R. Krell

Einführung in die objektorientierte Programmierung mit Java
einschl. der Verwendung von Stift & Co

Java mit Stift&Co

1. Objekte und Klassen; Grundsätzlicher Programmaufbau

Java ist eine objektorientierte Sprache. Objektorientiertes Programmieren („OOP“) bedeutet, ein Problem geschickt in Teilprobleme zu zerlegen und möglichst für jedes Teilproblem ein eigenes Objekt zu konstruieren. Das Zusammenspiel aller Objekte macht dann das Programm aus.
Objekte sind die konkreten Konstruktionen nach dem in einer Klasse (*class)* festgelegten Bauplan. Jedes Objekt kann weitere / andere Objekte besitzen, besitzt Datenfelder (globale Variablen zum Merken der Eigenschaften) und hat bestimmte, durch seine Methoden festgelegten Fähigkeiten. Im Interesse besserer Übersichtlichkeit sollten nur die Bauplan-/Klassen-Namen mit einem Großbuchstaben beginnen; Objekt-, Datenfeld- und Methodennamen werden hingegen üblicherweise mit einem Kleinbuchstaben begonnen.
Natürlich können nach einem Bauplan mehrere Objekte erzeugt werden. Diese können gleichartig sein, sich aber auch in ihren Eigenschaften unterscheiden. Werte für unterschiedliche Eigenschaften können schon beim Erzeugen des Objekts festgelegt werden durch Parameter (= übergebene Werte beim Aufruf einer Methode oder eines Konstruktors; sie stehen beim Aufruf ohne Typangabe in Klammern hinter dem Methodennamen) oder werden später gesetzt bzw. verändert.

Beispiel 1
Autos werden serienmäßig hergestellt, z.B. nach einem in der Datei public class VW_Golf festgelegten Bauplan. Trotzdem können verschiedene Farben oder Ausstattungsvarianten bestellt werden; wer nichts besonderes sagt, bekommt das Standardmodell in silber ohne Schiebedach. Deshalb enthält die Datei *public class VW_Golf* am besten zwei Konstruktoren [1]):

```
public class VW_Golf
{
  private Color farbe;                    // Hier sind (private) Datenfelder (nämlich Variable) an-
  private boolean mitSchiebedach;         // gelegt, in denen sich das spätere Objekt seine
  private int motorLeistung;              // Eigenschaften merken kann.
  private int gefahreneKilometer = 0;     // Da neue Autos 0 km auf dem km-Zähler haben,
  private String amtlichesKennzeichen;    // kann hier auch schon ein Anfangswert gesetzt werden.

  public VW_Golf ()  // Konstruktor, erste Version: ohne Parameter für das Standardmodell
  {
     farbe = Farbe.SILBER;  // Farbe gibt's in Stift&Co
     mitSchiebedach = false; // true = wahr bedeutet mit Schiebedach, false = falsch = ohne Scheibedach;
     motorLeistung = 55: // kW
  }

  public VW_Golf (Color lackFarbton, boolean mitStahlSchiebedach) // Konstruktor, 2.: mit Parametern
  {
     farbe = lackFarbton; //der als Parameter lackFarbton übergebene Wert wird im Datenfeld farbe gespei-
chert
     mitSchiebedach = mitStahlSchiebedach; // auch hier: Übernahme des Parameters für Kundenwunsch
     motorLeistung = 55;
  }
```

[1]) Es kann mehrere Methoden oder Konstruktoren gleichen Namens in einer Klasse geben, sofern sich diese in der Anzahl der Parameter (oder zumindest den Typen der Parameter) unterscheiden. Bei der Definition einer Methode werden hinter dem Methodennamen in Klammern Variable für die Parameter mit vorangestellter Typbezeichnung angegeben.

```
  public void fahre()  // eine 'normale' Methode, hier nicht weiter abgedruckt. Die Methoden verleihen
  {                    // dem Objekt Fähigkeiten, z.B. fahre, bremse, erhalteNeuesKennzeichen,
     ...               // nenneKilometerstand, usw.
```

In einem Firmenfuhrpark kann es jetzt durchaus mehrere Fahrzeuge vom Typ VW Golf geben. Der „Bauplan“ des Fuhrparks steht natürlich in einer eigenen Datei, z.B.

```
public class Fuhrpark
{
  private String verwalter = "Peter Müller, Kfz-Meister";         // Datenfeld mit Anfangswert
  private VW_Golf  chefAuto = new VW_Golf (Farbe.SCHWARZ, true);  // Sondermodell
  private VW_Golf  hausmeisterAuto = new VW_Golf ();              // Standardmodell
  ...
```

Wird später in einer dritten Datei irgendwann durch *Fuhrpark firmenfahrzeuge = new Fuhrpark()* ein Objekt namens *firmenfahrzeuge* der Klasse Fuhrpark erzeugt, so besitzt dieses Objekt neben dem Datenfeld *verwalter* weitere Objekte, nämlich die beiden Autos *chefAuto* und *hausmeisterAuto*! Sofern übrigens kein eigener Konstruktor definiert wird, kann (und muss) trotzdem hinter *new* immer der Klassenname mit leeren runden Klammern benutzt werden. Von diesem Konstruktor werden dann die Datenfelder (inkl. der dort genannten Objekte) angelegt.

Insofern konnte man die erste Version des Konstruktors in der VW-Golf-Klasse auch weg lassen, wenn man den Datenfeldern am Anfang schon direkt die Standardwerte zugewiesen hätte:

```
public class VW_Golf
{
  private Color farbe = Farbe.SILBER;       // Anfangswerte. Können vom zweiten Konstruktor oder später
  private boolean mitSchiebedach = false;   // durch andere Methoden (wie neuLackieren) geändert werden.
```

Das Hausmeister-Auto hätte dann trotzdem diese Werte erhalten, und beim Chef-Auto wären sie durch den zweiten Konstruktor mit den übergebenen Parametern überschrieben worden. Überhaupt lassen sich Variablen füllen und verändern. So kann man beim Wechsel des/der Verantwortlichen für den Fuhrpark natürlich später z.B. mit *firmenfahrzeuge.verwalter = "Uschi Flott, Kfz-Sachverständige";* den Inhalt des Datenfeldes verändern – allerdings sollte nach der reinen OOP-Lehre dazu lieber eine besondere Methode definiert werden, so dass besser vielleicht *firmenfahrzeuge.setzeVerwalter ("Uschi Flott, Kfz-Sachverständige";*) aufgerufen werden sollte!

Beispiel 2

Beim Haus vom Nikolaus befindet sich der Bauplan in der Datei „Nikohaus.java“ – der Dateiname muss (einschl. Groß- und Kleinschreibung) genau mit dem Klassen-Namen übereinstimmen. Bei Klassennamen sollte daher aauf Umlaute und Sonderzeichen verzichtet werden!

```
import stiftUndCo.*;   // macht Baupläne z.B. für Bildschirm und BuntStift verfügbar

public class Nikohaus  // Klassenname
{
 Bildschirm zeichenfläche = new Bildschirm(300,420);  //Datenfelder: Nikohaus besitzt 3 Objekte, nämlich
 BuntStift blei = new BuntStift();  // ..einen Bildschirm (hier nicht mit den Standardkonstruktor, sondern
 BuntStift kuli = new BuntStift();  // ..mit Sondergröße erzeugt) und zwei BuntStift-Objekte kuli und blei
 int hausnummer = 84;               // Damit der Name „Eigenschaften“ verständlich wird, hier noch ein
                                    // weiteres Datenfeld, in dem sich das Haus seine Hausnummer merkt.
```

```
// Nach den Datenfeldern kommen die Methoden

public void zeichne()  //erste Methode: gibt dem Nikohaus die Fähigkeit, sich zu zeichnen
{                      // void = ohne Rückgabewert
 blei.bewegeBis(50,350);       // Hier wird die Methode bewegeBis vom blei(-Stift) aufgerufen – mit ..
 blei.runter();                // Parametern, bei runter sind hingegen keine Parameter möglich/nötig.
 blei.bewegeBis(250,350);
 kuli.bewegeBis (50,400);      // Und hier werden – durch Methodenaufruf – Fähigkeiten des Objektes..
 kuli.schreibe ("Das ");       // kuli benutzt, die in der Klasse BuntStift definiert sind
 ...
```

Während diese Klasse auf Objekte und Methoden aus Stift&Co zugreift, können wir natürlich auch ein Objekt nach unserer eigenen Klasse *Nikohaus* erzeugen und dessen Fähigkeiten (sprich: die von uns geschriebene Methode *zeichne*) nutzen. Das wollen wir auch tun, um ein konkretes Nikolaus-Haus auf dem Bildschirm sichtbar zu machen. Der Bauplan zur Herstellung des eigenen Nikohaus-Objekts mit dem selbstgewählten Namen *hütte* steht in der Klasse NikoStart:

```
public class NikoStart
{
  public static void main(String[]s)
  {
   Nikohaus hütte =new Nikohaus();  // Definieren und Erzeugen des Objekts hütte
   hütte.zeichne();                 // Aufruf der Methode zeichne vom Objekt hütte
  }
}
```

Wir sehen: eigene Objekte (wie *hütte* vom Typ *Nikohaus*) werden genau so definiert und erzeugt, wie oben die Objekte *zeichenfläche*, *kuli* oder *blei* der importierten Klassen *Bildschirm* und *BuntStift*. Und so, wie wir dort die Methoden *runter* oder *bewegeBis* benutzen konnten, könne wir hier unsere Methode *zeichne* vom konkreten Objekt ausführen lassen.

Egal, aus wie vielen Klassen und Dateien ein Java-Programm besteht: in einer Klasse des Projektes muss es eine Methode namens *public static void main(String[]s)* geben, mit der das Projekt gestartet wird. Diese Startklasse sollte ansonsten so klein wie möglich gehalten werden: Am besten ruft die *main*-Methode nur eine einzige Methode aus der Klasse des Haupt-Objektes auf, wo aller Rest geschieht – auch weitere Objekte definiert und benutzt werden (weitere Beispiele später in den Kapiteln 4 und 8!).

Damit man zusammen gehörige Klassen leicht findet, sollten sie in einem eigenen Ordner (Unterverzeichnis) abgelegt werden und sollten ihre Namen möglichst gleich anfangen. Die Klasse (und damit auch die Datei, s.o) mit der *main*-Methode stets das Wort Start im Namen tragen, damit man nicht lange suchen muss!

2. Variable (Datenfelder), Lokalität, einfache und Objekt-Typen

Die Variablen stellen praktisch das Gedächtnis des Computers dar: ein Programm kann sich nur merken, was in Variablen gespeichert wurde. Und es kann sich nur erinnern, wenn konkret die Variable (durch ihren Namen) aufgerufen wird, in der ein gesuchter Wert (Inhalt) steht. Damit die Größe des Speicherplatzes festgelegt wird, die eine Variable für mögliche Werte bereit stellt, muss sie zunächst definiert werden.

In Java gibt es keinen festen Platz für Variablen-Definitionen: Sie können fast überall stehen. Wichtig ist allerdings, dass jede Variable vor ihrer ersten Benutzung definiert wird, damit klar ist, welche Art von Inhalt

die Variable aufnehmen kann. Bei der Definition kann man der Variablen schon einen ersten Wert geben, muss das aber nicht.

In den Klassen-Beispielen oben kamen Variable bisher nur als Datenfelder am Anfang einer Klasse vor („globale Variable“; nur da sind Zusätze wie *private* oder *public* sinnvoll, wie später noch erklärt wird) – mal mit, mal ohne Anfangswert. Aber auch in den Methoden können Variable definiert werden, wobei gilt: Variablen gelten ab ihrer Definition bis zum Ende des {}-Klammer-Paars, in dem sie definiert wurden (einschließlich aller darin enthaltenen weiteren Klammern, die jeweils so genannte Blöcke einschließen). Man sollte Variablen immer so lokal wie möglich definieren, d.h. immer nur in dem Block, wo sie auch wirklich gebraucht werden – nicht am Anfang auf Vorrat! Und es ist besser, in drei getrennten Blöcken oder in verschiedenen Methoden jeweils lokal immer wieder eine Variable für Zwischenergebnisse zu definieren, als eine globale Variable vom Anfang der Klasse für verschiedene, unabhängige Aufgaben zu nutzen. Übrigens kann man sogar in einem Unterblock durchaus nochmal eine namensgleiche Variable definieren (was allerdings meist ungeschickt ist und Programme schwer lesbar macht): Java benutzt immer die Variable, die am nächsten vor der Verwendung definiert wurde.

Beispiel

```
public int beispielMethodeFürVariablen()  // int (statt void) bedeutet, dass die Methode eine Ganzzahl
{                                         // als Ergebnis zurück geben wird!
  int x;      // Hier wird eine Variable x zur Aufnahme einer Ganzzahl eingerichtet („definiert“)
  x = 5;      // Jetzt wird die Variable x mit dem Wert 5 gefüllt. x musste vorher definiert sein!
  int y = 3;  // Hier wird y für Ganzzahlen definiert und direkt mit einem Anfangswert (hier 3) gefüllt
  if (y > 2)     // Hätte y noch keinen Anfangswert (hier 3), macht diese Frage keinen Sinn!
  {
     int x = 7;  // In diesem Block wird eine neue Variable x definiert, die nur in diesem Block gilt
     if (y < 4)  // In diesem Block und seinen Unterblöcken ersetzt dieses x das alte x von oben.
     {                                                    // Man spricht von „Überdeckung“
        y = 10 * x; // Weil hier das neue x gilt und 7 enthält, wird y zu 70. Das y ist noch das erste,
     }              // d.h. der ursprüngliche Wert 3 wird durch die 70 endgültig überschrieben
   }
  y = y + x;     // In das y wird der letzte Wert von y (nämlich 70) plus dem Wert von x gefüllt.
                 // Achtung: Hier außen gilt das erste x (nämlich 5), das x = 7 lebt hier nicht mehr!
  return (y);    // Also erhält y den Wert 75 und dieser Wert wird von der Methode zurück gegeben.
}
```

Einfache Datentypen

In Java können Variablen mit folgenden einfachen Typen definiert werden:

int	- für ganze Zahlen
double	- für Kommazahlen
char	- für einzelne Schriftzeichen
boolean	- für Wahrheitswerte, die nur entweder *true* oder *false* sein dürfen

Beispiel: Nach

```
int adam;
double eva;
```

kann *adam* eine ganze Zahl und *eva* eine Kommazahl aufnehmen. Natürlich kann auch *eva* eine ganze Zahl aufnehmen (aber Variablen für ganze Zahlen haben keinen Platz für Nachkommastellen, können als keine Kommazahlen speichern, die in Java mit Dezimalpunkt geschrieben werden müssen). Wir können also *eva = 34.5;* oder *eva = 7;* oder auch *eva = 3*4 + 7/5;* zuweisen. Außerdem ist *adam = 29;* oder *adam = 4 -11;* (Ganzzahlen können auch negativ sein!) oder *eva = adam;* erlaubt, während *adam = 3.14;* oder *adam = eva;* verboten sind (Unter Verlust der Nachkommastellen kann man *adam = (int)eva;* zuweisen).

Vorsicht bei der Division: Durch die Zuweisung *eva = 31/7.0;* oder *=31.0/7* erhält die Kommazahlen-Variable *eva* den Wert 4.4285714.., während die Ganzzahl-Variable *adam* bekanntlich keinen Platz für Nachkommastellen hat und ohne Fehlermeldung bei *adam = 31/7;* kurzerhand nur den Wert 4 speichert (nicht gerundet, sondern abgeschnitten; passiert auch bei *eva = 31/7*)! Da die Division 31 durch 7 nicht restlos aufgeht, könnte man den Rest in einer eigenen Variablen *int rest = 31%7;* speichern, wodurch *rest* den Wert 3 erhält!

Erlaubte **Rechenzeichen** für Zahlen sind also: + (plus) – (minus) * (mal) / (durch) und % (Divisionsrest). Java weiß, dass Punktrechnung (*, /) vor Strichrechnung (+,–) geht – ansonsten helfen runde Klammern!

Mit

```
char ersterBuchstabe = '$'; // Definition eines Schriftzeichens; immer nur einfache Anführungsstriche!
```

wird die Variable *ersterBuchstabe* als Variable vom Typ *char* definiert und ihr direkt ein erster Wert, nämlich das $-Zeichen, zugewiesen. Natürlich kann man später auch *ersterBuchstabe = 'A';* oder auch *ersterBuchstabe= '7';* füllen, wobei '7' mit einfachen Anführungsstrichen ein Schriftzeichen ist, mit dem nicht gerechnet werden kann (Unterscheide Ziffer und Zahl; die Ganzzahl 7 wird ohne Anführungsstriche geschrieben, gehört aber nicht in eine Variable vom Typ *char*, sondern in eine *int*-Variable).

Mit

```
boolean mitSchiebedach;
```

wird eine Wahrheitswert-Variable definiert, die *true* oder *false* aufnehmen kann. Soll ein Auto mit Luke produziert werden, liegt *mitSchiebedach = true;* nahe – sonst dürfte die Zuweisung *mitSchiebdach = false;* sinnvoll sein. Egal, wie man sich bisher entschieden hatte: Will man die Bestellung ändern, liefert *mitSchiebedach = ! mitSchiebedach* das Gegenteil des bisherigen Wertes, weil !false = true und !true = false ist. Das Ausrufezeichen ! ist also der Nicht-Operator, der jeden Wahrheitswert umdreht.

Man kann Java auch Wahrheitswerte bestimmen lassen. So füllt *mitSchiebedach = (4 > 8);* die Variable mit *false* (weil 4 eben nicht größer als 8 ist), während *mitSchiebedach = ((3==3) && (2 <= 10);* die Variable mit true füllt, weil tatsächlich 3 gleich 3 ist und 2 kleiner-gleich 10.

Für *boolean*-Werte gelten die **Rechenzeichen** && (und), || (oder) bzw. ! (nicht).
Vergleichsoperatoren sind == (gleich) < (kleiner) > (größer) <= (kleiner-gleich) >= (größer-gleich) sowie != (ungleich)

Objekt-Variable

Neben den einfachen Datentypen sind beliebige Objekte erlaubt, wobei hier die Variablen nicht nur definiert, sondern immer auch (normalerweise mit *new*) erzeugt werden müssen. Dies kann in einem Schritt – z.B. bei

```
BuntStift kuli = new BuntStift();     // natürlich muss es vorher schon einen Bildschirm geben
String meinName = new String ("Peter Hase");  // Nur bei Strings geht bequemerweise auch das Erzeugen
                                              // ohne new direkt mit String meinName = "Peter Hase";
```

oder auch in zwei Schritten geschehen:

```
Bildschirm fenster;                          // Definieren von fenster (als Objekt der Klasse Bildschirm)
int breite = 7 * 54;
int höhe = 200;
fenster = new Bildschirm (breite, höhe);  // Erzeugen des Objektes fenster
```

Die Klasse *String* für Textobjekte bietet die Besonderheit, dass nicht ausdrücklich *new* verwendet werden muss (aber kann). Im Gegensatz zu Schriftzeichen (*char*) müssen String-Konstanten immer mit doppelten(!)

Anführungszeichen umschlossen werden. Texte vom Typ *String* können mit + zusammengefügt werden, eingestreute *\n* - Zeichen sorgen für eine neue Zeile bei der Java-Ausgabe. So kann

```
String adresse = "Peter Ha" + "se\nDüsseldorf";   // + hängt Texte aneinander
```

Peter Hase
Düsseldorf

definiert, erzeugt und gefüllt werden, wobei *adresse* bei der Anzeige als ein Text in zwei Zeilen erscheinen würde (siehe Kasten; *\n* funktioniert leider nicht bei *schreibe* von Stift&Co)! Selbst Zahlen können in Java mit + an einen String angehängt werden: So erzeugt z.B. *adresse = "Ellerstr. " + 84;* ein Textobjekt mit dem Inhalt "Ellerstr. 84", mit dem natürlich nicht mehr gerechnet werden kann. + wird bei Strings nicht als Plus- oder Additionszeichen, sondern als Konkatenationszeichen verwendet und bezeichnet! Wichtig ist noch, dass in Objektvariablen (anders als in Variablen der einfachen Datentypen) nicht direkt die zugewiesenen Objekte, sondern ihre Speicherplatznummern(„Adressen“) stehen. Ein Vergleich mit <, = = oder > führt daher zu merkwürdigen Ergebnissen:

```
boolean gleich;
String rose = "Ellerstr. ";
String tulpe = "Ellerstr. ";
if (rose == tulpe)
{ gleich = true; }     // hier kommt man nicht hin,
else
{ gleich = false; }    // sondern landet hier!
```

Die beiden Variablen *rose* und *tulpe* stehen natürlich an verschiedenen Stellen im Hauptspeicher des Computers. Deshalb sind ihre Adressen verschieden und der else-Fall liefert gleich = false! Ob < oder > gilt, ist nicht vorhersagbar, weil nicht bekannt ist, welche Variable die niedrigere Speichernummer hat! Zum Vergleich von Objekten dürfen also nicht die einfachen Vergleichsoperatoren verwendet werden, sondern müssen Methoden benutzt werden, die Java (zumindest für Strings) schon bereithält, nämlich *equals* oder *compareTo*:

```
if (rose.equals (tulpe))
{  gleich = true; }     // Jetzt klappt's, wenn beide Variablen gleichen Inhalt haben!
```

3. Methoden sowie öffentliche und private Zugriffe

Methoden geben Objekten Fähigkeiten – bei ihrem Aufruf passiert etwas: Es wird gezeichnet oder ein Ergebnis berechnet oder ein Textobjekt mit einem anderen verglichen oder der Wert einer Variablen genannt oder das Objekt merkt sich etwas oder es wird eine gewisse Zeit gewartet oder oder oder... Jedenfalls stecken hier die Algorithmen drin, stehen Anweisungen und Kontrollstrukturen drin. Das Zusammenspiel der Methoden bewirkt den Programmablauf!
Trotz der Vielfalt der Möglichkeiten lassen sich immer zwei Fragen beantworten:

1) Braucht die Methode zusätzliche Informationen von außen, um ordentlich arbeiten zu können? Dann müssen geeignete Werte als Parameter hinter dem Methodennamen in den runden Klammern übergeben werden – sonst bleiben die Klammern leer!

2) Gibt die Methode ein Ergebnis zurück – sei es z.B. ein Namen (*String*), ein Buchstaben (*char*), eine Zahl (*int* oder *double*), ein Wahrheitswert (*boolean*) oder ein Objekt (dessen Klassenname angegeben werden muss) – , oder macht sie zwar irgendetwas (einschließlich Zeichnen oder Schreiben auf den Bildschirm), gibt aber kein Ergebnis an die aufrufende Stelle zurück? Im letzten Fall steht bei der Definition vor dem Methodennamen das Wort *void* (=keine Rückgabe), sonst der Typ des Ergebnisses!

Gibt eine Methode ein Ergebnis zurück, so muss als letzte auszuführende Anweisung in der Methode ein *return* mit dem Ergebnis stehen – nach *return (Ergebnis);* werden keine weiteren Anweisungen mehr ausgeführt. Trotzdem darf es mehrere *return*-Anweisungen geben – z.B. jeweils am Ende der beiden Blöcke in einer *if..else*-Verzweigung.

Wichtig ist auch, dass eine Methode immer nur höchstens ein Ergebnis (gar keines bei *void*, sonst genau eines) zurück geben kann. Möchte man mehrere Zahlen – z.B. die Koordinaten eines geometrischen Punktes – zurückgeben, muss zuvor ein geeignetes Objekt gebastelt werden, das den Punkt enthält. Man würde also zusätzlich eine eigene Datei „Punkt0.java“ für die Klasse

```
public class Punkt0
{
   double x, y;  // Koordinaten des Punktes können Kommazahlen sein!
}
```

anlegen und kompilieren . Damit wäre dann an andere Stelle die Methode *erzeugePunkt* möglich

```
public Punkt0 erzeugePunkt()     // erste Version
{
    Punkt0 pkt = new Punkt0 ();  // Objekte müssen definiert und mit new erzeugt werden!
    pkt.x = 4.7;   // Willkürliche Festlegung. Natürlich könnten hier auch sinnvolle Berechnungen stehen.
    pkt.y = 3.51;  // Es werden die Datenfelder x und y des Punktes pkt direkt angesprochen und verändert
    return (pkt);  // Gibt als Ergebnis den Punkt (4,7|3,51) zurück!
}
```

die eben nur ein Ergebnis – einen Punkt (auch wenn dieser intern aus zwei Kommazahlen besteht) – zurück gibt. Bei einem einfachen Punkt wird man nicht unbedingt die Datenfelder als *private* erklären und extra Methoden für den Zugriff schreiben wollen, was etwa wie folgt möglich wäre:

```
public class Punkt
{                           // Die Datenfelder x und y nehmen die Koordinaten des Punktes auf.
  private double x, y;      // Durch private kann von außerhalb dieser Klasse Punkt (also z.B. von
                            // der Methode erzeugePunkt) nicht mehr auf x, y zugegriffen werden!
  public void setzeX (double iks) // Will man trotzdem einen x-Wert festlegen oder ändern,..
  { x = iks; }              // muss man diese Methode setzeX nutzen (um x mit dem Parameter iks
                            // ..zu füllen oder zu überschreiben)
  public void setzeY (double üpsilon)  // Ebenso, um die y-Koordinate fest zu legen oder ..
  { y = üpsilon; }          // zu ändern

  public double holeX ()    // Auch Lesen von x ist von außen nicht möglich, weswegen man auch
  { return (x); }           // hierfür eine öffentliche (public !) Methode braucht

  public double holeY ()    // Ebenso für y:
  { return (y); }           // Als Ergebnis wir y zurückgegeben; Parameter sind nicht nötig ()

  public Punkt (double x, double üpsilon)  // Konstruktor, falls man einen Punkt direkt beim..
  { x = iks; y = üpsilon; }  // Erzeugen füllen will
}
```

Hiermit müsste dann die Methode *erzeugePunkt* (die in irgend einer anderen Klasse steht) etwa folgende Gestalt annehmen:

```
public Punkt erzeugePunkt()    // zweite Version
{
    Punkt pkt = new Punkt ();  // Objekte müssen definiert und mit new erzeugt werden!
    pkt.setzeX (4.7);   // Hier werden die Methoden des Punktes pkt aufgerufen,
    pkt.setzeY (3.51);  // um x und y zu füllen
    return (pkt);  // Gibt als Ergebnis den Punkt (4,7|3,51) zurück!
}
```

Konstruktoren

Da ein neuer Punkt aber normalerweise sofort mit Koordinaten gefüllt werden soll, wurde oben in der Klasse *Punkt* noch ein **Konstruktor** eingebaut. Konstruktoren sind spezielle Methoden, die direkt beim Erzeugen beliebige (im Konstruktor festgelegte) Aufgaben übernehmen. Konstruktoren dürfen niemals Ergebnisse zurück liefern – weshalb auf die Angabe *void* verzichtet werden kann und muss: jede Art von Typangabe fehlt vor einem Konstruktor! Hingegen sind sehr wohl Parameter erlaubt. Außerdem muss die Konstruktormethode genau so heißen wie der Klassenname (einschl. des Großbuchstabens am Namensanfang). Nutzt man den Konstruktor *Punkt* aus der Punkt-Klasse vereinfacht sich die Methode *erzeugePunkt* zu

```
public Punkt erzeugePunkt()    // dritte Version
{
    Punkt pkt = new Punkt (4.7, 3.51);  // Direkt beim Erzeugen wird gefüllt durch eigenen Konstruktor
    return (pkt);  // Gibt als Ergebnis den Punkt (4,7|3,51) zurück!
}
```

Natürlich können die anfangs mit dem Konstruktor gesetzten Koordinaten später bei Bedarf auch wieder mit *setzeX* oder *setzeY* verändert werden!

Der Vorteil des Versteckens der Datenfelder (durch den Vorsatz *private* – man spricht von „Kapselung“ bzw. ‘information hiding’) und damit des Zwangs zum Benutzen spezieller Methoden liegt darin, dass in die Methoden beispielsweise Gültigkeitsprüfungen integriert werden können, die bestimmte Punkte ausschließen oder dass man – durch Weglassen einer Methode wie *setzeY* – das nachträgliche Ändern der y-Koordinate unmöglich macht. Bei Programmen zur Flugsicherung ist so z.B. ausgeschlossen, dass irgendwo im Programm beim Herumrechnen mit den Koordinaten (etwa bei der Abstandsbestimmung) durch unachtsame Verwendung von *boeing.y* die vom Radarsystem gelieferte Koordinate verändert und so ein Flugzeug an falscher Stelle vermutet wird.

Parameterübergabe

Ebenfalls aus Sicherheitsgründen werden übergebene Parameter immer in eine Methode hinein kopiert (ausdrückliche Referenzparameter, wie man sie etwa in Pascal durch ein zusätzliches *var* auch vor/für einfache Datentypen erzwingen kann, gibt es in Java nicht). Ändert man in der Methode die übergebenen Werte, so ändert man die Kopie, die nur innerhalb der Methode sichtbar ist und mit dem Ende der Methode verschwindet. Das Original wird davon nicht beeinflusst (Auf der Druckvorlage zu diesem Skript erscheint auch kein Strich, wenn der Leser einen solchen auf seiner Kopie macht!). Allerdings ist Vorsicht beim Gebrauch von Objekten nötig. Da hier, wie bereits erwähnt, die Speicherstelle kopiert wird (und nicht das Objekt selbst), kann man trotzdem mit der kopierten Nummer direkt auf das Original zugreifen und es so – auch versehentlich – verändern. Nur beim Objekt vom vordefinierten *String*-Typ, der ja hinsichtlich *new* eine Sonderrolle spielt (s.o), wird bei der Zuweisung offenbar eine neue, andere Zeichenkette mit eigener Adresse erzeugt. Aber beim Objekt *pkt* von der Klasse *Punkt0* schlägt die Änderung (an der „Kopie“ *p* in *methode1*) aufs Original (aus *methode2*) durch. Auch Objekte der verbesserten Klasse *Punkt* wären genau so anfällig:

Schreibt man eine Datei „ParameterBeispiel.java“ und kompiliert diese

```
public class ParameterBeispiel
 {
   private void methode1 (int x, String wort, Punkt0 p)
   {
     x = x-3;  wort = "Welt";  p.y = 6;  //Auch eine Änderung mit setzeY hätte den gleichen Effekt!
     System.out.println ("In Methode 1: x = "+x+", wort = "+wort+", p.y="+p.y);
   }

   public void methode2 ()
   {
     int x = 7;
     String text = "Hallo";
     Punkt0 pkt = new Punkt0 ();  pkt.y = 3;  //Die y-Koordinate erhält den Wert 3
     methode1(x, text, pkt); //x, text u. pkt werden in methode1 kopiert (Kopien heißen dort x, wort, p)
     System.out.println ("Am Ende: x = "+x+", text = "+text+", pkt.y="+pkt.y);
   }
 }
```

und erzeugt ein Objekt *demo* nach diesem Bauplan, z.B. in der Datei „ParameterBeispielStart.java“,

```
public class ParameterBeispielStart
{
  public static void main(String[]s)
  {
   ParameterBeispiel demo = new ParameterBeispiel();
   demo.methode2 ();
  }
}
```

so liefert die Programmausführung die Ausgabe rechts: Das Objekt *pkt* wurde durch die Methode 1 verändert und hat seinen y-Wert 3 für immer verloren!

```
In Methode 1: x = 4, wort = Welt, p.y=6.0
Am Ende: x = 7, text = Hallo, pkt.y=6.0
```

Will man hingegen x vom einfachen Typ *int* durch einen Methodenaufruf dauerhaft ändern, ist eine Methode mit Ergebnisrückgabe nötig, z.B.

```
public int minus3 (int x)
{ return (x-3); }
```

die mit `x = minus3 (x);` aufgerufen werden kann und aus einer ursprünglichen 7 eine dauerhafte 4 macht!

Keine Methoden-Schachtelung

Methoden können in Java nicht geschachtelt werden, d.h, die *methode1* im vorstehenden Beispiel konnte – obwohl nur in der Methode 2 gebraucht – nicht innerhalb der *methode2* definiert werden. Sie muss unmittelbar in der Klasse definiert werden. Damit man auf solche nur intern gebrauchten Methoden wenigstens nicht von außen zugreifen kann, ist wieder der Vorsatz *private* sinnvoll, der den Zugriff auf die Benutzung innerhalb der Klasse einschränkt. Der Aufruf aus einer anderen Klasse wie *ParameterBeispielStart* etwa mit *demo.methode1 (7, “Hallo“,..);* ist dadurch verboten.

Aufrufe und Punkt-Schreibweise

Grundsätzlich kann auf klassenfremde, nicht-private Methoden und nicht-private globale Variable immer nur zugegriffen werden, wenn zunächst ein Objekt der gewünschten Klasse definiert und erzeugt wurde. Dann steht beim Aufruf <Objektname>.<Methodenname><Parameterliste> wie beispielsweise bei `demo.methode2();` in *ParameterBeispielStart* (oder auch bei <Objektname>.<Variablenname> wie bei `pkt.x` in der ersten Version von *erzeugePunkt*) das Gewünschte zur Verfügung.
Innerhalb einer Klasse können Methoden und globale Datenfelder einfach mit ihrem Namen ohne irgendetwas davor aufgerufen werden. Wer auch hier unbedingt eine Punkt-Schreibweise verwenden will, kann – da ja beim Erstellen der Klasse noch nicht bekannt ist, wie spätere Objekte nach diesem Bauplan einmal heißen werden – das Schlüsselwort *this* voranstellen. So könnte in der Klasse *ParameterBeispiel* (Seite 9) innerhalb der Methode 2 der Aufruf der Methode 1 auch `this.methode1(x, text, pkt);` lauten. Wäre die Variable x sogar global (also nicht in der Methode 2, sondern vor den Methodendefinitionen direkt nach dem Klassennamen) definiert, wäre auch `this.methode1(this.x, text, pkt);` möglich. Wirklich nötig wird das *this* allerdings nur, wenn man in einem Block während des vorübergehenden Überdeckens einer globalen Variable mit einer gleichnamigen lokalen Variable doch noch wieder auf die globale Variable zugreifen will.

Bei einigen besonderen, „statischen“ Klassen können übrigens auch „von außen“ Methoden der Klasse(n) benutzt werden, ohne dass extra ein passendes Objekt erzeugt werden muss (oder kann). Beispiele sind die Klasse *Math* der Mathematik-Funktionen von Java (Aufruf z.B. *y = Math.sin(3.14):*) oder die Klasse *Hilfe* aus der Stift&Co-Bibliothek (Aufruf-Beispiele: *Hilfe.warte();* oder auch *int std = Hilfe.stunde();* bzw. *int würfelZahl = Hilfe.zufall(1,6);*). Hier reicht der Klassename, der mit Punkt dem Methodennamen vorangestellt wird, anstelle eines Objektnamens!

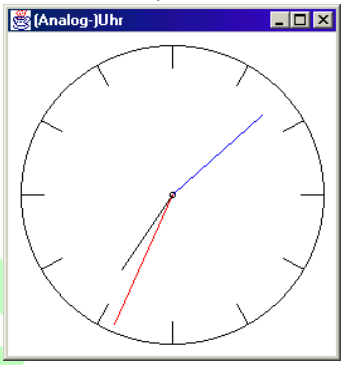
4. Objektanalyse und Programmentwurf (am Beispiel einer Uhr)

Nicht nur Anfängerinnen und Anfänger, sondern auch „gestandene“ Programmierer mit Erfahrung in nicht-objektorientierten Sprachen haben bei der Zerlegung eines Problems in geeignete Klassen oft Schwierigkeiten. Eigentlich sollte die Orientierung an Alltagsvorstellungen helfen. Soll beispielsweise eine Analog-Uhr programmiert werden, ist klar, dass diese aus einem Ziffernblatt und drei Zeigern besteht, die von einem Uhrwerk angetrieben werden. Entsprechend liegt es nahe, auch die programmierte Uhr mit solchen Objekten auszustatten. Dabei ist es natürlich wünschenswert, die drei Zeiger möglichst nach einem universelleren Bauplan (einer Zeigerklasse) erzeugen zu können – man möchte nicht unbedingt drei verschiedene Zeigerklassen schreiben müssen. Deshalb muss die Zeigerklasse eine variable Zeigerlänge und -farbe ermöglichen. Auch die Tatsache, dass eine Stundenzeiger-Umdrehung zwar 12 Stunden dauert, aber bei Minuten- und Sekundenzeiger eine 60-Teile-Einteilung des Vollkreises gilt, muss Berücksichtigung finden.

Die Anforderungen an die Klassen sollten zunächst in normaler Sprache auf einem Konzeptblatt festgehalten werden; hier können auch Beziehungen oder Abhängigkeiten der Klassen untereinander markiert werden. Außerdem muss man sich darüber klar werden, was Objekte der geplanten Klassen können müssen. So will man das Ziffernblatt der Uhr sicherlich sehen – es muss sich zeichnen können. Und die Zeiger sollen die Zeit anzeigen, müssen also vom Uhrwerk auf die richtige Zeit gedreht/gestellt werden können. Dass man ein Bildschirmfenster braucht (aber auch nur eines, nicht für jeden Zeiger ein anderes) um die Uhr zu sehen, und dass Stifte nötig sind, um Uhrblatt und Zeiger zu zeichnen, ist so selbstverständlich, dass man fast vergisst, dies ebenfalls zu notieren.

Schließlich kann man sich noch überlegen, ob man die Größe der Uhr von Anfang an festlegen möchte. Vorm Ausprobieren ist ja noch nicht klar, wie die Uhr in einer bestimmten Pixelgröße wirkt. Deshalb sollte das Programm so geschrieben werden, dass notfalls – und ohne dass Änderungen an verschiedenen Stellen des

Programmtextes nötig werden – eine konsistente Größenänderung durchgeführt werden kann. Am besten erschien es daher, wenn die Zeigerlängen nur relativ angegeben werden und sich die Zeiger die benötigten Mittelpunktkoordinaten und die Uhrgröße direkt beim Ziffernblatt holen – dann passt auch immer alles zusammen und selbst bei späteren Änderungen oder wegen Tippfehlern drehen sich die Zeiger nicht um verschiedene Punkte (Unter Verwendung der oben geschriebenen *Punkt*-Klasse hätten natürlich auch beide Mittelpunktskoordinaten noch eleganter mit nur einem Methodenaufruf geholt werden können).



Erst nach solchen Vorüberlegungen (und vollen Papierkörben mit verworfenen Konzepten) ist elegantes Programmieren möglich – sofortiges Eintippen führt selten zu schönen, später noch pflegbaren Ergebnissen.

Hier resultieren die Überlegungen in vier Klassen, die in beliebiger Reihenfolge jeweils in eigenen Dateien geschrieben und kompiliert werden konnten. Allerdings wurde die Startklasse (hier Teil 4) als zweite erstellt, und damit zunächst nicht (wie jetzt) das ganze Uhrwerk gestartet, sondern zuvor schon das Ziffernblatt getestet. Die Datenfelder im UhrWerk hätten übrigens auch alle *private* sein können.

```
// Teil 1/4 des Projekts (Analog-)Uhr.
// Java 1.1.8, R. Krell 25.1.03 (www.r-krell.de)

import stiftUndCo.*;

public class UhrBlatt
{
  Bildschirm fenster;  // einziges Bildschirmfenster im gesamten Programm
  private BuntStift stift;  // lokaler Stift zum Zeichnen des Ziffernblatts
  private int radius;  // Uhrgröße: global, da in versch. Methoden gebraucht
  private int mx, my;  // Datenfelder für die Koord. des Mittelpunktes
  private final int RAND = 10; // Abstand zwischen Uhr und Fensterrand

  public UhrBlatt (int gewünschterHalbmesser)  // Konstruktor
  {
    radius = gewünschterHalbmesser; // Speichern des Parameters in globaler Variable
    mx = radius + RAND;         // Berechnet aus dem radius sinnvolle
    my = radius + RAND;         // Koord. für den Mittelpunkt der Uhr
    fenster = new Bildschirm(2*(radius+RAND),2*(radius+RAND),"(Analog-)Uhr");
                                // Erzeugen eines Bildschirmfensters passender Größe
    stift = new BuntStift();   // stift darf erst nach dem Bildschirm erzeugt werden
    zeichne ();    // erstmaliges Zeichnen des Ziffernblattes
  }

  public void zeichne ()  // Diese Methode zeichnet das Ziffernblatt
  {                       // (aber ohne Zeiger)
    stift.hoch();
    stift.bewegeBis(mx,my);
    stift.zeichneKreis (radius);    // äußerer Kreis
    stift.zeichneKreis (2);         // dicker Mittel-"punkt"
    for (int stunde=0; stunde < 12; stunde = stunde + 1)
    {                               // 12 Stundenstriche am Rand:
      stift.hoch();
      stift.bewegeBis (mx,my);
      stift.dreheUm (30);           // 360//12 = 30/ pro Stunde
```

```
      stift.bewegeUm (0.85*radius);
      stift.runter();
      stift.bewegeUm (0.15*radius); // Strichlänge nur äußere 15% vom Radius
    }
  }

  public int nenneX ()       // Methode, um Mittelpunktskoordinate x auch ..
  { return (mx); }           // .. an die Zeiger weitergeben zu können

  public int nenneY ()       // Methode, um Mittelpunktskoordinate y auch ..
  { return (my); }           // .. an die Zeiger weitergeben zu können

  public int nenneRadius () // Methode, um den Radius auch ..
  { return (radius); }       // .. an die Zeiger weitergeben zu können
}
```

Die Uhr besitzt ein Objekt *ziffernblatt* dieser Klasse *UhrBlatt* sowie drei Zeiger nach folgendem Plan:

```
// Teil 2/4 des Projekts (Analog-)Uhr.
// Java 1.1.8, R. Krell 25.1.03 (www.r-krell.de)

import stiftUndCo.*;
import java.awt.*;           // für Color

public class UhrZeiger
{
  private BuntStift stift = new BuntStift();
  private int mx, my, länge; // Datenfelder
  private Color farbe;
  private double winkelStück;
  private double letzterWinkel = -1; // beliebiger Anfangswert.

  public UhrZeiger (UhrBlatt uhr, double relativeZeigerlänge, int teile,
                    Color zeigerFarbe)
  {
   mx = uhr.nenneX();        // Füllen der globalen Datenfelder mit Werten, ..
   my = uhr.nenneY();        // die vom Ziffernblatt geholt werden
   länge = (int) (uhr.nenneRadius() * relativeZeigerlänge);  // ganzzahl. Anteil
   farbe = zeigerFarbe;
   winkelStück = 360/teile;  // Berechnen des Winkelstücks pro Einheit
  }

  private void zeichne (double winkel, Color zFarbe) // lokale Hilfs-Methode:
  {                          // zeichnet Zeiger mit zFarbe und winkel
    stift.hoch ();
    stift.bewegeBis (mx, my);   // stift wird an absolute Position gesetzt
    stift.dreheBis (winkel);    // Ausrichtung bzgl. Stift&Co-Winkel-Skala (absolut)
    stift.setzeFarbe (zFarbe);
    stift.runter();
    stift.bewegeUm (länge);     // Bewegung in eingestellter Richtung um die
  }                             // durch länge angegebene relative Strecke
```

```
  public void stelleAuf (double zeit)
  {
    double winkel = 90 - winkelStück * zeit; // 90 und -, weil Stift&Co-Winkel-
     //Skala bei 3 Uhr anfängt und im Gegenuhrzeigersinn dreht. "12" also bei 90/
    if (winkel != letzterWinkel)     //Sofern nötig, wird der Zeiger ..
    {
      zeichne (letzterWinkel, Farbe.WEISS);  //..an alter Stelle gelöscht
      letzterWinkel = winkel;
    }
    zeichne (winkel, farbe); //immer neu, da evtl. durch andere Zeiger gelöscht!
  }
}
```

Damit ist das Uhrwerk jetzt sehr einfach und übersichtlich zu realisieren:

```
// Teil 3/4 des Projekts (Analog-)Uhr.
// Java 1.1.8, R. Krell 25.1.03 (www.r-krell.de)

import stiftUndCo.*;

public class UhrWerk
{
  final int RADIUS = 120; //Radius der Uhr, der nur hier geändert werden kann und braucht – alles weitere
                          //passt sich automatisch richtig an!

  UhrBlatt  ziffernblatt = new UhrBlatt (RADIUS);  // muss wegen des (nur) hier definierten und erzeugten
                           // Bildschirms an erster Stelle stehen!
  UhrZeiger stdZeiger = new UhrZeiger (ziffernblatt, 0.6,  12, Farbe.SCHWARZ); //0.6: Länge=60% vom Radius
  UhrZeiger minZeiger = new UhrZeiger (ziffernblatt, 0.8,  60, Farbe.BLAU);
  UhrZeiger sekZeiger = new UhrZeiger (ziffernblatt, 0.95, 60, Farbe.ROT);

  public void zeigeZeit ()
  {
    int std, min, sek;  // lokale Variablen für ganze Zahlen

    while (true) // läuft ewig!
    {
      std = Hilfe.stunde() % 12;  // 24 h auf 12 h reduzieren
      min = Hilfe.minute();  // in stiftUndCo eingebaute ..
      sek = Hilfe.sekunde(); // .. Systemzeit-Abfrage

      stdZeiger.stelleAuf (std + min/60.0); //.0 wichtig, damit Ergebnis double
      minZeiger.stelleAuf (min);
      sekZeiger.stelleAuf (sek);
      ziffernblatt.zeichne(); // immer wieder zeichen, damit durch Zeiger über-
    }               // schriebene bzw. gelöschte Striche wieder sichtbar werden
  }
}
```

Ein Uhr-Objekt (hier: *rolex*) entsprechend der vorstehenden Klasse *UhrWerk* muss jetzt nur noch erzeugt und mit *zeigeZeit* zur dauernden Zeitanzeige beauftragt werden:

```
// Teil 4/4 des Projekts (Analog-)Uhr.
// Java 1.1.8, R. Krell 25.1.03 (www.r-krell.de)

public class UhrStart extends java.applet.Applet
{
  public static void main(String[] s) // für den Start als "stand-alone"-Application
  {
    UhrWerk rolex = new UhrWerk();
    rolex.zeigeZeit();
  }

  public void init()  // für den Start als Applet aus einer Webseite heraus
  {
    UhrWerk rolex = new UhrWerk();
    rolex.zeigeZeit();
  }
}
```

Das Beispiel macht – bei der Übernahme von Radius und Mittelpunkt vom Ziffernblatt zu den Zeigern – auch deutlich, wie getrennte Objekte sozusagen „quer“ miteinander Informationen austauschen können. Dass hierzu ein ganzes Objekt (nämlich *ziffernblatt*) in *UhrWerk* an die Konstruktoren der drei Zeigerobjekte übergeben wird, ist überhaupt nicht schlimm: Wie mehrfach erwähnt, wird bei Objekten tatsächlich nur eine (ganzzahlige) Speicheradresse übergeben, so dass der Aufwand kleiner als bei der Übergabe einer Kommazahl oder dreier getrennter Ganzzahlen ist.
Natürlich hätte man die Werte auch einmal global in *UhrWerk* (statt letztlich in *UhrBlatt*) und zwar als Konstanten [= fianle Variablen mit Anfangswert] einrichten und von dort leicht nur „abwärts“ jeweils als Parameter an die (entsprechend veränderten) Konstruktoren von Blatt und Zeigern weitergeben können.
Auch das Anlegen einer fünften Datei ist möglich für die zusätzliche Klasse *UhrGlobals*, die nur die vier Konstanten *final int RADIUS = 120; final int RAND = 10; final int MX = RADIUS + RAND; final int MY = MX;* enthält. Diese Klasse bzw. jeweils ein Objekt dieser Klasse könnte von allen Klassen (wie *UhrBlatt* und *UhrZeiger*) benutzt werden, um auf die grundlegenden gemeinsamen Werte zuzugreifen...

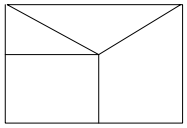
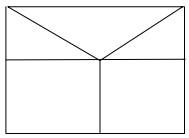
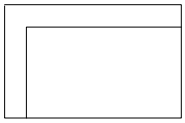
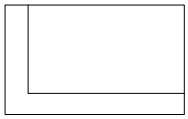
Bei anderen Projekten verläuft die Objektanalyse ähnlich: Will man beispielsweise einen Fuhrpark verwalten, sollten mindestens getrennte Klassen für ein allgemeines Auto, für eine(n) Fahrer(in) und für den Fuhrpark als Ganzes eingerichtet werden. Klassen fürs Kalenderdatum (sofern nicht die bei Java mitgelieferte Bibliothek benutzt wird) oder andere zusammengehörige Werte kommen hinzu (vgl. etwa die *Punkt*-Klasse, Seiten 7/8). Auch für grundsätzliche Datenstrukturen mit evtl. darauf arbeitenden Sortierverfahren empfehlen sich eigene Klassen. Und auf jeden Fall sollte die Benutzeroberfläche von der Funktionalität getrennt werden, damit leichter Austausch oder Weiterentwicklung möglich wird. Ziel ist jedenfalls, überschaubare, weitgehend unabhängige Module zu entwickeln, die möglichst auch in anderen Zusammenhängen wieder verwendet werden können. Speziell benötigte Zusatzeigenschaften oder -funktionen können auch durch Vererben hinzu gefügt werden, ohne einen bewährten Modul selbst verändern zu müssen. Interfaces helfen, die z.B. von einer Benutzer-Oberfläche erwarteten Methoden für die Funktionalität formal festzulegen.

5. Anweisungen und (Java-) Kontrollstrukturen

Eigentlich gibt es nur zwei Sorten „normaler“ Anweisungen: Zuweisungen wie *x = 5* oder *zahl = 17+4* bzw. *name = vorname+" "+zuname* (was rechts steht wird [ausgerechnet und] in die Variable links vom Zuweisungszeichen „=“ gefüllt) oder Methodenaufrufe (wie *stift.runter()*, *Hilfe.warte(300)*). Oft ist beides sogar

noch kombiniert, wenn nämlich Methoden aufgerufen werden, die nicht *void* sind, sondern einen Rückgabewert haben, der zugewiesen werden muss: *x = maus.hPosition()*; *stift = new BuntStift()*; *zahl=Hilfe.wurzel(16)*; o.ä.

Üblicherweise werden nach dem Aufruf einer Methode alle in ihr enthaltenen Anweisungen eine nach der anderen und je einmal nacheinander ausgeführt. Kontrollstrukturen erlauben Einschränkungen oder Wiederholungen und machen damit flexible Programme überhaupt erst möglich. Nach der oben beschriebenen Objektanalyse mit der Einteilung des Problems in Klassen und der Festlegung der benötigten Methoden und der umgangssprachlichen Beschreibung ihres Verhaltens müssen für jede Methode entsprechende Algorithmen mit den richtigen, oft geschachtelten Kontrollstrukturen entwickelt werden. Struktogramme helfen, den Überblick zu bewahren und erleichtern den Entwurf.

Kontrollstruktur		Java	Struktogramm
Verzweigung (bedingte Anweisung)	einseitige Verzweigung	`if (`*Bedingung*`)` `{` *AnweisungA ;* *AnweisungB; ... ;* `}`	
	zweiseitige Verzweigung	`if (`*Bedingung*`)` `{` *Anweisung1A; Anweisung1B; ..;* `}` `else` `{` *Anweisung2A; Anweisung2B; ...;* `}`	
	mehrseitige Verzweigung	`switch` (Selektor) `{` `case` *Konstante1: Anweisung1A; Anweisung1B; ...;* `break;` `case` *Konstante2: Anweisung2A; Anweisung2B; ...;* `break;` ... `default` *: AnweisungDA; AnweisungDB; ...;* `}`	
Wiederholung	Wdh. mit vorgeschalteter Kontrolle	`while (`*WdhBedingung*`)` `{` *AnweisungA; AnweisungB, ..;* `}`	
	Zählschleife (vorgeschaltete Kontrolle)	`for (int` *zähl = startwert; WdhBedingung; zähl=zähl+schritt*`)` `{` *AnweisungA; AnweisungB, ..;* `}`	
	Wdh. mit nachgeschalteter Kontrolle	`do {` *AnweisungA;* *AnweisungB, ..;* `} while (`*WdhBedingung*`);`	

Bedingungen müssen immer in runde Klammern eingeschlossen werden! Bei allen Strukturen können aber die geschweiften Klammern `{` *AnweisungA;* `}` auch weggelassen werden, wenn sie nicht mehrere, sondern nur eine einzige Anweisung umschließen würden.

Selektoren in der *switch..case*-Anweisung können nur Variablen vom Typ *int* oder *char* sein; Strings sind nicht möglich. Als Konstanten stehen entsprechend Ganzzahlen (*case 4:..*) oder Buchstaben (*case 'm':* ..). Es ist

immer nur eine Konstante pro Fall möglich – keine Aufzählungen oder Bereichsangaben. Fehlt allerdings das abschließende *break* hinter den Anweisungen, wird auch der nachfolgende Fall mit ausgeführt. Dies kann man ausnutzen, wenn z.B. zwei Buchstaben zum selben Fall führen sollen: *case 'm' : ; case 'x' : AnweisungX; break;* führt bei `m` und `x` die AnweisungX aus. Der *default*-Fall (der genau dann ausgeführt wird, wenn keiner der übrigen Fälle zutrifft), ist optional, kann also auch weggelassen werden.

In der For-Schleife entfällt das definierende *int* (oder auch *double* bei Kommazahlen) vor der Zählvariable, wenn die Zählvariable bereits vor der Schleife existiert und definiert wurde. Die Schrittweite kann positiv oder negativ sein und muss keineswegs immer 1 sein. Bei positiver Schrittweite wird als Wiederholungsbedingung oft *zähl* <= *endwert* benutzt; die Bedingung muss aber nicht unbedingt die Zählvariable explizit enthalten.

Auch bei der *do..while*-Wiederholung mit nachgeschalteter Kontrolle steht hinter dem *while* keine Abbruch-, sondern eine Wiederholungsbedingung, die wahr liefern muss, wenn (weiter) wiederholt werden soll. Bekanntlich werden die Anweisungen in der Schleife auch bei beliebiger Bedingung aber immer mindestens einmal ausgeführt, weil ja erst danach die Gültigkeit der Bedingung geprüft wird.

In den Beispielprogrammen in diesem Skript findet man immer wieder Kontrollstrukturen, so dass auf zusätzliche Beispiele an dieser Stelle verzichtet wird.

6. Einige wichtige Stift&Co-Befehle

Die mit ähnlicher Funktionalität zunächst für andere Sprachen entworfene Bibliothek Stift&Co hat Georg Dick in Java implementiert und stellt sie dankenswerterweise kostenlos für Unterrichtszwecke zur Verfügung. Die folgenden Angaben stützen sich auf seine Beschreibung; Details können der mitgelieferten Dokumentation entnommen werden.

6a. Konstanten

Konstanten für Füllmuster: Es gibt z.Z. nur zwei „Muster" für Rechtecke und Kreise:

```
final int Muster.DURCHSICHTIG = 0;
final int Muster.GEFUELLT = 1;
```

Beispiel: `meinStift.setzeFuellmuster(Muster.DURCHSICHTIG);`

Einige Farbwerte sind vordefiniert, u.a.

```
Color Farbe.SCHWARZ, Color Farbe.BLAU, Color Farbe.CYAN,...
```

Beispiele:

```
meinStift.setzeFarbe(Farbe.ROT);
meinStift.setzeFarbe(Farbe.rgb(10,200,40)); // Farbanteile je 0..255
```

Konstanten für besondere Zeichen auf der Tastatur:

```
char Zeichen.F1                    char Zeichen.PFEILRECHTS
char Zeichen.F2                    char Zeichen.PFEILOBEN
..                                 char Zeichen.PFEILUNTEN
char Zeichen.F12                   char Zeichen.PFEILLINKS
char Zeichen.POS1                  char Zeichen.ENDE
char Zeichen.BILDUNTEN             char Zeichen.ESC
char Zeichen.BILDAUF               char Zeichen.ENTER
```

Beispiel: `if (meineTastatur.zeichen()==Zeichen.F1) { .... }`

6b. Wichtige Methoden der Hilfe-Bibliothek

`Hilfe.zufall(`*von, bis*`)` liefert eine Zufallszahl zwischen den Ganzzahlen von..bis
`Hilfe.warte(`*zeit*`)` pausiert für die angegebene Anzahl von Millisekunden
`Hilfe.quadrat(`*x*`), Hilfe.wurzel(`*x*`), Hilfe.sinus(`*x*`)` rechnen
`Hilfe.stunde(), Hilfe.minute(), Hilfe.sekunde()` liefern die Zeit

6c. Die Klasse Bildschirm

Um einen Bildschirm zu erzeugen, muss ein Objekt z.B. mit
`Bildschirm` *bs* `= new Bildschirm();`
definiert und erzeugt werden. In Klammern kann auch die Breite und Höhe sowie evtl. zusätzlich noch ein Fenstertitel angegeben werden (z.B. *new Bildschirm (200,300,"Hallo")*;
Mit dem so erzeugten Objekt *bs* (oder jedem anderen Bildschirm-Objekt) können dann aufgerufen werden:
bs`.breite()` liefert die Breite der Zeichenebene (als Ganzzahl)
bs`.hoehe()` liefert die Höhe der Zeichenebene (als Ganzzahl)
bs`.loescheAlles()` macht die Zeichenebene leer.

6d. Die Klasse Tastatur

Auch hier muss zunächst ein Objekt beliebigen Namens erzeugt werden:
`Tastatur` *taste* `= new Tastatur();` // erzeugt neuen, leeren Tastaturpuffer
Dann kann mit dem erzeugten Objekt verwendet werden:
taste`.wurdeGedrueckt()` Falls eine Taste gedrückt wurde, die Tastatur also ein Zeichen enthält, ist wurdeGedrueckt wahr, sonst falsch.
taste`.zeichen()` enthält das zuerst eingegebene Zeichen der Tastatur. Dieses Zeichen ist danach nicht mehr in der Tastatur gespeichert

6e. Die Klasse Maus

Nach `Maus` *maus* `= new Maus()` sind möglich:
maus`.istGedrueckt()` Falls die linke Maustaste im Moment gedrückt ist, ist istGedrueckt wahr, sonst falsch.
maus`.hPosition()` enthält die gegenwärtige horizontale x-Koordinate der Position der Maus auf dem Bildschirm als ganze Zahl, unabhängig davon, ob die Maus gedrückt wurde.
maus`.vPosition()` ebenso für die vertikale y-Koordinate

6f. Die Klasse BuntStift

Der Stift ist ein Werkzeug, das sich auf dem Bildschirm bewegen kann. Er befindet sich stets auf einer genau definierten Position des Bildschirms, die durch Zeichenkoordinaten (horizontal nach rechts, vertikal nach unten) angegeben wird, und zeigt in eine Richtung, die durch Winkel beschrieben wird (0˚ rechts, Drehsinn mathematisch positiv).
Der Stift kennt zwei Zustände: Ist der Stift abgesenkt (*runter*) und bewegt er sich über den Bildschirm, so hinterlässt er eine Spur, die von einem Zeichenmodus abhängig ist. Ist der Stift angehoben (*hoch*), hinterlässt er keine Spur.

Beim Zeichnen kennt der Stift drei Modi: *normal* - der Stift zeichnet eine Linie in der Stiftfarbe; *wechseln* - der Stift zeichnet eine Linie, wobei die Untergrundfarbe in die Stiftfarbe und die Stiftfarbe in die Untergrundfarbe geändert wird; *radieren* - der Stift zeichnet eine Linie in der Farbe des Untergrunds.

Auch von der Klasse BuntStift muss zunächst ein Objekt erzeugt werden, z.B. durch
`BuntStift` *stift* `= new BuntStift();`

Jetzt können u.a. folgende Methoden genutzt werden:
stift`.bewegeUm (`*distanz*`)` Der Stift wurde von seiner aktuellen Position in die aktuelle Richtung bewegt. Die Zahl *distanz* gibt die Länge der zurückgelegten Strecke an
stift`.bewegeBis(`*x, y*`)` Der Stift wird unabhängig von seiner vorherigen Position auf die durch die Parameter angegebene Position bewegt.
stift`.dreheUm(`*winkel*`)` Der Stift wird ausgehend von seiner vorherigen Richtung um die durch die Zahl *winkel* angegebene Winkelgröße im mathematisch positiven Sinne weitergedreht.
stift`.dreheBis(`*winkel*`)` Der Stift wird unabhängig von seiner vorherigen Richtung auf die durch die Zahl *winkel* angegebene Winkelgröße gedreht. 0˚ ist rechts, 90˚ ist oben,...
stift`.hoch()` bzw. *stift*`.runter()` Der Stift wird angehoben bzw. zum Zeichnen abgesenkt
stift`.schreibe(`*text*`)` Der Stift schreibt den angegebenen Text auf die Zeichenebene unter Verwendung seines aktuellen Zeichenmodus, unabhängig vom Zustand hoch oder runter. Die aktuelle Stiftposition war die linke untere Ecke des Textes. Die neue Stiftposition ist die rechte untere Ecke des Textes.
stift`.normal()` *stift*`.wechsle()` *stift*`.radiere()` setzt einen Zeichenmodus (s.o.)
stift`.zeichneRechteck(`*breite*`,`*hoehe*`)` Der Stift zeichnet unabhängig von seinem Zustand im aktuellen Zeichenmodus ein achsenparalleles Rechteck mit der aktuellen Position als linker oberer Ecke und der angegebenen Breite und Höhe. Die Position und die Richtung des Stiftes bleiben unverändert.
stift`.zeichneKreis(`*radius*`)` Der Stift zeichnet unabhängig von seinem Zustand im aktuellen Zeichenmodus einen Kreis mit der aktuellen Position als Mittelpunkt und dem angegebenen Radius. Die Position und die Richtung des Stiftes bleiben unverändert.
stift`.zeichneLinie(`*x1,y1,x2,y2*`)` Der Stift zeichnet eine Linie von (x1 | y1) nach (x2 | y2). Position und die Richtung des Stiftes bleiben unverändert.
stift`.hPosition()` liefert die horizontale x-Koordinate der aktuellen Stiftposition.
stift`.vPosition()` liefert die vertikale y-Koordinate der aktuellen Stiftposition.
stift`.winkel()` liefert die momentan eingestellte Bewegungsrichtung des Stifts.
Außerdem sind – in Verbindung mit den oben unter 6a. genannten Konstanten – folgende Aufträge an jedes BuntStift-Objekt möglich:
stift`.setzeFarbe(`*Farbkonstante*`)`
stift`.setzeFuellmuster(`*Musterkonstante*`)`

7. Java-Befehle; wichtige Datenstrukturen; Fehlerbehandlung

Statt einer besonderen Übersicht wird auf die API-Dokumentation von Java verwiesen. Typische Bildschirmobjekte und die Befehle zu ihrer Verwendung sind dem Kapitel 9 (Java-AWT-Demo-Programm) zu entnehmen. Kontrollstrukturen wurden bereits im Kapitel 5 behandelt.

Auf komplexere Datenstrukturen (wie z.B. die Reihung [in anderen Sprachen auch als Array bezeichnet]) bzw. die Verwendung von Dateien wird noch im Unterricht eingegangen; Gleiches gilt für die Verwendung von *try..catch*-Blöcken zur Reaktion auf Fehler.

Auch die Vererbung wird später behandelt.

8. Typischer Programmaufbau – allgemein und mit Stift&Co

An zwei weiteren Beispielen soll nochmal der allgemeine Programmaufbau dargestellt werden (vgl. auch das Uhr-Programm auf den Seiten 11-14 sowie die etwas andere Nikohaus-Version auf Seite 2/3.) .

8a. Beispiel a

Stift & Co mit Zeichnungen

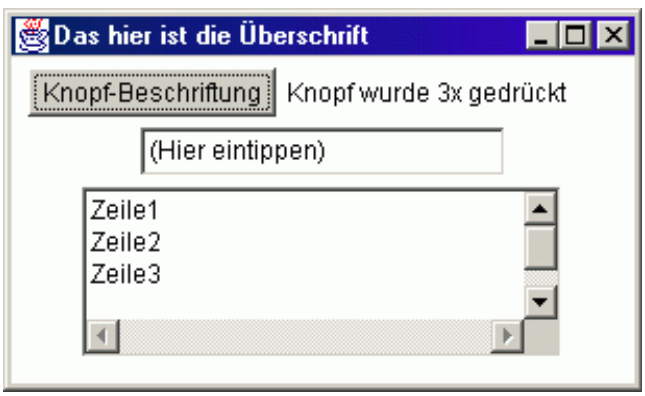
Allgemein: Objektbeschreibung.java	Beispiel NikoHaus.java
import *Benötigte Bibliotheken*; public class *Klassenname (=Dateiname)* { 1. *Deklaration von allgemein benutzten Objekten (auch mit Erzeugen) und Deklaration von globalen Variablen (Datenfeldern) (evtl. mit Anfangswert)* 2. *Methodendeklaration* public*) void**) *Methodenname* () { *2a. Erzeugen deklarierter, noch nicht erzeugter Objekte (mit new)* *2b. Deklaration lokaler Variablen (auch mit Anfangswert)* *2c. Anweisungen und Kontrollstrukturen: Aufträge (und Fragen) an die erzeugten Objekte* *) *public oder private (oder gar nichts). Es muss mindestens eine nicht-private Methode geben, die von der Startklasse (s.u.) aufgerufen werden kann* **) *void oder Ergebnistyp* } }	`import stiftUndCo.*;  //Bibliothek einbinden` `public class NikoHaus` `{` `  Bildschirm zeichenFläche; //Deklaration..` `  BuntStift meinStift;  //..von Objekten` `  Maus meinNagetier;    //..aus der Bibliothek` `   //Keine weiteren einfachen globalen Variablen` `  public void zeichne ()` `  //zentrale Methode der neuen Klasse` `  {` `     zeichenFläche = new Bildschirm ();//Erzeu` `     meinStift = new BuntStift (); //gen der` `     meinNagetier = new Maus ();  //deklarier-` `     meinStift.bewegeBis (180,250);//Aufträge..` `     meinStift.schreibe ("Nikolaus-Haus");` `     meinStift.bewegeBis (200,200);` `     meinStift.runter ();` `     meinStift.bewegeBis (300,200);` `     Hilfe.warte (300);` `      [..] //weitere Befehle, nicht abgedruckt` `     meinStift.hoch();` `     while (! meinNagetier.spezialKlick ())` `     {                    // Kontrollstruktur` `       meinStift.bewegeBis (170,270);` `       meinStift.schreibe ("Ende mit rechtem` `           Mausklick");` `     } //Ende von while` `  } //Ende von zeichne` `} //Ende von NikoHaus`
Allgemein: ObjektStart:java	**Beispiel: NikoStart.java**
public class *StartAnwendung (=Dateiname)* { public static void main (String[] s) { *1. Deklarieren eines Objekts/Exemplars der oben beschriebenen Klasse* *2. Erzeugen des Objekts* *3. Auftrag ans Objekt, sich mit einer Methode zu zeigen* } }	`public class NikoStart` `{` `  public static void main (String[] s)` `  {` `    NikoHaus meineHütte; //Obj. deklarieren` `    meineHütte = new NikoHaus ();//erzeugen` `    meineHütte.zeichne (); //Auftrag ans..` `  } //gerade erzeugte Objekt, sich darzustellen` `}`

8b. Beispiel b

Stift & Co mit Eingabe über die Tastatur und Textausgabe auf dem Bildschirm

Allgemein: Objektbeschreibung.java	Beispiel Gruss1.java
import *Benötigte Bibliotheken*; public class *Klassenname (=Dateiname)* { *1. Deklaration von allgemein benutzten Objekten und von globalen Variablen* *2. Methodendeklarationen* private *Ergebnistyp Methodenname* () { *2a. (Dekl. und) Erzeugen von Objekten (mit new)* *2b. Deklaration lokaler Variablen (auch mit Anfangswert)* *2c. Anweisungen und Kontrollstrukturen: Aufträge (und Fragen) an die erzeugten Objekte* public void *Methodenname* () { *2a. - -* *2b. Deklaration lokaler Variablen (auch mit Anfangswert)* *2c. Anweisungen und Kontrollstrukturen* } }	`import stiftUndCo.*;` `public class Gruss1` `{` `  String gruss = "Hallo Welt";  //globale Variable..` `  Bildschirm fenster = new Bildschirm(300,200,gruss);` `  BuntStift stift = new BuntStift(); //..und Objekte` `  private int wieOft()  //Methode für klasseninternen` `  {                    //Gebrauch (durch grüßen, s.u.)` `    Tastatur taste = new Tastatur(); //lokale Objekte` `    int zahl;            //..und Variable` `    do {` `      stift.schreibe ("Wie oft grüßen (0..9)?");` `      while (!taste.wurdeGedrueckt()) {}; //nichts!` `      char zchn = taste.zeichen();` `      zahl = (int)zchn - (int)'0'; //Typumwandlung` `    } while (zahl<0 || zahl>9);  //Bereichsprüfung` `    stift.schreibe (" "+zahl); //Eingabe auf Schirm` `    return (zahl);       //Ergebnis-Rückgabe` `  }` `  public void grüßen () //Methode für den Aufruf` `  {                                //von außen` `    stift.bewegeBis (10,15);` `    int anzahl = wieOft(); //Anfangswert aus wieOft` `    for (int zahl=1; zahl<=anzahl; zahl++)` `    {          //„zahl++“ kurz für „zahl = zahl + 1"` `      stift.bewegeBis (10, 30+15*zahl);` `      stift.schreibe(""+zahl+".) "+gruss);` `    }` `  }` `}`
Allgemein: ObjektStart:java	**Beispiel: Gruss1Start.java**
public class *StartAnwendung (=Dateiname)* { public static void main (String[] s) { *1. Deklarieren und 2. Erzeugen vom Objekt* *3. Auftrag ans Objekt: Methodenaufruf* } }	`public class Gruss1Start` `{` `  public static void main (String[] s)` `  {` `    Gruss1 proggi = new Gruss1();` `    proggi. grüßen();` `  }` `}`

9. Typisches Programm mit der Java-AWT



Auf der bzw. den nächsten Seiten wird ein Programm vorgestellt, dass nicht mehr mit Stift&Co, sondern mit der Java-AWT arbeitet und Labels, Schaltflächen (Knöpfe), Eingabezeilen und Textareas enthält.

Typische Java-AWT-Objekte und -Methoden:

```
1  // Demo: Ein-/Ausgabe mit der Java-AWT
2  // Java jdk 1.1.18 -- R. Krell, 8.11.01/28.11.01 – aber auch 2003 noch gültig
3  import java.awt.*;
4  import java.awt.event.*;
5
6  public class JavaEA extends Frame  // Frame ist Fensterklasse
7  {
8    Label ausgabezeile;  // Deklaration der Bildschirmobjekte
9    Button knopf;
10   TextField eingabezeile;
11   TextArea mehrzeiligesTextfenster;
12   int zähler = 0;  // für Beispiel: Knopfdrücke werden gezählt
13
14   private void richteFensterEin() // Fenster initialisieren und beschreiben
15   {
16     //WindowsListener hinzufügen, damit Schließknopf funktioniert
17     addWindowListener (
18       new WindowAdapter ()
19       {
20         public void windowClosing (WindowEvent ereignis)
21         {                 //ersetzt bisher leere Methode
22           setVisible (false);
23           dispose();
24           System.exit(0);
25         }
26       }
27     ); // runde Klammer vom Windowlistener geschlossen;
28
29     setTitle("Das hier ist die Überschrift"); // Fenster mit Titel versehen
30     setSize (420,200); //Fenstergröße (Breite und Höhe in Pixeln) festlegen
31     setLayout (new FlowLayout()); // innere Unterteilung des Fensters,
32     // kann auch später in führeAus() vor den add(..)-Befehlen stehen
33     // Ohne besondere Definition ist meist FlowLayout voreingestellt.
34   }
35
36   private void richteAusgabeEin()
37   {
38     ausgabezeile = new Label("-----------------------------------");
39     // Erzeugen des Ausgabelabels; Größe richtet sich nach Text!
40   }
41
42   private void richteKnopfEin()
43   {
44     // Knöpfe definieren und beschriften (Größe entsprechend Beschriftung)
45     knopf  = new Button ("Knopf-Beschriftung");
46     // oder: .. = new Button(); knopf.setLabel ("Knopf-Beschriftung");
47
48     //Funktion der Knöpfe festlegen
49     knopf.addActionListener (
50       new ActionListener ()
51       {
52         public void actionPerformed (ActionEvent e)
53         {
54           // hier steht der Programmtext, der ausgeführt werden
55           // soll, wenn der Knopf gedrückt wird. z.B.:
56           zähler++; // Kurzform für „zähler = zähler + 1;“
57           ausgabezeile.setText ("Knopf wurde "+zähler+"x gedrückt");
58         }
59       }); // runde Klammer (von addActionListener)
60    }
61
```

```
62   private void richteEingabezeileEin ()
63   {
64     eingabezeile = new TextField("(Hier eintippen)",30);
65     // 30 = Breite für 30 Zeichen
66     // spätere Textänderung mit eingabezeile.setText("Neuer Text");
67
68     eingabezeile.addActionListener (
69       new ActionListener ()
70       {
71         public void actionPerformed (ActionEvent e)
72         {
73           // hier steht der Programmtext, der nach der Eingabe (wenn die
74           // <Eingabe>-Taste gedrückt wurde) ausgeführt werden soll, z.B.:
75           ausgabezeile.setText ("Eingabe '"+eingabezeile.getText()+"'");
76           // Eingaben können wie folgt in Kommazahlen verwandelt werden:[2])
77           Double hilfszahlobjekt = Double.valueOf (eingabezeile.getText());
78           double kommazahl = hilfszahlobjekt.doubleValue();
79           mehrzeiligesTextfenster.append ("Als Dezimalbruch="
80               +kommazahl+"\n");
81           // oder -- weniger aufwändig -- in Ganzzahlen:
82           int ganzzahl = Integer.parseInt (eingabezeile.getText());
83           mehrzeiligesTextfenster.append ("Ganze Zahl="+ganzzahl+"\n");
84           // Falls die Eingabe gelöscht werden soll:
85           eingabezeile.setText("");
86         }
87       }); // runde Klammer (von addActionListener)
88    }
89
90   private void richteTextfensterEin ()
91    {
92     mehrzeiligesTextfenster = new TextArea("Zeile1\nZeile2\nZeile3\n",4,30);
93     // 4 Zeilen hoch, 30 Zeichen breit. (Anfangs-)Text kann auch später
94     // eingefügt werden mit:
95     // mehrzeiligesTextfenster.setText("Zeile1\nZeile2\nZeile3\n");
96     mehrzeiligesTextfenster.setEditable(false);
97     // bei true kann der Textfensterinhalt auf dem Bildschirm vom Benutzer
98     // verändert werden (Normalfall), bei false nicht.
99     // Üblicherweise kein ActionListener, sondern Button für Aktionen!
100   }
101
102  public void führeAus ()
103  {
104    richteFensterEin();       // Aufruf der Methoden zum Erzeugen der
105    richteKnopfEin();         // Bildschirmobjekte und der Definition
106    richteAusgabeEin();       // der bei Ereignissen auszuführenden Methoden
107    richteEingabezeileEin();
108    richteTextfensterEin();
109
110    add (knopf);              // Anordnung der Objekte auf dem Bildschirm
111    add (ausgabezeile);       // in der hier gewählten Reihenfolge bzw.
112    add (eingabezeile);       // mit dem hier gültigen Layout-Manager
113    add (mehrzeiligesTextfenster);
114    setVisible(true);         // Macht Bildschirm mit allen angeordneten
115      // Objekten sichtbar und startet damit das Programm
116  }
117 }
```

---

[2]) in neuer Java-JDK bzw. SDK ab 1.2 gelingt auch bei Kommazahlen vom Typ Float oder Double die Umwandlung wie bei Ganzzahlen ohne Zwischenobjekt (Z. 78/79) z.B. mit double kommazahl = Double.parseDouble (eingabezeile.getText());

Fehler bei der Zahl-Umwandlung (weil z.B. Buchstaben eingetippt wurden) führen hier noch nicht zu einer Reaktion. Der Programmtext nach dem fehlerhaften Umwandlungsversuch innerhalb der actionPerformed-Methode wird ignoriert – sonst passiert nichts.

Diese AWT-Demo kann - wie üblich - mit nachfolgender Startdatei als Application gestartet werden. Der ebenfalls leicht mögliche Start als Applet wurde bereits beim Uhr-Programm in Kapitel 4 gezeigt. Dann wäre allerdings noch eine geeignete Webseite (HTML-Datei) nötig.

```
public class JavaEAStart
{
  public static void main (String[] s)
  {
    JavaEA demo;
    demo = new JavaEA();
    demo.führeAus();
  }
}
```

Die vorstehenden Ausführungen – ohnehin viel umfangreicher, als ursprünglich geplant – sollten eine mehr als ausreichende Grundlage für den Start mit Java darstellen. Alles Weitere muss dem Unterricht entnommen bzw. selbst mitgeschrieben und notiert werden!