

class Henne{
    // wie üblich
    //...    int kuhAlter;
}
```

Um das Programm übersichtlicher werden zu lassen, kann man es auf mehrere Dateien verteilen. Dies wird mit den sogenannten packages erreicht.

5.2 Definition

Ein **package** (deutsch Paket) ist ein besonderes Verzeichnis (auf der Festplatte), in dem sich eine Menge von "zusammengehörigen" Klassen (ähnlich einer Programmbibliothek) befindet. Diese Klassen sind in einer oder mehreren Java-Dateien (mit der Endung "java") gespeichert. Unterhalb dieses Verzeichnisses müssen sich diese o.g. Java-Dateien befinden.

Um ein package zu bezeichnen, wird in der 1. Zeile der Java-Datei das Wort package gefolgt von einem frei gewählten Namen eingefügt. Dieser frei gewählte Name ist der Name des packages.

Will man nun von einer anderen Java-Datei aus auf eine Klasse dieses o.g. packages zugreifen, geschieht dies durch die Angabe des Schlüsselworts **import** und der Pfadangabe des packages, wobei die Verzeichnisse in der Pfadangabe durch jeweils einen Punkt getrennt werden müssen.
Am Schluss der Pfadangabe muss die Java-Datei (ohne die Endung java) angegeben werden (in der sich die Klassen befinden, auf die zugegriffen wird). Anstatt einer Datei kann auch der Platzhalter **`*`** verwendet werden.

1) Durch z.B. folgende Angabe kann von einem Java-Programm auf alle Klassen des packages io zugegriffen werden.
```
import java.io.*;
```
Dadurch wird dem Java-Quelltext, vor dem diese import Zeilen stehen, mitgeteilt wo (in welchem package) die Klassen gesucht werden müssen, auf die in dem Java-Quelltext zugegriffen werden. Es werden also keine Informationen (bzw. Quellcode, Maschinencode, Libraries, usw.) in den Java-Quelltext bzw. in den Java-Byte-Code hineinkopiert (wie dies z.B. in C mit der Anweisung include der Fall ist).

2) Die Anweisung
```
import java.io.*;
```
durchsucht nicht auch noch die Unterverzeichnisse von
java.io nach einer gesuchten Klasse (also z.B. nicht auch noch java.io.xyz)

3) In einer Java-Quellcode-Datei können mehrere Klassen hintereinander (hier sind nicht interne Klassen gemeint) stehen. **Maximal** eine davon kann public sein. Der Name dieser Klasse **muss** dann mit dem Dateinamen übereinstimmen !

4) classpath
Zuerst werden die Klassen in den Dateien des aktuellen Verzeichnisses gesucht, dann im sogenannten Klassenpfad (classpath).
Wie kann der Klassenpfad gesetzt werden?
a) durch das Setzen einer CLASSPATH Umgebungsvariablen (es wird empfohlen, darauf zu verzichten. Oft wird sie nicht angelegt und existiert deshalb nicht).
In der Umgebungsvariable CLASSPATH stehen das (oder die Verzeichnisse), ab denen die packages gesucht werden, die mit import ... angegeben wurden.
Wichtig: Das package (Verzeichnis) muß sich **direkt unterhalb** eines der in Classpath angegebenen Verzeichnisse befinden.

b) durch die Angabe des Klassenpfad (d.h. eines Verzeichnisse) nach Angabe des Schaltes -cp in dem Programm (wie z.B. javac).

5) Klassen des Java-Systems, wie z.B.
import java.io.BufferedReader;
werden **automatisch** ohne Zutun des Programmierers gefunden.
Klassen, die Programmierer erstellt hat, werden dagegen mit Hilfe des **Klassenpfads** gefunden.

6) Die sich im Java-System befindlichen Ordner java.lang, java.awt sind in einem Archiv versteckt und deswegen nicht sichtbar.
Die Quellcodes von allen Java-Standardklassen befinden sich aber auf dem Server von SUN.

7) Default-package
Wenn eine package-Anweisung fehlt, gehört die Klasse zum sogenannten **Default-package**.
Klassen des Default-package können ohne explizite import-Anweisung in anderen Dateien des (gleichen) Default-package verwendet werden. Dies gilt nicht **nur** für Default-packages: Klassen innerhalb desselben Package können unabhängig davon ob es das Default Package ist oder nicht auf protected oder public Methoden, Konstruktoren, Attribute zugreifen.
Allerdings können Klassen aus anderen Packages als dem Default Package nicht auf das Default Package zugreifen. Vom Gebrauch des Default Packages ist auch generell **abzuraten**.
Ein Java-Compiler braucht laut Spezifikation nur ein einziges Default-package zur Verfügung zu stellen. Typischerweise wird dieses Konzept aber so realisiert, dass jedes Verzeichnis (in dem java-Dateien ohne die package-Anweisung verwendet werden), ein Default-package ist
Wenn ein Member mit dem Modifier protected versehen ist, dann ist es erlaubt in allen Klassen des im gleichen package (d.h. auch des gleichen Default-packages) darauf zuzugreifen.

8) Automatisches Importieren
Es werden oft Klassennamen (wie beispielsweise String) verwendet, ohne dass eine zugehörige import-Anweisung zu erkennen wäre. In diesem Fall entstammen die Klassen dem Paket java.lang. Dieses Paket wurde von den Entwicklern der Sprache als so wichtig angesehen, daß es von jeder Klasse automatisch importiert wird. Man kann sich das so vorstellen, als wenn am Anfang jeder Quelldatei implizit die folgende Anweisung stehen würde:
import java.lang.*;

9) Beispiel (Kompilieren und Klassenpfad)

Voraussetzungen:
Dieses Dateien existieren
c:\temp\src\Maintest1.java

```
public class Maintest1{
  public static void main(String argv[]){
    Hund myh=new Hund();
    System.out.println("Hallo");
  }
}
```

c:\temp\src\Hund.java

```
public class Hund {
  private int alter;
  public void setAlter(int palter){
    alter=palter;
  }
}
```

In irgendeinem Verzeichnis der Eingabeaufforderung wird dann übersetzt (kompiliert):

```
javac -cp \temp\src -d \temp\classes \temp\src\*.java
```

Die Java-Dateien *.java in dem Verzeichnis \temp\src werden in class-Dateien übersetzt und wegen des Schalters -d in das Verzeichnis \temp\classes abgelegt.
In der Datei c:\temp\src\Maintest1.java wird die Klasse Hund benötigt, die es aber nicht in der gleichen Datei, sondern in der anderen Datei c:\temp\src\Hund.java gibt.

Wie wird gesucht (siehe oben):
a) Klassen des Java-Systems, wie z.B.
import java.io.BufferedReader;
werden automatisch ohne Zutun des Programmierers gefunden.
b) Klassen, die der Programmierer erstellt hat, werden mit Hilfe des Klassenpfads gefunden.
Zuerst werden die Klassen in den Dateien des aktuellen Verzeichnisses gesucht, dann im Klassenpfad (classpath).
Da es sich in diesem Beispiel um das Default package handelt, wird hier zuerst gesucht.
Damit ist in diesem Beispiel die Angabe des Klassenpfads mit cp unnötig.

5.2.1 Beispiel 1 (Verwendung eines packages)

Voraussetzungen:

Verzeichnisse:	Dateien	Classpath
`C:\JEP`	`C:\JEP\myp1\Kuh.java`	`C:\JEP`
`C:\JEP\myp1`	`C:\JEP\myp1\Henne.java`	

Inhalt der Datei C:\JEP\TiereTest.java:

```
import myp1.Kuh;
import myp1.Henne;
// Falls man den Namen der Dateien Kuh.java und Henne.java
// nicht kennt, kann man dies auch alternativ wie
// folgt (mit dem Platzhalter *) machen:
// import myp1.*;

public class TiereTest{
  public static void main(String[] args){
    // wie üblich
    //...    int kuhAlter;
  }
}
```

Inhalt der Datei C:\JEP\myp1\Kuh.java:

```
package myp1;
public class Kuh{
  // wie üblich
  //...    int kuhAlter;
}
```

Inhalt der Datei C:\JEP\myp1\Henne.java:

```
package myp1;
public class Henne{
  // wie üblich
  //...    int kuhAlter;
}
```

5.2.2 Beispiel 2 (Verwendung eines packages)

Voraussetzungen (wie im vorigen Beispiel, aber mit einem anderen Classpath, siehe unten)

Verzeichnisse:	Dateien
`C:\JEP`	`C:\JEP\myp1\Kuh.java`
`C:\JEP\myp1`	`C:\JEP\myp1\Henne.java`

Inhalt der Datei C:\JEP\TiereTest.java:

```
import JEP.myp1.Kuh;
import JEP.myp1.Henne;
// Falls man den Namen der Dateien Kuh.java und Henne.java
// nicht kennt, kann man dies auch alternativ wie
// folgt (mit dem Platzhalter *) machen:
// import JEP.myp1.*;

// Der Rest ist genau gleich wie im Beispiel 1
public class TiereTest{...}
```

Inhalt der Datei C:\JEP\myp1\Kuh.java:

```
package JEP.myp1;

// Der Rest ist genau gleich wie im Beispiel 1
public class Kuh{...}
```

Inhalt der Datei C:\JEP\myp1\Henne.java:

```
package JEP.myp1;

// Der Rest ist genau gleich wie im Beispiel 1
public class Henne{...}
```

5.2.3 Verhalten des Programms bei verschiedenen Classpath

1) Verzeichnisse in classpath: `C:\JEP;`
Beispiel 2 liefert Fehler, weil sich.
`JEP.myp1.Kuh` nicht direkt unterhalb von `C:\JEP` befindet.

2) Verzeichnisse in classpath: `C:\JEP;C:\;`
Beispiel 2 liefert keinen Fehler, weil sich
`JEP.myp1.Kuh` direkt unterhalb von `C:\` befindet.

3) Verzeichnisse in classpath: `C:\JEP;`
Beispiel 1 liefert keinen Fehler, weil sich.
`myp1.Kuh` direkt unterhalb von `C:\JEP` befindet.

4) Verzeichnisse in classpath: `C:\;`
Beispiel 1 liefert Fehler, weil sich
`myp1.Kuh` nicht direkt unterhalb von `C:\` befindet.

5.2.4 Packete mit Netbeans erstellen und importieren

5.2.4.1 Wie wird ein Paket erstellt?

Zuerst sollten alle Klassen public sein (die man verwenden will).
Dazu müssen diese Klassen jeweils in einem eigenen File erstellt werden.
Dann Run --> Build Main Project
Unter dem Ordner dist des Verzeichnisses wird dann das Paket mit der Endung .jar erzeugt (z.B:sound2.jar)

5.2.4.2 Wie wird ein Paket importiert? (z.B: sound2)

1)
Zuerst muss man im Quellcode als erste Anweisung das Paket importieren (z.B. folgendes einfügen):
import sound2.*;
Falls man das nicht macht, muss die verwendete Klasse mit dem **vollständigen** Namen (**full qualified name**) angegeben werden,
also z.B. nicht MyPlayer, sondern sound2.MyPlayer

2) Man muss Netbeans mitteilen, wo sich das Paket auf der Festplatte befindet:
Rechter Mausklick auf das Projekt --> Properties --> Librariers --->
Add Jar/Folder --> Name des Paktes eingeben.

Bemerkung:
Die vollständige Namensangabe betrifft alle Klassen (auch die von Java gelieferten), wie z.B:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;
import java.util.ArrayList;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Man kann also innerhalb des Quellcodes die Klasse ActionEvent verwenden, wenn man sie wie oben importiert hat.
Importiert man sie nicht, muss man sie innerhalb des Quellcodes mit
java.awt.event.ActionEvent bezeichnen.
Man kann auch mehrere Klassen mit Hilfe des Platzhalters * importieren (siehe oben).

6 Übung Bauernhof

6.1 Klassen

Auf einem Bauernhof gibt es u.a. Kühe und Hennen.
Jedes Tier hat einen Namen und ein (ganzzahliges) Alter.
Jede Kuh hat eine bestimmte Milchleistung pro Zeiteinheit, erzielt aber auf dem Markt den gleichen Preis für einen Liter Milch.
Jede Henne hat eine bestimmte Legeleistung pro Zeiteinheit, erzielt aber auf dem Markt den gleichen Preis für ein Ei.
Erstellen Sie die Klasse Kuh und Henne mit den entsprechenden Attributen und Methoden.
Erzeugen Sie außerdem jeweils in der Klasse Kuh bzw. Henne die Methode printAllAttributs(), die - zu Testzwecken - alle Werte der Attribute in der jeweiligen Klasse auf dem Bildschirm ausgibt.
Erzeugen Sie jeweils zwei Exemplare der Klasse Kuh bzw. Henne.

Die Klasse Kuh wird hier beispielhaft durch die sogenannte UML (Unified Modeling Language) beschrieben:.
UML = formale Sprache zur objektorientierten Spezifikation, Visualisierung, Konstruktion und Dokumentation von Softwaresystemen und Geschäftsmodellen.

Kuh
- name: String
- alter: int
- literpreis: double
- milchleistung: double
+ setName(String pname): void
+ getName(): String
+ setAlter(int palter): void
+ getAlter(): int
+ setMilchleistung(double pmilchleistung): void
+ getMilchleistung(): double
+ setLiterpreis(double pliterpreis): void
+ getLiterpreis(): double
+ printAllAttributs(): void

Bemerkung:
- bedeutet, dass das Attribut bzw. die Methode private ist.
+ bedeutet, dass das Attribut bzw. die Methode public ist.

6.2 Klassenvariable

Da es viel Schreibaufwand macht, für jede Kuh den gleichen Literpreis und für jede Henne den gleichen Eierpreis festzulegen, ist es viel sinnvoller, dies mit einer einzigen Anweisung zu machen.
Zuerst wird dazu eine sogenannte **Klassenvariable** definiert. Eine Klassenvariable unterscheidet sich von einem "normalen" Attribut durch den Vorsatz "static".
In der Klasse Kuh wird deshalb die Klassenvariable "literpreis" wie folgt definiert:

```
private static double literpreis;
```

Es gibt zwei Möglichkeiten mit einer Methode auf eine Klassenvariablen zuzugreifen:
1) Mit der "normalen" Methode (Exemplarmethode), die wir bis jetzt kennen gelernt haben.
2) Mit der sogenannten Klassenmethode, die sich von der "normalen" Methode (Exemplarmethode) durch den Vorsatz static unterscheidet.
In der Klasse Kuh wird dies also z.B. wie folgt realisiert:

```
public static void setLiterpreis(double pliterpreis){
  // ...
}
```

Eigenschaften der Klassenmethoden:
1) Klassenmethoden können nur auf Klassenvariablen zugreifen.

2) Klassenmethoden können nur Klassenmethoden der gleichen Klasse aufrufen.
Dagegen haben alle Exemplarmethoden Zugriff auf die Klassenvariablen und Klassenmethoden.

3) Durch die Benutzung des **Klassennamens** (und nicht des Namens eines Objekts einer Klasse) wird dann der Literpreis für **jedes** Objekt einer Klasse festgelegt. In der Klasse Kuh wird dies also z.B. wie folgt realisiert:

```
Kuh.setLiterpreis(20)
```

setzt für **jede** Kuh den Preis für ein Liter Milch auf 20 Cent.
Die Methode (hier `setLiterpreis`) nach der Benutzung des Klassennamens muss dann allerdings eine Klassenmethode sein!

Bemerkung:
Hier wird der Bezeichner static immer nach dem Bezeichner für den Zugriffschutz verwendet. Die Bezeichner für den Zugriffschutz und der Bezeichner static können aber auch in der Reihenfolge vertauscht werden. Dies ist dem Kompiler egal
Also statt:

```
private static double literpreis;
```

ist genauso möglich:

```
static private double literpreis;
```

6.3 Array (Feld)

Ein Array (Feld) ist in Java prinzipiell immer ein **Objekt** und wird deshalb durch eine **Referenzvariable** realisiert.
Für das Anlegen und Belegen des Feldes gibt es mehrere Möglichkeiten.
Wichtig:
Ein Array hat nach seiner Erzeugung eine konstante Länge, die nicht mehr verändert werden kann !!

6.3.1 Felder mit Klassen als Datentypen

Im Folgenden wird vorausgesetzt, dass die Klasse Kuh (siehe oben) existiert und den zweiparametrigen Konstruktor hat, mit dem man den Namen und das Alter einer Kuh festlegen kann. Es gibt zwei Möglichkeiten für das Anlegen von Arrays:

6.3.1.1 Erste Möglichkeit

1) Deklarieren der Feldvariablen

```
Kuh[] kuhherde;
```

// Folgende Schreibweise wäre auch möglich:

```
Kuh kuhherde[];
```

Dadurch wird die Referenzvariable kuhherde angelegt, die als Wert eine Referenz (technisch eine Adresse) auf kein Objekt, also auf nichts hat (es wurde ja noch nicht einmal die Länge des Feldes angegeben) hat, also hat die Referenzvariable kuhherde den Wert null.

Bemerkung:
Es wird die Schreibweise

```
Kuh[] kuhherde;
```

bevorzugt, weil hier besser ersichtlich ist, dass Kuh[] eine Klasse und kuhherde eine Objektvariable ist.

6.3.1.1.1 Technisch gesehen

Variable	Wert
kuhherde	null

2) Festlegen der Feldlänge
Jetzt muss man noch die Feldlänge angeben.

```
kuhherde = new Kuh[2];
```

Jetzt wurde zwar festgelegt, wie viele Elemente das Feld hat, aber die einzelnen Elemente (Objekte) noch nicht erzeugt. Deshalb werden die Elemente des Feldes mit den Standardwerten (das ist bei einer Referenzvariable null) initialisiert.

6.3.1.1.2 Technisch gesehen

Variable	Wert
kuhherde	04711

An der Adresse 04711 beginnt das Feld. Dieses Feld besteht aus zwei Adressen, die auf noch nicht erzeugte Objekte zeigen. Für diesen Fall werden die zwei Adressen null eingetragen.

Adresse	Wert
04711	null
	null

3)
Jetzt will man noch jedes Element des Feldes mit einem Objekt belegen.
Dazu muss man jedes Objekt einzeln erzeugen.
Dies kann man z.B. wie folgt machen:

```
kuhherde[0] = new Kuh("Berta", 4);
kuhherde[1] = new Kuh("Elsa", 5);
```

6.3.1.1.3 Technisch gesehen

Variable	Wert
kuhherde	04711

Adresse	Wert
04711	07000
	08000

Adresse	Wert
07000	Berta
	4

Adresse	Wert
08000	Elsa
	5

6.3.1.2 Zweite Möglichkeit

Der erste und zweite Schritt kann zusammengefasst werden zu:

```
Kuh[] kuhherde = new Kuh[2];
```

Jetzt braucht man nur noch den dritten Schritt hinzunehmen:

```
kuhherde[0] = new Kuh("Berta", 4);
kuhherde[1] = new Kuh("Elsa", 5);
```

6.3.1.3 Dritte Möglichkeit

Der erste, zweite und dritte Schritt kann zusammengefasst werden zu:

```
Kuh[] kuhherde =
           {new Kuh("Berta", 4), new Kuh("Elsa", 5)};
```

Das, was für Objekte als Elemente eines Arrays gilt, gilt auch für primitive Datentypen als Elemente eines Arrays:

6.3.2 Felder mit primitiven Datentypen

6.3.2.1 Erste Möglichkeit

```
int zahlen[];
// Folgende Schreibweise wäre auch möglich:
int[] zahlen;
zahlen = new int[3];
zahlen[0]=10;
zahlen[1]=20;
zahlen[2]=30;
```

6.3.2.2 Zweite Möglichkeit

```
int[] zahlen = new int[3];
zahlen[0]=10;
zahlen[1]=20;
zahlen[2]=30;
```

6.3.2.3 Dritte Möglichkeit

```
int[] zahlen = {10, 20, 30};
```

6.3.3 Länge eines Feldes bestimmen lassen

Mit der Angabe des Klassennamens und length kann man die Länge eines Feldes bestimmen lassen.

Beispiele zu oben:

```
System.out.println("Feldlänge kuhherde= "+kuhherde.length);
System.out.println("Feldlänge zahlen= "+zahlen.length);
```

Bemerkung:
length wird javaintern als Attribut wie folgt definiert:

```
public final length
```

Da length als final deklariert wurde (und als Konstante deshalb nicht mehr verändert werden kann), führt die folgende Anweisung zu einer Fehlermeldung des Compilers:

```
kuhherde.length = 123;
```

6.4 Mechanismus der Parameterübergabe

In Java wird bei der Parameterübergabe nur der Mechanismus "call by value" verwendet.
Beim Aufruf einer Methode werden also Kopien der Werte der Variablen an die formalen Parameter der aufrufenden Methode übergeben.

6.4.1 Beispiel (einfacher Datentyp)

```
...
int zahl=13;
verringere(zahl);
System.out.println("zahl= "+zahl);
// Bildschirmausgabe: 13
...

public static void verringere (int z){
  z=z-1;
}
...
```

6.4.1.1 Beschreibung

Die Variable zahl hat nach dem Aufruf immer noch den Wert 13.
Es wurde nur der Wert der Kopie z verändert, NICHT der Wert des Originals, also der Variablen zahl !!

6.4.1.1.1 Technisch gesehen

(Werte vor und nach der Anweisung `verringere(zahl)`)

Variable	Wert davor	Wert danach
zahl	13	13
z	?	12

Bemerkung:
Exakterweise müsste es bei z statt "Wert danach" besser "Wert kurz vor Verlassen der Methode" heissen.

6.4.2 Beispiel (Objekt)

```
...
Kuh myfriend = new Kuh("Elsa", 10);
changeKuhAlter(myfriend, -2);
...

public static void changeKuhAlter(Kuh k, int palter){
  int tempAlter;
  tempAlter = k.getAlter()+palter;
  k.setAlter(tempAlter);
// myfriend wird durch die folgende Anweisung nicht verändert:
  k = null;
}

...

// Die Klasse Kuh wie üblich implementieren:
class Kuh{
  ...
  public void setAlter(int palter){
    alter = palter;
  }
  ...
}
```

6.4.2.1 Beschreibung

Beim Aufruf wird die Variable myfriend, eine Referenzvariable (also eine Variable, deren Wert eine Adresse ist), in den formalen Parameter k kopiert.
Technisch gesehen wird der Wert von myfriend, also eine Adresse, in die Variable k kopiert.
k und myfriend zeigen also auf das gleiche Objekt (die Werte von k und myfriend - das sind Adressen - sind gleich).

1) Durch die Anweisung (Verwendung der Methode)
`k.setAlter(tempAlter);`
wird auf den Inhalt des Werts von k, also den Inhalt einer Adresse, hier kurz mit *k bezeichnet (das ist das Objekt, auf das k zeigt, besser ein bestimmter Teil, nämlich das Attribut alter) verändert. Da k und myfriend auf das gleiche Objekt zeigen, wird damit auch das Attribut alter von myfriend verändert.

Das Gleiche geschieht (was man aus Design-Gründen der OOP vermeiden sollte), wenn man direkt auf das Attribut zugreifen würde:
`k.alter = k.alter + palter;`
oder
`k.alter = 2;`

D.h. mit dem Zugriff über den Punkt . greift man auf den Inhalt der Adresse zu.

6.4.2.1.1 Technisch gesehen

(Werte vor und nach der Anweisung `k.setAlter(tempAlter)`)

Variable	Wert davor	Wert danach
myfriend	04711	04711
k	04711	04711

Adresse	Wert davor	Wert danach
04711	"Elsa"	"Elsa"
	10	**8**

Durch die Anweisung:
`k.setAlter(tempAlter);`
wird der Inhalt der Adresse 04711, also der Teil im Arbeitsspeicher, wo das Alter 10 gespeichert ist, um 2 auf 8 verringert.

2) Durch die Anweisung
`k = null;`
dagegen wird nicht der Inhalt einer Adresse verändert, sondern nur der Wert der Variablen k verändert. Technisch gesehen haben nach dieser Zuweisung k und myfriend nun andere Werte (das sind Adressen). Diese zeigen aber nun auf andere Objekte !
Diese Zuweisung hat also auf das Objekt, auf das myfriend zeigt keinen Einfluss.

6.4.2.1.2 Technisch gesehen

(Werte vor und nach der Anweisung `k.setAlter(tempAlter)`)

Variable	Wert davor	Wert danach
myfriend	04711	04711
k	04711	**null (0)**

Adresse	Wert davor	Wert danach
04711	"Elsa"	"Elsa"
	8	**8**

Durch die Anweisung:
`k=null;`
wird nur der Wert der Variablen k geändert, aber nicht das Objekt, auf das myfriend zeigt (also z.B. speziell nicht das Attribut alter dieses Objekts.

Bemerkung:
Exakterweise müsste es bei k statt "Wert danach" besser "Wert kurz vor Verlassen der Methode" heißen.

6.4.3 Beispiel (Array)

```
...
Kuh[] kuhherde = {new Kuh("Anna", 5),new Kuh("Berta", 6)};
swapKuhherde(kuhherde, 0, 1);
...

public static void swapKuhherde(Kuh[] herde, int i, int j){
  String tempstring;
  int tempAlter;
  tempstring = herde[i].getName();
  tempalter = herde[i].getAlter();
  herde[i].setName(herde[j].getName());
  herde[i].setAlter(herde[j].getAlter());
  herde[j].setName(tempstring);
  herde[j].setAlter(tempAlter);
}

// Die Klasse Kuh wie üblich implementieren:
// ...
```

6.4.3.1 Beschreibung

Die Übergabe eines Arrays funktioniert genauso wie die Übergabe eines Objekts, weil ein Array auch eine Referenzvariable ist.
Beim Aufruf wird die Variable kuhherde, eine Referenzvariable, in den formalen Parameter herde kopiert: herde ist also eine Kopie von kuhherde.

Durch Anweisungen wie
`herde[j].setName(tempstring);`
wird der Name des j-ten Elemets von herde (nicht der Wert von herde!!) und damit der Name des j-ten Elemets von kuhherde verändert.

6.4.4 Aufgaben

1) Schreiben Sie die Methode "anhaengen", die ein zusätzliches Element an ein bestehendes (volles) Array anhängt. Da ein Array nicht verlängert werden darf, muss in der Methode ein neues Array erstellt werden, in den der Inhalt des alten arrays kopiert wird.

2) Schreiben Sie ein Programm, das einen Kuhstall simuliert und eine Methode, die (wegen guter Fütterung mit Kraftfutter) das Gewicht jeder Kuh um den gleichen Betrag ändert.

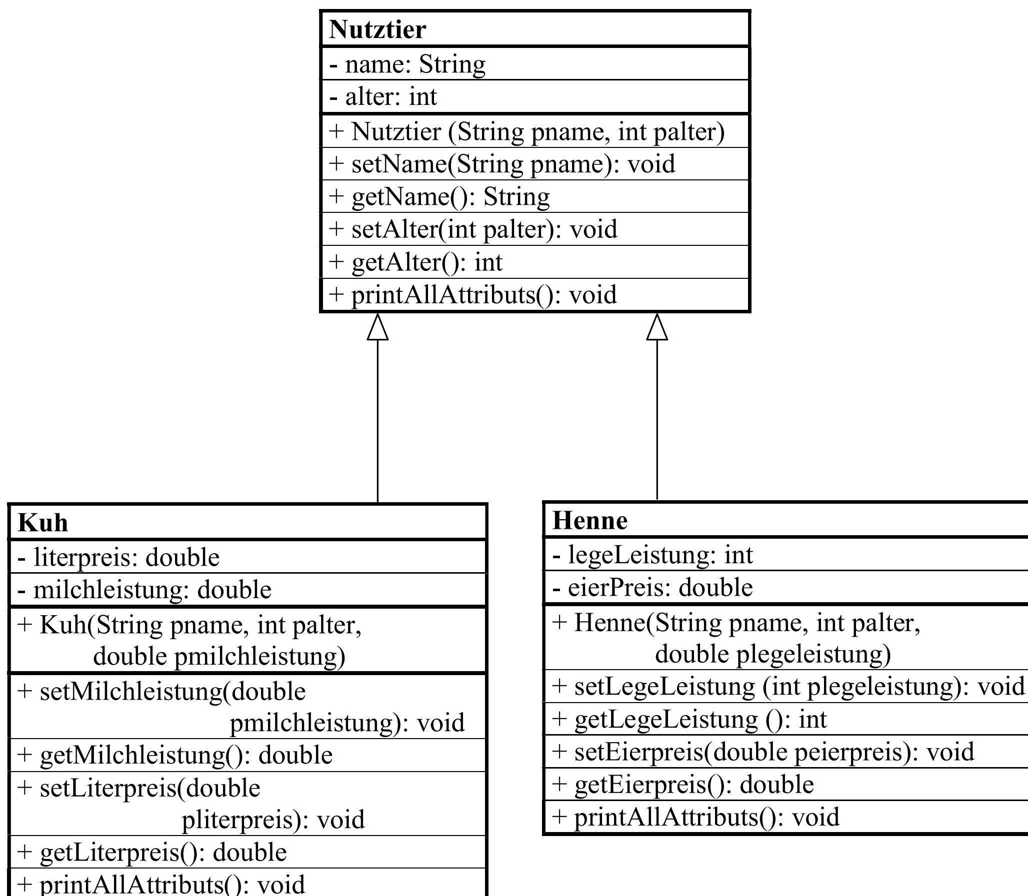
Lösung:

```
public class Parameteruebergabe2 {
  public static void main(String[] args) {
    int i;
    int arrayOld [] = {10,20,30};
    int arrayNew[];
    arrayNew=elementAnhaengen(arrayOld, 123);
    for(i=0;i<4;i++)
      System.out.print(arrayNew[i]+" ");
    }


  public static int[] anhaengen(int v[], int element){
    int i,len;
    len=v.length;
    int arrayNewTemp[]=new int[len+1];
    for(i=0;i<len;i++){
      arrayNewTemp[i]=v[i];
    }
    arrayNewTemp[len]=element;
      return arrayNewTemp;
  }
}
```

6.5 Vererbung

Da es viel Schreibaufwand macht, die gleichen Methoden für die Manipulation des Alters und des Namens der Kuh bzw. der Henne zu implementieren, ist es geschickter dies nur einmal in einer eigenen Klasse zu machen und diese Methoden dann in die Klassen Kuh und Henne zu vererben. In den Klassen Kuh und Henne werden dann nur noch die jeweiligen spezifischen Attribute und Methoden aufgenommen:



Die Klasse Kuh und die Klasse Henne erbt jeweils von der Klasse Nutztier alle Attribute und Methoden, kann aber nur auf die mit public (und protected) angegebenen Attributen bzw. Methoden der Oberklasse (= Klasse die vererbt = hier im Beispiel Nutztier) zugreifen.
Die Vererbung wird realisiert durch den Bezeichner **extends**
Die Klasse Kuh erbt von der Klasse Nutztier wie folgt:

6.5.1 Wie vererben

Mit dem Schlüsselwort extends:

```
class Kuh extends Nutztier{
  // hier werden die Attribute und Methoden
  // der Klasse Kuh definiert.
  // ...
}
```

6.5.2 Überschreiben

Frage:
Welche Methode wird im folgenden Programmausschnitt ausgeführt ?
Wird die Methode printAllAttributs() der Klasse Nutztier oder der Klasse Kuh (siehe oben) ausgeführt ?

```
...
Kuh k1 = new Kuh("Berta",500, 50);
k1.printAllAttributs();
...
```

Ergebnis:
Es wird die Methode der Unterklasse, also der Klasse Kuh, ausgeführt. Man sagt die Methode der Unterklasse überschreibt die Methode der Oberklasse.

Will man in (einer Methode) der Unterklasse auf die durch den gleichen Namen bezeichnete Methode der Oberklasse zugreifen, dann geschieht dies durch den Vorsatz "super" vor die Methode.

Beispiel:
Die folgende Anweisung:
`super.printAllAttributs();`
in einer Methode f() der Klasse Kuh ruft die Methode printAllAttributs() der Oberklasse Nutztier auf.

6.5.3 Konstruktoren

Konstruktoren werden nicht vererbt !
Wenn ein Konstruktor der Unterklasse nicht mittels super(...) einen Konstruktor der Oberklasse aufruft, dann wird automatisch der Standardkonstruktor (=Konstruktor mit 0 Parametern) der Oberklasse aufgerufen. Der Sinn davon ist, dass alle Attribute (auch die der Oberklassen) einen definierten Wert bekommen sollen, auch wenn der Programmierer vergisst, den Konstruktor der Oberklasse aufzurufen.
Falls dieser Standardkonstruktor nicht existiert, wird er automatisch vom Compiler erzeugt, außer ein anderer Konstruktor (mit mindestens einem Parameter) existiert schon. In diesem Fall bringt der Compiler dann eine Fehlermeldung.

Wenn man super(...) in einem Konstruktor benutzt, muss dies die erste Anweisung in dem Konstruktor sein, d.h. vor super(...) darf keine Anweisung stehen!

Hier in diesem Beispiel werden in der Oberklasse und in den Unterklassen Konstruktoren (keine Standardkonstruktoren) erzeugt. Der Konstruktor in der jeweiligen Unterklasse leitet durch super(...) an den Konstruktor der Oberklasse weiter.

```
class Nutztier{
  // Name des Tiers
  private String name;
  // Alter des Tiers
  private int alter;

  public Nutztier(String pname, int palter){
    name = pname;
    alter = palter;
  }
//...
}

class Kuh extends Nutztier{
  // Milchleistung (pro Zeiteinheit).
  private double milchLeistung;

  public Kuh(String pname, int palter,
             double pmilchleistung){
    // super(...) muß die erste Anweisung im Konstruktor sein!
    super(pname, palter);
    milchLeistung = pmilchleistung;
  }
//...
}

class Henne extends Nutztier{
  // Legeleistung der Henne (Anzahl der Eier pro Zeiteinheit)
  private double legeLeistung;

  public Henne(String pname, int palter,
               double plegeleistung){
    // super(...) muß die erste Anweisung im Konstruktor sein!
    super(pname, palter);
    legeLeistung = plegeleistung;
  }
//...
}
```

6.5.4 Konvertierung beim Methodenaufruf

1) Wenn eine Methode verlangt, dass ein Parameter einen primitiven Datentypen wie z.B. double haben muss, aber beim Methodenaufruf ein integer Wert benutzt wird, dann wird der "kleinere" Datentyp (hier also int) automatisch in den "größeren" Datentyp konvertiert (impliziter cast).

2) a) Wenn eine Methode verlangt, dass ein Parameter als Datentyp eine Klasse haben muss, aber beim Methodenaufruf ein Objekt einer Unterklasse davon übergeben wird, dann wird das Objekt vor der Übergabe an den Parameter automatisch in das Objekt der Oberklasse konvertiert (impliziter cast).

b) Umgekehrt geht es nicht:
Ein Objekt einer Oberklasse kann nicht an einen Parameter einer Methode übergeben werden, die ein Objekt einer Unterklasse erwartet.

6.5.4.1 Beispiel

6.5.4.1.1 Voraussetzungen

1) Die folgende Methode (siehe oben).

```
public static void changeKuhAlter(Kuh k, int palter){...}
```

2) Angus ist eine Unterklasse der Klasse Kuh und Kuh ist eine Unterklasse der Klasse Vierbeiner.

6.5.4.1.2 Programmausschnitt

```
...
Angus angie;
Kuh berta;
Vierbeiner fred;
// Dieser Aufruf ist nicht erlaubt: Keine Konvertierung in Typ der Unterklasse
changeKuhAlter(fred, 3);
// Dieser Aufruf ist erlaubt: Konvertierung in Typ der Oberklasse
changeKuhAlter(angie, 7);
...
```

6.5.5 Zugriff auf Klassen

In einer Klasse kann bei der Definition jeder einzelnen Komponente jeweils spezifiziert werden, in welcher Weise der Zugriff auf diese Komponente möglich ist:

Spezifizierer	in gleicher Klasse	in Unterklasse mit Vererbung	in gleichem Package	überall
Private	ja	nein	nein	nein
protected	ja	ja	ja	nein
Public	ja	ja	ja	ja
nichts angegeben	ja	nein	ja	nein

ja: Zugriff möglich
nein: Zugriff nicht möglich

6.6 Abstrakte Klassen

Es gibt in der realen Welt kein Tier, das ein allgemeines, unspezifiziertes Nutztier ist, sondern nur spezielle Unterarten von Nutztieren, z.B. eine Kuh oder eine Henne.
Es macht also keinen Sinn, ein Exemplar der Klasse "Nutztier" zu bilden. Dies realisiert man mit einer sogenannten **abstrakten Klasse**, wie z.B. der Klasse "Nutztier" durch den Vorsatz **abstract**.
Da es also keinen Sinn macht, ein Exemplar der Klasse Nutztier zu erzeugen, ergibt es einen Compilerfehler, falls man dies machen würde. Man darf also abstrakte Klassen nicht instanzieren (d.h. man kann keine Objekte davon erzeugen). Ansonsten verhalten sich abstrakte Klassen wie "normale" Klassen, sie können z.B. von anderen Klassen erben, Attribute und "normale" Methoden enthalten.
In der Klasse "Nutztier" wird dies also z.B. wie folgt realisiert:

```
abstract class Nutztier{
  // hier werden die Attribute und Methoden
  // der Klasse Nutztier definiert.
  // ...
}
```

6.6.1 abstrakte Methoden

Eine Oberklasse kann einer Unterklasse eine oder mehrere Methoden bereitstellen, von denen die Oberklasse aber nicht weiss, wie diese genau implementiert werden sollen. Diese nennt man eine **abstrakte Methode**.
Eine abstrakte Methode in der Oberklasse, wie z.B. "tierMarktwert()" definiert lediglich den Methodenkopf und drückt damit aus, dass die Oberklasse keine Ahnung von der Implemetierung hat und die jeweilige Unterklasse sich darum kümmern muss.
Dies macht in der Methode tierMarktwert() auch Sinn, da der Tiermarktwert von der Leistungsfähigkeit (z.B. der Legeleistung einer Henne, oder der Milchleistung einer Kuh) eines Tiers abhängt, von den Attributen "legeleistung" und "milchleistung" nur die jeweilige Unterklasse etwas weiss, aber die Oberklasse keine Ahnung davon hat.
In der Klasse "Nutztier" wird dies also z.B. wie folgt realisiert:

```
abstract class Nutztier{
  // hier werden die Attribute und Methoden
  // der Klasse Nutztier definiert.
  // ...
  /*
  Die folgende abstrakte Methode berechnet den Marktwert eines Tiers. Je nach Tier wird
  dieser mit einer anderen Formel berechnet, die u.a. von der Leistungsfähigkeit
  (z.B. der Legeleistung einer Henne, oder der Milchleistung einer Kuh) des Tiers abhängt.
  Da die Oberklasse nicht die Formeln in den Unterklassen kennt, muss die konkrete
  Implementierung der zugehörigen Methode in der jeweiligen Unterklasse gemacht werden.
  Zum Beispiel kann der Tiermarktwert einer Kuh aus dem 10-fachen der Milchleistung
  der Kuh berechnet werden und der Tiermarktwert einer Henne aus dem 2-fachen der
  Legeleistung der Henne.
  */
  abstract public double getTierwert();
}
```

Die Implementierung (Ausprogrammierung) der abstrakten Methode muss dann in der jeweiligen Unterklasse stattfinden, wie im folgenden Beispiel:

```
class Kuh extends Nutztier{
  // hier werden die Attribute und Methoden
  // der Klasse Kuh definiert, wie z.B:
  public double getTierwert(){
    return(10 * milchLeistung);
  }
}

class Henne extends Nutztier{
  // hier werden die Attribute und Methoden
  // der Klasse Henne definiert, wie z.B:
  public double getTierwert(){
    return(2 * legeLeistung);
  }
}
```

6.6.1.1 Wichtige Bemerkungen

1) Wenn eine abstrakte Methode in einer Klasse verwendet wird, dann muss diese Klasse eine abstrakte Klasse sein.

2) Wenn eine Klasse abstrakt ist, dann müssen die sich darin befindlichen Methoden nicht alle abstrakt sein (sie kann auch gar keine abstrakte Methoden besitzen).

3) Wenn eine abstrakte Methode in der Unterklasse nicht ausprogrammiert wird, dann bedeutet dies, dass diese abstrakte Methode an die Unterklasse vererbt wurde. Da sich dann in dieser Unterklasse eine abstrakte Methode befindet, muss die Klasse abstrakt sein (d.h. sie muss mit abstract gekennzeichnet werden).

4) Wenn man eine Klassenhierarchie (bzgl. Vererbung) entwickelt, ist es oft sinnvoll, die hierarchisch am weitesten oben stehende Klasse abstract zu machen.

6.6.2 Zuweisungen

Man darf zwar von abstrakten Klassen keine Objekte erzeugen. Man kann aber Variablen vom Typ dieser Klasse (bzw. Interfaces) deklarieren. Einer vom Typ einer abstrakten Klasse definierten Variable kann man nun eine Instanz einer Klasse zuweisen, die von dieser abstrakten Klasse abgeleitet ist.

Beispiel (hat Bezug zum obigen Programm):
Folgendes ist nicht erlaubt, da ein Exemplar der Klasse angelegt wird.

```
Nutztier n = new Nutztier();
```

Folgendes ist erlaubt, da nur eine Variable mit dem Typ der Klasse deklariert wird, aber kein Exemplar dieser Klasse angelegt wird.

```
Nutztier n;
```

Folgendes ist erlaubt, da ein Array der Länge 5 angelegt wird. Die einzelnen Elemente des Arrays sind mit "null" vorbelegt. Es wurde aber für kein Element ein Exemplar dieser Klasse angelegt.

```
Nutztier[] nn = new Nutztier[5];
```

Folgendes ist nicht erlaubt, da für ein Element des Arrays ein Exemplar der Klasse angelegt wird.

```
nn[2] = new Nutztier();
```

Dieser Variable vom Typ "Nutztier" kann man nun eine Instanz einer Klasse zuweisen, die von "Nutztier" abgeleitet ist.

```
Kuh k1 = new Kuh();
Nutztier n1;
n1 = k1;                        // erlaubt
k1 = n1;                        // nicht erlaubt
nn[3] = new Kuh();
```

Einem Objekt der Unterklasse kann kein Objekt der Oberklasse zugewiesen werden, aber umgekehrt, kurz:
"Unterklasse = Oberklasse" ist nicht erlaubt.
"Oberklasse = Unterklasse " ist erlaubt.

6.7 Interfaces (Schnittstellen)

Jede Klasse sollte - zumindest zu Testzwecken - jeweils die Methode printAllAttributs() enthalten, mit der die Attribute einer Klasse auf dem Bildschirm ausgegeben werden.
Um den Programmierer zu zwingen, diese Methode zu implementieren (auszuprogrammieren), könnte man sie (den Methodenkopf) in einer abstrakte Klasse ablegen. Es ist aber nicht besonders sinnvoll printAllAttributs() als abstrakte Methode in die Klasse Nutztier zu stecken, da printAllAttributs() nicht speziell was mit der Klasse Nutztier zu tun hat, sondern in jeder Klasse vorkommt.
Es ist sinnvoller, diese Methode z.B. in eine mit AllAttributs bezeichnete Klasse zu stecken und von dieser Klasse die Methode printAllAttributs() zu erben.
Da es in Java keine Mehrfachvererbung gibt, (konkret: wenn z.B. Kuh Unterklasse von Nutztier ist, kann Kuh nicht auch noch Unterklasse von der Klasse AllAttributs sein), hat man sich den Trick mit dem **Interface** einfallen lassen.
Eine Klasse kann beliebig viele Interfaces "einbinden" (mit dem Wort **implements**). Ein Interface ist so etwas ähnliches wie eine Klasse, nur dass dort von den Methoden nur die **Methodenköpfe** (mit den Parametern) angegeben werden dürfen (wie bei abstrakten Klassen). Die komplette Methode (mit Methodenrumpf) muss dann in der Klasse implementiert werden, die dieses Interface "einbindet".
Ein Interface besteht im Unterschied zu einer abstrakten Klasse **nur** aus Methodenköpfen (mit den Parametern).
Ein Interface muss in einer **eigenen Datei** erstellt werden, wenn es public ist (es gilt also das gleiche Schema wie bei Klassen: 2 public Klassen können nicht in der gleichen Datei sein).
Ein Interface ist von der Sache her abstract und es ist deswegen nicht nötig es als **abstract** zu deklarieren (dies sollte auch nicht gemacht werden).
In einem Interface dürfen nur Klassenvariable (mit public, static, final), aber keine "normalen" Attribute definiert werden.
Alle Methoden in Interfaces sind automatisch abstrakt und public. Deshalb sollten die Bezeichner public und abstract bei der Deklaration von Methoden in Interfaces nicht verwendet werden.
Dagegen müssen bei der Implementierung einer Schnittstelle die Methoden in den Unterklassen mit public implementiert werden, da die Methoden in Interfaces immer automatisch public sind.

Beispiel für ein Interface:

```
public interface AllAttributs{
  void printAllAttributs();
}
```

Beispiel für das "Einbinden" des Interface in die Klasse Kuh:

```
class Kuh extends Nutztier implements AllAttributs{
  // hier werden die Attribute und Methoden
  // der Klasse Kuh definiert.
  // ...

  // hier muß die entsprechende Methode des Interface
  // implementiert werden
  public void printAllAttributs(){
    System.out.println("Kuh-Name= "+getName());
    System.out.println("Kuh-Alter= "+getAlter());
    System.out.println("Kuh-Milchleistung= "+milchLeistung);
    System.out.println("Kuh-Literpreis= "+literpreis);
    System.out.println("Kuh-Preis= "+getPreis());
    System.out.println();
  }

  // hier muß die entsprechende Methode der abstrakten
  // Klasse implementiert werden
  public double getTierwert(){
    return(10 * milchLeistung);
  }
}
```

6.8 Polymorphie

Abhängig vom Zufall soll entweder ein Objekt der Klasse Kreis oder ein Objekt der Klasse Rechteck erzeugt werden. In jedem dieser zwei Klassen soll es die Methode (siehe unten) zeichneGeoForm() geben, die jeweils die Werte der Attribute (Datenmember) der Klasse ausgibt. Dazu gibt es die Technik der Polymorphie, mit der automatisch die JVM feststellt, ob zeichneGeoForm() des Objekts der Klasse Kreis oder des Objekts der Klasse Rechteck aufgerufen wird. Dazu wird die Technik des late binding verwendet.
In Java wird late binding **automatisch** gemacht. Es muss **nicht** wie in C++ das Schlüsselwort "virtual" verwendet werden.
Damit Polymorphie verwendet werden kann, müssen die Klassen Kreis und Rechteck eine gemeinsame Oberklasse besitzen, die hier mit GeoForm bezeichnet wird.

6.8.1 Abstrakte Methoden und Polymorphie

Wenn der Programmierer "vergessen" hat, die Methode in einer Unterklasse zu überschreiben, kann diese Methode nicht mehr die Methode der Oberklasse verdecken.
Deshalb wird dann die Methode der Oberklasse aufgerufen.
Mache dazu den Test: Kommentiere z.B. die Methode zeichneGeoForm() in der Klasse Rechteck aus (Programm unten).
Damit der Programmierer - falls er vergessen hat die Methode zeichneGeoForm() in den Unterklassen zu implementieren - durch eine Fehlermeldung des Compilers unterstützt wird, ist es hier vorteilhaft, die Methode zeichneGeoForm() abstrakt zu machen (damit muss die Klasse auch abstrakt sein und als solche bezeichnet werden).

Wenn nämlich jetzt der Programmierer "vergessen" hat, die Methode in einer Unterklasse auszuprogrammieren (überschreiben), gibt es einen Compilerfehler:
Wenn eine abstrakte Methode in einer Unterklasse nicht ausprogrammiert wird, wird sie in diese Unterklasse vererbt. Da aber eine abstrakte Methode in der Unterklasse zur Folge hat, dass diese Unterklasse selbst wieder abstrakt ist, kann von dieser Unterklasse kein Exemplar gebildet werden. Im Programm unten wären dann folgende Anweisungen nicht möglich:

```
f = new Kreis(2);
...
f = new Rechteck(10,20);
```

Weil durch "new Kreis(2)" und "new Rechteck(10,20)" Exemplare der Klassen Kreis bzw. Rechteck erzeugt werden, diese aber abstrakt sind, meldet der Compiler einen Fehler.

6.8.2 Beispiel

```
public class Polymorphie0 {
     public static void main(String[] args) {
          double zufall;
          GeoForm f;

          zufall = Math.random();
          if (zufall<0.5){
               f = new Kreis(2);
          }
          else{
               f = new Rechteck(10,20);
          }
          f. zeichneGeoForm();
     }
}
```

```
class GeoForm {
     private String name;

     public GeoForm(String pname){
          setName(pname);
     }

     public void setName(String pname){
          name = pname;
     }

     public String getName(){
          return(name);
     }

     public void zeichneGeoForm(){
     }
}
```

```
class Rechteck extends GeoForm{
     private double laenge;
     private double breite;

     public Rechteck(double plaenge, double pbreite){
          super("Rechteck");
          setLaenge(plaenge);
          setBreite(pbreite);
     }

     public void setLaenge(double plaenge){
          laenge = plaenge;
     }

     public void setBreite(double pbreite){
          breite = pbreite;
     }

     public double getLaenge(){
          return(laenge);
     }

     public double getBreite(){
          return(breite);
     }

     public void zeichneGeoForm(){
          System.out.println("Laenge= "+getLaenge());
          System.out.println("Breite= "+getBreite());
     }
}

class Kreis extends GeoForm{
     private double radius;

     public Kreis(double pradius){
          super("Kreis");
          setRadius(pradius);
     }

     public void setRadius(double pradius){
          radius = pradius;
     }

     public double getRadius(){
          return(radius);
     }

     public void zeichneGeoForm(){
          System.out.println("Radius= "+getRadius());
     }
}
```

Bemerkungen:
1) In den Fachklassen (das sind Klassen, in denen nur fachspezifische Aktionen gemacht werden), Rechteck und Kreis gibt es Methoden, die Ausgaben auf den Bildschirm bringen.
Das widerspricht der sogenannten **Dreischichten-Architektur**.
Die Dreischichten-Architektur verlangt, dass eine Trennung zwischen den Fachklassen und den Oberflächenklassen (das sind die Klassen, die sich mit der Eingabe und Ausgabe beschäftigen) herrscht. Das bedeutet, dass die Fachklassen "stumm" sein müssen.

2) Diese Trennung könnte man z.B. so realisieren, dass in den Klassen Rechteck und Kreis die Methoden zeichneGeoForm() eine Zeichenkette zurückgeben.
Diese Zeichenkette wird dann in main(), also einer anderen, davon getrennten Klasse auf dem Bildschirm ausgegeben.
Diese Zeichenkette kann entweder über return, oder über einen Parameter in der Methode (zeichneGeoForm()) zurückgegeben werden.
Die Zeichenkette kann mit Hilfe der Klasse String oder StringBuffer gebastelt werden.
(Welche Klasse ist bei der Verwendung eines Parameters in zeichneGeoForm(...) besser?)

3) Realisieren Sie die Zurückgabe der Zeichenkette mit
a) return und mit
b) einem Parameter

6.9 Ausnahme (Exception)

6.9.1 Motivation

6.9.1.1 Umgang mit Fehlern im Alltag

Im Englischunterricht in der Schule werden viele Fehler gemacht (Wörter falsch sprechen oder schreiben), in Firmen werden oft Fehler gemacht (z.B. Produktionsfehler), überall werden Fehler gemacht ...
Damit diese Fehler keine Auswirkungen haben (z.B. in die Hände des Kunden fallen), werden sie einer internen Kontrolle unterworfen (Qualitätssicherung?) und dann selbst gleich beseitigt, oder es werden die Fehler an eine übergeordnete Instanz weitergeleitet, die sie dann selbst bearbeitet oder wieder weiterleitet ...

6.9.1.2 Umgang mit Fehlern in Java

Zur Laufzeit eines Javaprogramms können bei der Abarbeitung einer Anweisung sogenannte Ausnahmen (Exception) eintreten - das sind außerordentliche - durch Fehler verursachte - Bedingungen, wie z.B. die Division durch Null oder der Zugriff auf ein sich außerhalb eines Feldes befindlichen Elements - die es nicht gestatten das Programm weiterhin "normal" abzuarbeiten.
In derartigen Fällen wird automatisch in der Methode, in der die Ausnahme aufgetreten ist, ein sogenanntes Fehlerobjekt (der Klasse Throwable oder einer Unterklasse davon) erzeugt und ausgeworfen.

Bemerkung:
1) Nur ganze Zahlen wie z.B. 16/0 bewirken bei einer Division die Erzeugung eines Fehlerobjekts. Bei float- oder double-Zahlen bewirkt dies keine Ausnahme, sondern liefert einen speziellen Wert ("unendlich"). Für die Abfrage dieser Werte gibt es in den Klassen Float und Double entsprechende Konstanten und Methoden. Bitte in einem Programm ausprobieren: 16.0/0 bzw. 16/0

6.9.1.3 Wer erzeugt Fehlerobjekte und wirft sie aus ?

Die Java Virtual Machine kann Fehlerobjekte erzeugen und auswerfen.
Aber genauso hat der Programmierer die Möglichkeit, eigene Fehlerobjekte zu erzeugen und diese auszuwerfen.

6.9.2 Wann Fehler einfangen

Für manche (aber nicht alle) Anweisungen verlangt die Java Language Specification (JLS), dass die im Fehlerfall geworfenen Fehlerobjekte "eingefangen" (abgefangen und bearbeitet) werden (programmtechnisch realisiert mit **try ... catch**), oder das Fehlerobjekt an die umgebende Methode weitergeworfen wird (programmtechnisch realisiert durch den Bezeichner "**throws**" im Methodenkopf der umgebenden Methode).
Man sagt auch: "Problembereiche einzäunen". Näheres dazu später.

Gemäß der Java Language Specification (JLS) gilt:
1) Fehler, die Fehlerobjekte vom Klassentyp Error oder RuntimeException (oder deren Unterklassen) werfen, müssen (können aber) nicht vom Programmierer "eingefangen" bzw. "weitergeworfen" werden. Man nennt sie ungeprüfte Ausnahmen (unchecked Exceptions).
Die Fehlerobjekte der - siehe unten **- fett dargestellten** Klassen (und deren Unterklassen) **müssen** also **nicht** vom Programmierer **eingefangen** werden.

2) Die Fehlerobjekte der - siehe unten - **NICHT fett dargestellten** Klassen (und deren Unterklassen) **MÜSSEN** vom Programmierer **eingefangen** werden.

Komplizierter ausgerückt:
Alle Fehlerobjekte der Klasse "Throwable" und deren Unterklassen (außer der Unterklasse RuntimeException bzw. Unterklassen davon und der Klasse Error bzw. Unterklassen davon) müssen vom Programmierer "eingefangen" bzw. "weitergeworfen" werden .

6.9.2.1 Bemerkung

Die Eingabe von Daten über Tastatur wird in Java auch über Exceptions realisiert.

6.9.3 Wie Fehler einfangen

Mit der try {...}... catch {...} Anweisung kann man Fehlerobjekte wieder einfangen.
Diese Anweisung besteht aus einem try-Bereich und dem catch-Bereich, der aus mindestens einem catch-Block besteht.

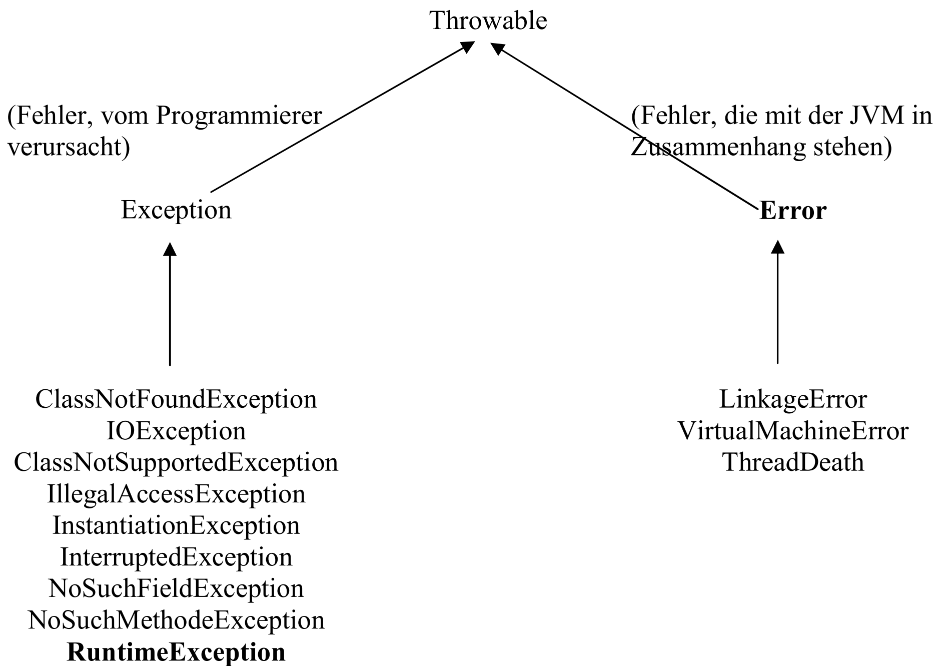
```
try{
  Anweisung(en);
}
catch(Throwable t){
  Anweisung(en);
}
finally{
  Anweisung(en);
}
```

Die (kritische) Anweisung, die einen Fehler verursachen kann, schreibt man in den try-Bereich der try {...}... catch {...} Anweisung. Die catch-Bezeichner werden auch Ausnahme-Handler genannt. Es muss **mindestens** einen catch-Bezeichner geben, der dann im Fehlerfall abgearbeitet wird (Ausnahme: Es gibt ein try ohne catch aber dann mit finally; näheres weiter unten).
Der formale Parameter des catch-Bezeichners (hier mit t bezeichnet) muss als Klassentyp "Throwable" oder eine Unterklasse davon haben.
Der finally-Teil ist optional, kann also weggelassen werden. Falls aber finally benutzt wird, werden die in im finally-Teil stehenden Anweisung auf jeden Fall abgearbeitet (egal, ob ein ein Fehlerobjekt geworfen wird oder nicht).

6.9.4 Ausnahmen und Klassen

Die sich in der Vererbungshierarchie am weitesten oben befindliche Klasse ist "Throwable". darunter befinden sich die Unterklassen.

6.9.4.1 Klassen Hierarchie



6.9.4.1.1 Unterklassen von LinkageError

AbstractMethodError: Aufruf einer abstrakten Methode
NoSuchFieldError: Zugriff auf eine nicht deklarierte Variable
NoSuchMethodError: Zugriff auf eine nicht deklarierte Methode

6.9.4.1.2 Unterklassen von VirtualMachineError:

OutOfMemoryError, StackOverFlowError, InternalError, UnknownError

6.9.4.1.3 Unterklassen von RuntimeException:

ArithmeticException
NullPointerException
NumberFormatException: Zahlen liegen in einem falschen Format vor, wenn z.B. "abc" in den Datentyp int konvertiert werden soll.
ArrayIndexOutOfBoundsException: Zugriff mit zu großem oder kleinen Index auf ein Feldelement

Bemerkung:
Eine Unterklasse der Klasse IOException ist die Klasse FileNotFoundException. Von dieser wird ein Objekt geworfen, wenn eine Datei, auf die zugegriffen wird, nicht existiert.
Eine Unterklasse der Klasse IOException ist die Klasse EOFException. Von dieser wird ein Objekt geworfen, wenn der Dateizeiger über das Dateiende hinaus verschoben wurde und dann noch von dieser Datei gelesen werden soll.

6.9.5 Beispiel (geprüfte Ausnahme: Compiler bringt Fehler)

```
public class MainException1 {
  public static void main(String[] args) {
    // Diese Anweisung verursacht beim Compilieren
    // eine Fehlermeldung.
    Class myC = Class.forName("Affe");
    String myS = myC.getName();
    System.out.println("myS="+myS);
  }
}
```

Bemerkungen:
1) Die Methode forName(...) der Klasse Class "lädt" eine Klasse mit dem angegebenen Namen.
Im Beispiel unten soll eine Klasse mit dem Namen Affe geladen werden.

2) Die Methode getName() der Klasse Class liefert den Namen der Klasse

3) Die Methode forName wirft eine Exception vom Klassentyp ClassNotFoundException aus und muss laut obiger Tabelle eingefangen werden.
Deshalb gibt es beim Kompilieren dieses Programms eine Fehlermeldung des Kompilers.

6.9.6 Beispiel (obiges, verbessertes Beispiel)

```
public class MainException2 {
  public static void main(String[] args) {
    try{
      Class myC = Class.forName("Affe");
      String myS = myC.getName();
      System.out.println("myS="+myS);
    }
    catch(Throwable t){
      System.out.println("Klasse existiert nicht");
      System.out.println("Klasse inexistent:" +t.toString());
      t.printStackTrace();
    }
  }
}
```

Bemerkungen:
1) Der Programmierer fängt durch eine try ... catch Anweisung das Fehlerobjekt ein.
2) Die Methode toString() liefert eine Zeichenkette, in der Infos über das Objekt stehen.
3) Die Methode printStackTrace() ruft intern zuerst die Methode toString auf und bringt dann noch die Zeilen des Quellcodes, wo sich ein Fehler ereignet hat.

Alternativ kann das letzte Beispiel auch so programmiert werden:

6.9.7 Beispiel (Alternative zum letzten Beispiel)

```
public class MainException3 {
  public static void main(String[] args) throws Throwable {
    Class myC = Class.forName("Affe");
    String myS = myC.getName();
    System.out.println("myS="+myS);
    }
}
```

Bemerkungen:
1) In der die Anweisung Class.forName("Affe") umgebenden Methode (hier also main) wird direkt nach dem Bezeichner "throws" die Klasse des Fehlerobjekts angegeben, das im Fehlerfall geworfen wird. In dem Beispiel ist das die Klasse Throwable.
Wenn das Fehlerobjekt von main (durch throws) weitergeworfen wird, wird von der JVM automatisch der sogenannte **Standard-Ausnahmehandler** aufgerufen und dieses Fehlerobjekt eingefangen und bearbeitet.
Es muß also kein try ... catch (hier in der Methode main(...)) verwendet werden, da ein eventuell auftretendes Fehlerobjekt an die umgebende Methode (hier also main) weitergeleitet wird.

6.9.8 Beispiel (ungeprüfte Ausnahme)

```
public class MainException4 {
  public static void main(String[] args) {
    double z;
    z = 15/0;
  }
}
```

Bemerkungen:
1) Die Anweisung `z=15/0` wirft ein Fehlerobjekt vom Klassentyp ArithmeticException (Unterklasse von RuntimeException) aus und muss (siehe oben) nicht eingefangen werden. Deshalb gibt es beim Kompilieren dieses Programms keinen Kompilerfehler.
Direkt nach der Erzeugung des Fehlerobjekts, wird der zu main gehörige Standard-Ausnahmehandler der JVM aufgerufen und das Programm beendet.

6.9.9 Genauere Beschreibung der Fehlerbehandlung

Voraussetzung:
Eine Methode wirft im Fehlerfall ein Fehlerobjekt aus.
Der Programmierer ruft diese Methode auf und es wird ein Fehlerobjekt geworfen.

Möglichkeiten des Programmierers:
1. Möglichkeit:
Wenn das Fehlerobjekt eingefangen werden muss, muss der Programmierer try ... catch oder throws verwenden.

2. Möglichkeit:
Wenn das Fehlerobjekt nicht eingefangen werden muss, kann der Programmierer try ... catch oder throws verwenden oder das Fehlerobjekt einfach ignorieren.
Falls der Programmierer das Fehlerobjekt ignoriert (keinen Ausnahme-Handler programmiert hat), wird von der JVM sofort nach Erzeugung des Fehlerobjekts automatisch der standardmäßig zu main gehörige Standard-Ausnahmehandler aufgerufen, der den Typ der Ausnahme und die Methodenaufrufe, die zur Ausnahme geführt haben, ausgibt.
Danach wird dann das Programm sofort beendet.

Im Folgenden behandeln wir die Möglichkeiten try ... catch oder throws:

6.9.9.1 Fall: Programmierer benutzt throws

Da sich dieser (wie jeder) Methodenaufruf innerhalb einer umgebenden Methode befinden muss (z.B. in main), muss im Methodenkopf der umgebenden Methode mit dem Bezeichner "throws" angegeben werden, welcher Fehlerklasse das ausgeworfene Fehlerobjekt angehört. Man hat das Problem also eine Ebene weiter zum Methodenaufruf der umgebenden Methode verschoben.
Wenn man dort auch nicht try ... catch benutzen will, muss man wieder mit "throws" das Fehlerobjekt an die umgebende Methode weiterwerfen.

6.9.9.1.1 Beispiel 1

```
public class MainException5{
  public static void main(String[] args){
    double erg;
    try{
      erg=myDivision(15, 0);
    }
    catch(Throwable t){
      System.out.println("Division durch 0 unmöglich");

    }
  }

  public static double myDivision(int a, int b) throws
                                Throwable{
    // muß nicht mit try...catch einezäunt werden
    double erg=a/b;
    return erg;
  }
}
```

Bemerkungen:
1) Da man (zu Demozwecken) in der Methode myDivision(...) nicht mit try ... catch das Fehlerobjekt einfangen will, hat man es einfach durch den Bezeichner throws in die umgebende Methode (dies ist hier main) weitergeleitet. Dort wird es dann mit try ... catch eingefangen. Falls man es dort nicht mit try ... catch einfangen will, kann man es wieder weiterleiten, indem man der Methode main den Bezeichner throws folgen lässt.

2) Da die Division ein Fehlerobjekt erzeugt, das durch den Programmierer nicht eingefangen werden muss, würde man im obigen Beispiel kein try ... catch oder throws benötigen.

3) ty ... catch ohne catch aber dann mit finally
setzt voraus, daß ein auftretendes Fehlerobjekt an die umgebende Methode weitergeleitet wird, aber vorher noch die Anweisungen des finally-Teils, abgearbeitet werden.

```
public void testMethode() throws Throwable{
  try{
    Anweisung(en);
  }
  finally{
    Anweisung(en);
  }
}
```

6.9.9.1.2 Beispiel 2

```
public class MainException5a{
  // Angabe von throws in main unnötig, da die hier in main
  // durch try .. catch das Fehlerobjekt eingefangen wird.
  // public static void main(String[] args) throws Throwable {
  public static void main(String[] args){
    double erg;
    try{
      erg=myTaschenrechner(5, 3, '/');
    }
    catch(Throwable t){
      System.out.println("Meine Fehlermeldung:");
      // mit getMessage() wird der String aus t geholt, der in der
      // Methode myTaschenrechner im Fehlerobjekt gesetzt wurde.
      System.out.println(t.getMessage());
      // auch möglich wie oben: toString() bzw. printStackTrace()
    }
  }

  public static double myTaschenrechner(int a, int b, char
                              rechenzeichen) throws Throwable{
    double erg;
    Throwable myt;

    if(rechenzeichen=='+'){
      erg=a+b;
    }
    else if(rechenzeichen=='-'){
      erg=a-b;
    }
    else{
      // Der durch myt = Throwable(...) festgelegte Text kann
      // mit getMessage() wieder gelesen werden (siehe main())
      myt = new Throwable("Taschenrechner kann nur + und -");
      // auch möglich, falls die Zeile oberhalb fehlt:
      // throw new Throwable("Mein Rechner kann nur + und -");
      // mit throw wird ein Fehlerobjekt geworfen.
      throw myt;
      // Das geworfene Fehlerobjekt myt wird oben im
      // Methodenkopf mit throws an die die umgebende Methode
      // weiter geworfen; return erg wird nicht mehr gemacht!
    }
    return erg;
  }
}
```

Bemerkungen:
Dieses obige Programm demonstriert eine Anweisung, in der der Programmierer das Werfen eines Fehlerobjekts (mit **throw**) in die umgebende Methode veranlasst.
Im Gegensatz zur Division durch 0 (wo die JVM das Fehlerobjekt wirf), veranlasst hier also der Programmierer durch die Verwendung des Bezeichnes **throw** das Werfen eines Fehlerobjekts!
In der umgebenden Methode (hier also main) muss dann das Fehlerobjekt mit try .. catch eingefangen werden oder wiederum an die dort umgebende Methode (mit throws) weitergeworfen werden.

6.9.9.2 Fall: Programmierer benutzt try ... catch

6.9.9.2.1 1. Fall

Wenn die Anweisungen des try-Bereichs ausgeführt sind, ohne dass ein Fehler passiert (kein Fehlerobjekt erzeugt wird), dann macht das Programm nach dem letzten catch-Block weiter (bzw. nach dem finally-Teil, falls es ihn gibt. Der finally-Teil wird vorher aber noch abgearbeitet).

6.9.9.2.2 2. Fall

Wenn bei einer Anweisungen des try-Bereichs ein Fehler geschieht, (ein Fehlerobjekt erzeugt wird), dann sucht das Programm (**ohne** die weiteren Anweisungen des try-Bereichs abzuarbeiten) sofort den **ersten** Ausnahme-Handler, dessen Parameter zu dem geworfenen Fehlerobjekt **passt** (es werden also nicht mehrere Ausnahme-Handler abgearbeitet). Das bedeutet, dass der Klassentyp des Fehlerobjekts und der Klassentyp des Parameters (in catch) gleich sind, bzw. der Klassentyp des Fehlerobjekts eine Unterklasse des Klassentyps des Parameters ist (siehe Konvertierung beim Methodenaufruf: das Objekt wird vor der Übergabe an den Parameter automatisch in das Objekt der Oberklasse konvertiert).
Dann macht das Programm nach dem letzten catch- Bezeichner weiter (bzw. nach dem finally-Teil, falls es ihn gibt. Der finally-Teil wird vorher aber noch abgearbeitet)).
Es wird also nicht notwendigerweise das Programm beendet!!

Beispiel:
```
public class MainException6{
  public static void main(String[] args){
    double erg;
    try{
      erg = 15 / 0;
    }
    catch(Exception mye){       // Zeile x1
      System.out.println("Division durch 0 unmöglich");
    }
    catch(Throwable myt){       // Zeile x2
      System.out.println("Irgendein anderer Fehler");
    }
  }
}
```

Bemerkungen:
1) Die Anweisung `z=15/0` wirft ein Fehlerobjekt vom Klassentyp ArithmeticException aus. Da aber ArithmeticException eine Unterklasse von RuntimeException und RuntimeException eine Unterklasse von Exception ist, wird dieses Fehlerobjekt dem ersten passenden Parameter, nämlich mye, zugewiesen.

2) Hätte man im obigen Beispiel Zeile x1 mit Zeile x2 vertauscht,
```
catch(Throwable myt){       // Zeile x2
...
catch(Exception mye){       // Zeile x1
```
würde der Compiler eine Fehlermeldung bringen, da das Programm dann nie zu Zeile x1 kommen würde: Da jedes geworfene Fehlerobjekt vom Klassentyp Throwable ist, würde es sofort vom ersten passenden Ausnahme-Handler "eingefangen" werden (das ist hier Zeile x2).

6.9.9.2.3 3. Fall

Wenn die Anweisungen des try-Bereichs ausgeführt sind und ein Fehler passiert (ein Fehlerobjekt erzeugt wird) und das geworfenen Fehlerobjekt zu keinem Parameter eines Ausnahme-Handlers passt, dann wird das Fehlerobjekt an die umgebende aufrufende Methode weitergeworfen (falls ein finally-Teil existiert wird, wird dieser noch ausgeführt).

6.9.10 Eigene Fehlerklassen basteln

Bei der ganzzahligen Division durch Null (z.B. 16/0) wird automatisch, d.h. ohne Zutun des Programmierers, ein Fehlerobjekt erzeugt. Der Programmierer kann aber auch durch "throw" selbst ein Fehlerobjekt einer schon existierenden Fehlerklasse, wie z.B. ArithmeticException oder NullPointerException, werfen.
Außerdem kann der Programmierer aber auch selbst eigene Fehlerklassen basteln und Objekte davon werfen.

In dem Beispiel unten soll in einer Methode festgestellt werden, ob in einer Zeichenkette ein ? enthalten ist. Wenn dies der Fall ist, kann diese Zeichenkette nicht als ein Verzeichnisname benutzt werden, da der Explorer in Windows dies nicht zulässt.
In diesem Fall soll dann ein Objekt einer selbst gebastelten Klasse geworfen werden.
Diese Fehlerklasse enthält im Wesentlichen eine Methode, die eine entsprechende Fehlermeldung auf dem Bildschirm ausgibt.

Bemerkungen:
1) "throw" veranlasst die Unterbrechung der "normalen" Abarbeitungsfolge des Programms: Es wird dann sofort (d.h. Anweisungen, die throw innerhalb derMethode folgen würden, wie z.B. return, werden nicht mehr ausgeführt) der entsprechende Ausnahme-Handler gesucht, der zu dem geworfenen Fehlerobjekt passt.

2) Die selbstgebastelte Klasse muss Unterklasse der Klasse Throwable oder einer Unterklasse davon sein.

6.9.11 Aufgaben

1) Programmieren Sie einen Kindertaschenrechner mit folgenden Eigenschaften:
a) Der Kindertaschenrechner soll bei Eingabe einer negativen Zahl sofort eine entsprechende Meldung bringen und dann das Programm sofort beenden (andere Idee: den Anwender die Eingabe wiederholen lassen).

b) Der Kindertaschenrechner soll im Fall eines negativen bzw. nicht ganzzahligen Ergebnisses sofort eine entsprechende Meldung bringen und dann das Programm sofort beenden (andere Idee: den Anwender die Eingabe wiederholen lassen).

c) Der Taschenrechner soll bei Division durch Null sofort eine entsprechende Meldung bringen und dann das Programm sofort (ohne Rechnung) beenden.

Bemerkungen:
1) siehe nächste Seite:
Die Methode indexOf (int zeichen) der Klasse String sucht das erste Vorkommen eines Zeichens (einer Zeichenkette) und gibt den zugehörigen Index zurück. Falls dieses Zeichens (Zeichenkette) nicht vorkommt, liefert die Methode -1 zurück.

6.9.11.1 Beispiel

```
public class MainException7 {
  public static void main(String[] args){
    Ordner v = new Ordner("Home?");
    try{
      v.checkOrdnerName();
    }
    catch(MyException_FalscherOrdnerName t){
      t.printFehlermeldung();
    }
  }
}

class Ordner{
  private String ordnerName;

  public Ordner(String pname){
    setOrdnerName(pname);
  };

  public void setOrdnerName(String pname){
    ordnerName = pname;
  }

  public String getOrdnerName(){
    return(ordnerName);
  }

  public void printAll(){
    System.out.println("Name des Ordners="+ordnerName);
  }

  public boolean checkOrdnerName() throws
                              MyException_FalscherOrdnerName{
    int zahl = (int)('?');
    if(ordnerName.indexOf(zahl)!=-1){
      throw new MyException_FalscherOrdnerName("Ordername
                                  "+ordnerName+" enthält ?");
    }
    return(true);
  }
}

class MyException_FalscherOrdnerName extends Throwable{
  private String zk;

  public MyException_FalscherOrdnerName(String pzk){
    zk=pzk;
  }

  public void printFehlermeldung(){
    System.out.println("Fehlermeldung: "+zk);
  }
}
```

Aufgaben

I) Mit den Methoden nextInt() bzw. nextDouble() der Klasse Scanner können Zahlen über Tastatur eingegeben werden.
Können dabei Fehlerobjekte geworfen werden, bzw. wie müssen diese verarbeitet werden (vom Programmierer eingefangen oder weitergeworfen werden)?
Siehe Java-Doku.

II) In der Java-Doku ist einer der RandomAccessFile-Konstrukturen wie folgt deklariert:

```
public RandomAccessFile(String name,
                        String mode)
         throws FileNotFoundException
```

Weiter unten in der Doku steht:
Throws:
- IllegalArgumentException
- FileNotFoundException
- SecurityException

Frage:
Warum kann dieser Konstruktor 3 verschiedene Exceptions werfen?
Laut der Deklaration oben wird aber nur eine Exception mit throws geworfen.
Und die Exceptions IllegalArgumentException und SecurityException sind nicht Unterklassen von FileNotFoundException.

Auszug aus den Java-Doku: (<--- bedeutet Oberklasse von)

---

java.lang.Object <--- java.lang.Throwable <--- java.lang.Exception <---
java.lang.RuntimeException <--- java.lang.IllegalArgumentException

---

java.lang.Object <--- java.lang.Throwable <--- java.lang.Exception <---
java.io.IOException <--- java.io.FileNotFoundException

---

java.lang.Object <--- java.lang.Throwable <--- java.lang.Exception <---
java.lang.RuntimeException <--- java.lang.SecurityException

III) Ist der fianally-Bezeichner überflüssig, oder ist es nur programmtechnisch eleganter mit ihm zu arbeiten? Vergleichen Sie die zwei folgenden Programme:

```
//Methode 1
public void f1()throws
                  Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
  }
  finally{
    schliesse_Datei
  }
}
```

```
//Methode 2
public void f2() throws
                 Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
    schliesse_Datei
  }
  catch (Exception ex){
    schliesse_Datei
    throw ex;
  }
}
```

Bem:
try...catch ohne catch mit finally ist möglich (siehe früher).

Lösungen:
II) (siehe oben die Bedingungen über einzufangende und nicht einzufangende Fehler).

1) Wenn in der Methode
public RandomAccessFile(String name, String mode) throws FileNotFoundException
ein Ausnahmeobjekt der Klasse **IllegalArgumentException** oder **SecurityException** geworfen wird, also Klassen der Oberklasse RuntimeException, dann müssen diese nicht eingefangen oder mit throws weitergeworfen werden.

2) Wenn dagegen in der Methode
public RandomAccessFile(String name, String mode) throws FileNotFoundException
ein Ausnahmeobjekt der Klasse **FileNotFoundException** geworfen wird, muss dies entweder mit try .. catch eingefangen oder mit throws weitergeworfen werden.
Wie man in der Deklaration von
public RandomAccessFile(String name, String mode) throws **FileNotFoundException**
sieht, wird es nicht eingefangen, sondern mit throws weitergeworfen.

III)

```
//Methode 1
public void f1()throws
                 Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
  }
  finally{
    schliesse_Datei
  }
}
```

```
//Methode 2
public void f2() throws
                  Throwable{
  try{
    öffne_Datei
    bewege_Dateizeiger
    schliesse_Datei
  }
  catch (Exception ex){
    schliesse_Datei
    throw ex;
  }
}
```

1) Diskussion
finally ist zwar überflüssig, aber es hat Vorteile:

Fall1:
Dateizeiger wird über das Dateiende hinaus verschoben:
Deshalb wird eine Exception ausgelöst.
In der Methode1 wird vor dem Werfen des Fehlerobjekts noch die Anweisung im finally-Teil ausgeführt (also schliesse Datei). Dann wird das Fehlerobjekt in die umgebende Methode geworfen (weil es kein catch gibt, muß im Methodenkopf der umgebenden Methode "throws" vorkommen).
In der Methode2 wird das Fehlerobjekt geworfen und im catch-Teil eingefangen, d.h. dort wird dann weitergemacht mit schliesse_Datei und throw ex. Dieses mit throw ex erzeugte Fehlerobjekt wird dann in die umgebende Methode geworfen.
D.h. Methode1 und Methode2 machen semantisch das Gleiche.

Fall2:
Dateizeiger wird nicht über das Dateiende hinaus verschoben. Methode1 und Methode2 machen dann die folgenden Anweisungen:
öffne_Datei, bewege_Dateizeiger, schliesse_Datei
D.h. Methode1 und Methode2 machen semantisch das Gleiche.

2)
Methode2 verstösst aber gegen DRY (don't repeat yourself), weil doppelter Code produziert wird (nämlich schliesse_Datei).

7 Reihenfolge der Initialisierungen (Vertiefung)

Um zu klären, wie die JVM beim Anlegen eines Objekts die Attribute initialisiert, müssen vorher noch ein paar Begriffe definiert werden:

7.1 Begriffe

7.1.1 Exemplarinitialisierer (nicht statischer Initialisierungsblock)

```
class MyTest{
  {
  System.out.println("Hallo Welt");
  }

  MyTest{
  }
}
```

Ein Exemplarinitialisierer besteht aus Anweisungen, die zwischen den geschweiften Klammern stehen. Er wird wie ein Konstruktor beim Erzeugen eines Objekts ausgeführt.
Das was der Exemplarinitialisierer macht, kann zwar auch durch einen Konstruktor erreicht werden, doch müsste man - falls man mehrere Konstruktoren hat - dann die Anweisungen des Exemplarinitialisierers jeweils in die Konstruktoren kopieren.

Empfehlung:
Exemplarinitialisierer möglichst vermeiden.

7.1.2 Statischer Initialisierungsblock

```
class MyTest{
  static
  {
  System.out.println("Hallo Welt");
  }
}
```

Der statische Block (der mit dem Bezeichner static beginnt) wird dann ausgeführt, wenn der Klassenlader eine Klasse in die Laufzeitumgebung geladen hat.

7.1.3 Statische, nicht statische Initialisierung

Eine statische Initialisierung wird durch den Bezeichner static charakterisiert, eine nicht statische Initialisierung ohne den Bezeichner static.

Beispiel:

```
class Kuh{
  // nicht statische Initialisierung
  public int gewicht = 450;
  // statische Initialisierung
  public static int literpreis = 30;
}
```

7.1.3.1 Bemerkung

Die statische Initialisierung wird vor der nicht statischen Initialisierung gemacht, obwohl diese im Quellcode der Klasse Kuh nach der nicht statischen Initialisierung gemacht wird. (siehe später)

7.1.4 Weitere Initialisierungsmöglichkeiten

Die statische Initialisierung (bzw. nicht statische Initialisierung) kann weiter unterteilt werden in folgende Möglichkeiten:

7.1.4.1 einfache Initialisierung

Beispiel:

```
class Kuh{
  // einfache nicht statische Initialisierung
  public int gewicht = 450;
  // einfache statische Initialisierung
  public static int literpreis = 30;
}
```

7.1.4.2 Initialisierung durch Aufruf einer statischen Methode

```
class Kuh{
  public int gewicht = MyPrinter.myprint("Hallo");
  public static int literpreis = MyPrinter.myprint("Hi");
}

class MyPrinter{
  public static int myprint(String ps){
    System.out.println(ps);
    return 123;
  }
}
```

7.1.4.3 Initialisierung durch Initialisierungsblock (statischer bzw. nicht statischer Initialisierungsblock)

```
class Kuh{
  {
  public int gewicht = 500;
  }

  static
  {
  public static int literpreis = 30;
  }
}
```

7.2 Reihenfolge

1) Alle Attribute aller Objekte sind prinzipiell mit den Standardwerten vorbelegt, das bedeutet, dass z.B. eine Integervariable den Wert 0 hat.
2) Alle **statischen** Attribute der **Oberklasse** des Objekts werden in der Reihenfolge ihres Auftauchens in der Klassenimplementierung initialisiert.
3) Alle **statischen** Attribute des Objekts werden in der Reihenfolge ihres Auftauchens in der Klassenimplementierung initialisiert.
4) Alle nicht statischen Attribute der Oberklasse des Objekts werden in der Reihenfolge ihres Auftauchens in der Klassenimplementierung initialisiert.
5) Der Konstruktor der **Oberklasse** wird aufgerufen.
6) Alle nicht statischen Attribute des Objekts werden in der Reihenfolge ihres Auftauchens in der Klassenimplementierung initialisiert.
7) Der Konstruktor der Klasse wird aufgerufen.

7.2.1 Bemerkungen

1) Die statischen Initialisierungen werden also zuerst gemacht, das Objekt der Oberklasse wird zuerst initialisiert.

2) Intern wird die Initialisierung so geregelt, dass wenn ein Attribut wie z.B.
public int gewicht =123;
mit 123 initialisiert wird, dies vom Compiler als Programmcode in den Konstruktor mit aufgenommen wird.

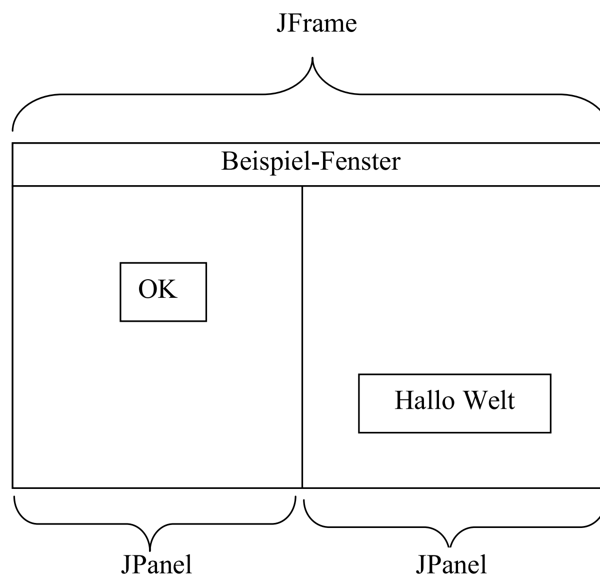
8 GUI (Graphical User Interface)

Die GUI (Graphical User Interface = Grafische Benutzeroberfläche) bietet dem Programmierer die Möglichkeit, die Fenster auf dem Bildschirm optisch ansprechend zu gestalten.
Das AWT (Abstract Window Toolkit) ist ein Package, das Klassen für die Zusammenstellung und Verwendung von grafischen Benutzeroberflächen (GUI) enthält.
Das Package Swing ist eine leistungsfähigere Alternative zu dem quick-and-dirty programmierten AWT.
Wir werden in Zukunft nur noch Swing verwenden.

8.1 Aufbau der grafischen Oberfläche

Das Prinzip der GUI in Swing sind Container (Behälter), die wiederum Container oder sichtbare Komponenten (wie z.B. Buttons, Textfelder, usw.) enthalten können.
Container sind eigentlich nur Gruppierungselemente (ähnlich Ordner im Explorer) und dienen nur zur übersichtlicheren Gestaltung.

8.1.1 Beispiel



8.1.1.1 Beschreibung

Das Hauptfenster (Top-Level-Container) ist ein Container, konkret ein Objekt der Klasse JFrame (bzw. eine von ihm abgeleitete, selbstgebastelte Unterklasse) und kann eine Menüleiste enthalten. In diesen Container (Behälter) kann man (mit dem Befehl add) wiederum Container (sozusagen Untercontainer, die wiederum Untercontainer oder sichtbare Komponenten enthalten können) oder sichtbare Komponenten (Buttons, Textfelder, Panels, usw.) montieren, die dann auf dem Bildschirm zu sehen sind.

Dieses Hauptfenster enthält hier zwei Container, konkret zwei JPanels (Zeichenflächen), auf denen etwas gezeichnet, sichtbare Komponenten oder wiederum Container angeordnet werden können. In diesem Beispiel enthalten die 2 JPanels als sichtbare Komponenten jeweils ein Button bzw. ein Textfeld.

Merke:
Jede graphische Komponente (wie z.B. Zeichenfläche, Textfeld, Schaltfläche, uw.) braucht ein Fenster (JFrame) in dem es dargestellt wird.
Man kann zwar eine DVD in einem DVD-Player abspielen lassen. Solange der DVD Player nicht am Fernseher angeschlossen ist, läuft zwar der Film aber sehen wird man ihn trotzdem nicht.

Wie zeichnet man in ein Panel?
Die Methode
public void paintComponent(Graphics g)
besitzt als Parameter einen grafischen Kontext g.
Diesen kann man sich als Leinwand vorstellen, auf die gezeichnet wird.
Wer stellt die Leinwand zur Verfügung, die irgendwann auf die Anzeigefläche des Bildschirms gelegt wird?
Diese Methode wird von der JVM bzw. automatisch von selbst augerufen (z.B. nachdem ein Fenster vergrößert (verkleinert) wurde oder nach dem Aufruf des Befehls repaint() im Programm).

Erklärung an einem Beispiel:
Voraussetzungen:
subJFrame ist eine Subklasse von JFrame
jPanel1 ist eine Subklasse von JPanel
jTextfield1 ist eine Subklasse von JTextfield
jPanel1 und jTextfield1 werden mit add... an das Fenster subJFrame montiert.
Deswegen nennt man jPanel1 und jTextfield1 die Kinder von subJFrame.
Die Kinder von subJFrame sind:
jPanel1 und jTextfield1, weil sie mit der Methode add... an das Fenster subJFrame montiert wurden.

2)
Das Kind von jPanel1 ist subJPanel, weil subJPanel an jPanel1 montiert wurde mit:
jPanel1.add(subJPanel, "Center");
Damit ist subJPanel ein Kindeskind von subJFrame.

3)
Wenn subJPanel noch Kinder hätte, dann würden durch subJPanel.repaint() auch die Kinder von subJPanel gezeichnet.

4)
Was würde gemacht werden, wenn man (statt subJPanel.repaint()) in einer Methode von subFrame nur repaint() aufrufen würde?
a) Es würde das Kind jPanel1 gezeichnet werden (und dann das Kind von jPanel1, also subJPanel)
b) Dann würde das andere Kind, also jTextField1 gezeichnet werden.
Kurz: Es würde alles gezeichnet werden.

5)
Warum macht man subJPanel.repaint(), wenn man mit repaint() auch alles zeichnen könnte ?
Wegen der Laufzeit.

Grafisch dargestellt:

subJFrame

jPanel1 jTextfield1

subJPanel

8.1.2 Layouts

Die einzelnen Panels kann man jeweils noch mit einem sogenannten Layout versehen.
Ein Layout gibt an, wie bzw. wo die mit **add** eingefügten sichtbaren Komponenten (z.B. Buttons, Textfelder, usw.) angeordnet werden sollen.
Wir benutzen hier oft das Gridlayout.
Ein Gridlayout wie z.B:
GridLayout myGL23 = new GridLayout(2,3);
ist ein Layout, das die Anordnung z.B. der Buttons und Textfelder in einer Zeichenfläche bzw. einem Fenster festlegt.
Das obige Gridlayout bedeutet, dass die sichtbaren Komponenten (z.B. Buttons, Textfelder, usw.) in einer Tabelle mit 2 Zeilen und 3 Spalten angeordnet werden.
Dabei wird zuerst die 1. Zeile spaltenweise (mit z.B. Textfeld, Button, usw.) aufgefüllt, dann 2. Zeile, usw.
Näheres dazu weiter unten.

8.1.3 Montieren der Komponenten und der Container

Die Panels und Komponenten müssen noch an (die eine Ebene) höheren Container montiert werden. Dies geschieht mit der Methode add.
Der Top-Level-Container JFrame (bzw. eine von ihm abgeleitete, selbstgebastelte Unterklasse) verlangt, dass ein Container (z.B.ein JPanel) oder eine Komponente bei ihm an eine bestimmte Stelle montiert wird.
Die Methode getContentPane() liefert diese Stelle.

8.1.3.1 Beispiel

```
// Liefert die Stelle, an der das JPanel an den Top-Level-Container JFrame
// (bzw. eine von ihm abgeleitete, selbstgebastelte  Unterklasse) montiert  wird.
Container mycont = getContentPane();
// Erstellen eines Objekts von  (Unterklasse von JPanel)
myp = new JPanel();
// mit add wird der Container-Panel myp in den Top-Level-Container montiert.
mycont.add(myp);
```

8.1.4 Zeichnen mit Swing

Mit Swing **muss** man in die Zeichenfläche **JPanel** zeichnen. Swing verlangt, dass die Zeichenbefehle sich in der Methode paintComponent(g) befinden müssen und diese Methode zu der Unterklasse von JPanel gehören muß, die in das Hauptfenster JFrame (oder einen sich darin befindlichen Container) montiert wird.

Die Methode paintComponent(...) darf aber niemals direkt innerhalb des selbst geschriebenen Programms aufgerufen werden !
Java (bzw. das Betriebssystem) entscheidet dagegen selbst, wann es paintComponent(...) aufruft. Dies geschieht:
a) bei der ersten Anzeige des Fensters (setVisible(true))
b) nachdem ein Fenster vergrößert (verkleinert) wurde,
c) nach dem Wegziehen überlagernder Fenster (verdeckter Elemente werden aufgedeckt)
d) nach einer Größenänderung des Fensters
e) nach dem Aufruf des Befehls repaint() im Programm

Bem:
1) Den formalen Parameter in paintComponent(...) kann man sich als den Teil des Bildspeicherbereichs vorstellen, der die zugehörige Komponente auf dem Bildschirm repräsentiert.

2) Die Stelle, an der das Panel an den Top-Level-Container JFrame montiert wird, ist auch ein Container.

Vorschlag:
Die Container und Panels bzw. Unterklassen von Panels sollen alle ein Layout verpasst bekommen.

8.1.5 Prinzipielles Vorgehen zur Erstellung einer GUI-Oberfläche

Für den Programmierer ist es am besten, wenn er eine passende Klasse zur Verfügung gestellt bekommt, die die gewünschte Oberfläche mit den entsprechenden Komponenten besitzt. Dazu ist es notwendig, dass er sich diese Klasse selbst bastelt und dann ein Objekt dieser Klasse erzeugt. Das bedeutet, dass die Komponenten dieser Klasse im Konstruktor aufgebaut werden müssen. Diese selbstgebastelte Klasse ist also ein spezielles vom Programmierer aufgebautes JFrame, d.h. eine Unterklasse von JFrame.
Im Hauptprogramm main könnte man also wie folgt eine selbst gebastelte GUI erzeugen:

```
public static void main(String[] args){
  MyFenster myf = new MyFenster();
  // Programm wird beendet (aus dem Arbeitsspeicher entfernt), wenn Fenster weggeklickt
  // wird. Nachprüfen mit Task-Manager
  myf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

8.1.6 Ein Java - Programm (Zeichnen mit Java)

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

// Hier wird ein Objekt der selbstgebastelten Klasse MyFenster erzeugt
public class MainFensterUndZeichnen1 {
  public static void main(String[] args) {
    MyFenster myf = new MyFenster();
    myf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}

// Ein Frame ist ein Fenster bzw. Fenstersegment
// Es ist nach seiner Erzeugung zuerst anfänglich unsichtbar.
// Hier wird die von JFrame abgeleitete Klasse MyFenster gebastelt, die
// damit die ganze Leistungsfähigkeit von JFrame erbt.
class MyFenster extends JFrame{
  // Buttons (Schalter) deklarieren
  private JButton myB1, myB2;
  // Textfelder deklarieren
  private JTextField myT1, myT2;
  // Labels deklarieren
  private JLabel myL1, myL2;
  // Zeichenflächen
  private JPanel myPan1;
  private MyZeichenflaeche myZf1;
  // Stelle in MyFenster deklarieren, an die montiert wird.
  private Container myCont;
  // Für myCont ein Layout
  private GridLayout myGL21;
  // Für myPan1 ein Layout
  private GridLayout myGL13;
  // Für myZf1 muss kein Layout erstellt werden, da
  // dort keine Buttons, Labels, usw. angebracht werden.
```



```
  // Konstruktor
```

```
  public MyFenster(){
    // Erzeugt jeweils ein Button
    myB1=new JButton("Go");
    myB2=new JButton("Go");
    // Erzeugt jeweils ein einzeiliges Textfeld mit dem vorgegebenen
    // Text und der vorgegebenen Spaltenzahl. Dieser Text kann
    // (im Gegensatz zu einem Label) editiert werden
    myT1=new JTextField("hier eingeben",30);
    myT2=new JTextField("Eingabe hier",30);
    // Erzeugt jeweils ein Label (Beschriftung) mit dem vorgegebenen
    // Text Dieser Text kann (im Gegensatz zu einem Textfeld) nicht
    // editiert werden
    myL1=new JLabel("Euro --> Dollar");
    myL2=new JLabel("Dollar --> Euro");
    // Erzeugt  Zeichenflächen
    myPan1 = new JPanel();
    myZf1 = new MyZeichenflaeche();
    // Erzeugt Layouts
    // Liefert die Stelle in MyFenster, an die montiert wird.
    myCont = getContentPane();
    myGL21 = new GridLayout(2,1);
    myGL13 = new GridLayout(1,3);
    // Ordnet das Layout dem Container myCont zu und
    // "formatiert" ihn damit.
    myCont.setLayout(myGL21);
    myPan1.setLayout(myGL13);

    // Montiert die grafischen Komponenten in myPan1
    myPan1.add(myL1);
    myPan1.add(myT1);
    myPan1.add(myB1);

    myCont.add(myPan1);
    myCont.add(myZf1);

    // Fensterüberschrift festlegen
    setTitle("Meine Zeichnung");
    // Koordinaten des linken, oberen Ecks des Fensters festlegen
    // Koordinate x = 30, Koordinate y = 60.
    setLocation(30,60);
    // Die Breite des Fensters in x-Richtung = 600
    // Die Breite des Fensters in y-Richtung = 400
    setSize(600,800);
    // Macht das Fenster sichtbar
    setVisible(true);
     }
}
```

```
//Eine Unterklasse des Containers JPanel erstellen
class MyZeichenflaeche extends JPanel{
/*
Die folgende Methode paintComponent(g) soll niemals direkt innerhalb des
selbstgeschriebenen Programms aufgerufen werden !
Java (bzw. das Betriebssystem) entscheidet selbst, wenn es
paintComponent(g) aufruft. Dies geschieht:
-> bei der ersten Anzeige des Fensters (setVisible(true))
-> nachdem ein Fenster vergrößert (verkleinert) wurde,
-> nach dem Wegziehen überlagernder Fenster (verdeckter Elemente
   werden aufgedeckt)
-> nach einer Größenänderung des Fensters
-> nach dem Aufruf des Befehls repaint() im Programm

Bem:
Den Parameter in paintComponent(...) kann man sich als den Teil des
Bildspeicherbereichs vorstellen, der die zugehörige Komponente auf dem
Bildschirm repräsentiert.
*/
  public void paintComponent(Graphics myG){
  /*
  Eigene Farben basteln:
  150, 180, 234 sind die Rot-, Grün- und Blauteile einer Farbe
  jeweils im Bereich zwischen 0 und 255
  */
    Color myFarbe = new Color(100, 250, 200);
    // Diese Farbe setzen
    myG.setColor(myFarbe);
    /*
    Zeichnet die Randlinie einer Ellipse, die sich in einem gedachten
    Rechteck befindet. Der linke obere Punkt hat die Koordinaten
    Koordinate x = 100, Koordinate y = 50.
    Die Breite des Rechtecks in x-Richtung = 250
    Die Breite des Rechtecks in y-Richtung = 300
    */
    myG.drawOval(100, 50, 250, 300);
    // Füllt die Ellipse mit der vorher gesetzten Farbe
    myG.fillOval(100, 50, 250, 300);

    // Die Farbe rot auswählen (setzen)
    myG.setColor(Color.red);
    // Text ausgeben
    myG.drawString("Das bin ich",200,20);
    // Linie zeichnen
    myG.drawLine(150,160,180,160);
    myG.drawLine(270,160,300,160);
    myG.drawLine(230,200,230,230);
    myG.drawRect(200,280,50,30);
  }
}
```

8.1.7 Jede Komponente ein eigenes Layouts

8.1.7.1 Standardverwendung

Jeder Komponente wird standardmäßig ein Layout zugeordent:

JPanel() --> FlowLayout
Container --> BorderLayout

8.1.7.2 Das Layout GridLayout

Ein Gridlayout wie z.B:
GridLayout myGL23 = new GridLayout(2,3);
ist ein Layout, das die Anordnung z.B. der Buttons und Textfelder in einer Zeichenfläche bzw. einem Fenster festlegt.
Das obige Gridlayout bedeutet, dass die sichtbaren Komponenten (z.B. Buttons, Textfelder, usw.) in einer Tabelle mit 2 Zeilen und 3 Spalten angeordnet werden.
Dabei wird zuerst die 1. Zeile spaltenweise (mit z.B. Textfeld, Button, usw.) aufgefüllt, dann 2. Zeile, usw.

Besonderheiten:
1) Wenn man mit:
myGL62 = new GridLayout(6,2);
mycont.setLayout(myGL62);
6 Zeilen und 2 Spalten vorgibt, und dann nicht alle 6*2=12 Elemente befüllt, sondern z.B. nur wie hier 2 Elemente:
mycont.add(myp1);
mycont.add(myz1);
dann wird myp2 nicht neben myp1 in der ersten Zeile dargestellt, sondern unter myp1 in der 2. Zeile. Es wird also nicht zuerst die 1. Zeile komplett gefüllt, dann die 2.Zeile, usw. wie es GridLayout eigentlich machen müsste.
Warum? Offensichtlich gruppiert dann Java um.
Man **muss** also alle Elemente befüllen. Notfalls benutzt man dazu verschiedene "DummyLabels", also hier im Beispiel noch 10 mal diese Anweisung machen:
mycont.add(new JLabel());
Beachte: Jedesmal wird durch new JLabel() ein neues, anderes Objekt erzeugt!!

2) Wenn man sowohl die Anzahl der Zeilen als auch dei Anzahl der Spalten mit "nicht Null" angibt, dann wird die Anzahl der Spalten ignoriert. Statt dessen wird die Spaltenanzahl festgelegt durch die angegebene Zeilenanzahl und die Gesamtanzahl der Komponenten im Layout. Zum Beispiel, wenn drei Zeilen und zwei Spalten angegeben wurden und neun Komponenten werden dem Layout hinzugefügt, dann werden sie in drei Zeilen und drei Spalten angezeigt. Die Angabe einer Spaltenanzahl berührt das Layout nur, wenn die Anzahl der Zeilen mit "Null" angegeben wird.

Beispiele:
a) GridLayout(0,2);
bedeutet zwei Spalten, unbegrenzte Zeilen

b) GridLayout(1,2);
bedeutet eine Zeile, unbegrenzte Spalten

8.1.7.3 Eltern

Ein und dieselbe Swing-Komponente (z.B. myl1) darf nicht mehrere (verschiedene) Eltern haben (z.B. myp1 und myp2). Folgendes darf also nicht sein:
myp1.add(myl1);
myp2.add(myl1);

8.1.7.4 Größe einstellen

1) getContentPane(), also mycont (der Klasse Container), verwendet BorderLayout, welches die Eigenschaft hat, die eingefügten Komponenten auf die maximale Fläche zu vergrößern, unabhängig davon was PreferredSize macht.
Deswegen funktioniert folgendes:
mycont.add(myp1);
mycont.add(myz);
und Nichtimplementierung von getPreferredSize().

2) Ein normales JPanel hat standardmäßig als Layout FlowLayout, welches nicht maximiert und auf PreferredSize achtet.
Deswegen funktioniert folgendes nicht:
mypan.add(myz);
mycont.add(mypan);
und Nichtimplementierung von getPreferredSize().

8.1.7.5 Eigene Layouts

Es wurde im Java-Forum geraten, pro JPanel ein eigenes Layout zu verwenden:
Also folgendes nicht machen:
myp1.setLayout(myGL23);
myp2.setLayout(myGL23);

Richtig wäre z.B. folgendes:
myp1.setLayout(myGL23_1);
myp2.setLayout(myGL23_2);

8.2 Wanzen

8.2.1 Motivation

Damit man mit der grafischen Oberfläche arbeiten kann, ist es wichtig, dass durch Klicks auf die erzeugten Komponenten (Buttons, Textfelder, usw.) vom Anwender ausgelöste Aktionen ausgelöst werden. Zum Beispiel wird durch einen Klick auf die entsprechende Taste eines Taschenrechners die Summe zweier Zahlen berechnet.

8.2.2 Funktionsweise

Ein Anwender klickt mit der Maus auf einen Button. Der Klick wird vom Betriebssystem registriert. Dies ermittelt, welcher Anwendung der Klick gilt und gibt die Klick-Informationen (Koordinaten, Taste, Zeitpunkt, usw.) an die Java-Laufzeitumgebung (JRE) weiter.
Die JRE ermittelt, welcher Komponente (Button, Textfeld, usw.) der Klick galt. Wenn an dieser Komponente eine Wanze angebracht wurde, wird an die entsprechende Methode der Wanze ein von der JRE erzeugtes Ereignisobjekt (das die Klick-Informationen enthält) als Parameter übergeben. In dieser Methode muss die gewünschte Benutzeraktion veranlasst werden.

8.2.2.1 Vorgehensweise des Programmierers

Bemerkung:
Die Vorgehensweise wird am Beispiel des Ereignisobjekts ActionEvent durchgeführt.
Diese Vorgehensweise lässt sich auf andere Ereignisobjekte übertragen.

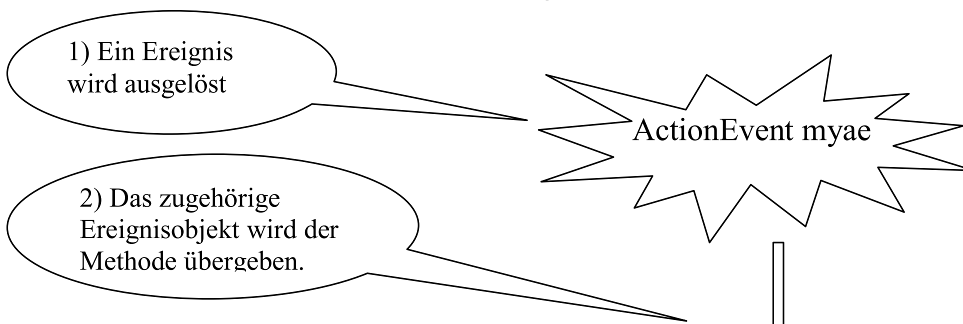
1) Es wird durch z.B. einen Mausklick letztendlich von der JRE ein Ereignisobjekt erzeugt, z.B. **ActionEvent**
b) Den Namen der Ereignisklasse nehmen, das Wort Event entfernen und durch das Wort Listener ersetzen, also **ActionListener**. Das ist der Name eines Interfaces.

2) Es muss eine Wanzenklasse erzeugt werden, die das bei 1b) erzeugte Interface implementiert, also in unserem Beispiel:
```
class MyWanze implements ActionListener
```

3) Man erzeugt ein Objekt der Wanzenklasse, also z.B:
```
MyWanze myw;
```
und montiert es an die gewünschte Komponente, z.B. einen Button. Die dazu benötigte Methode erhält man, indem man an das Wort add das bei 2) konstruierte Wort für das Interface anfügt, also in unserem Beispiel:
```
mybutton.addActionListener(myw);
```

4) Alle Methoden, die im Interface deklariert wurden, müssen nun in der Klasse MyWanze implementiert werden. Selbst, wenn nur ein einziges erzeugtes Ereignisobjekt in der zugehörigen Methode ausgewertet werden soll (z.B. soll nur ausgewertet werden, ob der Mauszeiger eine Schaltfläche betritt, nicht ob er sie verlässt), müssen alle in dem Interface deklarierten Methoden implementiert werden.
Falls ein Ereignisobjekt nicht ausgewertet werden soll, besteht die zu dem Ereignisobjekt zugehörige Methode aus einem leeren Befehl (Methodenkörper enthält keine Anweisungen).
In unserem Beispiel gibt es nur die Methode **actionPerformed** (ActionEvent myae), die implementiert werden muss.

Zeichnung:




```
class MyWanze implements ActionListener {

  public void actionPerformed   ActionEvent myae   {
    ...
  }

  public MyWanze(MyFenster f){
    ...
  }
}
```

```
class MyFenster extends JFrame {
  JButton myb;
  private MyWanze myw;

  public MyFenster() {
    myw = new MyWanze(this);
    myb.addActionListener(myw);
  }
}
```

Bemerkung:
Wenn sich bei einem Klick die Eigenschaft eines Fensters ändern soll (z.B. die Farbe eines Buttons), dann muss dem Konstruktor der Wanze, also MyWanze() als Parameter das Objekt des erzeugten Fensters übergeben werden, damit dann über dieses Fenster auf den Button zugegriffen werden kann. Dies wird durch den Parameter this realisiert.

8.2.3 Konkretes Beispiel

```
/*
P R O G R A M M B E S C H R E I B U N G
In diesem Programm wird ein Button erzeugt, der angeklickt werden muss.
Nach einem Mausklick wird die Hintergrundfarbe des Buttons (Schalter) blau
und bleibt danach immer blau.
Außerdem wird nach jedem Mausklick die Meldung "Ich wurde angeklickt"
ausgegeben.
*/

package de;
import javax.swing.*;          // GUI Komponente

import java.awt.*;             // Layouts
import java.awt.event.*;       // Ereignisse

// Hier wird ein Objekt der selbstgebastelten Klasse MyFenster erzeugt
public class MainListener2 {
     public static void main(String[] args) {
          MyFenster myf = new MyFenster();
          myf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     }
}


/*
Ein Frame ist ein Fenster bzw. Fenstersegment. Es ist nach seiner Erzeugung zuerst
anfänglich unsichtbar. Hier wird die von JFrame abgeleitete Klasse MyFenster gebastelt, die
damit die ganze Leistungsfähigkeit von JFrame erbt.
*/
class MyFenster extends JFrame {
  // Stelle in MyFenster deklarieren, an die montiert wird.
  private Container mycont;
  // Eine Zeichenfläche deklarieren
  private JPanel mypan;
  // Ein Button (Schalter) deklarieren
  private JButton myb;
  // Eine Wanze wird deklarieren
  private MyWanze myw;
```

```
  // Konstruktor
  public MyFenster() {
    // Liefert die Stelle in MyFenster, an die montiert wird.
    mycont = getContentPane();
    // Erzeugt eine Zeichenfläche
    mypan = new JPanel();
    // Erzeugt einen Button
    myb = new JButton("Bitte klicke mich");
    // Erzeugt eine Wanze
    myw = new MyWanze(this);
    // Montiert die Wanze an den Button
    myb.addActionListener(myw);
    // Montiert den Button an die Zeichenfläche
    mypan.add(myb);
    // Montiert die Zeichenfläche in das Fenster MyFenster
    mycont.add(mypan);
    // Fensterüberschrift festlegen
    setTitle("Ein Klick-Test");
    // Koordinaten des linken, oberen Ecks des Fensters
    //festlegen. Koordinate x = 100, Koordinate y = 200.
    setLocation(100,200);
    // Die Breite des Fensters in x-Richtung = 400
    // Die Breite des Fensters in y-Richtung = 200
    setSize(400,200);
    // Macht das Fenster sichtbar
    setVisible(true);
  };

  public JButton getMyb(){
    return(myb);
  }
}


class MyWanze implements ActionListener {
  private MyFenster myfVerweis;

  // Konstruktor
  public MyWanze(MyFenster f){
    myfVerweis = f;
  }

  public void actionPerformed (ActionEvent myae) {
    myfVerweis.getMyb().setBackground(Color.blue);
    // Bringt Meldung auf dem Bildschirm
    System.out.println("Ich wurde angeklickt");
  }
}
```

8.2.3.1 Aufgabe

Verändern Sie das Programm so, dass sich bei jedem Klick auf das Button die Hintergrundfarbe von blau auf rot bzw. von rot auf blau ändert.

8.2.4 Infos für die Programmierung

Ereignisse und unterstützende Komponenten

Ereignisse, Lauscherschnittstellen, add- und remove-Methoden	Komponenten, die das Ereignis unterstützen
ActionEvent **ActionListener** addActionListener( ) removeActionListener( )	**Button**, **List**, **TextField, MenuItem,** und davon abgeleitete einschließlich **CheckboxMenuItem**, **Menu,** und **PopupMenu**
AdjustmentEvent **AdjustmentListener** addAdjustmentListener( ) removeAdjustmentListener( )	**Scrollbar** **Alles erzeugte, das das Adjustable** Interface implementiert
ComponentEvent **ComponentListener** addComponentListener( ) removeComponentListener( )	**Komponenten und ihre Ableitungen**, einschließlich **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog**, **Frame**, **Label**, **List**, **Scrollbar**, **TextArea**, und **TextField**
ContainerEvent **ContainerListener** addContainerListener( ) removeContainerListener( )	**Container** und ihre Ableitungen, einschließlich **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog,** und **Frame**
FocusEvent **FocusListener** addFocusListener( ) removeFocusListener( )	**Komponenten und ihre Ableitungen**, einschließlich **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog**, **Frame Label**, **List**, **Scrollbar**, **TextArea,** und **TextField**
KeyEvent **KeyListener** addKeyListener( ) removeKeyListener( )	**Komponenten und ihre Ableitungen**, einschließlich **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog**, **Frame**, **Label**, **List**, **Scrollbar**, **TextArea,** und **TextField**
MouseEvent (für Klicks und Bewegung) **MouseListener** addMouseListener( ) removeMouseListener( )	**Komponenten und ihre Ableitungen**, einschließlich **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog**, **Frame**, **Label**, **List**, **Scrollbar**, **TextArea,** und **TextField**
MouseEvent[1] (für Klicks und Bewegung) **MouseMotionListener** addMouseMotionListener( ) removeMouseMotionListener( )	**Komponenten und ihre Ableitungen**, einschließlich **Button**, **Canvas**, **Checkbox**, **Choice**, **Container**, **Panel**, **Applet**, **ScrollPane**, **Window**, **Dialog**, **FileDialog**, **Frame**, **Label**, **List**, **Scrollbar**, **TextArea, und TextField**
WindowEvent **WindowListener** addWindowListener( ) removeWindowListener( )	**Window** und seine Ableitungen, einschließlich **Dialog**, **FileDialog,** und **Frame**
ItemEvent **ItemListener** addItemListener( ) removeItemListener( )	**Checkbox**, **CheckboxMenuItem**, **Choice**, **List,** und Alles, das das **ItemSelectable** interface implementiert
TextEvent **TextListener** addTextListener( ) removeTextListener( )	Alles von **TextComponent abgeleitete**, einschließlich **TextArea** und **TextField**

Man sieht, dass jeder Komponententyp nur bestimmte Typen von Ereignissen verarbeiten kann!

---

[1]Es gibt keinen **MouseMotionEvent** obwohl es so aussieht, dass es einen geben müsste. Klicken und bewegen ist im **MouseEvent kombiniert**, so dass das zweite Auftreten von **MouseEvent** in der Tabelle kein Fehler ist.

Gelöscht: ,

Komponenten und unterstützte Ereignisse

Komponententyp	Ereignis, das durch diese Komponente unterstützt wird
Adjustable	**AdjustmentEvent**
Applet	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Button	**ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Canvas	**FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Checkbox	**ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
CheckboxMenuItem	**ActionEvent, ItemEvent**
Choice	**ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Component	**FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Container	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Dialog	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
FileDialog	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Frame	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Label	**FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
List	**ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent**
Menu	**ActionEvent**
MenuItem	**ActionEvent**
Panel	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
PopupMenu	**ActionEvent**
Scrollbar	**AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
ScrollPane	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
TextArea	**TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
TextComponent	**TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
TextField	**ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Window	**ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**

Wenn man weiß, welches Ereignis eine bestimmte Komponente unterstützt, muss man nicht mehr als einfach Folgendes tun:

1. den Namen der Ereignisklasse nehmen und das Wort "Event" entfernen. An den verbleibenden Namen fügt man das Wort "Listener" an. Das ist die Listener–Schnittstelle (interface), die man in seiner Inneren Klasse (oder auf andere Art und Weise) implementieren muss.
2. Man implementiert das obige Interface and schreibt die Methoden für das Ereignis aus, die man verwenden möchte. Wenn man z.B. die Mausbewegungen beobachten möchte, muss man Code für die **mouseMoved( )**–Methode des **MouseMotionListener**–Interfaces ausschreiben. (Man muss natürlich auch die anderen Methoden schreiben (evtl. nur mit Methodenkörper {;}, aber dafür gibt es auch einen einfacheren schnelleren Weg: Adapterklassen)
3. Man erzeugt ein Objekt der Listenerklasse aus Schritt 2 und meldet es an der Komponente mit der Methode an, die man erhält, wenn man das Wort "**add**" vor dem Listenername einfügt. Im Beispiel: **addMouseMotionListener( )**.

9 Listener-Schnittstellen und ihre Methoden

Listener-Schnittstellen, zugehörige Adapter-Klassen	Methoden der Schnittstelle/Adapterklasse
ActionListener	**actionPerformed(ActionEvent)**
AdjustmentListener	**adjustmentValueChanged(AdjustmentEvent)**
ComponentListener **ComponentAdapter**	**componentHidden(ComponentEvent)** **componentShown(ComponentEvent)** **componentMoved(ComponentEvent)** **componentResized(ComponentEvent)**
ContainerListener **ContainerAdapter**	**componentAdded(ContainerEvent)** **componentRemoved(ContainerEvent)**
FocusListener **FocusAdapter**	**focusGained(FocusEvent)** **focusLost(FocusEvent)**
KeyListener **KeyAdapter**	**keyPressed(KeyEvent)** **keyReleased(KeyEvent)** **keyTyped(KeyEvent)**
MouseListener **MouseAdapter**	**mouseClicked(MouseEvent)** **mouseEntered(MouseEvent)** **mouseExited(MouseEvent)** **mousePressed(MouseEvent)** **mouseReleased(MouseEvent)**
MouseMotionListener **MouseMotionAdapter**	**mouseDragged(MouseEvent)** **mouseMoved(MouseEvent)**
WindowListener **WindowAdapter**	**windowOpened(WindowEvent)** **windowClosing(WindowEvent)** **windowClosed(WindowEvent)** **windowActivated(WindowEvent)** **windowDeactivated(WindowEvent)** **windowIconified(WindowEvent)** **windowDeiconified(WindowEvent)**
ItemListener	**itemStateChanged(ItemEvent)**
TextListener	**textValueChanged(TextEvent)**