Informatik I für Hörer anderer Fachrichtungen

(Java)

Sommersemester 2002

Dr. G. Bauer

Inhaltsverzeichnis

0 Einleitung **1**
- 0.1 Was ist Informatik ? 1
- 0.2 Wegmarken 1
- 0.3 Entwicklung der Rechenleistung 3

1 Rechner, Betriebssystem, Grundlagen von UNIX **4**
- 1.1 Rechneraufbau : 4
- 1.2 Betriebssysteme : 4
- 1.3 Dateiverwaltung 5
- 1.4 Shell 6
- 1.5 Standarddateien 6
- 1.6 Kommandosprache 6

2 Java : Applikationen und Applets **7**
- 2.1 Grundbegriffe und erste Beispiele 7
- 2.2 Applikationen 8
- 2.3 Konventionen : 8
- 2.4 Kommandozeilenargumente : 9
- 2.5 Applets 9
- 2.6 Sicherheitsaspekte 11

3 Programmentwurf **11**
- 3.1 Schema 11
- 3.2 Anforderungsphase 12
- 3.3 Systementwurfsphase 12
- 3.4 Detailentwurfsphase 13
- 3.5 Kodierungsphase 14

4 Algorithmen **16**
- 4.1 Grundbegriffe 16
- 4.2 Darstellung von Algorithmen 16
- 4.3 Beispiel : Sortieren durch Minimumsuche 19

5 Syntaxdiagramme **23**
- 5.1 Erstes Beispiel 23
- 5.2 Bestandteile und Interpretation eines Syntaxdiagramms (allgemein) 24
- 5.3 Weitere Beispiele 25

6 Grundelemente der Sprache Java **26**
- 6.1 Zeichensatz 26
- 6.2 Format von Java-Programmen 26
- 6.3 Kommentare 26
- 6.4 Operatoren 27
- 6.5 Schlüsselworte 27

7 Einfache Datentypen, Variable und Konstanten **27**
- 7.1 Datentypen 27
- 7.2 Variable und Variablendeklarationen 30
- 7.3 Benannte Konstanten 31

8 Ausdrücke, Wertzuweisungen **31**
- 8.1 Operanden, Operatoren und Prioritäten 31
- 8.2 Wertzuweisung 34

9 Kontrollstrukturen **35**
- 9.1 Blöcke 35

9.2 Verzweigungen 35
9.3 Schleifen (Wiederholungsanweisungen) 37
9.4 Beispiele : 38
9.5 Marken und Sprunganweisungen 39

10 Klassen, Objekte und Methoden 40
10.1 Beispiel 40
10.2 Klassendeklaration 41
10.3 Attribute 41
10.4 Methoden 42
10.5 Bemerkungen 43
10.6 Objekte 44
10.7 Zugriff auf Attribute und Methoden 44
10.8 Konstruktoren 46
10.9 Beispiel : Die Klasse Punkt 47
10.10 Überladen von Methoden 47
10.11 Parameterübergabe-Mechanismen 49
10.12 Das Schlüsselwort `this` 51
10.13 Klassen- und Instanzattribute 53
10.14 Klassenmethoden 54

11 Felder 55
11.1 Vorbemerkungen 55
11.2 Felder 55

12 Vererbung 58
12.1 Komposition von Klassen 58
12.2 Erweitern von Klassen; Vererbung 60
12.3 Das Schlüsselwort `super` 61
12.4 Zuweisungskompatibilität 62
12.5 Abstrakte Klassen 62
12.6 Zugriffsrechte 63
12.7 Interfaces 64

13 Fehlerbehandlung 65
13.1 Vorbemerkungen 65
13.2 Auswerfen von Ausnahmen 65
13.3 Abfangen von Ausnahmen (try, catch) 66
13.4 Der `finally`-Block 68
13.5 Hierarchie der Ausnahmeklassen 69
13.6 Zusammenfassung: Prinzip der Ausnahmebehandlung 69

14 Grafikprogrammierung 70
14.1 Vorbemerkungen 70
14.2 Aufbau eines Grafik-Programms 71
14.3 Die Klasse Frame 72
14.4 Die Klasse Graphics 73
14.5 Farben (die Klasse `Color`) 73
14.6 Schriften (die Klassen `Font` und `FontMetrics`) 74

15 Applets 77
15.1 Vorbemerkungen 77
15.2 Initialisierung und Finalisierung 77
15.3 Parameterübergabe an Applets 78

16 Literatur 80

0 Einleitung

0.1 Was ist Informatik ?

Informatik (engl. : Computer science) ist

die Wissenschaft und Technik der Verarbeitung von Informationen mit Hilfe elektronischer Rechenanlagen.

Informationsverarbeitung ist ein grundlegendes Element menschlichen Lebens. Die spezielle Betonung der Informatik liegt dabei auf einer Informations- (oder Daten-) Verarbeitung, die **von einer Maschine** (in Abgrenzung etwa zu einem menschlichen Bearbeiter) durchgeführt werden kann. Heute lassen sich bereits sehr viele Datenverarbeitungsaufgaben mit Hilfe von Computern lösen, so z.B.

- Verwaltung von Konten einer Bank
- Textverarbeitung
- Wettervorhersage
- Verarbeitung von physikalischen und chemischen Messdaten
- .....u.v.m.

Grundlage hierfür sind jeweils systematische Verarbeitungsvorschriften, die so präzise formuliert sind, dass sie von einer Rechenanlage automatisch durchgeführt werden können.

Eine präzise formulierte Verarbeitungsvorschrift in diesem Sinne heißt **Algorithmus**. Dieser Begriff ist ein zentraler Begriff der Informatik, um den sich ein weites Spektrum von wissenschaftlichen Themenbereichen gruppiert, so z.B.

- Aufbau und Arbeitsweise der Maschinen, die Algorithmen ausführen können
- praktischer Einsatz von Rechenanlagen bei der Durchführung von Algorithmen
- maschinengerechte Darstellung von Daten und Algorithmen
- Konstruktion von Algorithmen
- prinzipielle Grenzen der maschinellen Lösbarkeit von Datenverarbeitungsaufgaben

Die Beschäftigung mit der **Hardware**, d.h. mit Aufbau und Wirkungsweise von elektronischen Rechenanlagen, gehört zum Teilbereich der **technischen Informatik**. Diese hat zahlreiche Berührungspunkte mit der Elektrotechnik und der Physik. Die **praktische Informatik** umfasst alle Bereiche der Konstruktion, Darstellung und Ausführung von Algorithmen (**Software**). Zur **theoretischen Informatik** zählen z.B. die o.g. Frage nach den prinzipiellen Grenzen der maschinellen Lösbarkeit von Datenverarbeitungs-problemen sowie ganz allgemein die theoretische Durchdringung und Grundlegung von Fragen und Konzepten der Informatik. Die praktische und vor allem die theoretische Informatik beziehen einen großen Teil ihrer Methoden und Techniken aus der Mathematik. Darüber hinaus gibt es eine Reihe von Frage-stellungen, die sich nicht in diese drei Bereiche einordnen lassen (so z.B. Fragen nach den gesellschaftlichen Auswirkungen).

0.2 Wegmarken

Ein **Rechner** ist allgemein eine Maschine zur Verarbeitung von Informationen aller Art (nicht nur Zahlen !) wie z.B. Zahlen , Texte, Bilder, Meßwerte .....

Altertum : mechanische Rechenwerke der einfachsten Art (Abakus,...)

1623 : erste urkundlich nachweisbare Rechenmaschine ("Rechenuhr"; konstruiert von Schickard in Tübingen). Die Pläne gehen leider verloren und werden erst 1956 wiederentdeckt.

1641 : Blaise Pascal stellt den Entwurf einer mechanischen Addiermaschine vor, die lange als die erste galt. Diese Maschine kann allerdings keine Subtraktionen.

1672 : Leibniz : Konstruktion einer Rechenmaschine für alle 4 Grundrechenarten

1801: Joseph-Marie Jaquard entwickelt ein Lochkartensystem für den Betrieb von Webstühlen. Diese Methode wird später für Computer verwendet.

1825-1840: Charles Babbage entwirft Pläne für eine Rechenmaschine, die gespeicherte Programme und auf Lochkarten gespeicherte Daten verwenden kann. Die Maschine hat er nie gebaut, da er die nötigen Finanzmittel nicht auftreiben konnte. Die "Analytic Engine" wurde nach den alten Plänen 1991 gebaut und funktioniert problemlos.

1833-1842: Die grundlegenden Konzepte zur Programmierung der "Analytic Engine" wurden von Ada Augusta Lovelace (Mitarbeiterin von Babbage) entwickelt.

1889: Herman Hollerith vom MIT entwickelt ein Buchhaltungssystem, das Lochkarten verwendet. Dieses System wird bei der Volkszählung 1890 in den USA verwendet. Sein Unternehmen, die Tabulating Machine Co., wird später in International Business Machines Corp. umbenannt.

1936: Alan Turing schreibt eine Abhandlung über die Theorie berechenbarer Zahlen. Der wesentliche Teil ist dabei ein Gedankenmodell über den grundsätzlichen Aufbau eines universellen Rechenautomaten. Seine Arbeiten sind von grundlegender Bedeutung für die spätere Entwicklung des digitalen Computers.

1939: Konrad Zuse baut den ersten funktionierenden mechanischen Rechenautomaten (Z1); die Grundstruktur entspricht der "Analytic Engine"; Zuse kannte allerdings die Arbeiten von Babbage und Lovelace nicht.

ab 1940 : Zuse u. Mitarbeiter : erste Konzepte elektronischer Rechner (Schaltkreise realisiert durch Röhren)

1941: John Atanasoft und Clifford Berry (Iowa State Collage) entwickeln gemeinsam den ersten Digitalcomputer, den sie ABC nennen. Sie zeigen ihre Konstruktion John Mauchly (University of Pennsylvania), der dann zusammen mit seinem Kollegen Prepser Eckert ein Digital-Computer-Modell patentieren lässt und gemeinsam mit diesem eine Firma (die spätere Sperry-Rand Corp.) gründet. Erst 1972 wird Atanasoft rehabilitiert, als die Honeywell Inc. die Sperry-Rand wegen der Unrechtmäßigkeit obigen Patents verklagt und das Patent für ungültig erklärt wird.

1939-1943: Howard Aiken (Harvard University) entwickelt unabhängig einen programmgesteuerten Computer auf elektromechanischer Basis (Schaltkreise realisiert durch Relais). 1944 erfolgt die Fertigstellung des ersten Typs (MARK I). Das Gerät ist über 15 Meter lang und wiegt 4,5 Tonnen. Die US-Marine benutzt ihn zur Berechnung von Raketenflugbahnen.

1943: Eckert und Mauchly entwickeln den elektronisch-numerischen Integrator und Computer (Eniac). Er wiegt 27 Tonnen und enthält 18 000 Vakuumröhren. Die US-Armee kauft ihn und verwendet ihn zur Berechnung von Raketenflugbahnen und für die Entwicklung der Wasserstoffbombe.

1948: In den Bell Labs erfinden William Shockley, Walter Brattain und John Bardeen den Transistor (Nobelpreis für Physik 1956).

1950: Eckert und Mauchly entwickeln den UNIVAC-1, einen Computer, der im Handel erhältlich ist. Es ist der erste allgemein gebräuchliche Computer, der Zahlen und Texte verarbeiten kann.

1952: Grace Hopper, eine Programmiererin für die US-Marine, konstruiert den ersten Compiler, der eine dem Englischen verwandte Sprache in Maschinensprache übersetzt. 1957 entwickelt sie die Programmiersprache COBOL.

1954-1958: Jim Backus (IBM) und eine Mitarbeiterin entwickeln FORTRAN, die erste anspruchsvolle Programmiersprache.

1959: Jack Kilby und Robert Noyce (Texas Instruments) entwickeln den integrierten Schaltkreis. Noyce macht sich später selbständig und gründet die Intel Corp.

1963: Der erste Mini-Computer, der PDP-8, wird bei Digital Equipment Corp. entwickelt. Als die Harvard-Universität ihn ausleiht, wird er dort gestohlen.

1968: Ken Thompson und Dennis Ritchie entwicken bei Bell Labs das Betriebssystem UNIX.

1969: Das US-Verteidigungsministerium entwickelt ARPAnet, ein Netzwerk aus 4 Computern. ARPAnet bildet die Grundlage für das heutige Internet.

1971: Intel bringt den ersten Microprozessor, den 4004-Chip, auf den Markt.

1973: Der erste Microcomputer, Micral, wird in Frankreich entwickelt. Er setzt sich jedoch nicht durch.

1975: IBM bringt seinen Portable Mercury 5100 auf den Markt. Das Gerät wiegt 25 kg und kostet 9000 $.

1976: Wang Labs stellt das erste vollautomatisierte Textverarbeitungsgerät vor. Preis: 30 000$

1980: IBM bestellt bei Microsoft ein Betriebssystem für seinen vor der Markteinführung stehenden PC. Gates und Allan haben kein solches System und erwerben deshalb für weniger als 100 000 $ die Rechte an Q-DOS, das sie dann in DOS umbenennen.

1981: IBM bringt den IBM-PC auf den Markt.

1985: Microsoft führt Windows ein. IBM ist nicht interessiert; deshalb vermarktet Microsoft das System selbst.

1989: Das WWW wird ins Leben gerufen. Entwickelt hat es Timoty Berners-Lee für internationale Forscher bei CERN in Genf. Die zugehörige Software wird 1992 veröffentlicht.

0.3 Entwicklung der Rechenleistung

Die Rechenleistung wird in der Regel als Verarbeitungsgeschwindigkeit von Operationen angegeben. Übliche Maßeinheiten : Typische Rechnerleistungen :

mips : **m**illion **i**nstructions **p**er **s**econd : Eine Instruktion ist dabei ein Datentransfer von einem Speicher zum anderen

mops : **m**illion **o**perations **p**er **s**econd : Hier gehen nur die Rechenoperationen ein

mflops : **m**illion **fl**oating-point **o**perations **p**er **s**econd : Hier gehen nur die (zeitintensiven) Gleitkommaoperationen ein

Jahr	Rechnertyp	**Leistung in mflops**
1946	ENIAC	10^{-5}
1964	CDC 6600	1
1970	IBM 368/85	1
1975	Cray 1	10
1980	Cyber 205	60
1985	IBM 3090	10
1985	Cray 2	150
1990	FPST	5000
1993	Parsytec GC	400000
1997	ASCI Option Red	1068000

Die hohe Rechenleistung z.B. des letztgenannten Rechners wird durch hohe Parallelität erreicht (dieser Rechner hat über 7000 Prozessoren !).

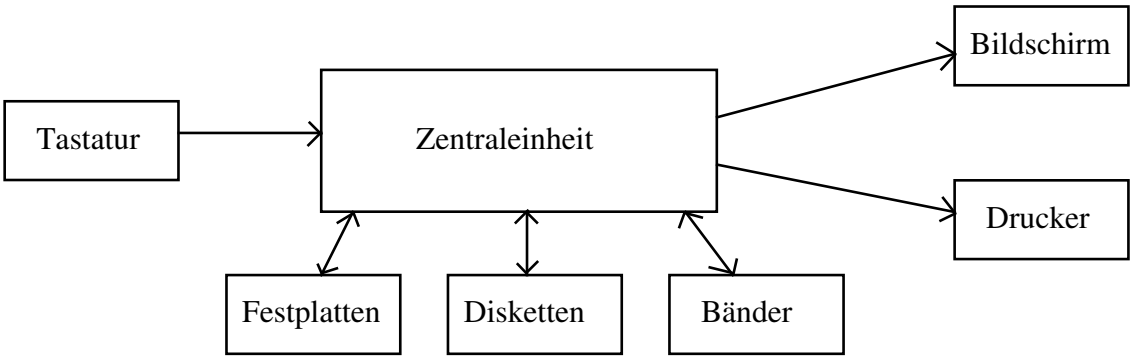
1 Rechner, Betriebssystem, Grundlagen von UNIX

1.1 Rechneraufbau :

Ein Rechner besteht immer aus

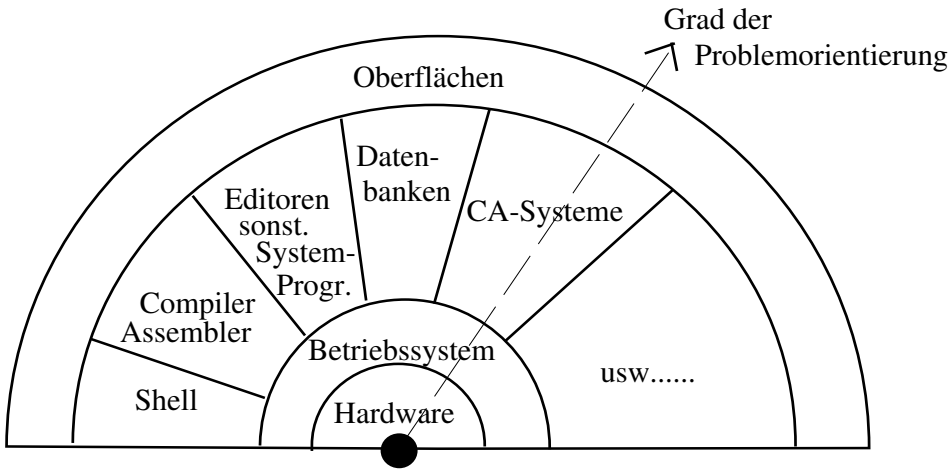
- einer Zentraleinheit
  und
- der Peripherie (Ein/Ausgabegeräte, Massenspeicher)

Schema :



1.2 Betriebssysteme :

Bei jedem Rechner gibt es eine Fülle von (lästigen) Aufgaben, die der Benutzer nicht übernehmen möchte und in der Regel auch nicht übernehmen kann. Die Bearbeitung dieser rechnerinternen Organisationsprobleme ist die Aufgabe des **Betriebssystems**.



Schalenmodell

Aus Anwendersicht ist das Betriebssystem nichts anderes als ein notwendiges Hilfsmittel, um dem Anwender die

- Ein/Ausgabe von Daten
- Dateiverwaltung
- Speicherverwaltung
- Job- und Prozessverwaltung

abzunehmen.
Von diesen Aufgaben soll an dieser Stelle lediglich die Dateiverwaltung beschrieben werden, wie sie von UNIX (und in ähnlicher Form von DOS etc.) durchgeführt wird.

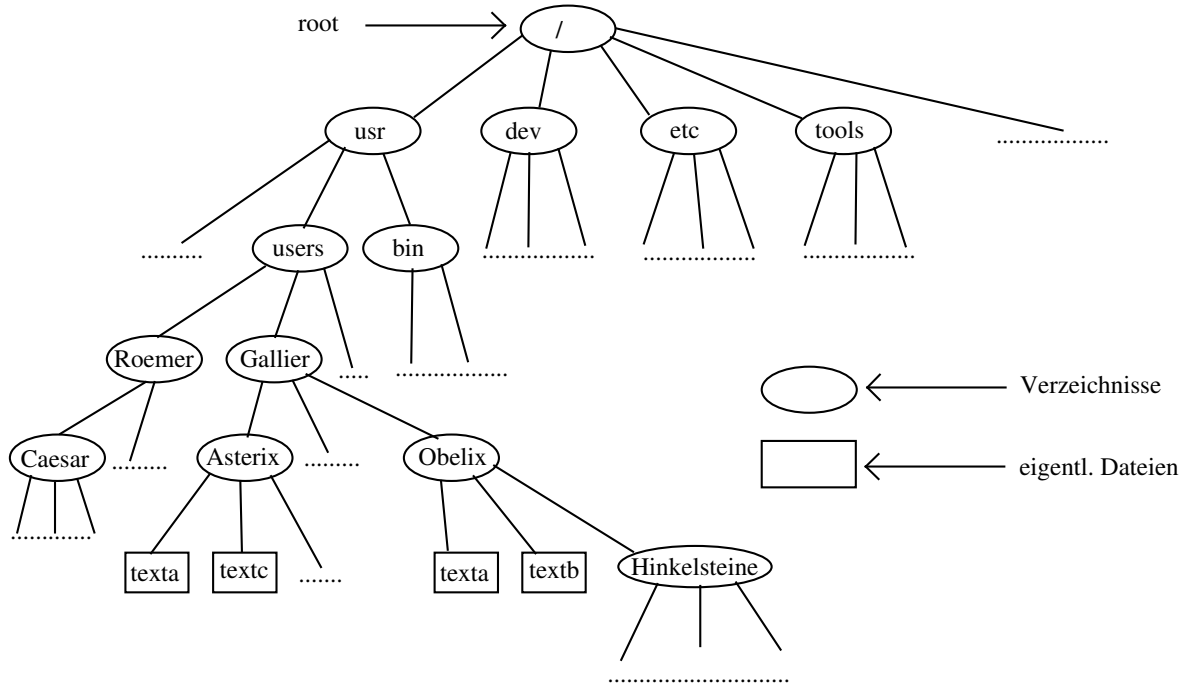
1.3 Dateiverwaltung

Alle Informationen, Programme, Texte, Bilder und sonstige beliebige Daten sind in UNIX in **Dateien** abgelegt. Ebenso werden Geräte wie z.B. Terminals und Drucker als Dateien verwaltet. Das Dateisystem ist **hierarchisch organisiert** und hat die Struktur eines Baums. Jeder Knoten im Baum stellt eine Datei dar. Die inneren Knoten heißen **Verzeichnisse** (directory). Dieser Dateityp enthält nur Verwaltungsinformationen und verweist auf weitere Dateien. Die eigentlichen Dateien wie Texte, Programme, Geräte usw. bilden die Blätter des Baums.

Die Organisation des **Dateisystems** könnte dabei z.B. wie in dem Bild „Dateiorganisation“ aussehen.

Der Name einer Datei wird in einem solchen Dateisystem durch den **absoluten Pfadnamen** eindeutig beschrieben, der den Weg von der Wurzel des Baums bis einschließlich des Zielknotens bezeichnet. Die einzelnen Elemente des Pfads werden durch "/" voneinander getrennt. Knoten auf verschiedenen Pfaden dürfen auch gleiche Namen haben ("texta" im angegebenen Beispiel).
Die Wurzel des Dateisystems (root) hat den vorgegebenen Namen "/"; alle anderen Namen sind frei wählbar.

Beispiel : `/usr/users/Gallier/Asterix/texta`
Um die Eingabe langer Pfadnamen zu vermeiden, gibt es für jeden Benutzer ein Arbeitsverzeichnis (**working directory**). Als Arbeitsverzeichnis können Sie ein beliebiges Verzeichnis auswählen. Fehlt nun die Angabe "/" am Anfang eines Pfadnamens, so beginnt der Abstieg nicht an der Wurzel des Dateisystems, sondern beim Arbeitsverzeichnis.



Dateiorganisation in Unix

Beispiel : Ist im vorangehenden Beispiel "users" das Arbeitsverzeichnis, so wird durch den **relativen Pfadnamen**

`Gallier/Asterix/texta`

dieselbe Datei wie oben bestimmt.

Weiter gibt es für jeden Benutzer ein **Home-Directory**, in dem die Daten des Benutzers liegen. Beim Öffnen einer Shell wird das Home-Directory automatisch zum Arbeitsverzeichnis. Unterhalb dieses Home-Directories kann der Benutzer beliebig (im Rahmen des ihm zur Verfügung stehenden Speicherplatzes) selbst neue Dateien und Verzeichnisse anlegen.

1.4 Shell

Während einer Sitzung tritt der Benutzer nicht direkt mit dem Betriebssystem in Verbindung, sondern mit der **Shell**, dem **Kommando-Interpretierer** (siehe Diagramm). Die Shell ist ein Programm, das nach einer korrekten Anmeldung standardmäßig gestartet wird und auf die Eingabe von Kommandos wartet. Nach Eingabe von "RETURN" analysiert die Shell die Kommandozeile, zerlegt sie in ihre Bestandteile und führt ggf. Expandierungen aus. Ist das erste Wort der Kommandozeile ein Shellkommando, so führt die Shell dieses aus. Andernfalls interpretiert die Shell dieses Wort als Programm und startet dieses.
In der Regel sind auf einer Anlage mehrere Shells verfügbar.

1.5 Standarddateien

In UNIX sind als Standard-Eingabedatei bzw. als Standard-Ausgabedatei die beiden Dateien *stdin* bzw. *stdout* definiert; daneben für Fehlerausgaben die Datei *stderr*. *stdin* , *stdout* und *stderr* sind **logische Namen** für real existierende Dateien. Ein/Ausgabegeräte wie z.B. Terminals werden in UNIX wie Dateien behandelt. Beim Anmelden des Benutzers am Terminal wird *stdin* der Tastatur und *stdout* dem Bildschirm (bei zeichenorientierten Terminals) bzw. dem gerade aktiven Eingabefenster (bei graphischen Terminals) zugewiesen. Diese Primärzuweisungen können auf andere Dateien umgelenkt werden. Dabei gelten folgende Regeln :

`<datei`	nimmt den Inhalt von *datei* als Eingabe
`>datei`	schreibt die Ausgabe (ohne Fehlermeldungen) in *datei*
`>>datei`	fügt die Ausgabe (ohne Fehlermeldungen) an *datei* an
`>&datei`	schreibt die Ausgabe und Fehlermeldungen in *datei*
`>>&datei`	fügt die Ausgabe und Fehlermeldungen an *datei* an

Eine getrennte Ausgabe der eigentlichen Ausgabedaten und der Fehlermeldungen ist nicht vorgesehen.

1.6 Kommandosprache

Kommandos haben die folgende allgemeine Form (Angaben in [..] sind optional):

`Kommandoname [- Optionen ] [Argumente]`

Beispiel : `ls -l *.pas`

Die Optionen bewirken, dass ein Kommando nicht in der standardmäßig vorgegeben Form ausgeführt wird, sondern in der Form, die durch die Optionen festgelegt wird. Die Optionen werden durch "-" eingeleitet und stehen vor den Argumenten.

Metazeichen (Wildcarts) dienen der Arbeitserleichterung bei der Eingabe von Kommandos. Sie sind Sonderzeichen, die jeweils für ganze Zeichenketten stehen und unabhängig vom Kommando sind. Die Shell ersetzt die Metazeichen zunächst durch existierende Zeichenketten ("Expandierung") und führt dann erst das Kommando aus. Die Shell kennt folgende Metazeichen :

*	steht für jede (einschließlich der leeren) Zeichenkette
?	steht für ein einzelnes Zeichen
[...]	steht für jedes der eingeschlossenen Zeichen. Ein durch "-" getrenntes Zeichenpaar steht dabei für alle ASCII - Zeichen zwischen den beiden Zeichen des Paars

Beispiele : Enthält das Arbeitsverzeichnis die Dateien test, test.pas, test.lst, testa.pas, testb.pas, testc.pas , so werden jeweils folgende Dateien aufgelistet durch

`ls t*`	alle genannten Dateien
`ls t*.pas`	test.pas, testa.pas, testb.pas, testc.pas
`ls *[a-c].pas`	testa.pas, testb.pas, testc.pas
`ls t???.*`	test.pas, test.lst

2 Java : Applikationen und Applets

2.1 Grundbegriffe und erste Beispiele

Java-Applikation : ein Programm, das nur auf dem eigenen Rechner ausgeführt wird

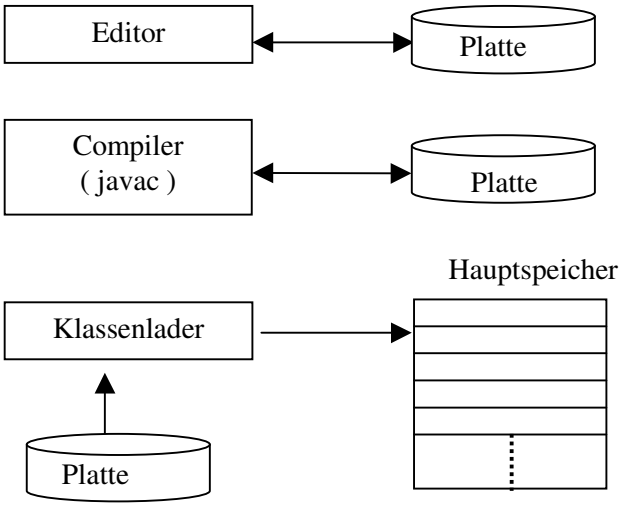
Java-Applet : ein Programm, das (in der Regel über das Internet) von einem Browser geladen und dann auf dem eigenen Rechner ausgeführt wird.

Verarbeitungsschritte :

- editieren
- compilieren
- laden / prüfen
- ausführen

Zur Durchführung dieser Verarbeitungsschritte stehen für den PC eine Reihe kommerzieller und recht bequemer Entwicklungsumgebungen zur Verfügung; auf den UNIX-Anlagen des RHRK sind nur die von den Sun Corp. im Rahmen des frei verfügbaren **J**ava **D**evelopment **K**it (JDK) gelieferten freien Werkzeuge installiert, die in gleicher Form auch für den PC zur Verfügung stehen.

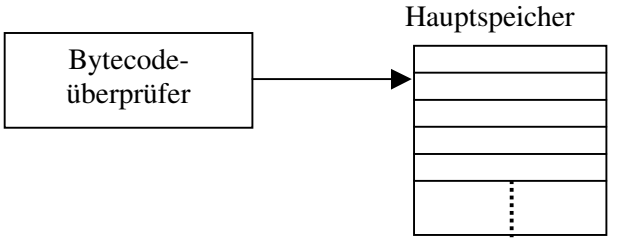
Eine typische Java-Entwicklungsumgebung hat folgendes Aussehen :



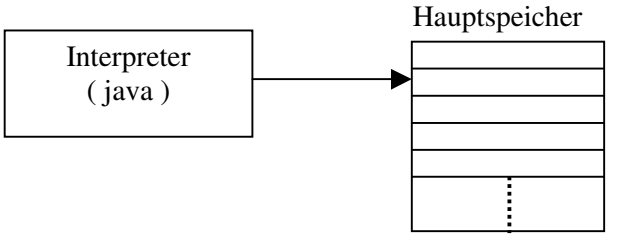
Das Programm wird im Editor geschrieben und in einer Datei auf der Platte abgespeichert.

Der **Compiler** liest das in einer Datei auf der Platte gespeicherte Programm und erzeugt einen plattformunabhängigen **Bytecode.**

Der **Klassenlader** lädt die Bytecodes in den Hauptspeicher.



Der **Bytecodeüberprüfer** untersucht, ob alle Bytecodes gültig sind und die Sicherheits-restriktionen von Java nicht verletzen.



Der (plattformspezifische) Interpreter liest Byte-codes, übersetzt sie in Maschinenbefehle und führt diese aus.

2.2 Applikationen

Beispiel : Zeige die Textzeile " Mein erstes Java-Beispiel " auf dem Bildschirm an.

Java-Applikation `Beispiel_1.java` :

Zeile	**Programmtext**
1	`// Ein erstes Java-Programm`
2	`import java.io.*;`
3	`class Beispiel_1`
5	`{`
6	`public static void main ( String[] Argument)`
7	`{`
8	`System.out.println("Mein erstes Java-Programm");`
9	`}`
10	`}`

Zu Zeile	Bemerkungen /Erläuterungen
1	"//" leitet eine Kommentarzeile ein; der Kommentar endet mit dieser Zeile.
2	Java enthält eine Reihe vordefinierter Bausteine (*Classes*), die in *Packages* zusammengefasst werden. Die Packages tragen die *Namen Java Class Library* bzw. *Java Application Programming Interface* (*Java API*). Mit der *import*-Anweisung werden Elemente dieser Packages, die im vorliegenden Programm benötigt werden, geladen. Java-Klassen unterstützen die Wiederverwendung von Code.
3	Hier wird eine neue Klasse, nämlich **Beispiel_1**, definiert : Allgemein wird eine Klasse durch das Schlüsselwort `class` gefolgt von dem Namen der Klasse eingeführt. Jedes Java-Programm besteht aus mindestens einer Klasse.
5, 10	Der Rumpf der Klassendefinition steht innerhalb der Begrenzungszeichen **{** und **}.**
6	Jede Java-Applikation enthält zumindest die *Methode* `main`. Diese wird beim Aufruf der Applikation automatisch ausgeführt. Die Bedeutung der Schlüsselworte `public`, `static` und `void` wird später erklärt. Bei der Ausführung der Klasse können Kommandozeilenargumente berücksichtigt werden. Durch `String[] Argument` wird festgelegt, dass diese Argumente Strings sind, auf die mit `Argument[0], Argument[1],...` zugegriffen werden kann.
7, 9	Auch der Rumpf der Methodendefinition von main wird durch **{** und **}** begrenzt.
8	Anweisung ; abgeschlossen durch ein Semikolon. Die Anweisung besteht hier aus einem Methodenaufruf der Methode `println`; das *Objekt* `System.out` (es entspricht i.w. der Standarddatei *stdout* mit einigen Ausgaberoutinen) ist vordefiniert. Allgemein wird eine für ein Objekt definierte Methode durch <*Name des Objekts*>**.** <Na*me der Methode*> (<*Parameterliste*>) aufgerufen.

2.3 Konventionen :

1. Der Programmtext **muss** in der Datei `Beispiel_1.java` abgespeichert werden.
   (Allgemein : Abspeicherung in <*`Name der Klasse`*>`.java`)
2. Der Java-Compiler *javac* erzeugt beim Aufruf
   `javac Beispiel_1.java`
   den zugehörigen (plattformunabhängigen) Bytecode und speichert ihn in der Datei
   `Beispiel_1.class`

3. Ausgeführt wird der Bytecode durch den Java-Interpreter *java* durch den Aufruf
   java Beispiel_1
4. Sollen Kommandozeilenargumente übergeben werden, so geschieht das in folgender Form :
   java Beispiel_1 *Arg0 Arg1* ....

2.4 Kommandozeilenargumente :

Beispiel :
```
// Ein zweites Java-Programm
import java.io.*;
class Beispiel_2
{
  public static void main ( String[] Argument)
  {
    System.out.println("Mein zweites Java-Programm");
    System.out.println("geschrieben von "+Argument[0]);
    System.out.println("..... am "+Argument[1]);
  }
}
```

Der Aufruf

```
java Beispiel_2 "Donald Duck" 12.5.1999
```

liefert die folgende Ausgabe :

```
Mein zweites Java-Programm
geschrieben von Donald Duck
..... am 12.5.1999
```

Die Kommandozeilenargumente sind Strings, die jeweils durch mindestens ein Leerzeichen getrennt sind. Soll (wie im Beispiel) einer der Strings selbst ein Leerzeichen enthalten, so muss er in Anführungszeichen gesetzt werden. Das obige Programm liefert einen Laufzeitfehler, falls weniger als zwei Kommandozeilenargumente geliefert werden.

2.5 Applets

Java-Applet `Beispiel_3.java`

Zeile	Programmtext
1	`//Java-Applet Beispiel_3`
2	`//es werden drei Zeilen auf dem Bildschirm angezeigt`
3	`import java.applet.Applet;  // import Applet class`
4	`import java.awt.Graphics;   // import Graphics class`
5	
6	`public class Beispiel_3 extends Applet {`
7	`   public void paint( Graphics g )`
8	`   {`
9	`      g.drawString( "Mein erstes Applet", 25, 25 );`
10	`      g.drawString( "im Praktikum zu ",25,40);`
11	`      g.drawString( "'Informatik I für Hörer anderer Fachrichtungen'", 25, 55 );`
12	`   }`
13	`}`

Zu Zeile	Bemerkungen /Erläuterungen
6	Hier wird die neue Klasse **Beispiel_3** definiert . Jede Klassendefinition basiert auf einer vorhandenen Klasse. Dafür gibt es einen Mechanismus, der als *Vererbung* bezeichnet wird (Schlüsselwort: **`extends`**). Wird wie in den vorangegangenen Beispielen keine Basisklasse explizit angegeben, so wird vom Compiler die aktuell definierte Klasse als Erweiterung der Basisklasse **`Object`** angesehen. Eine Klassenbeschreibung definiert keine Systeminstanz, sondern eine Menge von Klassenelementen, den sog. Objekten. Ein System entsteht durch Instanziierung von Objekten. Das Schlüsselwort **`public`** erlaubt dem Browser, das Applet zu erzeugen.
7	Beginn der Definition der Methode `paint`. Jedes Java-Applet enthält mindestens eine Methode, die das Verhalten des Applets steuert. Die Methode `paint` wird bei der Ausführung des Applets automatisch aufgerufen; die übergebenen Parameter stehen eingegrenzt durch **(** und **)** in der Parameterliste.
9, 10	Die in der Klasse `Graphics` definierte Methode `drawstring` gibt den 1. Parameter (Zeichenkette) ab der durch den 2. und 3. Parameter festgelegten xy-Koordinate des Applets aus. Gezählt werden dabei die Pixel ab der linken oberen Ecke des Applets.

Im Unterschied zu Applikationen sind Applets keine eigenständigen Programme, sondern sie müssen durch einen javafähigen Browser (z.B. Netscape, InternetExplorer,...) ausgeführt werden. Hierzu muss in einer entsprechenden HTML-Datei das entsprechende Applet angegeben werden.

Beispiel einer HTML-Datei zur Anzeige des obigen Applets:

Zeile	Text
1	**`<html>`**
2	**`<h2>`**
3	**`Das ist mein erstes Applet:`**
4	**`</h2>`**
5	**`<P>`**
6	**`<applet code="Beispiel_3.class" width=300 height=100>`**
7	**`</applet>`**
8	**`</html>`**

Zu Zeile	Bemerkungen /Erläuterungen
1	HTML (HyperText Markup Language) ist die Formatierungssprache des WWW. Das html-tag `<html>` steht am Beginn einer HTML-Datei. Allgemein stehen die verschiedenen html-tags in spitzen Klammern.
2	Formatierungstag, das die Darstellung des folgenden Textes in einer größeren Schriftart erzwingt.
3	Dieser Text wird ungeändert dargestellt.
4	Hier wird die in Zeile 2 angeforderte Darstellungsart wieder aufgehoben. Allgemein wird der Wirkungsbereich des Tags `<xxx>` durch `</xxx>` beendet.
5	Dieses Tag erzeugt einen Absatz
6	`applet`-tag mit Parametern; Beginn des Applets. - `code` **:** Datei mit dem kompilierten Applet - `width` **:** Breite des Applets in Pixeln - `heigth` **:** Höhe des Applets in Pixeln
7	Ende des Applets
8	Ende der HTML-Datei

Zur reinen Anzeige von Applets steht mit ***appletviewer*** ein Minimalbrowser zur Verfügung, der ausschließlich das `applet`-tag interpretieren kann und alle weiteren Tags ignoriert. Durch diesen Browser wird im Unterschied zu z.B. Netscape der Text aus Zeile 3 nicht angezeigt.

2.6 Sicherheitsaspekte

Applets werden üblicherweise von einem fremden Rechner heruntergeladen und dann auf dem eigenen Rechner durch den Browser ausgeführt. Dies ist wie bei jedem fremden Programm offensichtlich ein erhebliches Sicherheitsrisiko, denn ein unbekanntes Programm könnte z.B.

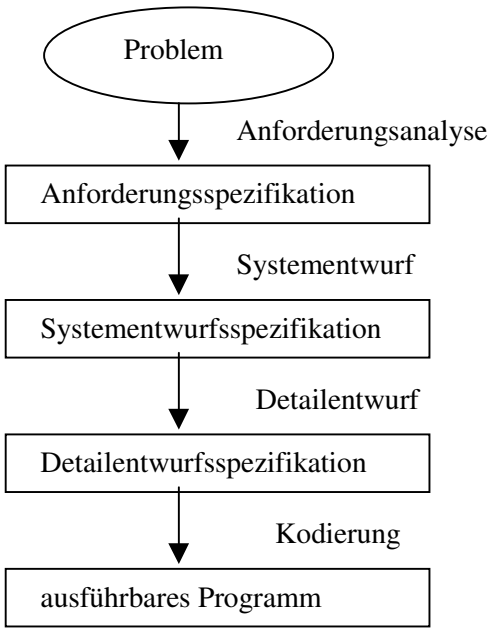
- den Rechner zum Absturz bringen
- Dateien löschen
- Informationen auf dem eigenen Rechner ermitteln und zu einem anderen Rechner schicken
- die Festplatte mit Datenmüll füllen
- Viren einschleusen
- ....

Dabei ist es ziemlich unerheblich, ob der Schaden bewusst oder ungewollt durch einen Programmierfehler angerichtet wird. Java enthält zur Vermeidung dieser Probleme eine Reihe eingebauter Sicherheitsmechanismen, die an dieser Stelle noch nicht im Detail erläutert werden können. Allerdings ist kein Sicherheitsmechanismus vollkommen, so dass beim Herunterladen von Internetseiten fragwürdiger Herkunft größte Vorsicht anzuraten ist. Insbesondere sollte der Browser so konfiguriert sein, dass er beim Ausführen von Applets keinen Zugriff auf lokale Dateien erlaubt.

3 Programmentwurf

3.1 Schema

Beim Entwurf von Programmen / Systemen kann man nebenstehende Phasen unterscheiden:



Jede Phase hat dabei eine definierte Eingabe (Produkte / Dokumentation) und eine definierte Ausgabe (Produkte / Dokumentation).

3.2 Anforderungsphase

- Durchführung der Anforderungsanalyse, um das konkret vorliegende Problem zu verstehen.
- Beteiligt sind
  - der anvisierte Benutzer
  - das für die Realisierung zuständige Team
- Fragestellung : **Was** soll das System leisten? (hier ist noch ohne Bedeutung, **wie** die Leistung erbracht werden soll)
- Ergebnis : Anforderungsspezifikation
  Hier werden u.a.
  - Eingaben
  - Ausgaben
  - Funktionalität
  - Benutzerschnittstellen
  - .....

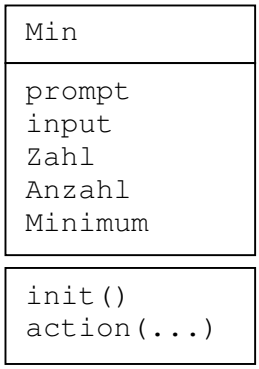
  festgelegt.

Beispiel : Minimum einer Zahlenfolge
Anforderungsspezifikation : Das System erhält eine Folge ganzer Zahlen per Tastatur; nach jeder Eingabe sollen sowohl die Anzahl als auch die kleinste der bisher eingelesenen Zahlen angezeigt werden und es soll eine weitere Eingabe möglich sein.

3.3 Systementwurfsphase

- Ausgangsdokument : Anforderungsspezifikation
- Frage : aus welchen Komponenten besteht das System?
- (abstrakter) Lösungsentwurf
  - Bestimmung der Komponenten
  - Zuordnung von Aufgaben zu den einzelnen Komponenten
  - Schnittstellenspezifikation (Beschreibung der Interaktion von Komponenten; Ein/Ausgabeverhalten)
  - Ergebnis : Systementwurfsspezifikation

Beispiel : Minimum einer Zahlenfolge
Das System besteht aus einem Objekt der Klasse Min. Die Merkmale dieser Klasse sind :



- `prompt` Aufforderungstext zur Eingabe der nächsten ganzen Zahl
- `input` Eingabebereich (per Tastatur)
- `Anzahl` Anzahl und Minimum der bisher eingegebenen Zahlen
  `Minimum`
  `Zahl` gerade eingegebene Zahl
- `init` Operation zur Initialisierung: die Objekte `prompt` und `input` werden erzeugt und in einem gleichfalls erzeugten Fenster auf dem Bildschirm angezeigt. `Anzahl` wird mit 0 initialisiert und die erste Eingabe wird erwartet.
- `action` Hier wird die Eingabe verarbeitet:
  `Anzahl` wird um 1 erhöht;

ist Anzahl = 1, so wird Minimum auf die gerade gelesene Zahl gesetzt;

ist Anzahl > 1, so wird Minimum auf die kleinere der beiden Zahlen Minimum und die gerade gelesene Zahl gesetzt.

Anschließend werden Anzahl und Minimum auf dem Bildschirm angezeigt und der Eingabebereich zurückgesetzt; die nächste Eingabe kann erfolgen.

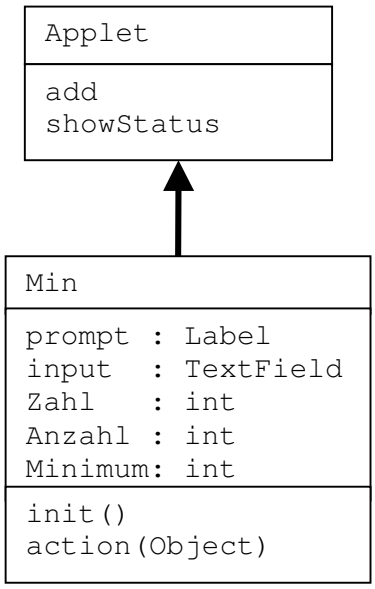
3.4 Detailentwurfsphase

- Ausgangsdokument : Anforderungsspezifikation
- Fragestellung : wie sollen die einzelnen abstrakten Komponenten realisiert werden?
- Verfeinerung der Systementwurfsspezifikation : dabei für jede Komponente
  - strukturelle Verfeinerung der abstrakten Komponente
  - Zuordnung von Aufgaben zu den internen Komponenten
  - Festlegung der Interaktionen
- Ergebnis : Detailentwurfsspezifikation

Beispiel : Minimum einer Zahlenfolge

Das vollständige System besteht aus einem Objekt der Klasse **Min**. Diese Klasse erbt die Merkmale der Klasse Applet, dabei insbesondere die Operationen (Methoden) add und showStatus.

Die Klasse besitzt weiterhin die folgenden zusätzlichen Merkmale :



- prompt Aufforderungstext zur Eingabe der nächsten ganzen Zahl (Objekt der Klasse Label)
- input Eingabebereich (ein Objekt der Klasse TextField)
- Anzahl Minimum Anzahl und Minimum der bisher eingegebenen Zahlen
- Zahl gerade eingegebene Zahl
- init Operation zur Initialisierung: die Objekte prompt und input werden erzeugt und in dem bereits erzeugten Applet auf dem Bildschirm angezeigt. Anzahl wird mit 0 initialisiert und die erste Eingabe wird erwartet.
- action Operation zur Verarbeitung einer Eingabe; die Operation wird immer nach erfolgter Eingabe durchgeführt (d.h. nach Drücken der Taste ↵).
  Die Operation hat einen Eingabeparameter der Klasse Object, der die Eingabe als String enthält. Dieser String wird in einen ganzzahligen Wert Zahl konvertiert (falls möglich).
  Anschließend :
  - Anzahl wird um 1 erhöht;
  - ist Anzahl = 1, so wird Minimum auf die gerade gelesene Zahl gesetzt;
  - ist Anzahl > 1, so wird Minimum auf die kleinere der beiden Zahlen Minimum und die gerade gelesene Zahl gesetzt.

  Anzahl und Minimum werden auf dem Bildschirm angezeigt und der Eingabebereich wird zurückgesetzt; die nächste Eingabe kann erfolgen.

3.5 Kodierungsphase

- Ausgangsdokument : Detailentwurfsspezifikation
- Fragestellung : wie soll die vollständige Lösung aussehen
- Kodierung und Zusammenbau der Komponenten des Detailentwurfs
- Ergebnis : strukturiertes ausführbares Programm

Beispiel : **Minimum einer Zahlenfolge** (**`Min.java`**)

```
1   // Minimumprogramm
2   import java.awt.*;
3   import java.applet.Applet;
4   public class Min extends Applet {
5      int Zahl;         // zu lesende Zahl
6      int Anzahl;       // Anzahl der bisher gelesenen Zahlen
7      int Minimum;      // Minimumm der bisher gelesenen Zahlen
8      Label prompt;     // Eingabeaufforderung
9      TextField input;  // Eingabefeld
10     public void init(){
11        prompt = new Label( "Ganze Zahl eingeben;  Return:" );
12        input = new TextField( 10 );
13        add( prompt );          // platziere prompt im Applet
14        add( input );           // platziere input im Applet
15        Anzahl = 0;             // initialisiere Anzahl mit 0
16     }
17     public boolean action( Event e, Object o ) {
18        Zahl = Integer.parseInt(o.toString() ); //ermittle Zahl
19        input.setText( "" );   // leere Eingabefeld
20        Anzahl = Anzahl + 1;   // erhöhe Anzahl um 1
21        if ((Anzahl == 1) | ((Anzahl > 1) & ( Zahl < Minimum)))
22        {
23           Minimum = Zahl;
24        }
25        showStatus( "Anzahl : " + Anzahl +
          "  bisheriges Minimum : " + Minimum);   //zeige Ergebnis
26        return true;  // Benutzeraktion verarbeitet
27     }
28  }
```

Zu Zeile	Bemerkungen
2,3	Das komplette `java.awt` Package (abstract window toolkit) und die Klasse `java.applet.Applet` werden importiert. Der Compiler verwendet daraus allerdings nur die Klassen, die im aktuellen Applet verwendet werden.
5-9	Deklarationen : Alle Variablen (d.h. einfache Variable und Objektreferenzen), die in einem Programm verwendet werden, müssen vorher durch Angabe des Namens und des Typs deklariert werden. Namenskonventionen : Namen sind (beliebig lange) Folgen von Buchstaben, Ziffern, Unterstreichungszeichen (_) und Dollarzeichen ($); die Folge darf nicht mit einer Ziffer beginnen. Groß- und Kleinbuchstaben werden unterschieden. Namen gleichen Typs können gemeinsam deklariert werden; im obigen Beispiel wäre also

auch möglich : `int Zahl, Anzahl, Minimum;`
Die hier deklarierten Variablen heißen Instanzvariable, da sie bei der Erzeugung eines Objekts der hier definierten Klasse neu angelegt werden und genauso lange wie das Objekt existieren.

5-7 Die hier deklarierten Variablen sind ganzzahlige Variable. Die Eigenschaften werden durch den vordefinierten Datentyp `int` festgelegt

8,9 `prompt` und `input` sind Objektreferenzen, deren Eigenschaften jeweils durch eine Klassendefinition festgelegt sind.

10 Bei der Erzeugung eines Applets werden immer 3 Methoden ausgeführt, nämlich `init`, `start` und `paint`. Wird eine der Methoden nicht explizit definiert, so wird sie von der Klasse `Applet` geerbt; dies sind dort Methoden mit leerem Rumpf.

11 Ein Objekt der Klasse Label mit dem Parameterwert `"Ganze Zahl eingeben; Return:"` durch den Aufruf der Operation `new` erzeugt; eine Referenz auf dieses Objekt wird durch den Zuweisungsoperator (=) an `prompt` übergeben.

12 analog Zeile 11; ein Eingabebereich für 10 Zeichen wird erzeugt.

13,14 mit `add` wird ein Objekt auf dem Applet platziert. Werden keine Vorgaben gemacht, so werden die Objekte einfach in der angegebenen Reihenfolge hintereinander platziert (ähnlich wie Fließtext); mit weitergehenden Hilfsmitteln lässt sich der Platzierungsort praktisch beliebig steuern.

15 Initialisierung von `Anzahl`.

17 Die Methode `action` verarbeitet Interaktionen zwischen dem Benutzer und den Komponenten der graphischen Benutzeroberfläche (GUI , graphical user interface) des Applets. Eine Interaktion des Benutzers (z.B. "RETURN" oder Mausklick auf einem Button) erzeugt ein Ereignis (`event`). Dies führt seinerseits automatisch zur Ausführung der Methode `action`. Die Eingabeparameter von `action` sind Objektreferenzen auf ein Ereignis (e) und ein Objekt (o). Das Objekt enthält ereignisspezifische Informationen; hier z.B. den Text aus dem Textfeld.

18 `o.toString` ist eine in der Klasse `Object` definierte Methode, die das Objekt in eine Zeichenkette (`String` ; elementarer Datentyp in Java) konvertiert. Dieser String ist dann Eingabeparameter für die Methode `Integer.parseInt`, die Strings in `integer`-Werte konvertiert. Sollte diese Konvertierung nicht möglich sein, so wird eine Fehlerbedingung erzeugt.

19 Die Methode `setText` ist eine Methode der Klasse `TextField`, um das Textfeld auf einen definierten Wert zu setzen (hier : das Textfeld wird geleert).

21-24 In Abhängigkeit von einem logischen Wert
(Anzahl == 1) | ((Anzahl > 1) & (Zahl < Minimum))
wird der Wert von `Minimum` ermittelt und gesetzt.

25 Der genannte Wert wird auf dem Bildschirm in der Statuszeile angezeigt. Zahlen werden zur Anzeige automatisch in Strings gewandelt; mehrere Strings werden mit dem Operator "+" zu einem String zusammengefügt.

Bemerkung :
Bei der Übersetzung dieses Applets erfolgt eine Warnung, nämlich

```
Note: Min.java uses or overrides a deprecated API.
Recompile with "-deprecation" for details.
1 warning
```

Diese Warnung bedeutet, dass eine veraltete Methode (hier "action") benutzt wurde, die aus Kompatibilitätsgründen noch zur Verfügung steht, aber durch bessere Methoden ersetzt werden sollte. Der Ersatz an dieser Stelle erfordert aber Sprachkonstrukte, die hier noch nicht zur Verfügung stehen.
Bei der schnellen Entwicklung der Sprache Java ist zu erwarten, dass diese Warnung in Abhängigkeit von der aktuell installierten Version häufiger auftritt. Sie verhindert aber nicht die Lauffähigkeit eines Programms.

4 Algorithmen

4.1 Grundbegriffe

Der Begriff des **Algorithmus** ist einer der zentralen Begriffe der Informatik.

Ein **Algorithmus** ist eine Berechnungsvorschrift, die aus mehreren elementaren Schritten besteht, die in einer bestimmten Reihenfolge ausgeführt werden müssen. Die Anwendung eines Algorithmus führt nach einer endlichen Zeit zu einem Funktionswert oder einer Datenmenge. Ein Algorithmus ist eine allgemeine Berechnungsvorschrift; d.h. er ist unabhängig von der Realisierung in einer bestimmten Programmiersprache. Zur Darstellung der Daten werden (wie in einer Programmiersprache) **Variable** verwendet. Ein Algorithmus legt weder Datenstrukturen noch Systemstrukturen fest, sondern setzt sie als gegeben voraus.

Für den Begriff "**Algorithmus**" gibt es eine Reihe formaler Definitionen (z.B. von Turing, Church, Kleene, Post, Markov ,...). Allgemein können einem Algorithmus folgende Merkmale zugeordnet werden :

- Ein Algorithmus muss von einer Maschine durchgeführt werden können (d.h. Intuition darf keine Rolle spielen). Die zum Ablauf des Algorithmus benötigten Informationen müssen zur Verfügung stehen.
- Ein Algorithmus besteht aus einer Reihe von Einzelschritten und Anweisungen über die Reihenfolge. Jeder Schritt muss in seiner Wirkung genau definiert sein.
- Ein Algorithmus muss nach einer endlichen Zeit und einer endlichen Zahl von Schritten enden. Für das Ende des Algorithmus muss eine Abbruchbedingung formuliert sein.

Beispiel: Bestimmung des Minimums einer (nichtleeren) Reihe einzulesender Zahlen.

1. Lese **Zahl**
2. Setze **Minimum = Zahl**
3. weitere **Zahl** vorhanden?
   nein: weiter bei 7
   ja: weiter bei 4
4. Lese **Zahl**
5. **Zahl < Minimum**?
   ja : Setze **Minimum = Zahl**
   nein: -
6. weiter bei 3
7. Ausgabe von **Minimum**

Ein **Programm** ist die Realisierung eines Algorithmus in einer Programmiersprache.

4.2 Darstellung von Algorithmen

Die informelle Beschreibung von Algorithmen wie im vorangegangenen Beispiel durch Aufzählung der einzelnen Schritte sowie der Vorschrift, in welcher Reihenfolge die Schritte durchgeführt werden sollen, ist in der Regel nicht sehr übersichtlich.

Besser geeignet sind formale Beschreibungsmittel wie z.B. formal definierte **Metasprachen** (Pseudocode) oder graphische Beschreibungen durch **Flussdiagramme** bzw. **Struktogramme**.

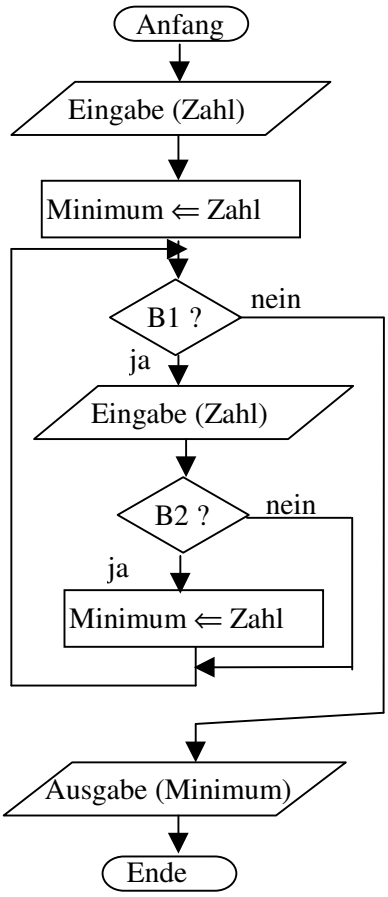
Pseudocode für den Algorithmus Minzahl :

```
Eingabe(Zahl)
Minimum ⇐ Zahl
solange (Eingabeende nicht erreicht)
tue     Eingabe(Zahl)
        wenn Zahl < Minimum
        dann  Minimum ⇐ Zahl
        endwenn
endtue
Ausgabe (....)
```

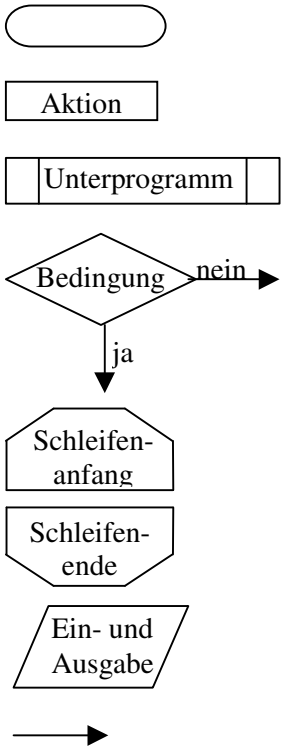
Flussdiagramme

sind graphische Darstellungsformen für Algorithmen. Dabei dienen spezielle graphische Symbole zur Beschreibung der Aktionen. Diese werden mit sog. Flusslinien zur Darstellung des Kontrollflusses verbunden.

Beispiel : **Flussdiagramm** für den Algorithmus Minzahl
(Zur Verkürzung der Schreibweise :
B1 steht für "weitere Zahl vorhanden"
B2 steht für "Zahl < Minimum")



Symbole(Auswahl):



Klammersymbol (begin , end)

Aktionssymbol

vordefinierte Routine ev. mit Parametern

Verzweigung

Schleifenanfang

Schleifenende

Ein/Ausgabe

Flussliniensymbol

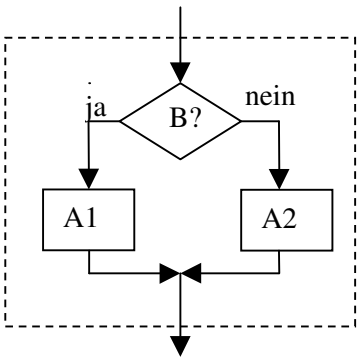
Kontrollstrukturen

Bei der Darstellung durch Flussdiagramme sind keine eigenen Symbole für die bei der strukturierten Programmierung üblichen komplexeren Kontrollstrukturen (= Anweisungen zur Steuerung der Reihenfolge der Ausführung von Anweisungen) wie z.B. Wiederholungsanweisungen vorgesehen. Diese Kontroll-strukturen werden dort direkt durch Abfragen und Sprünge dargestellt. Im Sinne der durch die gängigen Programmiersprachen unterstützten strukturierten Programmierung sollte man aber Sprunganweisungen ausschließlich so wie im folgenden beschrieben benutzen; Ausnahmen hiervon sind lediglich Anweisungen zum direkten Verlassen eines Blocks bzw. einer Routine.

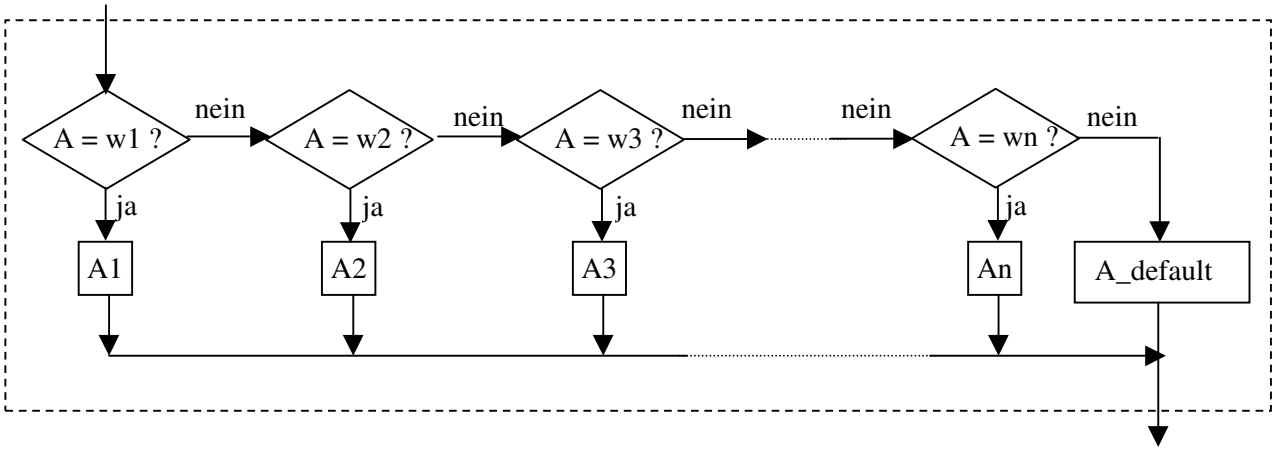
Selektionsblöcke

Bedingte Verzweigung :

Häufig ist es in Algorithmen erforderlich, eine Auswahl zwischen zwei Alternativen zu treffen. Ist die ***Bedingung*** B erfüllt, so wird A1 und ansonsten A2 durchgeführt. Der Block A2 kann leer sein.



Fallunterscheidung :



Eine Auswahl mit mehreren Alternativen ist eine **Fallunterscheidung**. In Abhängigkeit vom Wert des Ausdrucks **A** wird eine der Aktionen **A1, A2**,... durchgeführt. Der „Fehlerausgang“ **A_default** kann fehlen.

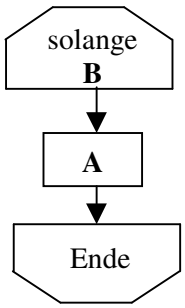
Iterationsblöcke :

Iterationsblöcke beschreiben das mehrfache Ausführen eines oder mehrerer Strukturblöcke. Hierfür stehen folgende Formen zur Verfügung :

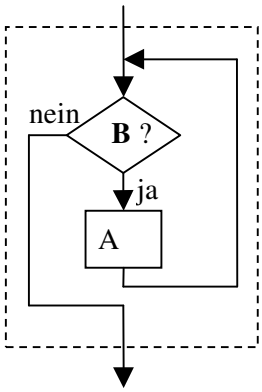
a) abweisende Schleife

Hier wird zunächst die Ausführungsbedingung ausgewertet. Wenn die Bedingung beim Eintritt in den Block nicht erfüllt ist, wird der Block **A** überhaupt nicht ausgeführt; ist die Bedingung immer erfüllt, so liegt eine Endlosschleife vor.

Diagramm:



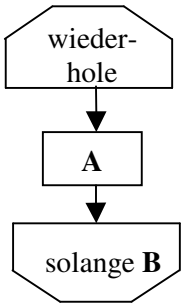
Semantik:



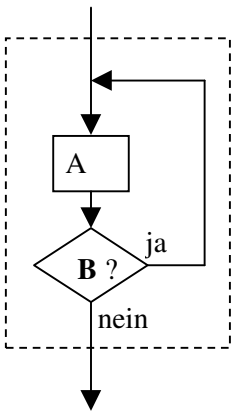
b) nicht abweisende Schleife

Hier wird der Block **A** mindestens einmal ausgeführt. Ist die Bedingung erfüllt, so wird die nächste Ausführung angeschlossen, ansonsten wird die Ausführung beendet.

Diagramm:



Semantik:



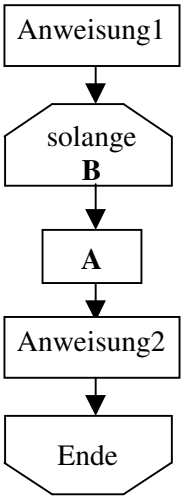
c) Zählschleife

Diese Schleife ist eine spezielle abweisende Schleife, die es erlaubt, einfache Initialisierungen und Zählvorgänge übersichtlich zu gestalten.

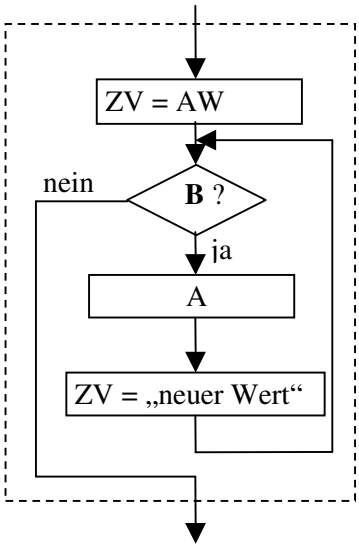
Die Zählvariable **ZV** wird mit dem Anfangswert **AW** initialisiert. Die Schleife wird nun sooft durchgeführt, bis erstmals die Bedingung **B** nicht mehr erfüllt ist. In der Regel ergibt sich der neue Wert der Zählvariablen durch Addition oder Subtraktion einer Konstanten; die Bedingung B ist in der Regel die Abfrage, ob ZV ≤ EW bzw. ZV ≥ EW mit einem geeigneten Endwert EW ist.

Allgemein: *for* (Anweisung1; Bedingung B; Anweisung2)

Diagramm:



Semantik:



4.3 Beispiel : Sortieren durch Minimumsuche

Ein Feld von n (> 0) ganzen Zahlen wird durch folgendes informell beschriebene Verfahren aufsteigend sortiert :

Schritt **1** : Die kleinste Zahl der Liste wird ermittelt und auf Platz **1** gesetzt (d.h. mit dem auf Platz 1 stehenden Element vertauscht)

Schritt **2** : Die zweitkleinste Zahl der Liste wird ermittelt und auf Platz **2** gesetzt (d.h. mit dem auf Platz **2** stehenden Element vertauscht)

....

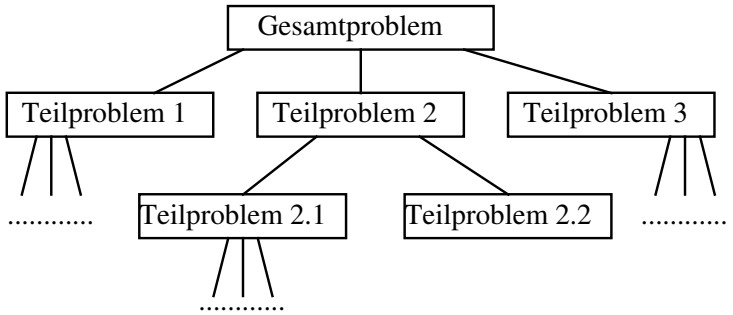
Schritt **k** : Die k-kleinste Zahl der Liste wird ermittelt und auf Platz **k** gesetzt (d.h. mit dem auf Platz **k** stehenden Element vertauscht)

.....

Abgebrochen wird, wenn die (n-1)-te Zahl ihren korrekten Platz erhalten hat. Berücksichtigt wird dabei, dass im Schritt k die k-kleinste Zahl der gesamten Liste bei obigem Vorgehen offensichtlich die absolut kleinste Zahl ist, die auf den Listenpositionen ≥ k gespeichert ist.

Der Algorithmus wird durch das Verfahren des **Top-down-Entwurfs** ("schrittweise Verfeinerung ") entwickelt. Bei diesem Entwurfverfahren wird das Problem in mehrere einfachere Teilprobleme (Module) aufgespalten, die dann getrennt und unabhängig voneinander behandelt werden. Die Teilprobleme werden entsprechend aufgespalten, bis man endlich vernünftig handhabbare Probleme erhalten hat.

Schema :



Im vorliegenden Beispiel wird nebenstehender Entwurf benutzt :

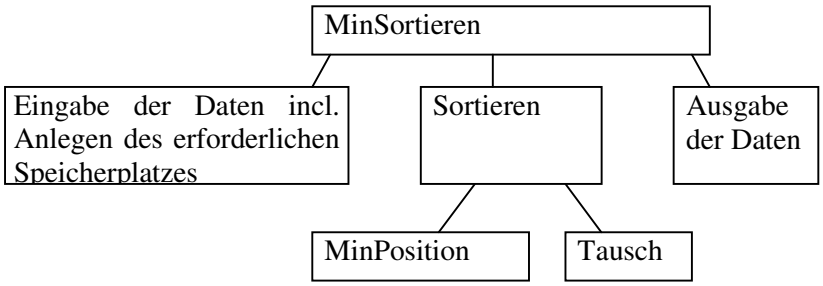


Diagramm des Gesamtprojekts Minsortieren:

Daten:

- aktAnzahl (ganze Zahl)
- aktFeld (Zahlenfeld)

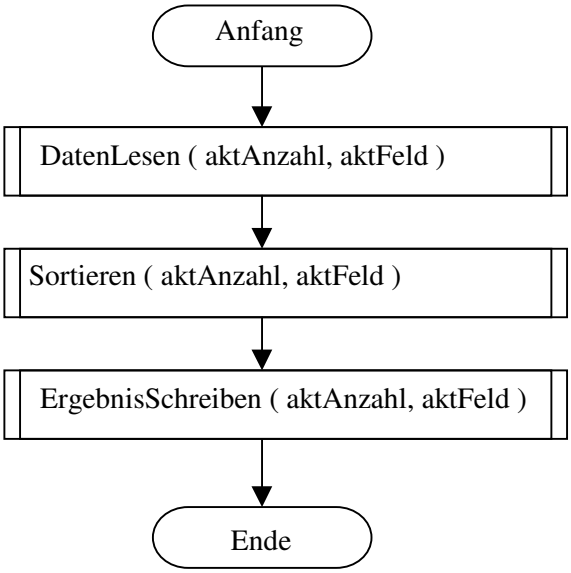


Diagramme der Einzelprojekte

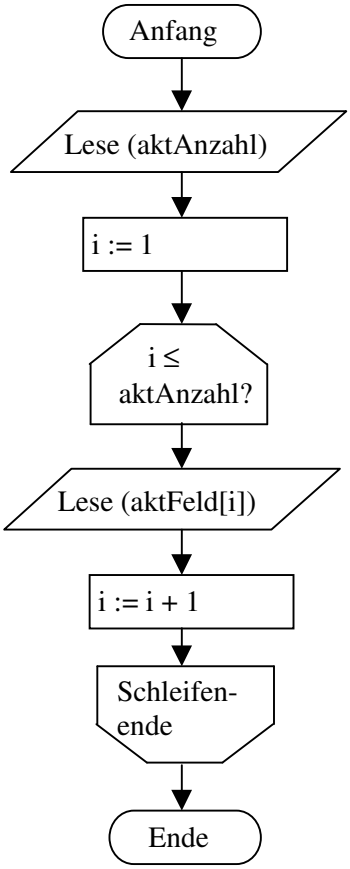
Zur Beschreibung der Einzelroutinen („Prozeduren“ bzw. „Funktionen“) gehört nicht nur die Angabe der Aktionen, sondern auch die Angabe der Parameter und der lokalen Variablen.

DatenLesen

Diese Prozedur liest zunächst die aktuelle Anzahl der zu sortierenden Daten und legt einen entsprechend großen Speicherbereich an. Anschließend werden die zu sortierenden Daten gelesen.

Parameter:
- aktAnzahl (ganze Zahl, Ausgabe)
- aktFeld (indiziertes Zahlenfeld, Ausgabe)

lokale Variable:
- i (ganze Zahl)

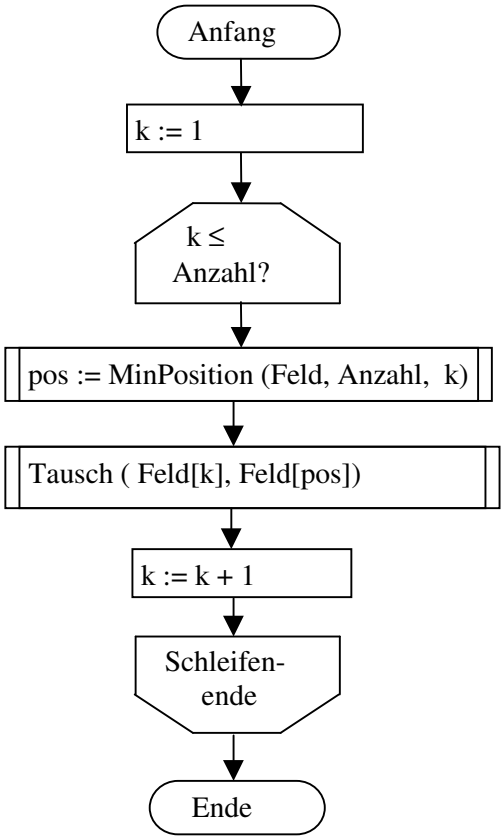


Minsortieren

An dieser Stelle wird erst der oben beschriebene grobe Ablauf formuliert. Die Routinen Tausch und MinPosition müssen erst noch definiert werden. Die Prozedur Tausch vertauscht dabei die Feldelemente auf den Positionen k und pos; die Funktion MinPosition ermittelt bei Eingabe des Feldes, seiner Größe und einer Anfangsstelle k die Indexposition der kleinsten Zahl auf den Positionen ab k.

Parameter:
- Anzahl (ganze Zahl, Eingabe)
- Feld (indiziertes Zahlenfeld, Ein/Ausgabe)

lokale Variable:
- k (ganze Zahl)



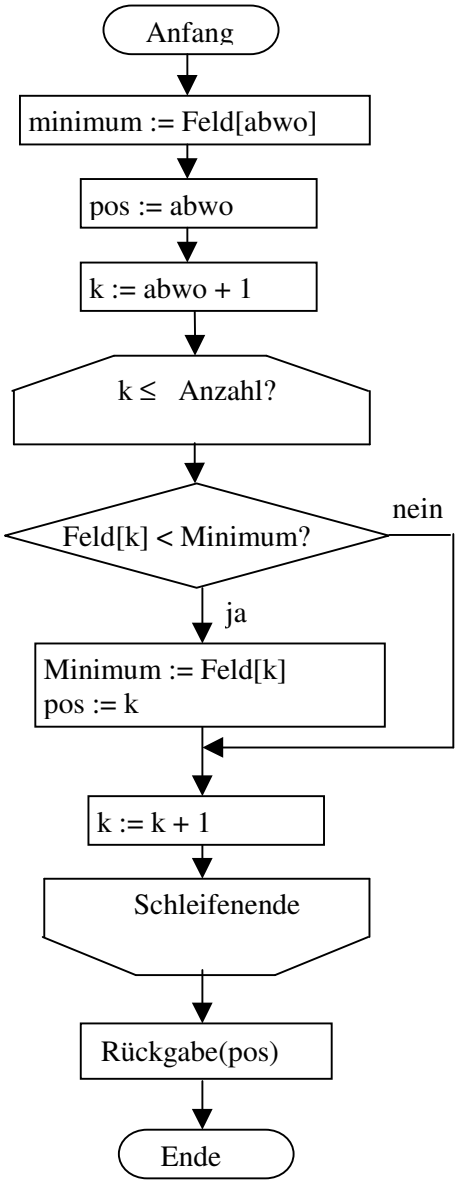
MinPosition

Diese Funktion ermittelt die Position (d.h. den Index) des kleinsten Elements im Bereich der Indexpositionen abwo bis Anzahl

Parameter:
- Anzahl , abwo (ganze Zahlen, Eingabe)
- Feld (indiziertes Zahlenfeld, Eingabe)

lokale Variable:
- k (ganze Zahl)
- Minimum (Zahl entsprechend Daten in Feld)
- pos (ganze Zahl)

Rückgabewert: pos

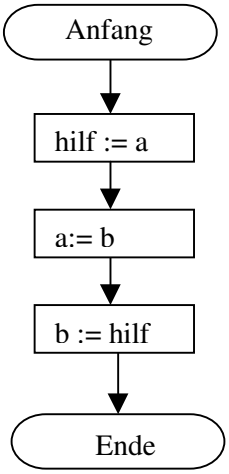


Tausch

Diese Prozedur vertauscht zwei Zahlen a und b

Parameter:
- a, b (Zahlen, Ein/Ausgabe)

lokale Variable:
- hilf (Zahl)

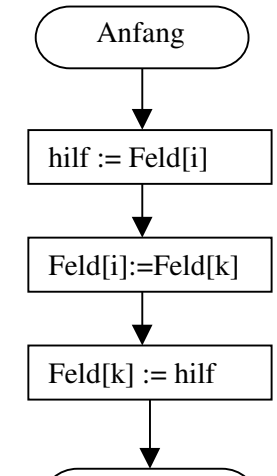


Tausch (modifiziert)

Im hier vorliegenden Beispiel ist es aus technischen Gründen erforderlich, die Prozedur **Tausch** in folgender Weise umzuformulieren:
Tausch erhält als Parameter das Feld und die beiden Indizes der zu tauschenden Feldelemente, d.h.

Parameter:
- i,k (ganze Zahlen, Eingabe)
- Feld (Zahlenfeld, Ein/Ausgabe)

lokale Variable:
- hilf (Zahl)



ErgebnisSchreiben

Diese Prozedur gibt die (sortierten) Daten aus

Parameter:
- Anzahl (ganze Zahl, Eingabe)
- Feld (indiziertes Zahlenfeld, Eingabe)

lokale Variable:
- i (ganze Zahl)

5 Syntaxdiagramme

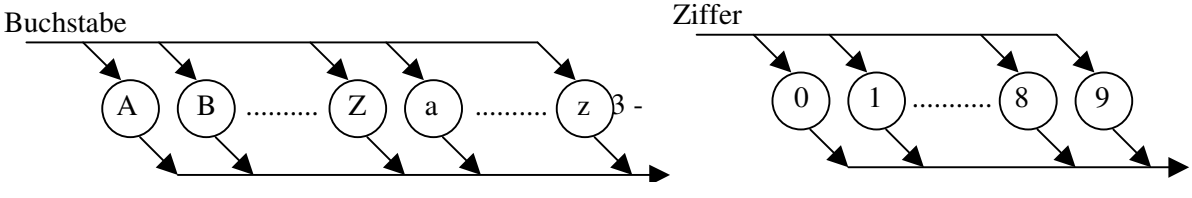
Die verbale Beschreibung syntaktischer Klassen ist nur in einfachsten Fällen eindeutig und leicht handhabbar. Zur formalen Beschreibung gibt es eine Reihe von Verfahren; weit verbreitet ist die Beschreibung durch Syntaxdiagramme.

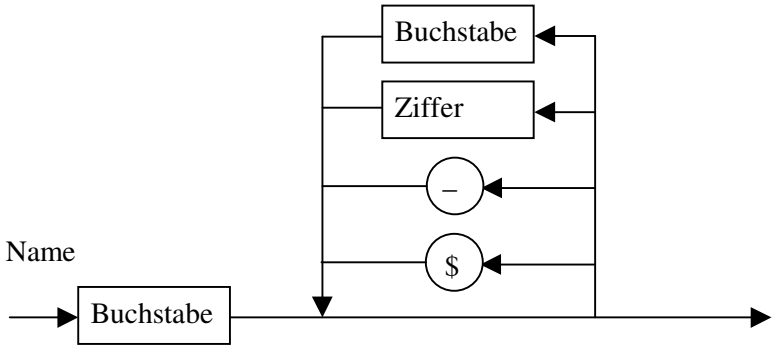
5.1 Erstes Beispiel

Namen (identifier) sind in Java wie folgt festgelegt :

- Namen müssen mit einem (großen oder kleinen) (Unicode-)Buchstaben beginnen
- anschließend kann eine beliebige Anzahl von (Unicode-)Buchstaben und Ziffern kommen
- das Unterstreichungszeichen (_) und das Dollarzeichen ($) sind Buchstaben im Sinne von Java

Die entsprechenden Syntaxdiagramme :





Bemerkungen :

In Java werden bei Namen Großbuchstaben und die entsprechenden Kleinbuchstaben unterschieden.

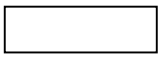
Prinzipiell dürfen neben den Standardbuchstaben wie oben beschrieben alle Buchstaben des Unicode-Zeichensatzes verwendet werden, der u.a. die nationalen Sonderzeichen enthält (z.B. ÊÅåÿç). Da dies aber in der Regel zu Problemen bei der Terminaldarstellung und beim Drucken führt, sollte man darauf verzichten.

Namen dürfen keine reservierten Worte sein (dies kann nicht im Syntaxdiagramm dargestellt werden)

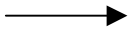
Allgemein werden (ohne Verpflichtung !) folgende Konventionen beachtet :
- Variablennamen beginnen mit einem Kleinbuchstaben (radius)
- Klassennamen beginnen mit einem Großbuchstaben (MinSortieren)
- besteht ein Name aus mehreren Worten, so werden in der Regel die Wortanfänge durch Großbuchstaben und nicht durch Unterstreichungszeichen gekennzeichnet
  (Beispiel : SortierenDurchMinimumsuche anstatt Sortieren_durch_Minimumsuche)

5.2 Bestandteile und Interpretation eines Syntaxdiagramms (allgemein)

In diesen rund begrenzten "Kästen" werden weiter nicht erklärte Grundsymbole eingetragen



In den Rechtecken stehen durch weitere Syntaxdiagramme beschriebene Begriffe (Rekursivität ist möglich). Der Eingang in das Rechteck entspricht dem Einstiegspunkt, der Ausgang dem Ausgang des entsprechenden Syntaxdiagramms.

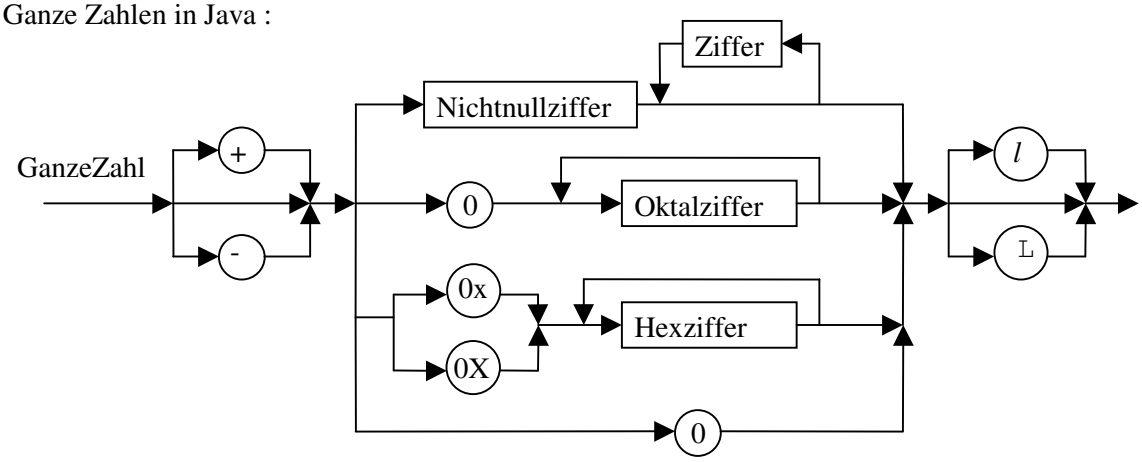


"Fahrwege" : diese Kästen sind durch "Fahrwege" verbunden, die im Sinne eines Eisenbahnnetzes durchlaufen werden können. Wenden und das Befahren von Spitzkehren ist dabei nicht erlaubt.

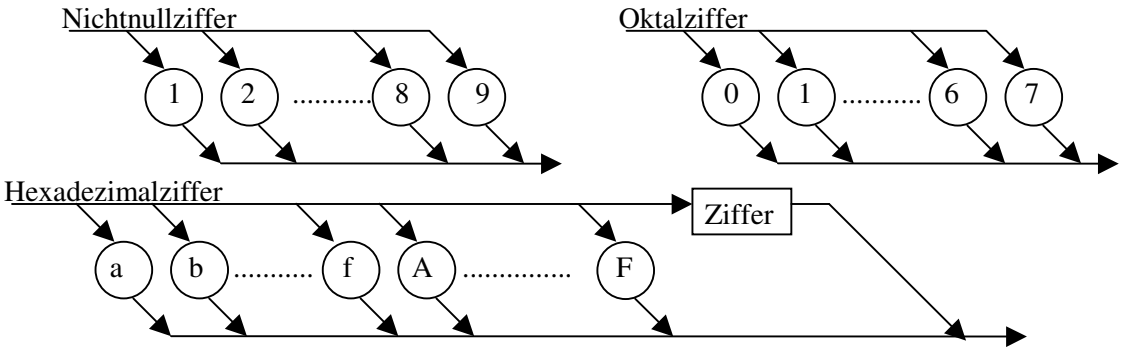
Ein Syntaxdiagramm hat immer **genau einen Einstiegspunk**t und **genau einen Ausgang**. Beginnend am Einstiegspunkt muss das Syntaxdiagramm "fahrgerecht" durchlaufen und irgendwann am Ausgang verlassen werden. Jede Folge von Objekten, die dabei besucht werden (und nur diese !) , stellt ein korrektes Objekt der gerade definierten syntaktischen Klasse dar.

5.3 Weitere Beispiele

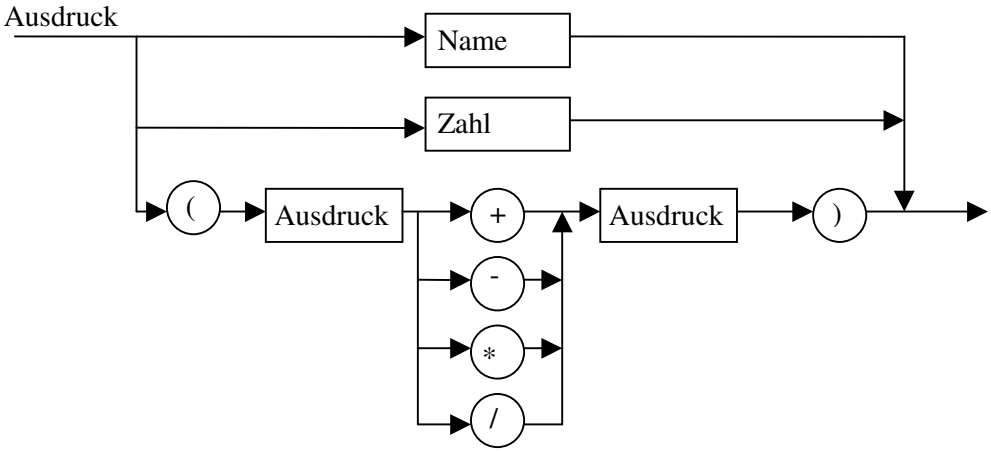
1. Ganze Zahlen in Java :



mit



2. Voll geklammerte arithm. Ausdrücke (ein Syntaxdiagramm für "Zahl" wird als bekannt vorausgesetzt):



Ein im Sinne dieses Syntaxdiagramms korrekter Ausdruck ist z. B.

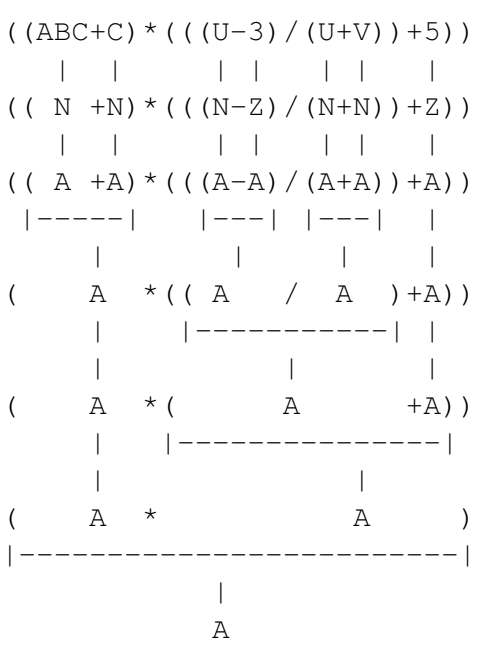
((ABC + C) * (((U - 3) / (U + V)) + 5))

Die nebenstehende Ableitung benutzt die Abkürzungen N für Name , Z für Zahl und A für Ausdruck .

Beachten Sie, dass nicht jede mögliche Ersetzung zum Ziel führt; so führt die im ersten Schritt auch mögliche Ableitung

```
ABC
|||
NNN
```

nicht weiter !



6 Grundelemente der Sprache Java

6.1 Zeichensatz

Java benutzt (anders als die meisten anderen Programmiersprachen) den standardisierten 16-Bit-Zeichensatz Unicode 1.1.5. Insbesondere wird hierdurch nicht nur die Zeichenmenge festgelegt, sondern auch deren explizite Kodierung. Dieser Zeichensatz umfasst neben den üblichen ASCII-Zeichen auch praktisch alle nationalen Sonderzeichen und viele weitere Zeichen (es können mit 16 Bit mehr als 65000 verschiedene Zeichen kodiert werden).

Da aber die Darstellung "ungewöhnlicher" Zeichen häufig sowohl auf dem Terminal als auch auf dem Drucker Probleme bereitet, sollte man sich trotz der weitergehenden Möglichkeiten in der Regel auf den weltweit verbreiteten ASCII Zeichensatz beschränken, da dieser von fast allen Betriebssystemen unterstützt wird. Die ersten 256 Zeichen des Unicode Zeichensatzes stimmen mit den Zeichen der ASCII-Variante Latin-1 überein; hierbei sind wiederum die ersten 128 Zeichen weitgehend identisch mit dem 7-Bit-ASCII Zeichensatz.

6.2 Format von Java-Programmen

Java-Quelltexte sind nicht formatgebunden; insbesondere gibt es keine verbindliche Zeilenstruktur. Als Trennzeichen werden außerhalb von Zeichenketten (Strings) Zeilenenden, Zeilenvorschub, Leerzeichen, Tabulatoren und Kommentare behandelt, wobei eine beliebige Folge von Trennzeichen äquivalent zu einem Trennzeichen ist.

Auch wenn somit erlaubt ist, z. B. den gesamten Quelltext in eine Zeile zu schreiben, so sollte man im Sinne einer besseren Übersichtlichkeit den Programmtext sinnvoll strukturieren (z.B. durch Einrückungen), so dass man die Struktur mit einem Blick überschauen kann.

6.3 Kommentare

Kommentare dienen zur besseren Lesbarkeit und Verständlichkeit eines Programmtextes; sie werden beim Übersetzen ignoriert.

In Java gibt es folgende drei Schreibweisen für Kommentare :

// Kommentar */* Kommentar */* */** doc-Kommentar */*

Zu // *Kommentar* :
Alle dem Kommentarzeichen "//" folgenden Zeichen der aktuellen Zeile sind Kommentar; der Kommentar endet mit dem Zeilenende.

Zu /* *Kommentar* ***/ :**
Alle zwischen den Kommentarzeichen "/*" und "*/" stehenden Zeichen sind Kommentar; der Kommentar kann sich über mehrere Zeilen erstrecken. Eine Schachtelung von Kommentaren dieser Art ist nicht erlaubt; Kommentare der ersten Art dürfen aber enthalten sein.

Zu /** doc-*Kommentar* ***/ :**
Dieser Kommentar stellt eine Anmerkung für das *javadoc*-Programm des JDK dar, das automatisch Dokumentationen erstellen kann.

6.4 Operatoren

Java kennt über 40 Operatoren. Eine Liste mit Detailangaben folgt im nächsten Paragraphen.

6.5 Schlüsselworte

Schlüsselworte sind reservierte Worte der Sprache. Sie stellen den Grundwortschatz dar und dürfen nicht für eigene Bezeichner verwendet werden. Die Kleinschreibung ist signifikant.
Java kennt folgende Schlüsselworte (*kursiv* geschriebene Worte sind zwar reserviert, werden aber z.Z. noch nicht benutzt) :

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

7 Einfache Datentypen, Variable und Konstanten

7.1 Datentypen

Ein **Datentyp** wird beschrieben durch

- die Menge seiner *Werte*
  und
- durch die auf ihm definierten *Operationen* und *Relationen*.

Beispiel : integer

Wertemenge :	ganze Zahlen
Operationen :	Addition, Subtraktion, Multiplikation, ganzzahlige Division
Relationen :	kleiner, kleiner gleich, gleich, ungleich, größer, größer gleich

Java-Programme sollen unabhängig von der jeweiligen Plattform immer gleich ablaufen. Dieses Ziel hatten auch die meisten anderen Programmiersprachen, aber es wurde nie erreicht : schon die Darstellung der Basisdatentypen (ganze Zahlen, reelle Zahlen, ...) ist in allen anderen höheren Programmiersprachen wie z.B. Pascal, C, C++,... maschinenabhängig. Java legt dagegen bereits in der Sprachdefinition die Darstellung und die Initialwerte für die verschiedenen Basisdatentypen fest. Im einzelnen kennt Java die folgenden Basisdatentypen :

Datentyp	**Initialwert**	**Beschreibung**
`boolean`	`false`	Wahrheitswerte
`char`	`\u0000`	16-Bit Zeichen des Unicode-Zeichensatzes
`byte`	`0`	8-Bit Ganzzahl
`short`	`0`	16-Bit Ganzzahl
`int`	`0`	32-Bit Ganzzahl
`long`	`0`	64-Bit Ganzzahl
`float`	`0.0`	32-Bit Gleitkommazahl
`double`	`0.0`	64-Bit Gleitkommazahl

Zahlen haben grundsätzlich ein Vorzeichen.

Hier soll zunächst die Beschreibung der jeweiligen Wertemengen erfolgen; die Operationen und Relationen werden später erläutert.

boolean

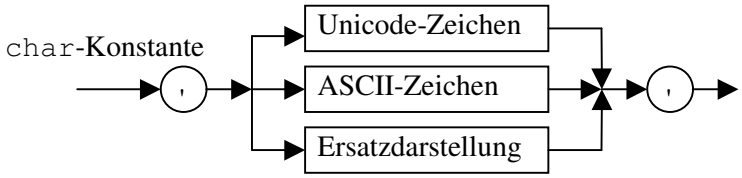
Wertebereich : `false, true`

char

Wertebereich : `\u0000 ... \uffff` (Zeichen des Unicode-Zeichensatzes)
Mit `'\uxxxx'` wird dabei das Zeichen des Unicode mit dem Hexadezimalwert xxxx beschrieben. Zeichen des ASCII-Codes kann man einfacher dadurch angeben, indem man das Zeichen direkt mit umgebenden Apostrophen angibt. Für einige Zeichen (speziell für Steuerzeichen) gibt es vereinfachte Ersatzdarstellungen :

Ersatzdarstellung	**Zeichen**	**Wert**	Beschreibung
\b	BS	\u0008	Backspace
\t	HAT	`\u0009`	horizontaler Tabulator
\n	LF	`\u000a`	neue Zeile (linefeed)
\f	FF	`\u000c`	Neue Seite (form feed)
\r	CR	`\u000d`	Wagenrücklauf (carriage return)
\"	"	`\u0022`	doppeltes Anführungszeichen
\'	'	`\u0027`	Apostroph
\\	\	`\u005c`	Backslash
\xxx			Zeichen mit dem oktalen ASCII-Wert xxx (0000 ≤ xxx ≤ 0377)

Konstante :



Beispiele : `'a'  '\n'  'ä'  '\u0084'  '\204'`
Die drei letzten Konstanten stellen dasselbe Zeichen dar.

Bemerkungen: Der Unicode-Zeichensatz ist ein normierter plattformunabhängiger Zeichensatz, der neben den üblichen Zeichen auch Sonderzeichen und nationale Zeichensätze wie z.B. Kyrillisch, Chinesisch und Japanisch enthält. Prinzipiell werden alle diese Zeichen zwar von Java unterstützt, aber keineswegs von allen Plattformen. Aus diesem Grund sollte man sich möglichst auf die ersten 128 Zeichen beschränken, die mit dem 7-Bit ASCII-Code identisch sind (bereits die folgenden Zeichen des 8-Bit-ASCII-Codes sind nicht mehr normiert; sie werden z.B. unter DOS und Windows unterschiedlich dargestellt).

Zeichenkonstante, Strings

Zeichenkonstante sind Folgen von Zeichen der zuvor definierten Art, die mit Doppelapostrophen geklammert werden. Ersatzdarstellungen dürfen enthalten sein.

Beispiele : "Kaiserslautern" "Hallo, \n wie geht es ?"

Strings sind zwar keine Grunddatentypen, aber in Abweichung vom sonstigen Java-Schema werden Strings vom Compiler in einer Reihe von Punkten wie Basisdatentypen behandelt; insbesondere ist für Strings der Operator "+" (Konkatenation) definiert..

byte, short, int, long

Java kennt die genannten 4 ganzzahligen Datentypen. Bei der Deklaration einer ganzzahligen Variablen kann dieser ein Anfangswert mitgegeben werden; wird darauf verzichtet werden die entsprechenden Variablen jeweils mit dem Wert "0" initialisiert. Die wesentlichen Charakteristika wie z. B. Speicherbedarf und Zahlbereich ist in folgender Tabelle dargestellt :

Typ	**Speicherbedarf (in Bit)**	**Zahlbereich**	
byte	8	-128 .. 127	$-2^7 .. 2^7-1$
short	16	-32 768 .. 32 767	$-2^{15} .. 2^{15}-1$
int	32	-2 147 438 648 .. 2 147 438 647	$-2^{31} .. 2^{31}-1$
long	64	$-92 * 10^{17} .. 92 * 10^{17}$	$-2^{63} .. 2^{63}-1$

Ganzzahlige Konstanten können dezimal, hexadezimal oder oktal geschrieben werden; long-Konstanten werden mit einem angehängten `L` oder `l` gekennzeichnet (`L` ist wegen der Verwechslungsgefahr mit der Ziffer `1` vorzuziehen). Die Syntax der ganzzahligen Konstanten wurde bereits früher vorgestellt.

Beispiele :

```
1999          03715            0x7CD            0X007CD
-4711         03777776631      0xfffed99        0XFFffEd99
1234567890L   011145401322L    0x499602D21L
```

(die Zahlen einer Zeile haben jeweils denselben Wert)

float, double

Diese beiden Datentypen sind die in Java verfügbaren reellen Datentypen. Wie bei der Deklaration einer ganzzahligen Variablen kann diesen ein Anfangswert mitgegeben werden; wird darauf verzichtet werden die entsprechenden Variablen jeweils mit dem Wert "0.0" initialisiert. Die wesentlichen Charakteristika wie z. B.

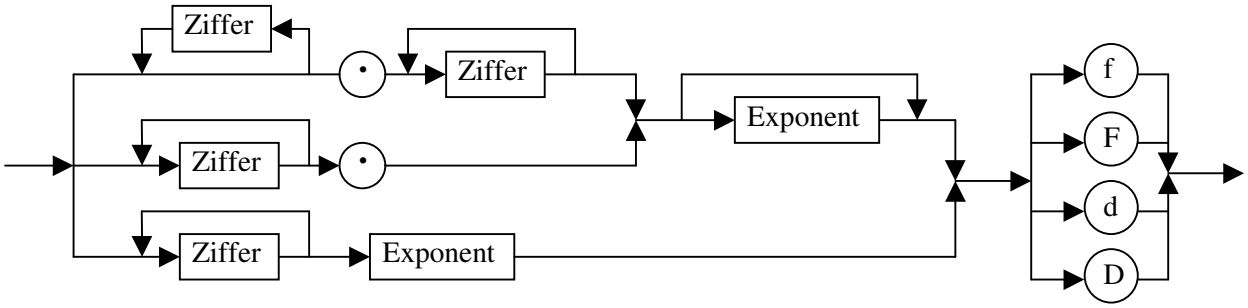
Speicherbedarf und Zahlbereich ist in folgender Tabelle dargestellt. Dabei ist zu beachten, dass diese beiden Zahltypen im positiven und negativen Bereich jeweils symmetrisch sind; angegeben sind deshalb jeweils die kleinste und die größte darstellbare positive Zahl.

Typ	Speicherbedarf (in Bit)	kleinste pos. Zahl	größte pos. Zahl
float	32	$1.4 * 10^{-45}$	$3.4 * 10^{38}$
double	64	$4.9 * 10^{-324}$	$1.8 * 10^{308}$

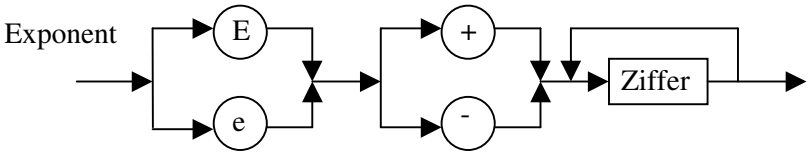
Die interne Darstellung dieser Zahlen erfolgt in dem im *IEEE ANSI/IEEE Standard 754-1985* festgelegten Format. Insbesondere ist die Darstellung durch die Programmiersprache und nicht wie bei fast allen anderen Programmiersprachen durch die jeweilige Plattform festgelegt. Neben den üblichen Zahlen sind für diese beiden Zahltypen noch folgende spezielle Werte definiert :

- positive und negative Null
- positiv und negativ unendlich
- NaN (not a number)

vorzeichenlose reelle Konstante :



mit



7.2 Variable und Variablendeklarationen

In jeder höheren Programmiersprache gibt es den Begriff der Variablen. Variablennamen sind symbolische Namen für Speicherbereiche. Eine Variable wird durch 3 Angaben festgelegt :

Name : `summe` **Datentyp** : `int` **Wert** : `125`

Der **Name** ist dabei ein Bezeichner und identifiziert die Speicheradresse. Der **Datentyp** legt den Wertebereich fest, in dem die möglichen Werte der Variablen liegen. Der **Wert** der Variablen ist der momentane Inhalt des zugehörigen Speicherbereichs.

Variable müssen grundsätzlich vor ihrer ersten Verwendung deklariert werden. Bei der Deklaration werden immer Name und Datentyp festgelegt; ein Anfangswert kann (muss aber nicht !) angegeben werden. In Java gibt es eine Reihe verschiedener Möglichkeiten der Deklaration; das folgende Beispiel zeigt die wohl einfachste Version :

```
public class DeklarationsDemo
{ public static void main ( String[] args)
    {  int gz = 100;    // ganze Zahl mit Initialisierung
```

```
        int sum ;         // ganze Zahl ohne Initialisierung
        double d ;        // Gleitkommazahl ohne Initialisierung
        float f = 123.34e6f;  // Gleitkommazahl mit Initialisierung
        sum = 25 + gz;
        String s = "Zahlenwerte";
        System.out.println(s);
        System.out.println("Ganzzahl gz  : " + gz);
        System.out.println("Ganzzahl sum : " + sum);
        System.out.println("reelle Zahl d : " + d);
        System.out.println("reelle Zahl f : " + f);
    }
}
```

7.3 Benannte Konstanten

Neben Variablen können auch Konstanten benannt werden, deren Name dann im Programm anstelle des jeweiligen Wertes benutzt wird. Die Deklaration einer benannten Konstanten ist ähnlich wie die Deklaration einer Variablen; es wird aber zusätzlich durch den *Modifizierer* **`final`** festgelegt, dass der Wert konstant ist und somit nie geändert werden kann.

Beispiel : final float pi = 3.141592654;

8 Ausdrücke, Wertzuweisungen

Der entscheidende Teil eines Programms, nämlich Rechnungen und Veränderungen von Variablen, wird i.w. durch Ausführung von arithmetischen und logischen Operationen vorgenommen, die in formelähnlichen Konstruktionen - Ausdrücken - symbolisiert sind.

Beispiele :
1. $x * 3 + 5 * (2 * x - 1)$ (mit x: Variable eines Zahltyps)
2. $(3 * a < 5 + b) \ \& \ (b > 0)$ (mit a, b : Variable eines Zahltyps)

Man unterscheidet hierbei

- **arithmetische Ausdrücke** (Wert von einem Zahltyp; Beispiel 1)
- **logische Ausdrücke** (Wert vom Typ `boolean`; Beispiel 2)
  Logische Ausdrücke werden in der Regel zur Steuerung des Programmablaufs mittels Verzweigungen und Schleifen benötigt.
- sonstige Ausdrücke : Ausdrücke, deren Wert weder eine Zahl noch ein Wahrheitswert ist

Der Aufbau von Ausdrücken entspricht weitgehend der üblichen mathematischen Schreibweise; sie sind aufgebaut aus **Operatoren, Operanden** und Klammern. Dabei wird durch die Operatorprioritäten festgelegt, in welcher Reihenfolge die Operatoren auf die Operanden anzuwenden sind. Bei Operatoren gleiche Priorität muss zudem noch festgelegt werden, ob sie von rechts nach links oder umgekehrt ausgeführt werden.

8.1 Operanden, Operatoren und Prioritäten

Operanden : Objekte, auf die die Operatoren angewendet werden können (Zahlen, logische Werte,....)

Operatoren (in Java) :
In der folgenden Tabelle sind die in Java verfügbaren Operatoren mit Prioritäten, Stelligkeit und

Auswertungsrichtung aufgelistet. Im Verlauf dieser Veranstaltung werden diese Operatoren nur zum Teil benötigt und besprochen.

Priorität	**Operator**	**Stelligkeit**	**Zeichen**	**Auswertungs-richtung**

13	logische Negation
	Bitkomplement
	positives Vorzeichen
	negatives Vorzeichen
	Inkrement
	Dekrement
	cast-Operator
		1	`!`
			`~`
			`+`
			`-`
			`++`
			`--`
			`(typ)`
				⇐

12	Multiplikation
	Division
	Rest
		2	`*`
			`/`
			`%`
				⇒

11	Addition Subtraktion	2	`+` `-`	⇒

10	Linksshift
	Rechtsshift mit Vorzeichen
	Rechtsshift mit Nullen
		2	`<<`
			`>>`
			`>>>`
				⇒

9	Vergleiche (Zahlen) Typvergleich	2	`<,>,<=,>=` `instanceof`	⇒ ⇐
8	Gleichheit Ungleichheit	2	`==` `!=`	⇒
7	bitweises UND logisches UND	2	`&`	⇒
6	bitweises exlusives ODER logisches exklusives ODER	2	`^`	⇒
5	bitweises ODER logisches ODER	2	`|`	⇒
4	bedingt UND	2	`&&`	⇒
3	bedingt ODER	2	`||`	⇒
2	bedingter Ausdruck	3	`? :`	⇐
1	Zuweisung Zuweisung mit Operation (Die durch die Operatoren gegebenen Prioritäten können durch Klammer-setzung überspielt werden. Hiervon sollte man weitgehend Gebrauch machen, da dies die Fehleranfälligkeit beim Schreiben von Ausdrücken deutlich reduziert. op ∈ {*,%,/,+,-,&,^,|,<<,>>,>>> })	2	`=` `op=`	⇐

Interne Typkonvertierungen

Die meisten 2-stelligen Operatoren verlangen, dass beide Operanden vom selben Typ sind. Dies ist in der Regel gerechtfertigt, aber nicht bei arithmetischen Operation auf Zahlen. Diese Situation wird dadurch berücksichtigt, dass der Java-Compiler in diesen Fällen ohne Zutun des Benutzers eine automatische Typkonvertierung durchführt, wobei Zahloperanden jeweils auf den komplizierteren der beiden beteiligten Datentypen und ganzzahlige Operanden mindestens auf den Typ `int` gewandelt werden.

Arithmetische Operatoren (`++,--,+,-,*,/,%`)
Neben den 4 Grundrechenarten **`+,-,*,/`** kennt Java noch die modulo-Operation (Teilungsrest) **`%`**, die hier sowohl für ganzzahlige wie auch reelle Operanden definiert ist. Außerdem gibt es die Vorzeichen **`+`** und **`-`** sowie den Operator **`+`** zum Aneinanderhängen von Strings und die Inkrement-bzw. Dekrementoperatoren **`++`** und **`--`**.

Während die Operatoren **`+,-,*`** wie gewohnt arbeiten, sind bei den Divisionsoperatoren **`/,%`** zu beachten :

`/` Bei ganzzahligen Operanden wird der ganzzahlige Quotient ermittelt; bei reellen Operanden wird wie üblich dividiert.

`%` Bei ganzzahligen Operanden liefert **`Zähler % Nenner`** den ganzzahligen Teilungsrest; bei reellen Operanden wird der folgendermaßen festgelegte Rest geliefert :

`Rest = Zähler - Anzahl * Nenner`

wobei **`Anzahl`** ganzzahlig und **`Rest < Nenner`** ist.

Bei beiden Operatoren muss der Nenner ≠ 0 sein. Ist der Nenner = 0, so wird bei ganzzahligen Operanden das Programm mit einer Fehlermeldung abgebrochen; bei reellen Operanden ist das Ergebnis unendlich (**`Infinity`**).

Bei gemischt ganzzahligen und reellen Operanden wird der ganzzahlige Operand zuerst in den entsprechenden reellen Operanden gewandelt und dann die Operation gemäß den Vorgaben für reelle Operanden ausgeführt.

Mit den Inkrement- bzw. Dekrementoperatoren wird der Wert einer Zahl um 1 erhöht bzw. erniedrigt. Diese Operatoren können jeweils in Präfix als auch in Postfixnotation verwendet werden. Bei der Verwendung als Präfix-Operator wird das Objekt vor der Verwendung im Ausdruck geändert, ansonsten nach der Verwendung.

Beispiel :
```
int x=1, y;
y = ++x + 1;      // x = x + 1; y = x + 1    ( x=2, y=3 )
y = x++ + 1;      // y = x + 1; x = x + 1    ( x=3, y=3 )
x = x++ + 2;      // x = x + 2               ( x=5)
x = --x + 2;      // x = x – 1; x = x + 2    ( x=6)
```

Logische und bitweise Operatoren (`&, &&,|, ||,^, ~,!`)
Für einzelne Bits sind die logischen Operatoren wie in der folgenden Tabelle definiert. Interpretiert man dabei **`1`** als **`true`** und **`0`** als **`false`**, so werden hierdurch die logischen Verknüpfungen auf den Typ **`boolean`** gegeben. Die bitweisen Operatoren wirken auf die Bits von Bitfolgen jeweils einzeln. Dabei müssen die Operanden jeweils ganzzahlig sein; das Ergebnis ist jeweils mindestens vom Typ **`int`**.

A	B	not A	A and B	A or B	A xor B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Bemerkung :

Bei **&** und **|** werden immer beide Operanden ausgewertet; bei **&&** bzw. **||** wird der 2. Operand nicht mehr ausgewertet, wenn das Ergebnis bereits nach der Auswertung des ersten Operanden feststeht, d.h. wenn der 1. Operand **false** (bei **&&**) bzw. **true** (bei **||**) ist.

Vergleiche

Arithmetische Vergleiche (<, <=, >, >=)

Mit diesen arithmetischen Vergleichen können nur Zahlgrößen verglichen werden. Bei ungleichen Operandentypen erfolgt wie bei den arithmetischen Operatoren eine interne Typkonvertierung vor dem Vergleich.

Vergleiche auf Gleichheit (==, !=)

Hiermit können Operanden des gleichen Basisdatentyps und Strings verglichen werden.

Vergleich instanceof

Hiermit kann festgestellt werden, ob ein Objekt zu einer bestimmten Klasse gehört.

Bedingter Ausdruck (? :)

Sind **a1** und **a2** Ausdrücke desselben Typs und ist **bool** ein logischer Ausdruck, so hat der Ausdruck

```
bool ? a1 : a2
```

den Wert **a1**, falls **bool**=**true** ist; ansonsten ist der Wert **a2**.

8.2 Wertzuweisung

Die Wertzuweisung erfolgt über den Operator =. Man sollte beachten, dass dieser Operator von rechts nach links ausgewertet wird; d.h.

```
a = b = c  ≙  a = ( b = c )  ≙  b = c ; a = b
```

Neben diesem einfachen Wertzuweisungsoperator sind einige Abkürzungen definiert, die immer dann eingesetzt werden können, wenn der Parameter auf der rechten Seite der Zuweisung dem Ergebnisobjekt entspricht.

Kurzform	x += y	x -= y	x *= y	x /= y	....und entsprechend für
Langform	x = x + y	x = x – y	x = x * y	x = x / y	die weiteren Formen

Regeln, Wirkung:

Die linke Seite einer Wertzuweisung besteht in der Regel aus einer Variablen, die eine Speicheradresse bezeichnet. Die rechte Seite besteht aus einem Ausdruck, der ausgewertet wird.

Stimmen bei der Zuweisung **E1 = E2** die Datentypen von **E1** und **E2** nicht überein, wird sofern möglich der Datentyp von **E2** "nach oben" in den Datentyp von **E1** erweitert (z.B. **int** ⇒ **double**). Ist dies nicht möglich, so ergibt sich ein Fehler.
Ist der Datentyp von **E2** allgemeiner als der Datentyp von **E1**, so muss der Datentyp von **E2** explizit gewandelt werden.

Beispiel :

falsch:
```
int a;
double b, c ;
a = b * c;
```

richtig:
```
int a;
double b, c;
a = (int) b * c;
```

Warnung : während bei der Konvertierung von "einfachen" zu komplizierteren Typen keine Datenverluste auftreten, sind diese im umgekehrten Fall nicht ausgeschlossen !

Die Zuweisung liefert einen Wert, der weiter verwendet werden darf. Deshalb sind Konstrukte wie z.B.

```
a = b = c + d    oder    a =  b + ( c =  d + e )
```

prinzipiell möglich.

9 Kontrollstrukturen

Kontrollstrukturen sind allgemein Sprachkonstrukte, mit deren Hilfe die Reihenfolge der Ausführung von Anweisungen gesteuert werden kann. Dies sind

- Verzweigungen, Fallunterscheidungen
- Wiederholungsanweisungen (Schleifen)
- Sprunganweisungen

Sprunganweisungen sind im Sinne der strukturierten Programmierung möglichst zu vermeiden, da sie in der Regel zu nicht klar strukturierten und damit unübersichtlichen und fehleranfälligen Programmen führen. Mit aus diesem Grund stehen in Java Sprunganweisungen nur in sehr eingeschränkter Form zur Verfügung; mit ihnen kann man i.w. Anweisungsblöcke vorzeitig in kontrollierter Form verlassen.

Im Zusammenhang mit Kontrollstrukturen haben Blöcke, die eine Anweisungsfolge zu einer (Verbund-) Anweisung zusammenfassen, eine erhebliche Bedeutung :

9.1 Blöcke

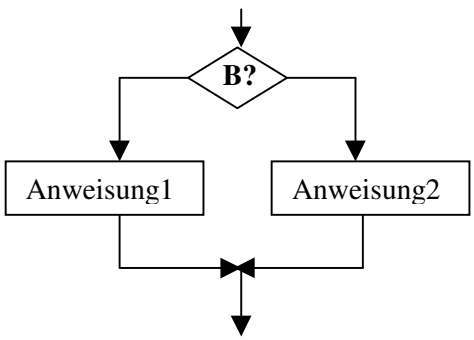
Ein Block enthält eine Folge von Anweisungen, die nacheinander ausgeführt werden. Ein Block kann (z.B. innerhalb von Kontrollstrukturen) als eine Anweisung angesprochen werden. Die Klammerung der Anweisungen erfolgt durch die Klammersymbole "**{**" und "**}**".

Beispiel :

```
{// Anfang von Block1
   Anweisung1;
   Anweisung2;
   {  // Anfang von Block2
      Anweisung3;
      Anweisung4;
      .....;
   }  // Ende von Block2
   Anweisung5;
   .....;
} // Ende von Block1
```

Blöcke können wie im Beispiel zu sehen geschachtelt werden. Werden in einem Block Objekte definiert, so sind sie lokal für diesen Block; d.h. sie sind nur in diesem Block und in seinen Unterblöcken bekannt.

9.2 Verzweigungen



if-Anweisung

Java : if (B)
Anweisung1
else
Anweisung2

mit : B : boolescher Ausdruck
Anweisung1, Anweisung2 : Anweisung oder
Anweisungsblock

Der else-Zweig kann entfallen.

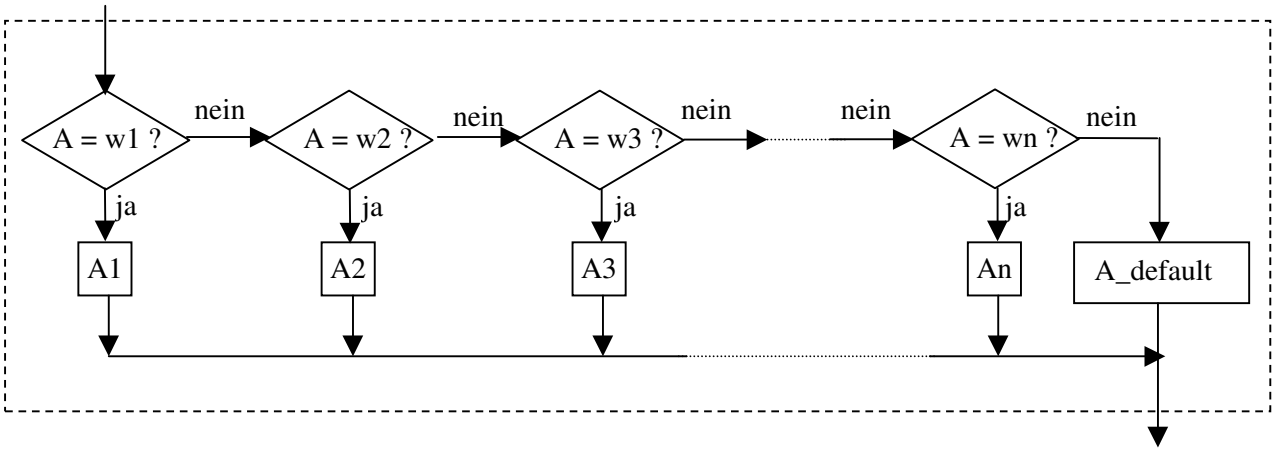
In der vorangehenden Beschreibung darf Anweisung1 bzw. Anweisung wieder eine if-Anweisung sein; d.h. eine Schachtelung ist erlaubt. Dabei ist zu beachten, dass sich jeder else-Zweig auf den letzten vorhergehenden noch nicht mit else abgeschlossenenen if-Zweig bezieht.

Beispiele :

```
int a, b;
....
if   (a <= b)                    // Anfang der 1. if-Anweisung
     { a += b - 2;
       b = b * a % b;
     }
else                             // else-Zweig der 1. if-Anweisung
     a = 2 * a;

if   ( b > 10)                   // Anfang der 2. if-Anweisung
     { b -= 10;
       a = b + 2 * a;
     }                           // Ende der 2. if-Anweisung; kein else
if   ( a != 0 )                  // Beginn der 3. if-Anweisung
     { if ( b / a > 3 )          // Beginn der 4. if-Anweisung
          { a = b - 3;
            System.out.println("a = " + a );
          }                      // Ende der 4. if-Anweisung; kein else

      }
else                             // else-Zweig der 3. if-Anweisung
     if ( b == 0 )               // Beginn der 5. if-Anweisung
          System.out.println("a und b sind beide Null" );
     else                        // else-Zweig der 5. if-Anweisung
          System.out.println("b = " + b );  // Ende !!
```

switch-Anweisung

Die switch-Anweisung realisiert die Fallunterscheidung mit Fehlerausgang :



```
switch ( A ) {     case Fall1:      Anweisung1;
                                    break;
                   case Fall2:      Anweisung2;
                                    break;
                                    ......;
                   default:         defaultAnweisung;
                   }
```

mit : **A** : Ausdruck von einem der folgenden Datentypen : **byte, char, short, int, long**
Fall1, Fall2 ,...: **Konstanten** des durch den Auswahlausdruck **A** gegebenen Datentyps
Anweisung1, Anweisung2 : Anweisung oder Anweisungsblock

Es wird zu dem case-Label gesprungen, das als Konstante den Wert des Ausdrucks hat. Falls keine der Konstanten mit dem Wert des Ausdrucks übereinstimmt, wird zu **default:** gesprungen. Die (noch zu besprechende) **break**-Anweisung dient dazu, den Anweisungsblock abzuschließen und an das Ende der **switch**-Anweisung zu springen. Fehlt die **break**-Anweisung, so werden auch die Anweisungen der noch folgenden case-Labels ausgeführt; es erfolgt keine erneute Überprüfung der Bedingungen. Dies ist in der Regel unerwünscht.

Beispiel :

```
int a, b;
....
switch (a) {
   case 1:
   case 3:
          b = 1;    // kein break
                    // deshalb weiter mit dem nächsten Zweig
   case 4:
          b = 2;
          break;    // Ende der Zweige mit 1:, 3:, 4:
   case 6:
          b = 5;
          break;    // Ende des Zweigs mit 6:
   default :
          b = 0;
}
```

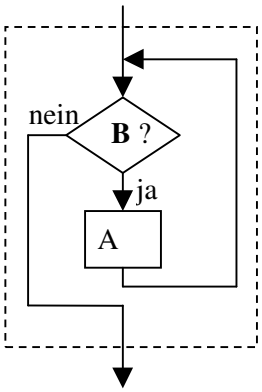
9.3 Schleifen (Wiederholungsanweisungen)

a) abweisende Schleife (while)

Java :

```
while ( B )
Anweisung;
```

mit : ***B*** : boolescher Ausdruck

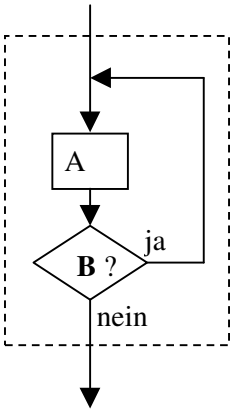


b) nicht abweisende Schleife (do-while)

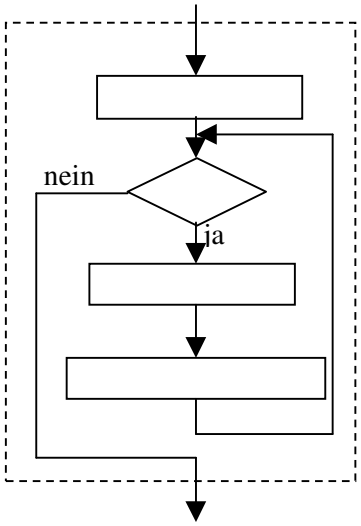
Java :

```
do
    Anweisung;
while ( B );
```

mit : ***B*** : boolescher Ausdruck



c) Zählschleife (for)



Java :

```
for (Init; B; Incr )
Anweisung;
```

mit : ***Init :*** Initialisierung
B : boolescher Ausdruck
Incr : Inkrementanweisung

Diese Zählschleife ist zu folgender while-Schleife äquivalent :

```
Init;
while ( B ){
                Anweisung;
                Incr;
              }
```

Bemerkungen :

- Durch die Anweisung ***Init*** wird der Zählvariablen ein Anfangswert zugewiesen.
- Der boolesche Ausdruck ***B*** ist die Durchlaufbedingung für diese Schleife. Wird der Wert **false** erreicht, so wird die Schleife beendet. Die erstmalige Überprüfung findet vor dem ersten Schleifendurchlauf statt.
- Die Inkrementanweisung ***Incr*** dient zur Veränderung (Erhöhung oder Erniedrigung) des Schleifenzählers.
- Alle Ausdrücke im Schleifenkopf sind optional. Werden sie weggelassen, so liegt eine Endlosschleife vor : **for (; ;)** **Anweisung ;**

9.4 Beispiele :

a) Aufsummieren der Zahlen 1 ... n und Ausgabe der Zwischenergebnisse

(i) while-Schleife

```
int i, summme, n=100;
...;
//while-Schleife
i = 1;
summe = 0;
while ( i <= n ) {
     summe += i;
     System.out.println(summe);
     i = ++i;
}
```

(ii) do-while-Schleife

```
int i, summme, n=100;
...;
i = 1;
summe = 0;
do {
     summe += i;
     System.out.println(summe);
}  while ( i++ < n )
```

(iii) for-Schleife

```
int i, summme, n=100;
...;
summe = 0;
for ( i = 1; i <= 100; i++) {
     summe += i;
     System.out.println(summe);
}
```

9.5 Marken und Sprunganweisungen

Marken

Jede Anweisung kann durch eine Marke mit einem Namen versehen werden.

```
Marke : Anweisung
```

Marken werden hauptsächlich im Zusammenhang mit der break- bzw. continue-Anweisung verwendet, um Blöcke zu verlassen bzw. zu ihrem Anfang zurückzugehen.

break

Die break-Anweisung dient zum Verlassen eines beliebigen Blocks.

Beispiel :

```
while ( B1 ) {
     ...;      // Anweisung(en) 1
       if (B2)
           break;   //Schleife verlassen
     ...;      // Anweisung(en) 2
     }
```

Hat **B2** den Wert **true**, so werden die hinter break stehenden Anweisungen übersprungen; die Schleife wird sofort verlassen.

Bei geschachtelten Blöcken wird durch break immer der innerste Block verlassen, sofern dies nicht durch Angabe einer Marke überspielt wird.

Beispiel :

```
Block1:
     while ( B1 ) {
          ...;               // Anweisung(en) 1
          if (B2)
              break;         //Schleife verlassen; identisch mit
                             // break Block1
          ...;               // Anweisung(en) 2
        Block2:
          while ( B3 ) {
               ...;          // Anweisung(en) 3
               if (B4)
                  break;     //innere Schleife verlassen; identisch
                             // mit break Block2
               if (B5)
                  break Block1;   //äußere Schleife verlassen
               ...;          // Anweisung(en) 4
          }
        }
```

continue
Die **`continue`**-Anweisung dient zum Beenden eines Schleifendurchlaufs. Durch diese Anweisung wird die Schleife als ganzes nicht beendet; es werden lediglich die hinter continue stehenden Anweisungen des Schleifenrumpfes übersprungen und es wird der nächste Schleifendurchlauf begonnen.

Beispiel :
```
int summe=0;
....;
for ( int i = 1, i <= 100, i++ ) {
   summe += i;
   if ( i % 10 != 0 )
       continue;
   System.out.println(summe);
}
```

Hierdurch wird die Summe nur alle 10 Schleifendurchläufe ausgegeben.

return
Die **`return`**-Anweisung dient zum Beenden von Methoden und zum Zurückliefern eines Rückgabewertes.

10 Klassen, Objekte und Methoden

Die bisherigen Beispiele von Java-Applikationen waren Klassen, die neben wenigen Daten i.w. eine einzige Methode, nämlich die Methode `main`, enthalten haben.

Allgemeiner stellen ***Klassen*** die Beschreibung von Gegenständen (***Objekten***) gleicher Struktur dar; sie enthalten ***Attribute*** (Datenkomponenten) und ***Methoden*** (Beschreibung der Aktionen auf den Attributen).

10.1 Beispiel

Verbale Beschreibung der Klasse **`Fahrzeug:`**

Attribute :

- Besitzer
- Fahrgestellnummer
- Hersteller
- Art des Motors (Benzin, Diesel, Elektro)
- Leistung des Motors
- .....

Diese Daten sind i.w. im Kfz-Schein vermerkt.
Weiter soll das Fahrzeug ein Fahrtenbuch haben, das u.a. folgende Attribute enthält :

- momentaner Standort
- km-Stand
- Tankinhalt
- Ölstand
-

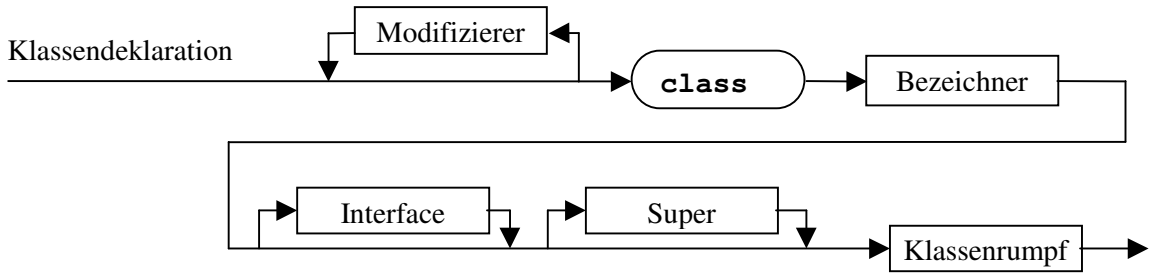
Methoden (Verhalten des Fahrzeugs) :

- fährt vom momentanen Standort nach Ort A
- erhält eine Inspektion
- gibt seine aktuellen Daten getrennt nach Kfz-Schein und Fahrtenbuch aus

Hiermit sind die Attribute und das Verhalten irgendeines Fahrzeugs festgelegt. Die Attribute haben allerdings selbst noch keine spezifizierten Werte, hier ist lediglich ein "Formular" festgelegt, das noch speziell ausgefüllt werden muss. Jedes einzelne Fahrzeug hat ein nach obigem Muster speziell ausgefülltes Formular (nämlich Kfz-Schein und Fahrtenbuch) und kann die angegebenen Aktionen ausführen. Ein solches speziell ausgefülltes Formular ist ein *Objekt* oder eine *Instanz* der Klasse. Objekte sind also nichts anderes als Variable vom Typ der zugehörigen Klasse.

10.2 Klassendeklaration

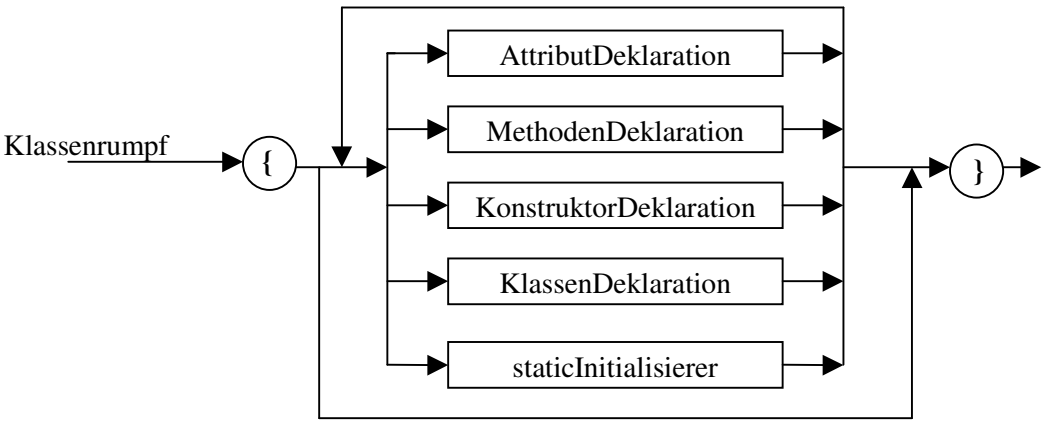
Allgemein ist die Syntax einer Klassendeklaration durch folgendes Diagramm gegeben (einige Teile des folgenden Syntaxdiagramms werden erst später erklärt).



Die (optionalen) Modifizierer legen u.a. Zugriffsrechte fest. In dieser Veranstaltung werden nur einige der möglichen Modifizierer verwendet; die restlichen werden hier nur aus Gründen der Vollständigkeit aufgeführt :

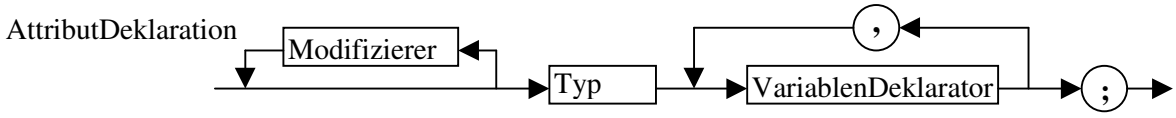
```
abstract  final     native          private      protected
public    static    synchronized    transient    volatile
```

Der Klassenrumpf hat folgende Struktur :



10.3 Attribute

Die Attribute sind die **Datenkomponenten** einer Klasse. Die zugehörigen Datentypen können einfache Datentypen wie z.B. **`int`** oder auch selbst wieder Klassen, Felder oder Strings sein (Bemerkung : Felder und Strings sind genau betrachtet ebenfalls Klassen).



Der VariablenDeklarator ist dabei im einfachsten Falle lediglich der Name einer Variablen; diese können mit Anfangswerten versehen werden.

Beispiel :

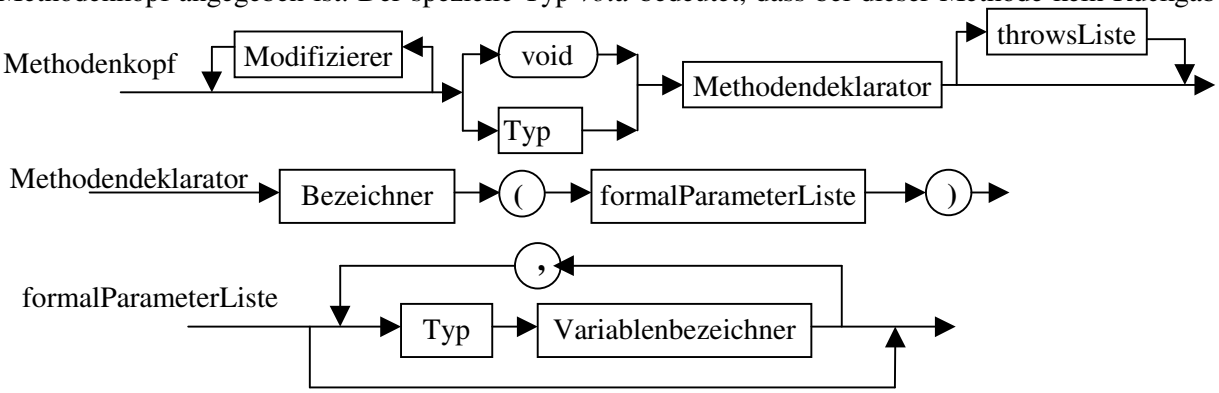
```
public class Fahrzeug
 {// Attribute
    String Besitzer;
    final static byte OTTO=0,DIESEL=1,ELEKTRO=2;
    long Fahrgestellnummer;
    String Hersteller;                    // z.B. "VW"
    byte MotorArt;                        // z.B. OTTO
    int Leistung;                         // in kw
    int LeerGewicht,zulGesamtGewicht;     // in kg
    int AnzahlSitzPlaetze;
    String Standort;
    int kmStand;
    double Tank;                          // in Liter
    boolean Oelstand;                     // ok ?

 // Methoden
    ............
    ............
}
```

10.4 Methoden

Die Methoden beschreiben die Aktionen, die mit den Attributen der Klasse durchgeführt werden können. Sie entsprechen in vielen Punkten den Prozeduren und Funktionen anderer Programmiersprachen.
Jede Methode besteht aus einem Kopf, in dem die Schnittstelle zur Umgebung beschrieben wird, und dem Methodenblock, in dem die Aktionen der Methode in geeigneter Weise aufgeführt sind.

Syntax :

Die Formalparameterliste kann leer sein; die umgrenzenden Klammern müssen aber auch dann geschrieben werden. Für jede Variable muss der Typ getrennt angegeben werden; eine gemeinsame Typangabe ist auch für Variable gleichen Typs nicht vorgesehen. Prinzipiell liefern Methoden Rückgabewerte, deren Typ im Methodenkopf angegeben ist. Der spezielle Typ *void* bedeutet, dass bei dieser Methode kein Rückgabewert



vorgesehen ist.

Beispiele :

1. Methode **faehrtNach**
   Das fragliche Fahrzeug erhält einen neuen Standort und der km-Stand ändert sich dementsprechend. Hierzu muss die Methode sowohl den neuen Standort als auch die Entfernung zwischen dem alten und dem neuen Standort mitgeteilt bekommen. Dies geschieht über die *formalen Parameter* **Ziel** und **Entfernung**. Die Methode soll den aktuellen km-Stand (nach der Standortänderung) als Ergebnis liefern :

```
/*
 *  Vom Standort an anderen Ort fahren
 *  @param  Ziel         Zielort und neuer Standort
 *  @param  Entfernung   Entfernung vom Standort zum Zielort
 *  @return neuer kmStand
 */
public int faehrtNach(String Ziel, int Entfernung)
   {  System.out.println(Besitzer+" faehrt von "+Standort+" nach "
                +Ziel+", Entfernung: "+Entfernung+" km");
      Standort=Ziel;
      kmStand+=Entfernung;
      return kmStand;
   }
```

   Diese Methode ändert die Werte der Attribute **Standort** und **kmStand** entsprechend den in den Parametern angegebenen Werten. Der zurückgegebene Wert wird durch die **return**-Anweisung festgelegt.

2. Methode **Inspektion**
   Durch diese Methode wird ein Ölwechsel vorgenommen; das Attribut **Oelstand** wird auf **true** gesetzt. Ein Rückgabewert ist nicht vorgesehen.

```
void Inspektion()
   {  Oelstand=true;
      // ... weitere Aktionen, falls gewünscht
   }
```

3. Weitere Methoden
   In der Klasse werden noch Methoden zum Ausgeben von Daten definiert, nämlich :
   void KfzScheinAusgeben()
   void FahrtenbuchAusgeben()
   void AlleDatenAusgeben()
   Die Definition und Wirkung dieser Methoden ist offensichtlich.

10.5 Bemerkungen

Der Wert, der durch eine Methode zurückgegeben wird, wird durch eine **return**-Anweisung festgelegt :

return; **// falls eine void-Methode vorliegt**
oder
return *Ausdruck*;

Bei einer **return**-Anweisung wird die Methode an dieser Stelle sofort verlassen; der Wert des Ausdruck ist das Ergebnis der Methode. Eine Methode kann mehrere **return**-Anweisungen enthalten; ist die Anweisung **return;** (ohne Ausdruck !) die letzte Anweisung der Methode, so kann sie weggelassen werden.

10.6 Objekte

Die zuvor definierte Klasse **Fahrzeug** legt kein konkretes Objekt fest, sondern sie ist lediglich die Beschreibung ("Formular") für noch einzuführende Objekte. Jedes konkrete Fahrzeug erhält ein bestimmtes Exemplar eines solchen Formulars, das dann entsprechend den aktuellen Gegebenheiten ausgefüllt werden kann. Von der Programmiertechnik her ist mit der Klasse **Fahrzeug** ein neuer Datentyp eingeführt worden, von dem bei Bedarf analog zur Einführung ganzzahliger oder reellwertiger Variabler neue Variablen eingeführt werden können. Solche Variable von einem Klassentyp heißen Objekte (oder Instanzen) der Klasse.

Erzeugung von Objekten

Neue Objekte können in Java analog zur Deklaration von Variablen der Basisdatentypen durch Angabe eines Klassentyps gefolgt von einem Namen deklariert werden.
Beispiel : Einführung zweier Variabler vom Typ Fahrzeug

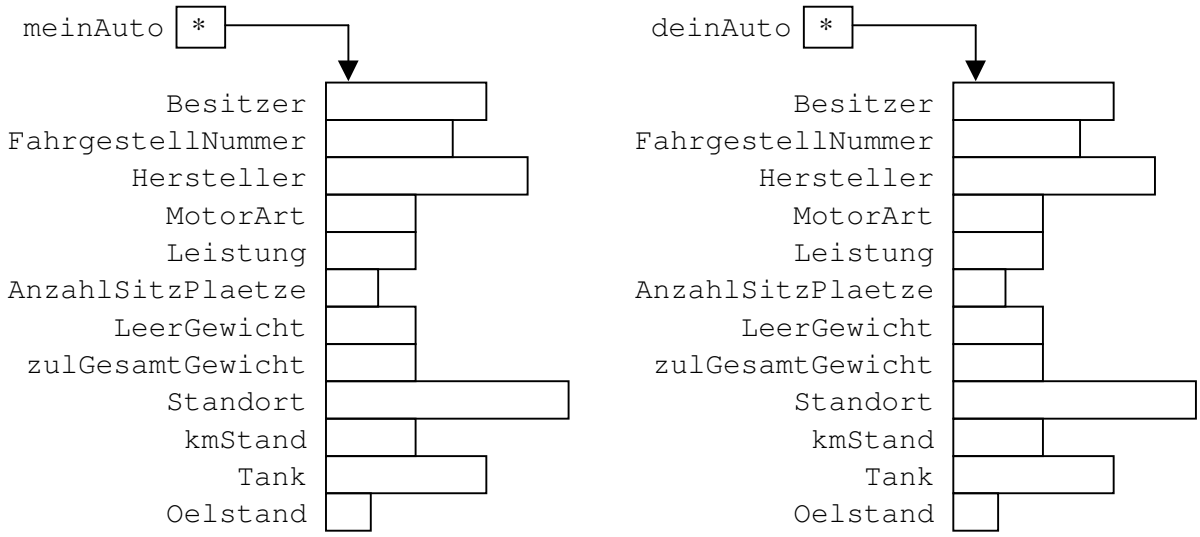
```
Fahrzeug  MeinAuto, DeinAuto;
```

Diese Objekte (**MeinAuto, DeinAuto**) stellen lediglich Referenzen (Adressen !) auf Klassenobjekte diesen Typs dar; hierdurch wird noch kein Speicherplatz für diese Objekte angelegt.
Der Speicherplatz für die Objekte wird dynamisch (d.h. zur Laufzeit bei Bedarf) wie folgt reserviert :

```
MeinAuto = new Fahrzeug();
DeinAuto = new Fahrzeug();
```

Erst nach dieser Reservierung kann auf die einzelnen Attribute bzw. Methoden zugegriffen werden.

Ergebnis der Deklarationen :



Bemerkung : Bei dieser Deklaration wird für jedes neu angelegte Objekt eigener Speicher für die jeweiligen Attribute (Ausnahmen werden noch besprochen) angelegt. Die Methoden dagegen sind für alle Objekte der Klasse gemeinsam.

10.7 Zugriff auf Attribute und Methoden

Zum Test der Klasse **Fahrzeug** benötigt man ein Hauptprogramm **main**, in dem Objekte vom Typ **Fahrzeug** deklariert und bearbeitet werden. Prinzipiell könnte man dieses Hauptprogramm als Methode in **Fahrzeug** selbst deklarieren, aber in der Regel ist es sinnvoller eine neue Applikation zum Test dieser Klasse zu entwerfen. Hier soll dazu die Klasse **FahrzeugTest** dienen.

In der Methode main dieser Klasse werden nun zwei Objekte vom Typ Fahrzeug deklariert, einige Initialisierungen der Attribute durchgeführt und Methoden aufgerufen.

Der Zugriff auf Attribute eines Objekts erfolgt durch Angabe des Objekts und (durch einen Punkt getrennt) des Namens (Beispiel: **meinAuto.kmStand = 12700;)**

```
class FahrzeugTest
{  /*
   * Testumgebung fuer Klasse "Fahrzeug"
   * Hier werden nur einige Attribute initialisiert.
   */
   public static void main(String[] argv)
   {  Fahrzeug meinAuto=new Fahrzeug(),
               deinAuto=new Fahrzeug();
      meinAuto.Besitzer="Bauer";
      meinAuto.kmStand=12345;
      // weitere Attribute initialisieren
      deinAuto.Hersteller="VW";
      deinAuto.Oelstand=true;
      // jetzt koennte man weitere Attribute initialisieren
      //............
      // einige Ausgaben :
      meinAuto.alleDatenAusgeben();
      deinAuto.alleDatenAusgeben();
      // Verwendung von faehrtNach mit Verwendung des Ergebnisses
      System.out.println("Fahrt ist "
            +meinAuto.faehrtNach("Stuttgart",222)
            +" km weit");
      // ... und hier wird das Ergebnis von faehrtNach ignoriert
      deinAuto.faehrtNach("Stuttgart",67);
      meinAuto.Inspektion();           // Oelstand setzen
      meinAuto.FahrtenbuchAusgeben();
      deinAuto.FahrtenbuchAusgeben();
      deinAuto.faehrtNach("Karlsruhe",89);
      deinAuto.FahrtenbuchAusgeben();
   }
}
```

Allgemein ist beim Aufrufen von Methoden zu beachten:

- Die formalen Parameter (d.h. die Parameter bei der Deklaration) und die aktuellen Parameter (d.h. die Parameter beim Aufruf müssen in ihrer Anzahl übereinstimmen und die Typen müssen zueinander passen).
- Ist der Ergebnistyp einer Methode **void**, so bildet der Methodenaufruf eine Anweisung.
- Ist der Ergebnistyp nicht **void**, so sind zwei Aufrufmöglichkeiten vorhanden :
  - Die Methode kann als Anweisung aufgerufen werden; in diesem Fall wird das Ergebnis ignoriert.
  - Die Methode wird als Funktion aufgerufen. Dies kann genau an den Stellen geschehen, an denen auch eine Variable des Ergebnistyps stehen könnte.

Beim Aufruf einer Methode werden folgende Aktionen durchgeführt:

- Die Ausdrücke der aktuellen Parameterliste werden ausgewertet und die formalen Parameter werden mit den so ermittelten Werten initialisiert. Gegebenenfalls werden die Werte entsprechend den Regeln bei der Wertzuweisung konvertiert.
- Es wird an die erste Anweisung des Methodenrumpfs verzweigt und die Anweisungen werden wie gewohnt abgearbeitet, bis eine **return**-Anweisung oder das Ende des Methodenrumpfs erreicht wird.

- Ist der Ergebnistyp der Methode nicht **void**, so muss sie eine **return**-Anweisung enthalten, der ein Ausdruck des Ergebnistyps folgt. Dieser Ausdruck wird ausgewertet und sein Wert als Ergebnis der Methode zurückgereicht.
- Anschließend wird die Programmabarbeitung unmittelbar hinter der Aufrufstelle der Methode fortgesetzt.

10.8 Konstruktoren

Bisher wurden einzelne Attribute eines Objekts durch Wertzuweisungen initialisiert. Besser ist es in der Regel allerdings, solche Initialisierungen bereits bei der Erzeugung einer Instanz zu erledigen.
Eine solche Initialisierung wird durch eine spezielle Methode, dem ***Konstruktor***, beschrieben. Ein Konstruktor besitzt denselben Namen wie seine Klasse; er wird bei der Erzeugung eines Objekts durch **new** ohne Zutun des Benutzers automatisch aufgerufen und ausgeführt. Die Syntax eines Konstruktors entspricht i.w. der Syntax von Methoden, allerdings wird dem Konstruktor kein Ergebnistyp (auch nicht **void** !) zugeordnet. Ein Konstruktor kann wie Methoden Parameter haben. Konstruktoren werden im **new**-Ausdruck aufgerufen, wobei auch die aktuellen Parameter übergeben werden.

Beispiel : Erweiterung der Klasse **Fahrzeug** um einen Konstruktor, der einige Attribute initialisiert.

```
Fahrzeug ( String N, long FgNr, String Herst, byte MA,
           int Lst, int leer, int zul, String StdO,
           int km, boolean Oel)
{  Besitzer = N;
   FahrgestellNummer = FgNr;
   Hersteller = Herst;
   MotorArt = MA;
   Leistung = Lst;
   LeerGewicht = leer;
   zulGesamtGewicht = zul;
   Standort = StdO;
   kmStand = km;
   Oelstand = Oel;
}
```

Eine Deklaration für ein neues Objekt vom Typ **Fahrzeug** könnte dann so aussehen :

```
Fahrzeug nocheins = new Fahrzeug (
             "Donald Duck",            // Besitzer
             121212345,                // Fahrgestellnummer
             "Opel",                   // Hersteller
             DIESEL,                   // Motorart
             60,                       // Leistung
             1100,                     // Leergewicht
             1620,                     // zul. Gesamtgewicht
             "Kaiserslautern",         // Standort
             37845,                    // km-Stand
             true);                    // Oelstand
```

Durch diese Deklaration werden die Attribute wie angegeben initialisiert.

Bemerkungen :

- Auch bei den bisherigen Deklarationen von Objekten wurde jeweils ein Konstruktor aufgerufen. Java generiert nämlich von sich aus für jede Klasse ohne explizit deklarierten Konstruktor ohne Zutun des Benutzers einen parameterlosen Default-Konstruktor mit leerem Rumpf.

- Ein Konstruktor kann genau wie Methoden neben Wertzuweisungen wie im Beispiel auch noch beliebige Anweisungen enthalten.

10.9 Beispiel : Die Klasse Punkt

Ein Punkt in der 2-dimensionalen Ebene wird durch die beiden ganzzahligen Koordinaten x, y beschrieben; in der Regel gibt man ihm zur Identifizierung noch einen Namen.
Ein Punkt soll neben Konstruktoren noch folgende Methoden besitzen :

- **`laenge`** berechnet aus den Koordinaten den Abstand zum Koordinatenursprung
  Formel : $\sqrt{x^2 + y^2}$
- **`ausgeben`** gibt die Koordinaten und den Namen als Text auf dem Bildschirm aus.

Definition der Klasse :

```
public class Punkt
{  int x, y;
   String name;

// Konstruktor
   Punkt ( int a, int b, String n )
   {  x = a; y = b ; name = n; }

// Methoden
   public double laenge()
   { return Math.sqrt(x*x+y*y);
   }

   public void ausgeben()
   {  System.out.println(name + "(" + x + "," + y + ")"); }

}
```

10.10 Überladen von Methoden

In Java ist es erlaubt, verschiedenen Methoden den gleichen Namen zu geben, sofern sie sich in der Zahl oder/und dem Typ der Parameter unterscheiden. Dieses Konzept wird in der Regel eingesetzt, wenn die fraglichen Methoden zwar im Prinzip dieselben Aufgaben erledigen, aber mit unterschiedlichen Parametern arbeiten. Dieses Verfahren heißt ***Überladen*** von Methoden.

Beispiel : Es soll das Maximum von zwei Werten der Typen **`int`**, **`double`**, **`String`** und **`Punkt`** sowie von drei Werten des Typs **`int`** berechnet werden. Dabei soll ein String **`s1`** größer als der String **`s2`** sein, falls **`s1`** lexikographisch hinter **`s2`** steht. Bei Punkten soll die Länge maßgebend sein.

```
public class Ueberladen
{
   //  Maximum zweier int-Zahlen bestimmen
   static int maximum(int a,int b)
   {  return a>b?a:b;  }

   //  Maximum von drei int-Zahlen bestimmen
   static int maximum(int a,int b,int c)
   {  return maximum(maximum(a,b),c);   }

   //  Maximum zweier double-Zahlen bestimmen
   static double maximum(double a,double b)
   {  return a>b?a:b;  }

   // Maximum zweier Strings bestimmen (lexikogr. Ordnung).
   //  a.compareTo(b) liefert >0, wenn a hinter b kommt
   static String maximum(String a,String b)
   {  return (a.compareTo(b)>0)?a:b; }

   //  Maximum zweier Punkte bestimmen,
   // bei gleicher Länge wird der zweite Punkt geliefert

   static Punkt maximum(Punkt p1,Punkt p2)
   {  return (p1.laenge()>p2.laenge())?p1:p2; }

   // Testumgebung
   public static void main(String[] argv)
   {  int i=12,j=17,k=13;
      double f=321.345,g=3.21245e2;
      String s1="Java",s2="Jawa";
      Punkt p1=new Punkt(12,3,"P1"),p2=new Punkt(5,5,"P2");

      System.out.println("Maximum von "+i+" und "+j
            +" ist "+maximum(i,j));
      System.out.println("Maximum von "+f+" und "+g
            +" ist "+maximum(f,g));
      System.out.println("Maximum von \""+s1+"\" und \""+s2
            +"\" ist \""+maximum(s1,s2)+"\"");
      System.out.println("Maximum von "+i+" und "+j
            +" und "+k+" ist "+maximum(i,j,k));
      System.out.print("Maximum von ");p1.ausgeben();
      System.out.print(" und ");p2.ausgeben();
      System.out.print(" ist ");
      maximum(p1,p2).ausgeben();
      System.out.println();
   }
}
```

Genau wie Methoden können auch Konstruktoren überladen werden; eine Klasse kann also mehrere unterschiedliche Konstruktoren besitzen. Dies wird dann benutzt, wenn mehrere unterschiedliche Initialisierungen möglich sein sollen.

Beispiel: Erweiterung der Definition der Klasse Punkt

```
public class Punkt
{  int x, y; String name;
// Konstruktoren
   Punkt ( int a, int b, String n )
   {  x = a; y = b ; name = n; }
   Punkt ( )
   {  x = 0; y = 0 ; name = "unbenannt"; }
   Punkt ( int a, int b )
   {  x = a; y = b ; name = "unbenannt"; }
   Punkt ( String n )
   {  x = 0; y = 0 ; name = n; }
// Methoden wie zuvor
}
```

10.11 Parameterübergabe-Mechanismen

Bei der Übergabe von Parametern an Methoden gibt es prinzipiell zwei Möglichkeiten :

1. **Werteparameter** (*call by value*): Die Methode erhält den **Wert** der Parameter. Dieser wird auf einem lokalen Hilfsspeicherplatz abgelegt, der dann innerhalb der Methode bei jedem Zugriff auf den Parameter benutzt wird. Änderungen dieser Parameter haben außerhalb der Methode keine Auswirkung.
2. **Referenzparameter** (*call by reference*): Die Methode erhält eine **Referenz** auf den Parameter (d.h. die Speicheradresse). Jeder Zugriff auf den Parameter erfolgt über die Adresse auf den realen Speicherplatz des jeweiligen aktuellen Parameters. Somit dringen Änderungen des jeweiligen Parameters sofort nach außen durch).

Im Unterschied zu manchen anderen Programmiersprachen hat man in Java keinen direkten Einfluss auf die Art der Parameterübergabe. Durch die Sprachdefinition wird hier festgelegt :

- Parameter der Grunddatentypen (**char, boolean, byte, short, int, long**,.....) werden immer als Werteparameter übergeben.
- Objekte werden immer als Referenzparameter übergeben; d.h. die Methode erhält als Information die Adresse des Objekts. Wie zuvor beschrieben dringen Änderungen von Attributen des Objekts direkt nach außen durch; die Adresse selbst ist aber ein Werteparameter, dessen Änderung in der Methode außen ohne Wirkung ist. (Hinweis : neben den selbst eingeführten Objekten sind auch Strings und Felder Objekte und keine Grunddatentypen; sie werden also immer als Referenzparameter übergeben).

Beispiel : Tausch zweier Zahlen

Version 1 :

```
public class tausche_falsch
{public static void tausche(long a,long b)
   {  long h=a;
      System.out.println("alte Werte : a="+a+", b="+b);
      a=b;  b=h;
      System.out.println("neue Werte : a="+a+", b="+b);
   }
   /* Testumgebung für tausche
   * zwei long-Zahlen sollen vertauscht werden
   */
    public static void main(String[] argv)
   {  long x=10,y=12;
```

```
      // "falscher" Tausch
         System.out.println("alte Werte : x="+x+", y="+y);
         tausche(x,y);
         System.out.println("neue Werte : x="+x+", y="+y);
      }
   }
```

Das Programm liefert :
```
alte Werte: x=10, y=12
alte Werte: a=10, b=12
neue Werte: a=12, b=10
neue Werte: x=10, y=12
```

Da der Typ der übergebenen Variablen ein Grunddatentyp war, hatte die Änderung innerhalb der Methode keine Auswirkung nach außen. Getauscht wurden nur die auf lokalen Speicherplätzen der Methode abgelegten Werte, auf die von außen nicht zugegriffen werden kann. Sollen die Werte auch für die Außenwelt getauscht werden, muss man sich die Referenzen beschaffen; dies geht nur durch Einbettung der Variablen in Objekte durch Deklaration einer Hilfsklasse.

Version 2 :
```
public class ReferenzParameter
{  public static void tausche(Lang a,Lang b)
   {  long h=a.x;
      a.x=b.x;  b.x=h;
   }
   // Testumgebung für Referenzparameter
   public static void main(String[] argv)
   {  long x=10,y=12;
      System.out.println("alte Werte : x="+x+", y="+y);
      Lang xx=new Lang(x);
      Lang yy=new Lang(y);
      tausche(xx,yy);
      x=xx.x;  y=yy.x;
      System.out.println("neue Werte : x="+x+", y="+y);
   }
}
/*
Hilfsklasse für "call by reference";
lediglich "Klassenkapsel" um primitiven Typ gelegt.
*/
class Lang
{  long x;
   // Default-Konstruktor
   public Lang()  {}
   // "Umwandlung" von long nach Lang
   public Lang(long z)
   {  x=z; }
}
```

Diese Version tauscht die beiden Zahlen korrekt.

Bemerkung:

Bei der Übergabe von Strings entstehen auf den ersten Blick eigentümliche Effekte:

```
class Stringtest
{ public static void main(String arguments[])
```

```
  {  String s="Hallo";
     Aendern(s);
     System.out.println(s);
  }
  static void Aendern( String s)
  { s="xxx";
    System.out.println(s);
  }
}
```

liefert xxx
Hallo

Der Grund: Strings können nach der Initialisierung nicht mehr geändert werden; bei jeder vermeintlichen Änderung (z.B. s = s + "xxx") wird vom Compiler ein neues Stringobjekt angelegt. Im Beispiel wird die in s stehende Adresse auf die Adresse des neuen Objekts geändert; da aber die Adresse selbst ein Werteparameter ist, hat dies außen keine Wirkung.

10.12 Das Schlüsselwort `this`

Die zuvor schon angegebene Methode zur Bestimmung des Maximums zweier Punkte sollte eigentlich eine Methode der Klasse **`Punkt`** sein. Hierbei ergibt sich ein Problem:

```
public class Punkt
{ int x, y; String name;
//...
  public Punkt maximum (Punkt p )
  { if (laenge() > p.laenge())
      // <<-- return ????
    else
      return p;
  }
}
```

Die Methode soll wie folgt aufgerufen werden können

```
Punkt m = p1.maximum(p2)
```

und den größeren der beiden Punkte **`p1, p2`** zurückliefern. An der mit ???? markierten Stelle soll das Objekt zurückgegeben werden, auf das die Methode angewendet wird, aber dessen Name ist nicht bekannt!!! Um diese Situation ausdrücken zu können, gibt es das Schlüsselwort **`this`**, das eine Referenz auf das Objekt liefert, auf das die Methode angewendet wird. Damit sieht obiges Beispiel wie folgt aus:

```
public class Punkt
{ int x, y;
  String name;
//...
  public Punkt maximum (Punkt p )
  { if (laenge() > p.laenge())
      return this;
    else
      return p;
  }
}
```

Weitere Anwendungen von **this**:

1. Zugriff auf verdeckte Klassenvariable
   Bei der Definition der Klasse **Punkt** wurden bei dem dort angegebenen Konstruktor Variablennamen benutzt, die nichts mit den Namen der Attribute zu tun hatten :

```
public class Punkt
{  int x, y;
   String name;
// Konstruktor
   Punkt ( int a, int b, String n )
   {  x = a; y = b ; name = n; }
//...
}
```

   Will man im Konstruktor nun die eigentlichen Namen x, y für die Parameter benutzen, so ergibt sich das Problem, dass die Attributnamen **x,y** durch die Parameternamen überdeckt sind und man somit nicht ohne weiteres auf sie zugreifen kann. Eine Lösung des Problems ergibt sich auch hier wieder durch die Verwendung von **this**:

```
public class Punkt
{  int x, y;
   String name;
// Konstruktor
   Punkt ( int x, int y, String n )
   {  this.x = x; this.y = y ; name = n; }
//...
}
```

2. Vereinfachte Deklaration der Konstruktoren
   Bei der Definition der Klasse Punkt mit mehreren Konstruktoren wurde der gesamte Code des Konstruktorrumpfes jeweils wiederholt. In vielen Fällen ergibt sich eine Vereinfachung, wenn man bereits vorhandene Konstruktoren nutzen kann. Dies soll hier am Beispiel der Klasse **Punkt** verdeutlicht werden :

```
public class Punkt
{  int x, y; String name;
// Konstruktoren
   Punkt ( int x, int y, String n )
   {  this.x = x; this.y = y ; name = n; }
   Punkt ( )
   {  this(0,0,"unbenannt"); }

   Punkt ( int a, int b )
   {  this(a,b,"unbenannt"); }
   Punkt ( String n )
   {  this(0,0,n); }
// Methoden wie zuvor
}
```

   Diese Aufrufe von **this(...)** dürfen nur in Konstruktoren auftreten; sie müssen dann jeweils die erste Anweisung im Konstruktorrumpf sein.

10.13 Klassen- und Instanzattribute

Wie bereits erwähnt hat jedes Objekt einer Klasse für die dort definierten Attribute eigene Kopien, während die Methoden nur einmal pro Klasse existieren. Manchmal ist es aber sinnvoll, dass alle Objekte einer Klasse bestimmte Attribute gemeinsam benutzen. In der Klasse **Punkt** könnte dies z.B. das Attribut **PunkteAnzahl** sein, das die Anzahl aller definierten Punkte zählt. Ein solches gemeinsames Attribut wird mit dem Schlüsselwort **static** deklariert und belegt pro Klasse nur einen Speicherplatz. Ein solches statisches Attribut heißt ***Klassenattribut***, die anderen (also die nicht statischen) Attribute heißen ***Instanzattribute*** oder ***Objektattribute***.

Beispiel : Erweiterung der Klasse **Punkt** um das Klassenattribut **PunktAnzahl**

```
public class Punkt
{  int x, y;
   String name;
   static int PunktAnzahl=0;
// Konstruktor
   Punkt ( int x, int y, String n )
   {  this.x = x; this.y = y ; name = n; PunktAnzahl++;}

// Methoden wie zuvor
// .. und dazu
   public void zeige()
   {  System.out.println(name + "(" + x + "," + y +
                         "), Anzahl = " + PunktAnzahl);
   }
   public void zeige(String s)
   {  System.out.println(s + name + "(" + x + "," + y +
                         "), Anzahl = " + PunktAnzahl);
   }
}
```

Der Zugriff auf ein Klassenattribut kann wie gewohnt über einen Objektnamen geschehen. Dabei wird aber unabhängig vom aktuellen Objekt immer auf denselben Speicherplatz zugegriffen. Andererseits existiert ein Klassenattribut bereits, wenn noch kein Objekt der Klasse erzeugt wurde. Deshalb kann man auch über den Klassennamen anstelle eines Objektnamens auf ein Klassenattribut zugreifen:

Klassenname.Klassenattribut

Im folgenden Beispiel sind beide Methoden verwendet.

```
public class PunktTest
{  public static void main(String[] argv)
   {  System.out.println("AnzahlPunkte = "
             +Punkt.AnzahlPunkte); // Init.wert
      Punkt p1=new Punkt(1,2,"p1"),
            p2=new Punkt(3,3,"p2");
      p1.zeige();
      p2.zeige();
      Punkt max,
            p3=new Punkt();
      p3.zeige();
      max=p1.maximum(p2).maximum(p3);
      max.zeige("Maximaler Punkt: ");
```

```
        System.out.println("Es gibt jetzt "
             +p1.AnzahlPunkte +" verschiedene Punkte");
        System.out.println("Es gibt jetzt "
             +p2.AnzahlPunkte+" verschiedene Punkte");
    }
}
```

10.14 Klassenmethoden

Ebenso wie Attribute kann man Methoden als **`static`** kennzeichnen. Wie Klassenattribute werden statische Methoden über den Klassennamen aufgerufen; man benötigt also kein Objekt der Klasse. Solche statischen Methoden heißen ***Klassenmethoden***; nicht-statische Methoden heißen ***Objektmethoden*** oder ***Instanzenmethoden***.

Beispiel : Die Klasse Punkt wird um eine statische Methode **`maximum`** erweitert, die wie zuvor das Maximum zweier Punkte ermittelt. Da die Methode über den Klassennamen **`Punkt`** aufgerufen wird, benötigt sie zwei Parameter vom Typ **`Punkt`**.

```
public class Punkt
{  int x, y; String name;
   static int PunktAnzahl=0;
// Konstruktor
   Punkt ( int x, int y, String n )
   {  this.x = x; this.y = y ; name = n; PunktAnzahl++;}
// Methoden "laenge", "zeige" wie zuvor ... und dazu
// Instanzenmethode maximum ( zum Vergleich )
   public Punkt maximum(Punkt p)
   { if (laenge() > p.laenge())
         return this;
     else
         return p;
   }
// Klassenmethode maximum
   public static Punkt maximum(Punkt p1, Punkt p2)
   { if (p1.laenge() > p2.laenge())
         return p1;
     else
         return p2;
   }
}
```

Eine Testklasse hierzu :

```
public class PunktTest
{  public static void main(String[] argv)
   { Punkt p1 = new Punkt(1,2,"p1"),
           p2 = new Punkt(3,3,"p2");
     Punkt max;
   // Aufruf der Instanzenmethode
   max = p1.maximum(p2);
   max.zeige("Max. Punkt (Instanzenmethode)");
```

```
        // Aufruf der Klassenmethode
        max = Punkt.maximum(p1,p2);
        max.zeige("Max. Punkt (Klassenmethode)");
        }
    }
```

Offensichtlich müssen Methoden, die ohne die Existenz eines Objekts aufgerufen werden, immer Klassenmethoden sein. Dies ist der Grund dafür, dass z.B. die Methode **`main`** in Java-Applikationen immer statisch sein muss.

11 Felder

11.1 Vorbemerkungen

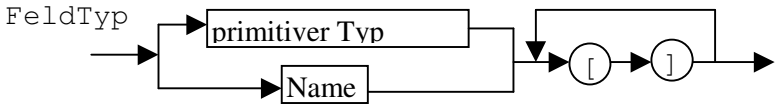
In Java kann man wie in den meisten anderen Programmiersprachen auch eine Folge gleichartiger Objekte unter einem gemeinsamen Namen zusammenfassen. Auf die einzelnen Elemente der Folge wird dann über Indizes zugegriffen.
Java kennt hierzu zwei Konstrukte : **Felder** und die Klasse **`Vector`**.
Diese beiden Konstrukte unterscheiden sich wie folgt :

- Bei **Feldern** wird bei der Erzeugung die Anzahl der Elemente festgelegt; diese ist anschließend nicht mehr änderbar.
- Bei Objekten der Klasse **`Vector`** kann die Zahl der Elemente zur Laufzeit noch geändert werden; außerdem können die einzelnen Elemente unterschiedlichen Typ haben. Die Klasse **`Vector`** ist somit weitaus flexibler als Felder, aber sie ist keine Basisklasse und steht erst nach dem Importieren des Package **`java.util`** zur Verfügung. Die detaillierte Besprechung dieser Klasse sprengt den Rahmen dieser Veranstaltung und soll deshalb hier unterbleiben.

11.2 Felder

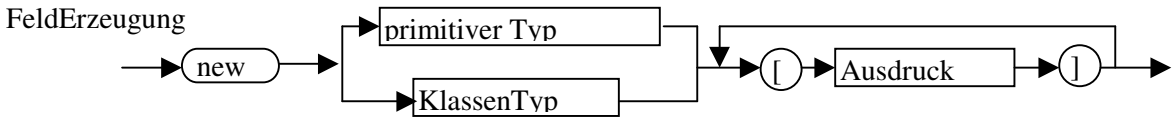
Felder sind wie Klassen Referenzvariable.

Deklaration:



Beispiele:
```
int []   feld;
Punkt[]  Bildpunkte;
String[] Texte;
```

Erzeugung von Feldobjekten (vereinfacht):



Der Ausdruck in [..] muss einen ganzzahligen Wert liefern; die Nummerierung beginnt ab dem Indexwert 0.

Beispiele:
```
feld  = new int[5];          // 5 int's; Nummerierung: 0..4
Bildpunkte = new Punkt[100]; // 100 Punkte
Texte = String[10];          // 10 Strings
```

Die Feldelemente werden hierbei mit dem Wert 0 bzw. mit zu 0 äquivalenten Werten initialisiert. Die Strings in **`Texte`** enthalten Leerstrings. Man kann aber auch ein Feld bei der Deklaration mit Werten initialisieren :

Beispiel: `int[] feld = {1,2,3};`

Hiermit wird ein ganzzahliges Feld mit den 3 Elementen **feld[0]**=1, **feld[1]**=2 und **feld[2]**=3 erzeugt.

Zugriffe sind nur auf die Feldelemente erlaubt, die wirklich existieren. Ein Zugriffsversuch auf ein nicht-existierendes Feldelement ist illegal und erzeugt eine Fehlerbedingung. Aus diesem Grund sollte man bei Zugriffen sicherstellen, dass die fraglichen Indizes im erlaubten Bereich liegen. Der untere Index ist immer 0; der obere Index ist die Länge des Feldes - 1. Jedes Feldobjekt kennt seine Länge (= Zahl der Elemente); diese kann man über das Attribut **length** abfragen.

Beispiel: Erzeugung eines Feldes und Ausgabe

```
public class Feldtest
{ public static void main (String[] args)
  {
    // Einlesen der gewuenschten Feldlaenge
    int anzahl = Input.leseInteger("Feldgroesse : ");
    // ... und Erzeugung des Feldes mit den Quadratzahlen
    //     als Inhalten
    int[] feld = new int[anzahl];
    int i;
    for ( i = 0; i<feld.length; i++)
       feld[i] = i*i;
    // Index einlesen und das entsprechende Feldelement
    // ausgeben.
    // Abbruch bei negativem Index
    int index = Input.leseInteger("\nIndex : ");
    while (index >= 0 )
    { System.out.println("feld[" + index + "] = " + feld[index]);
      index = Input.leseInteger("\nIndex : ");
    }
  }
}
```

Rechteckige mehrdimensionale Felder werden analog zu den bisher besprochenen eindimensionalen Feldern erzeugt und behandelt. Hier müssen dann aber sowohl bei der Deklaration als auch bei der Behandlung jeweils für jeden Indexbereich ein Klammerpaar **[...]** angegeben werden.

Beispiel: Erzeugung und Behandlung 2-dimensionaler Felder

```
public class Matrix
{  private static void matrix_ausgeben(String titel,int[][]m)
   {  System.out.println(titel);
      for (int i=0;i<m.length;i++)
      {  for (int j=0;j<m[i].length;j++)
            System.out.print(m[i][j]+",");
         System.out.println();
      }
   }

   public static void main(String[] args)
   {  int[][] matrix1=new int[3][3],
              matrix2={{ 0, 1, 2, 3, 4},
                       {10,11,12,13,14},
                       {20,21,22,23,24} }
```

```
      // matrix1 einlesen
      for (int i=0;i<matrix1.length;i++)
      {  for (int j=0;j<matrix1[i].length;j++)
         {  System.out.print("matrix1["+i+"]["+j+"]: ");
            matrix1[i][j]=Input.leseInteger();
         }
      }
      matrix_ausgeben("Matrix1",matrix1);
      matrix_ausgeben("Matrix2",matrix2);
   }
}
```

Prinzipiell lassen sich auch nicht rechteckige mehrdimensionale Felder erzeugen; die hierzu erforderlichen Verfahren sollen hier lediglich durch ein Beispiel angedeutet werden.

Beispiel: Erzeugung und Behandlung eines beliebigen zweidimensionalen Feldes mit variierender Zeilenlänge und einer Dreiecksmatrix

```
public class Dreiecksmatrix
{  static int [][] Dreieck;
   static private void mat_ausgeben(String titel,int[][]m)
   {  System.out.println(titel);
      for (int i=0;i<m.length;i++)
      {  for (int j=0;j<m[i].length;j++)
            System.out.print(m[i][j]+" , ");
         System.out.println();
      }
   }
   public static void main(String[] argv)
   {  int [][] zweidimFeld={{ 0, 1, 2, 3},
                            {10,11,12},
                            {31,32,33,34,35}
                           }
      int laenge;
      System.out.print(
         "Wie gross soll die Dreiecksmatrix sein? ");
      System.out.flush();
      laenge=Input.leseInteger();

      /* Dreieck instanziieren; die erste Dimension muss
         angegeben werden. Die zweite Dimension kann spaeter
         fuer die einzelnen Zeilen unterschiedlich festgelegt
         werden.
      */
      Dreieck=new int[laenge][];
      // Jetzt werden die einzelnen Zeilen instanziiert
      // mit abnehmender Laenge
      for (int i=0;i<Dreieck.length;i++)
         Dreieck[i]=new int[laenge-i];
      // Belegung der Feldelemente mit Zufallszahlen
      for (int i=0;i<Dreieck.length;i++)
         for (int j=0;j<Dreieck[i].length;j++)
            Dreieck[i][j]=(int)(1000*Math.random());
```

```
            // Ausgabe der 2-dim Matrix
            mat_ausgeben("2-dim. Feld",zweidimFeld);
            // Ausgabe der Dreiecksmatrix
            mat_ausgeben("Dreieck",Dreieck);
         }
      }
```

Diese Applikation liefert folgende Ausgabe :

```
Wie gross soll die Dreiecksmatrix sein? 3
2-dim. Feld
0 , 1 , 2 , 3 ,
10 , 11 , 12 ,
31 , 32 , 33 , 34 , 35 ,
Dreieck
248 , 17 , 427 ,
639 , 136 ,
828 ,
```

12 Vererbung

Klassen können in zwei Formen miteinander verknüpft werden :

- Die Klasse **K1** hat eine Komponente, die vom Klassentyp **K2** ist (*Komposition* von Klassen)
- Eine bestehende Klasse **K3** wird zu einer neuen Klasse **K4** erweitert (*Vererbung*). Dabei erbt die neue Klasse **K4** alle Bestandteile (Attribute und Methoden) von **K3**; zusätzlich kann man in der Klasse **K4** neue Attribute und Methoden hinzufügen oder bestehende Methoden umdeklarieren.

12.1 Komposition von Klassen

In der folgenden Klasse **Figur** wird der allgemeine Aufbau einer geometrischen Figur beschrieben. Die Klasse hat einen Punkt, der die Rolle eines Bezugspunktes für die Figur spielt. Bei einem Kreis kann dies der Mittelpunkt sein, bei einem Rechteck z.B. die linke obere Ecke. Daneben gibt es einige Methoden, die das Verhalten von Figuren beschreiben.

Die Klasse `Figur`

Eine **Figur** hat außer einem Bezugspunkt noch eine Farbe, eine Art der Darstellung (Linie oder gefüllt), einen Namen zur Identifikation sowie die Information, ob die Figur sichtbar ist oder nicht. Aus Gründen der Einfachheit sollen im folgenden Beispiel bei den Ausgaben die Attribute als Text angezeigt werden.

```
// das Package java.awt.* wird benoetigt ( Farben !)
import java.awt.*;
public class Figur
{  final static int LINIE=1,GEFUELLT=2;
   Punkt Bezugspunkt;
   Color Farbe;
   String name;
   int Form;
   boolean sichtbar;
```

```
// Konstruktoren
   Figur(String n,int x,int y,Color f)
   {  Bezugspunkt=new Punkt(x,y);
      Farbe=f; Form=LINIE;
      name=n;  sichtbar=true;  }
   Figur()
   {  this("nichts",0,0,Color.black);  }

   // textuelle Ausgabe der Figur
   public void ausgeben()
   {  System.out.print("Figur : "+name+", Bezugspunkt: ");
      Bezugspunkt.ausgeben();
      System.out.print(", ");
      ausgebenFarbe(Farbe);
      System.out.print(", Form="+Form);
      if (Form==LINIE)
         System.out.print(", LINIE ");
      else
         System.out.print(", GEFUELLT ");
      System.out.println(", sichtbar = "+sichtbar);
   }
   // Figur soll sichtbar angezeigt sein
   public void zeigen()
   {  sichtbar=true; }

   // Figur soll unsichtbar angezeigt sein
   public void verbergen()
   {  sichtbar=false;  }

   // Figur verschieben; es wird der Bezugspunkt verschoben.
   // dx, dy :  Verschiebung in x- und y-Richtung
   public void verschieben(int dx,int dy)
   {  Bezugspunkt.verschieben(dx,dy);  }

   /* Farbe ändern. Die möglichen Werte findet man in der
      Klasse Color.
      neuFarbe : neue Farbe
   */
   public Color aendernFarbe(Color neuFarbe)
   {  Color alt=Farbe;
      Farbe=neuFarbe;
      return alt;
   }

   // Darstellungsform ändern.
   // form  : Wert für Form: LINIE oder GEFUELLT
   public int aendernForm(int form)
   {  int alteForm=Form;
      Form=form;
      return alteForm;
   }
```

```
   /* Gibt die gewählte Farbe als deutsches Wort aus.
      Dabei werden die Standardfarben aus Color berücksichtigt.
      f  : Farbe
   */
   public static void ausgebenFarbe(Color f)
   {  if (f.equals(Color.black))
         System.out.print("schwarz");
      else if (f.equals(Color.blue))
         System.out.print("blau");
      // usw. fuer die weiteren Farben ......
      // .......
      else
         System.out.print("andere Farbe");
      System.out.print("  ");
   }
}
```

12.2 Erweitern von Klassen; Vererbung

Es sollen nun zwei geometrische Figuren definiert werden, nämlich ein Kreis und ein Rechteck. Ein Kreis ist eine Figur, die zusätzlich zum Bezugspunkt (Mittelpunkt des Kreises) noch einen Radius hat. Die Attribute und Methoden von Figur sollen beibehalten werden, als weitere Methoden sollen die Berechnung der Fläche und die Ausgabe des Radius definiert werden. Zu diesem Zweck wird die Klasse **`Figur`** zur Klasse **`Kreis`** erweitert:

```
public class Kreis extends Figur
{  int radius;
   Kreis(String n,int x,int y,int rad,Color farbe)
   {  name=n;
      Bezugspunkt=new Punkt();
      Bezugspunkt.x=x;Bezugspunkt.y=y;
      sichtbar=false;
      Farbe=farbe;
      radius=rad;
   }
   public double flaeche()
   {  return Math.PI*radius*radius;  }
   // Kreisradius ausgeben
   public void ausgeben()
   {  System.out.print("Radius ="+radius+" ,"); }
}
```

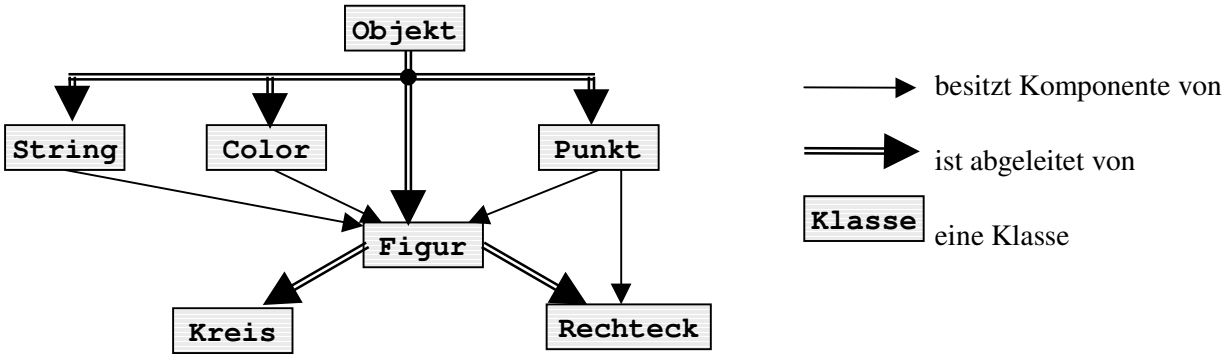
Die Methode **`ausgeben()`** wird in dieser Klasse **`Kreis`** neu definiert; d.h. die bereits in **`Figur`** vorhandene Methode wird überschrieben.

Die ursprüngliche Klasse (hier: **`Figur`**) heißt *Superklasse*; die abgeleitete Klasse (hier: Kreis) heißt *Subklasse*. In der Subklasse kann man nur neue Attribute und Methoden hinzufügen oder Methoden der Superklasse überschreiben. Ein Löschen von Attributen oder Methoden der Superklasse ist nicht möglich.

Weiteres Beispiel: die Klasse Rechteck

```
public class Rechteck extends Figur
{  Punkt zweiteEcke;
   Rechteck(String n,int x,int y,Punkt zweite,Color farbe)
   {  name=n;
      Bezugspunkt=new Punkt();
      Bezugspunkt.x=x;Bezugspunkt.y=y;
      sichtbar=false;
      Farbe=farbe;
      zweiteEcke=new Punkt(zweite.x,zweite.y,"2. Ecke");
   }
   // Rechteckflaeche
   public double flaeche()
   {  int f=(Bezugspunkt.x-zweiteEcke.x)
            *(Bezugspunkt.y-zweiteEcke.y);
      return (f<0)?-f:f;
   }
   // Rechteck textuell ausgeben
   public void ausgeben()
   {  zweiteEcke.ausgeben();  }
}
```

Die in diesen beiden Beispielen hergestellten Beziehungen lassen sich in folgendem Diagramm darstellen:



12.3 Das Schlüsselwort `super`

Die in der Klasse **`Kreis`** definierte Methode **`ausgeben()`** gibt nur den Radius des Kreises aus. Wenn man nun die restlichen Angaben zur Figur ausgeben möchte, könnte man natürlich den entsprechenden Code von **`Figur`** in **`Kreis`** kopieren. Eleganter und einfacher ist es aber, die Methode der Superklasse in der Methode **`ausgeben()`** von Kreis aufzurufen. Hier entsteht aber das Problem, dass beide Methoden gleich heißen. Um anzugeben, dass die entsprechende Methode der Superklasse gemeint ist, verwendet man das Schlüsselwort super:

Beispiel: Neudefinition von **`ausgeben()`** in der Klasse **`Kreis`**

```
public void ausgeben()
{ System.out.println("Radius = " + radius + " , ");
  super.ausgeben();
}
```

Auf die gleiche Weise kann man auch Konstruktoren der Superklasse verwenden :

Beispiel:
```
Kreis(String n, int x, int y, int rad, Color farbe)
{ super(n, x, y, farbe);
  radius = rad;
}
```

12.4 Zuweisungskompatibilität

Regel : In Java kann eine Variable vom Typ der Klasse **K** jeder Variablen vom Typ einer Superklasse von **K** zugewiesen werden.

So können z.B. Objekte vom Typ **Kreis** bzw. **Rechteck** einem Objekt vom Typ **Figur** zugewiesen werden, wie das folgende Beispiel zeigt:

```
public class FigurenTest
{  public static void main(String[] argv)
   {  Figur f1,f2;
      f1=new Kreis("Kreis",10,20,8,Color.red);
      f1.ausgeben();
      f1.verschieben(3,3);
      f1.ausgeben();
      f2=new Rechteck("Rechteck",1,2,new Punkt(3,4),Color.green);
      f2.ausgeben();
      f2.aendernFarbe(Color.blue);
      f2.aendernForm(Figur.GEFUELLT);
      f2.ausgeben();
      f1=new Kreis("Kreis 2",40,40,10,Color.blue);
      f1.ausgeben();
      f2=f1;
      f2.ausgeben();
   }
}
```

Am Programmende stellen **f1** und **f2** Referenzen auf dasselbe Objekt dar.

12.5 Abstrakte Klassen

Wenn im vorangehenden Beispiel in **main** die Methode **f1.flaeche()** aufgerufen wird, so ergibt sich die Fehlermeldung des Compilers

```
Method flaeche() not found in class Figur
```

Diese Meldung ist korrekt, da die Klasse **Figur** keine Methode **flaeche()** enthält; diese wurde erst in den Klassen **Kreis** und **Rechteck** eingeführt. In Figur lässt sich aber nicht in sinnvoller Weise eine Fläche definieren; dazu sind konkrete Figuren wie z.B. Kreise oder Rechtecke erforderlich.

Also: die Klasse Figur ist noch nicht so konkret, dass man eine Methode **flaeche()** implementieren könnte. Deshalb sollte die Klasse **Figur** wie folgt umdefiniert werden:

- Die Klasse **Figur** wird durch das Schlüsselwort **abstract** als abstrakte Klasse festgelegt.
- Die Methode **flaeche()** wird ebenfalls mit **abstract** als abstrakte Methode gekennzeichnet; der Rumpf der Methode entfällt und wird durch ein Semikolon ersetzt.

```
abstract public class Figur
{  // alle Attribute und Methoden wie vorher
   // ... und zusaetzlich
   abstract public double flaeche();
}
```

Hierdurch wird festgelegt, dass die Klasse **Figur** zwar die Methode **flaeche()** kennt; die eigentliche Implementierung der Methode erfolgt aber erst in einer Subklasse. Werden in einer Subklasse nicht alle abstrakten Methoden ihrer Superklasse implementiert, so ist sie ebenfalls abstrakt und muss entsprechend gekennzeichnet werden. Von einer abstrakten Klasse können keine Objekte mit **new** erzeugt werden.
Mit diesen Änderungen könnte man die Flächenberechnung wie folgt durchführen:

```
public class FigurenTest
{  public static void main(String[] argv)
   {  Figur f1,f2;
      f1=new Kreis("Kreis",10,20,8,Color.red);
      f1.ausgeben();
      System.out.println("Flaeche: "+f1.flaeche());
      f2=new Rechteck("Rechteck",1,2,new Punkt(3,4),Color.green);
      f2.ausgeben();
      System.out.println("Flaeche: "+f2.flaeche());
   }
}
```

Als Methode `flaeche()` wird hier jeweils die Methode der zugehörigen konkreten Klasse aufgerufen.

12.6 Zugriffsrechte

In Java gibt es vier Stufen von Zugriffsrechten zu Attributen und Methoden einer Klasse. Diese Zugriffsrechte werden durch eines der Schlüsselworte **public, private, protected** oder durch den Verzicht auf eines dieser Schlüsselworte gekennzeichnet.

- **public**
  Beginnt die Deklaration einer **Klasse** oder **Methode** mit public, so können diese überall benutzt werden. Auf mit **public** gekennzeichnete **Attribute** kann lesend und schreibend von überall her zugegriffen werden. Dies widerspricht aber der in der Regel gewünschten Kapselung von Attributen in Klassen beim objektorientierten Programmieren, da in der Regel nur der Programmierer einer Klasse genau weiß, welche Zugriffe auf ein Attribut legal sind. Wenn man einem Benutzer der Klasse einen lesenden oder schreibenden Zugriff auf gewisse Attribute geben möchte, sollte die aus dem genannten Grund nur durch die Zurverfügungstellung von öffentlichen Lese- oder Schreibmethoden geschehen und nicht durch direkte Zugriffsmöglichkeiten auf das jeweilige Attribut.
- **private**
  Mit **private** gekennzeichnete Attribute und Methoden können nur in Methoden dieser Klasse benutzt werden; andere Klassen (auch Subklassen!) haben keinen Zugriff.
  Eine Klasse, die ein Objekt ihres eigenen Klassentyps verwendet, hat dagegen Zugriff auf die privaten Komponenten (Attribute und Methoden) dieses Objekts.
  Attribute sollten in der Regel als **private** gekennzeichnet sein; Klassen können nicht als private gekennzeichnet werden.
- kein Modifizierer
  Wird bei einer Klasse, einem Attribut oder einer Methode kein Modifizierer verwendet, so kann man diese in jeder Klasse verwenden, die im selben **Package** definiert sind. In dieser Veranstaltung wurden Packages nicht besprochen; wenn keine Package-Deklaration verwendet wird, interpretiert Java alle im aktuellen Arbeitsverzeichnis liegenden Klassen als Teile eines von Java selbst definierten Package.

- **`protected`**
  Auf mit **`protected`** gekennzeichnete Komponenten (Attribute, Konstruktoren und Methoden) einer Klasse können alle Subklassen und die Klassen desselben Package zugreifen.

Schema :

Zugriff aus	**`public`**	ohne	**`protected`**	**`private`**
Klasse	ja	ja	ja	ja
Subklasse	ja	ja: falls im selben Package wie die Superklasse nein : sonst	ja	nein
Package	ja	ja	ja	nein
alle	ja	nein	nein	nein

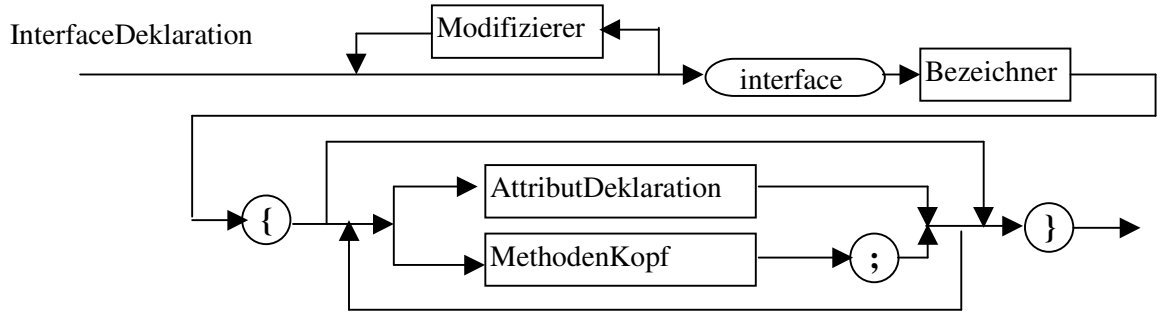
Beim Überschreiben von Methoden muss das alte Zugriffsrecht berücksichtigt werden: **`public`**-Methoden dürfen beim Überschreiben nicht als **`private`** gekennzeichnet werden. Es sind nur Änderungen des Zugriffsrechts in die allgemeinere Richtung erlaubt (hier: von rechts nach links in der Tabelle); dazu die Änderung nach **`final`**.

- **`final`**
  Attribute: **`final`**-Attribute sind Konstante; sie können nicht mehr geändert werden.
  Methoden: **`final`**-Methoden können in keiner Subklasse überschrieben werden; sie werden also unverändert vererbt, sofern sie nicht **`private`** sind.
  Klassen: Aus **`final`**-Klassen können keine Subklassen mehr abgeleitet werden.

12.7 Interfaces

Die Verwendung abstrakter Klassen oder abstrakter Methoden erlaubt es, in den abgeleiteten Klassen die Existenz bestimmter Methoden unter einem festgelegten Namen zu erzwingen. Dies ist allerdings auf die jeweiligen Subklassen beschränkt. Es gibt aber Situationen, in denen es sinnvoll ist, analog zur beschriebenen Situation die Existenz bestimmter Methoden zu erzwingen, ohne dass eine entsprechende abstrakte Superklasse mit den fraglichen abstrakten Methoden existiert.
Beispiel:
Soll die schon vorher vorgestellte Klasse **`Figur`** nicht nur als Superklasse für ebene Figuren wie Kreise, Dreiecke etc. sondern auch als Superklasse für räumliche Körper wie Kugeln, Zylinder, Quader dienen, so ist es offensichtlich unsinnig, in Figur abstrakte Methoden für die Flächen- oder Volumenberechnung zu definieren, da diese jeweils nur für einen Teil der abgeleiteten Subklassen sinnvoll sind. Andererseits ist es wiederum sinnvoll, z.B. für ebene Figuren Methoden für Flächen- und Umfangberechnung und für räumliche Körper Methoden zur Oberflächen- und Volumenberechnung zu verlangen.

Als Ausweg bietet sich die Verwendung von ***Interfaces*** an. Ein Interface ähnelt einer abstrakten Klasse, in der ausschließlich abstrakte Methoden und Konstanten enthalten sind. Ein Interface wird unter Verwendung des Schlüsselworts **`interface`** nach folgendem (vereinfachten) Schema deklariert:



Implizit sind alle im Interface angegeben Attribute **`final public static`** und die Methoden **`abstract public`**; diese Modifizierer brauchen also nicht (und sollten auch nicht) angegeben werden. Ein Interface selbst ist ebenfalls implizit **`abstract`**, ohne dass der entsprechende Modifizierer angegeben wird.

Beispiele:

```
interface EbeneFigur {
  double flaeche();
  double umfang();
}
```

```
interface RaumFigur {
  double volumen();
  double oberflaeche();
}
```

Verwendung von Interfaces

Die abstrakten Methoden eines Interface werden von einer Klasse *implementiert*, indem im Kopf der Klassendeklaration das Schlüsselwort **`implements`** gefolgt vom Namen des Interface angegeben wird.

Beispiele:
```
class Polyeder extends Figur implements EbeneFigur {...}
class Kegel extends Figur implements RaumFigur {...}
```

Jede Klasse, die ein Interface implementiert, muss alle abstrakten Methoden, die im Interface angegeben sind, implementieren. Eine Klasse kann ohne weiteres mehrere Interfaces implementieren.

13 Fehlerbehandlung

13.1 Vorbemerkungen

Eine Klasse ist ein eigenständiges Programm, das eine bestimmte Anwendung implementiert. Dabei kann das Programm eine fehlerhafte Benutzung bemerken. Dies können z.B. sein

- Zugriffe auf nicht vorhandenen Indizes eines Feldes
- der Versuch, eine nicht vorhandene Datei zu öffnen
- für das aktuelle Problem illegale Daten
- .....

Wie auf eine solche fehlerhafte Benutzung reagiert werden soll, kann in der Regel nur der Benutzer und nicht der Autor der Klasse entscheiden. Die Ausnahmebehandlung von Java stellt relativ bequem zu benutzende Hilfsmittel hierzu zur Verfügung.

13.2 Auswerfen von Ausnahmen

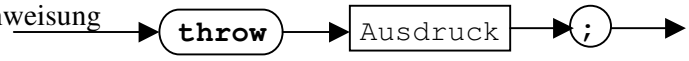
Beispiel: Die Klasse **`MonatLesen`** stellt eine Lesemethode zur Verfügung, die eine Monatsziffer einliest. Wird eine gültige Monatsziffer (d.h. 1..12) eingelesen, so wird diese zurückgegeben. Wird eine Zahl außerhalb dieses Bereichs eingelesen, so ist dies offensichtlich ein Fehler und die Methode wirft eine Ausnahme aus. Dabei wird die in Java definierte **Ausnahmeklasse `Exception`** verwendet, der man beim Konstruktor einen Text zur Fehlerbeschreibung mitgeben kann.

```
public class MonatLesen
{  // legale Monatsnummer einlesen; sonst Auswurf einer Exception
   static int MonatLesen() throws Exception
   // "throws": die Ausnahme wird ggf. an
   // einen Aufrufer weitergereicht
```

```
    {
       int monat=Input.leseInteger();
       if ((monat<1)|(monat>12))
          throw new Exception("\""+monat+"\" ist kein Monat");
                        // hier wird die Ausnahme mit einer
                        // geeigneten Information ausgeworfen
       return monat;
    }
 }
```

Die Ausnahme wurde durch die **throw**-Anweisung ausgeworfen.

Syntax: throw-Anweisung

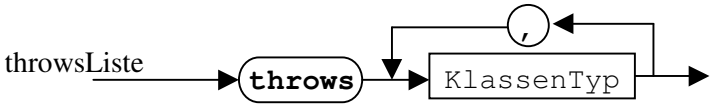


Die Methode **MonatLesen** möchte die Ausnahme an den Aufrufer der Klasse weiterreichen; hierzu muss im Methodenkopf diese Ausnahme mit **throws** angegeben werden.

Ruft nun eine Methode andere Methoden auf, in denen (wie hier in **MonatLesen()**) Ausnahmen ausgeworfen werden können, so ist der Aufrufer für die weitere Behandlung zuständig. Dazu muss die aufrufende Methode diese Ausnahme

- entweder in einem **catch**-Block auffangen
- oder im Methodenkopf in einer throwsListe ihrerseits an ihren Aufrufer weiterreichen.

Syntax:



Wird bei der Methode MonatLesen keine throwsListe angegeben, so bemerkt der Compiler, dass zwar eine Ausnahme ausgeworfen, diese aber weder behandelt noch weitergereicht wurde.

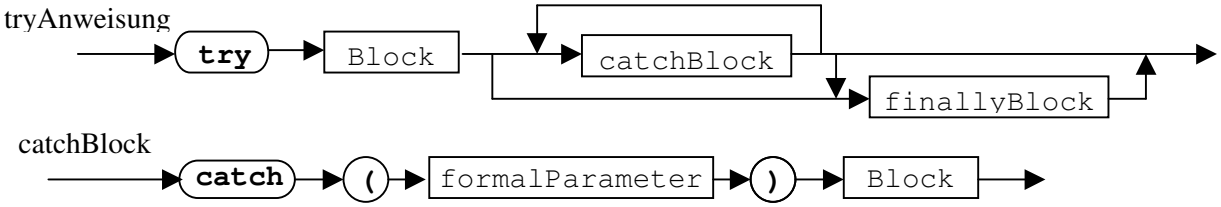
13.3 Abfangen von Ausnahmen (try, catch)

Beispiel: Die folgende Applikation verwendet die Methode MonatLesen und fängt bei Bedarf die ausgeworfene Ausnahme ab.

```
public class MonatEinlesen
{  public static void main(String[] argv)
   {  int monat1,monat2;
      System.out.print("1. Monat:");
      try
      {  monat1=MonatLesen.MonatLesen();
         System.out.println("monat1="+monat1);
         System.out.print("2. Monat:");
         monat2=MonatLesen.MonatLesen();
         System.out.println("monat2="+monat2);
      }
      catch (Exception e)
      {  System.out.println(e.getMessage());
      }
      System.out.println("Programmende");
   }
}
```

Werden hier zwei korrekte Werte eingegeben, so wird der **try**-Block ausgeführt und anschließend hinter dem **catch**-Block weitergemacht. Bei einer fehlerhaften Eingabe wird die Methode **MonatEinlesen** an

der fraglichen Stelle beendet und in den **catch**-Block verzweigt. Die hier im **catch**-Block verwendete Methode **getMessage** liefert den Text, der bei der Definition der Ausnahme im Konstruktor angegeben wurde. Die **try**-Anweisung und der **catch**-Block haben dabei folgende Syntax:



Der formale Parameter muss dabei eine **Exception** oder eine selbstdefinierte Ausnahme sein. Selbstdefinierte Ausnahmen sind dabei Subklassen der Klasse **Exception**.

Beispiel: Im Monatsbeispiel soll nun bei einer fehlerhaften Eingabe die Ausnahmebehandlung so erfolgen, dass nach der Fehlermeldung ein Defaultwert (z.B. 1) benutzt wird.

```
class MonatAusnahme extends Exception
{  private int monat;
   public MonatAusnahme()
   {  super(); }
   public MonatAusnahme(String s,int d)
   {  super(s);
      monat=d; }
   /* Erlaubt einen neuen Versuch der Monatseingabe.
   *  Ist auch dies fehlerhaft, wird Monat 1 benutzt.
   */
   public int erneuterVersuch()
   {  try
      {
         System.out.println("monat war falsch:"+monat);
         System.out.print("Neuer Versuch:");
         monat=Input.leseInteger();
         if ((monat<1)|(monat>12))
            throw new MonatAusnahme(
               "Nochmals falsch. Es wird 1 genommen",monat);
      }
      catch (MonatAusnahme d)
      {  System.out.println(d.getMessage());
         monat=1;
         // monat = erneuterVersuch();    // Stelle 1
      }
      return monat;
   }
}
```

In der Methode **erneuterVersuch** wird die Ausnahme **MonatAusnahme** ausgeworfen und sofort selbst behandelt; deshalb muss sie nicht im Methodenkopf in der **throws**-Liste weitergereicht werden. Das folgende Beispiel demonstriert die Verwendung dieser Ausnahmeklasse:

```
public class MonatEinlesen2
{  public static void main(String[] argv)
   {  int monat;
      System.out.print("Monat:");
```

```
        try
        {  monat=MonatLesen2.MonatLesen();
        }
        catch (MonatAusnahme m)
        {  System.out.println(m.getMessage());
           monat=m.erneuterVersuch();
        }
        System.out.println("monat="+monat);
     }
  }
```

Die hierbei verwendete Klasse **MonatLesen2** stimmt i.w. mit **MonatLesen** überein; lediglich **Exception** wurde durch **MonatAusnahme** ersetzt:

```
  public class MonatLesen2
  {  static int MonatLesen() throws MonatAusnahme
     // "throws": die Ausnahme wird ggf. an
     // einen Aufrufer weitergereicht
     {
        int monat=Input.leseInteger();
        if ((monat<1)|(monat>12))
           throw new MonatAusnahme("\""+monat+"\" ist kein Monat",1);
        return monat;   }
  }
```

Wird im **catch**-Block von **MonatAusnahme** die mit **Stelle 1** markierte Zeile anstatt der beiden davorstehenden Anweisungen verwendet, so werden neue Eingaben angefordert, bis endlich eine Eingabe korrekt ist.

Alle Ausnahmeklassen haben als gemeinsame Superklasse die Klasse **Throwable**, die außer der bereits benutzten Methode **getMessage** eine Reihe von weiteren Methoden zur Verfügung stellt. Diese sollen an dieser Stelle nicht besprochen werden.

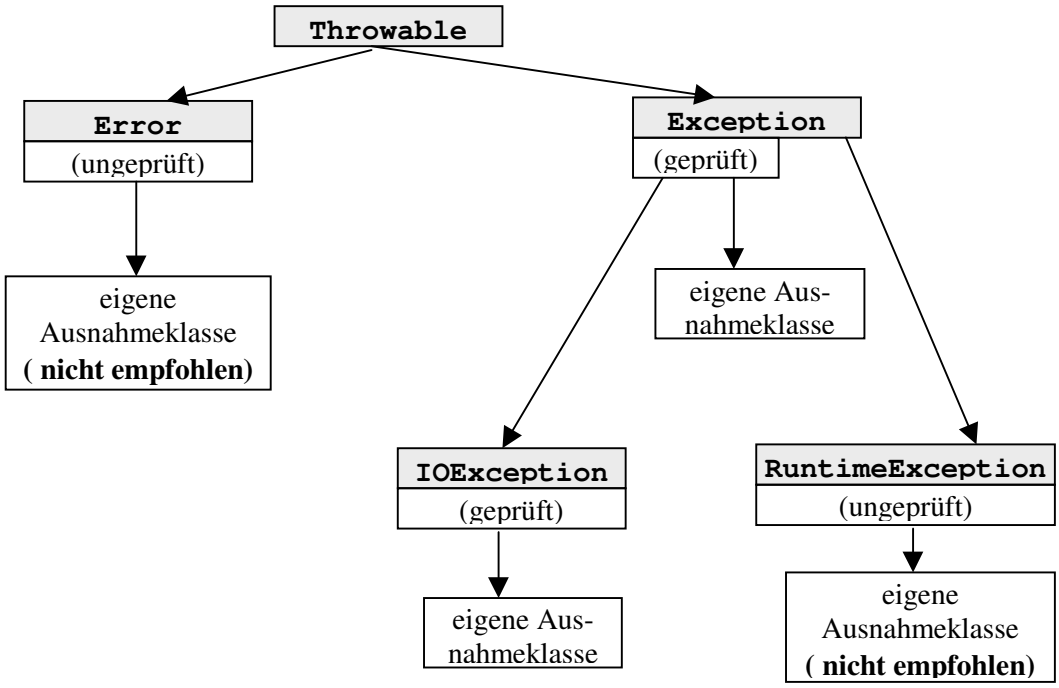
13.4 Der `finally`-Block

Wie zuvor erklärt wird beim Auftreten einer Ausnahme der **try**-Block sofort verlassen. In der Regel wird dann bei der Ausnahmebehandlung die aktuelle Methode beendet. Es gibt aber Situationen, bei denen vor Beendigung der Methode zwingend eine Anweisungsfolge durchlaufen werden muss, auch wenn die Methode durch die aufgetretene Ausnahme irregulär beendet wird (Beispiel: eine durch die Methode geöffnete Datei sollte zum Erreichen eines definierten Zustands wieder geschlossen werden; ebenso sollten teilweise geöffnete Netzwerkverbindungen geschlossen werden,...). Diese Aufgabe übernimmt der **finally**-Block.
Hinter einem **try**-Block können keine oder mehrere **catch**-Blöcke oder ein **finally**-Block stehen (siehe Syntaxdiagramm); mindestens einer dieser Blöcke muss aber vorkommen. Ein **finally**-Block muss hinter allen **catch**-Blöcken stehen.
Der **finally**-Block wird durchlaufen

- nach dem normalen Ende des **try**-Blocks (keine Ausnahme aufgetreten)
- nach einer behandelten Ausnahme
- vor einer Ausnahme, die aus dieser Methode weitergereicht, aber nicht selbst behandelt wird
- nach dem Verlassen des **try**-Blocks über **break**, **continue** oder **return**

13.5 Hierarchie der Ausnahmeklassen

Viele Methoden des Java-Package definieren eigenen Ausnahmeklasse, um möglichst umfassend über die Art der Ausnahmen informieren zu können. Diese Ausnahmeklassen sind entsprechend ihren Aufgaben in folgender Klassenhierarchie eingeordnet:



Geprüfte Klassen:

- Diese Klassen und deren Subklassen werden wie bereits besprochen vom Compiler daraufhin überprüft, ob die Ausnahmen behandelt oder weitergereicht werden.

Ungeprüfte Klassen:

- Zur Laufzeit eine Programms können eine Reihe von Ausnahmesituationen wie z.B. die (ganzzahlige) Division durch 0 auftreten. Im Prinzip müsste man bei jeder z.B. bei jeder ganzzahligen Division die hierfür vordefinierte Ausnahme **`ArithmeticException`** behandeln oder weiterreichen; es gibt eine Reihe von weiteren Beispielen dieser Art. Hierdurch würden die Programme aber sehr lang und kaum mehr lesbar. Deshalb werden die Klassen **`RuntimeException`** und Subklassen standardmäßig ohne Angabe in einer **`throws`**-Liste weitergereicht und am Ende von der virtuellen Java-Maschine behandelt. Ebenfalls ungeprüft sind die in der Klasse **`Error`** behandelten Fehler der virtuellen Java-Maschine wie z.B. Speicherüberlauf ...
  Ausnahmen ungeprüfter Klassen können aber ebenso wie die anderen Ausnahmen behandelt oder explizit weitergereicht werden.

Selbst definierte Ausnahmeklassen sollen immer geprüft sein; d.h. sie sollten Subklassen von **`Exception`** oder **`IOException`** sein.

13.6 Zusammenfassung: Prinzip der Ausnahmebehandlung

1. Die Methodendeklaration einer Klasse erkennt eine Ausnahme und wirft diese in einer **`throw`**-Anweisung aus, wobei ein Objekt dieser Ausnahme erzeugt wird.
   Die Methode kann die Ausnahme selbst behandeln, aber in der Regel wird sie diese an den Aufrufer weiterreichen. Zum Weiterreichen muss im Kopf der Methode die fragliche Ausnahme in einer throws-Liste angegeben sein:

```
int Methode (....) throws Exception
{  ...
   if (.. es ist etwas passiert ...)
      throw new Exception (....);
... ab hier restlicher Code für den Normalfall
}
```

2. Eine Anwendermethode kann in einem bestimmten Codeteil auf das Auswerfen von Ausnahmen achten und diese behandeln. Dieser Teil des Codes wird durch **try** eingeleitet. In den folgenden **catch**-Anweisungen wird beschrieben, welche Ausnahmen behandelt werden und wie diese zu behandeln sind. Alle im **try**-Block ausgeworfenen Ausnahmen, die nicht behandelt werden, müssen im Methodenkopf in der **throws**-Liste aufgelistet sein (wird durch den Compiler geprüft).
3. Anweisungsfolgen, die unabhängig vom Auftreten einer Ausnahme auf jeden Fall durchlaufen werden sollen, werden in dem **finally**-Block im Anschluss an die **catch**-Blöcke angegeben.
4. Bei der Verwendung mehrerer **catch**-Blöcke ist zu beachten, dass die Reihenfolge von Bedeutung ist. Es wird grundsätzlich immer der erste passende **catch**-Block ausgeführt; deshalb müssen Subklassen von Ausnahmen immer vor ihren jeweiligen Superklassen stehen, da sonst der **catch**-Block für die Subklassen nie erreicht wird.

```
int AnwenderMethode (....) throws Ausnahme1,Ausnahme2
// diese beiden Ausnahmen werden hier nicht behandelt,
// sondern an den Aufrufer weitergereicht
{ ...
  try  // hier folgt der Codeteil mit Ausnahmebehandlern
  { .... der normale Code, der auf Ausnahmen achtet
  }
  catch (andereAusnahme a)
  { ... Ausnahmebehandlung für andereAusnahme }
  catch (nochEineAusnahme a)
  { ... Ausnahmebehandlung für nochEineAusnahme }
  catch (Exception e)
  { ... Ausnahmebehandlung für Exception }
          .... weiterer Java-Code
}
```

Bemerkung:

Ausnahmen können nicht nur in den Methoden der Klasse ausgeworfen werden, sondern auch bereits in den Konstruktoren. Diese Ausnahmen müssen dann entsprechend dem Vorgehen bei Methoden entweder im Konstruktor direkt oder beim Aufruf des Konstruktors (**new ...**) abgefangen werden.

14 Grafikprogrammierung

14.1 Vorbemerkungen

Java stellt im Package **java.awt** und dessen Unterpackages die wesentlichen Klassen für die Grafik-Programmierung zur Verfügung. Diese Packages sind rechnerunabhängig implementiert und werden jeweils von der virtuellen Java-Maschine so realisiert, dass die jeweilige Anwendung auf den verschiedenen Plattformen jeweils gleich aussieht. Zur Behandlung von Ereignissen (events; z.B. Tastendruck; Mausaktionen,...) werden die Klassen in **java.awt.event** und den zugehörigen Unterpackages verwendet.

Für ein Grafik-Programm muss zunächst ein **Rahmen** (*frame*) zur Verfügung gestellt werden, in dem dann alle möglichen grafikorientierten Aktionen ablaufen. Die folgende Klasse ist ein einfachstes Beispiel hierfür; es wird lediglich ein Rahmen mit einer Titelleiste erzeugt:

```
import java.awt.*;
public class einfachsteGrafik
{  public static void main(String[] args)
   {  Frame f=new Frame("einfachste Grafik");
      // Rahmen mit Titel erzeugen
      f.setSize(300,200);
      // Größe einstellen, sonst "unsichtbar"
      f.show();    // Grafik anzeigen
   }
}
```

14.2 Aufbau eines Grafik-Programms

- Zuerst wird ein Rahmen erzeugt, in dem die Grafik gezeichnet werden kann. Dies geschieht durch die Erzeugung eines Objekts der Klasse **`Frame`** (aus **`java.awt`**).
- Anschließend wird die (anfängliche) Größe des Rahmens mit **`setSize`** festgelegt; die Parameter sind die Breite und Höhe des Rahmens in Pixeln.
- Mit **`show`** wird nun der Rahmen angezeigt.
- Der Titel des Rahmens wird im Konstruktor angegeben.

Dieser Rahmen kann (fast) wie jedes Fenster bearbeitet werden; insbesondere kann man seine Größe ändern oder ihn zu einem Icon reduzieren. Der Versuch, diesen Rahmen auf die übliche Art zu beenden (d.h. Mausklick auf ✕), scheitert allerdings; dies ist unter Windows nur noch mit dem Taskmanager und unter UNIX mit dem kill-Kommando möglich.

Der Grund für dieses Verhalten:
Jede Benutzereingabe (also auch der Mausklick auf ✕) ist im Sinne der Grafikprogrammierung ein **Ereignis** (event), das behandelt werden muss. Dies geschieht hier durch Einführung einer Subklasse **`GrafikMitBeenden`** von **`Frame`**, in der das Ereignis "Fenster schließen" abgefangen und behandelt wird:

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

public class GrafikMitBeenden extends Frame
{  GrafikMitBeenden()
   {  addWindowListener(new FensterSchliessen()); }

   // lokale Klasse, die ein Fenster schliesst
   class FensterSchliessen extends WindowAdapter
   {  public void windowClosing(WindowEvent e)
      {  System.exit(0);  }
   }

   //"Hauptprogramm" für Grafikprogramme
   public void paint(Graphics g)
   {  g.drawString("Beispieltext",50,75); }

   public static void main(String[] argv)
   {  Frame f=new GrafikMitBeenden(); // Rahmen erzeugen
```

```
        f.setTitle("Grafik beenden mit Alt F4");
        f.setSize(300,200);    // Größe einstellen
        f.show();
    }
}
```

Diese neu eingeführte Klasse **GrafikMitBeenden** ist also ein **Frame** mit den folgenden zusätzlichen Eigenschaften:

- Die innere Klasse **FensterSchliessen** behandelt das Fensterereignisse (und zwar hier nur das Ereignis "Fenster schließen") über die Methode **windowClosing**. Diese Deklaration überschreibt die entsprechende Methode aus der Klasse **WindowAdapter**.
- Im Konstruktor der Klasse **GrafikMitBeenden** wird dieser Ereignisbehandler durch den Methodenaufruf **addWindowListener** registriert; in der Folge reagiert der Rahmen auf den Befehl "Fenster schließen".
- Die Methode **paint** ist das eigentliche Hauptprogramm des Rahmens. In ihr werden alle Aktionen beschrieben, die im Rahmen ausgeführt werden sollen. Der Parameter **g** vom Typ **Graphics** ist der sogenannte Grafikkontext, über den man alle Methoden der Klasse Graphics benutzen kann. Die Methode **paint** wird aus **main** indirekt durch **f.show()** aufgerufen; mit **f.repaint()** erfolgt ein neuerlicher Aufruf von **paint**.

14.3 Die Klasse Frame

Vererbungsstruktur : **Object** → **Component** → **Container** → **Window** → **Frame**

Dabei sind :

Object:	Superklasse, aus der alle Klassen abgeleitet werden
Component:	Objekte mit einer grafischen Darstellung, die auf dem Bildschirm angezeigt werden. Diese Objekte können interaktiv mit dem Benutzer kommunizieren. Beispiele sind Buttons, Kontrollkästchen, Rollbalken,....
Container:	Objekte , die andere AWT-Komponenten enthalten können
Window:	Fenster ohne Rahmen
Frame:	Fenster mit Rahmen und Menuleiste

Einige Methoden der Klasse **Frame**:

- **public void setTitle(String Titel)** (aus **Frame**)
  public String getTitle() (")
  setzt bzw. liefert den Rahmentitel
- **public void setSize(int breite, int hoehe)** (aus **Component**)
  setzt die Rahmengröße
- **public void show()** (aus **Window**)
  zeigt das Fenster auf dem Bildschirm an
- **public void addWindowListener(WindowListener l)** (aus **Window**)
  registriert den **WindowListener l** für den Rahmen

14.4 Die Klasse Graphics

Diese Klasse stellt die wesentlichen Methoden zum Zeichnen zur Verfügung; im Beispiel wurde die Methode **`drawString`** zum Zeichnen eines Textes verwendet.

Zur Kennzeichnung von Fensterpositionen hat jedes Fenster ein Koordinatensystem, bei dem die x-Achse von links nach rechts und die y-Achse von oben nach unten verläuft. Die linke obere Ecke hat dabei die Koordinaten (0,0); die Koordinaten sind ganzzahlig und zählen die Pixel.

Methoden (Auswahl):

a) Zeichenmethoden

- **`public void drawString(String str, int x, int y)`**
  schreibt den Text **`str`** ab der Position (x,y) im Rahmen
- **`public void drawLine(int x1,int y1,int x2,int y2)`**
  `ze`ichnet die Linie zwischen den Punkten (x1,y1) und (x2,y2)
- **`public void drawRect(int x,int y,int breite,int hoehe)`**
  `public void fillRect(int x,int y,int breite,int hoehe)`
  zeichnet ein Rechteck der linken oberen Ecke (x,y) und der angegebenen Breite und Höhe (**`draw...`**: nur den Umriss; **`fill...`**: mit der aktuellen Farbe gefüllt)
- **`public void drawOval(int x,int y,int breite,int hoehe)`**
  `public void fillOval(int x,int y,int breite,int hoehe)`
  zeichnet eine Ellipse in das durch die Parameter gegebene Rechteck
- eine Reihe weiterer Methoden **`drawXXX(...)`** und **`fillXXX(...)`**, wobei XXX eine Beschreibung der jeweiligen Figur ist.

b) weitere Methoden

- **`public abstract void setColor(Color c)`**
  setzt die aktuelle Farbe auf den angegebenen Parameter. Diese Farbe wird anschließend bei allen Grafikoperationen benutzt
- **`public abstract void setFont(Font font)`**
  Der angegebene Font wird bei allen folgenden Schreiboperationen benutzt.
- `.....`

14.5 Farben (die Klasse `Color`)

Farben werden durch die Klasse Color geliefert; am einfachsten sind die dort als statische Konstante definierten Standardfarben zu verwenden.

Standardfarben :

`black, blue, cyan, darkGrey, gray, green, lightGray, magenta, orange, pink, red, white, yellow`

Daneben lassen sich durch den entsprechenden Konstruktor

`public Color(int r, int g, int b)`

beliebige weitere Farben nach RGB-Konvention erzeugen.

Mit z.B. **`g.setColor(Color.pink)`** wird dann im Grafikkontext **`g`** die aktuelle Farbe auf die Standardfarbe **`pink`** gesetzt; das anschließend durch **`g.fillRect(20,50,30,60)`** gezeichnete Rechteck wird mit **`pink`** ausgefüllt.

14.6 Schriften (die Klassen **Font** und **FontMetrics**)

Diese Klassen im Package **java.awt** stellen eine Reihe von Schriften in unterschiedlichen Größen und Auszeichnungen (normal, fett, kursiv) und Methoden zur Verfügung.

Konstanten: **public final int BOLD, ITALIC, PLAIN** (aus **Font**)
Konstanten für den Schriftstil

Konstruktor: **public Font(String name, int stil, int groesse)**
erzeugt ein neues Schriftobjekt mit übergebenen Namen, Stil und Größe

Methoden zur Bearbeitung von Schriften (Auswahl):

- **public Toolkit getToolkit()** (aus **Component**)
  liefert das Toolkit der Komponente
- **public String[] getFontList()** (aus **Toolkit**)
  liefert die Namen der verfügbaren Schriften
- **public FontMetrics getFontMetrics(Font font)** (aus **Component**)
  liefert die plattformabhängige Fontmetrik (z.B. Laufweite der Buchstaben,...)
- **public int getHeigth()** (aus **FontMetrics**)
  liefert die Standardhöhe einer Textzeile (Abstand zweier Zeilen) in der Schrift, die man über **getFontMetrics** angibt
- **public int stringWidth(String str)** (aus **FontMetrics**)
  liefert die Länge des Strings **str** der eingestellten Schrift. Ab dieser Stelle kann das nächste Zeichen ausgegeben werden
- `...` und eine Reihe weiterer Methoden

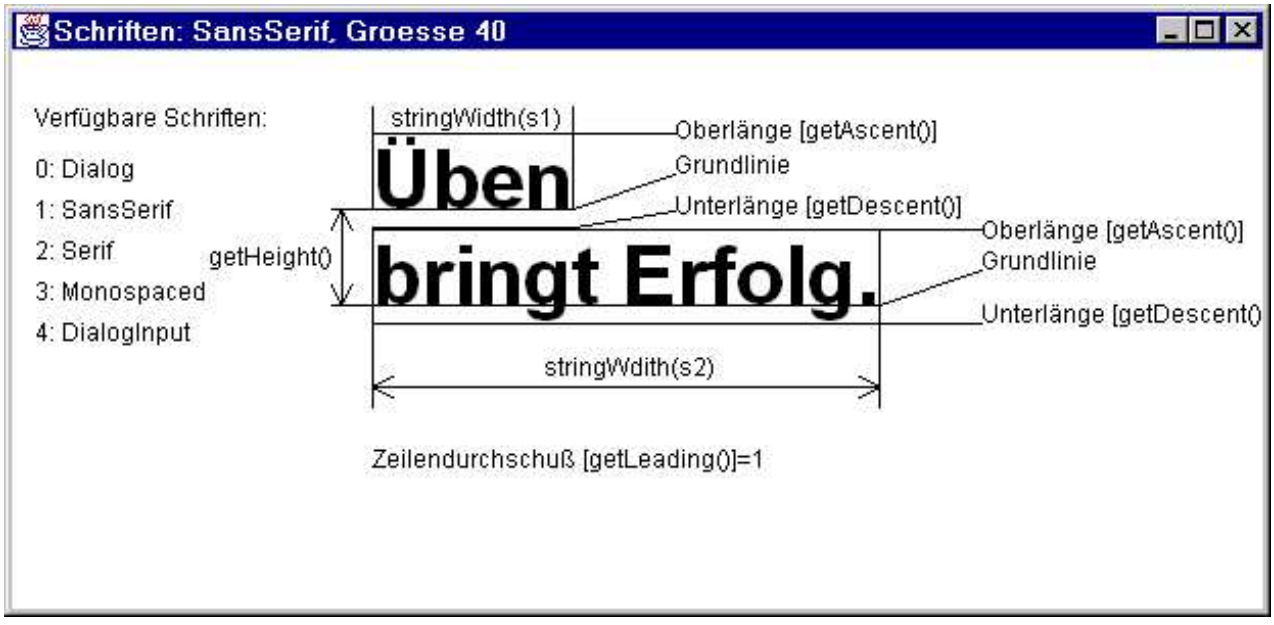
Das folgende Programm demonstriert einige Möglichkeiten zur Manipulation und Ausgabe von Schriften. Es wird ein fester Text ausgegeben, dessen Schriftart und Größe über die Tastatur ausgewählt werden.

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

public class Schriften extends GrafikMitBeenden
{  static String []Schriften;
   static int textx,Schrift,Zeile1,Zeile2,groesse;
   static String s1,s2;

   /* Dialog für Auswahl und Größe der Schrift
    * sowie Standardinitialisierungen.
   */
   Schriften()
   {  Schriften=getToolkit().getFontList();
      textx=180;Zeile1=100;
      s1="Üben";s2="bringt Erfolg.";
      for (int i=0;i<Schriften.length;i++)
         System.out.println(i+": "+Schriften[i]);
      Schrift=Input.leseInteger("Schrift auswaehlen: ");
      groesse=Input.leseInteger("Groesse auswaehlen: ");
      setTitle("Schriften: "+Schriften[Schrift]+", Groesse "+groesse);
      addWindowListener(new FensterBeenden());
   }
```

```
// Aufbau der Grafik
public void paint(Graphics g)
{  g.drawString("Verfügbare Schriften:",15,60);
   for (int i=0;i<Schriften.length;i++)
      g.drawString(i+": "+Schriften[i],15,85+20*i);
   Font f=new Font(Schriften[Schrift],Font.BOLD,groesse);
   FontMetrics fm=getFontMetrics(f);
   int hoehe=fm.getHeight();
   int breite1=fm.stringWidth(s1);
   int breite2=fm.stringWidth(s2);
   int oberlaenge=fm.getAscent();
   int unterlaenge=fm.getDescent();
   int mitte=(oberlaenge+unterlaenge)/2-6;

// 1. Zeile
   g.drawLine(textx,Zeile1-oberlaenge,textx+breite1+50,
              Zeile1-oberlaenge);
   g.drawString("Oberlänge            [getAscent()]",textx+breite1+50,
                Zeile1-oberlaenge+3);
   g.drawLine(textx-20,Zeile1,textx+breite1,Zeile1);
   g.drawLine(textx+breite1,Zeile1,textx+breite1+50,Zeile1-mitte);
   g.drawString("Grundlinie",textx+breite1+50,Zeile1-mitte);
   g.drawLine(textx,Zeile1+unterlaenge,textx+breite1,
              Zeile1+unterlaenge);
   g.drawLine(textx+breite1,Zeile1+unterlaenge,textx+breite1+50,
              Zeile1+unterlaenge-8);
   g.drawString("Unterlänge     [getDescent()]",     textx+breite1+50,
                Zeile1+unterlaenge-6);
// 2. Zeile
   Zeile2=Zeile1+hoehe;
   g.drawLine(textx,Zeile2-oberlaenge,textx+breite2+50,
              Zeile2-oberlaenge);
   g.drawString("Oberlänge            [getAscent()]",textx+breite2+50,
                Zeile2-oberlaenge+5);
   g.drawLine(textx-20,Zeile2,textx+breite2,Zeile2);
   g.drawLine(textx+breite2,Zeile2,textx+breite2+50,Zeile2-mitte);
   g.drawString("Grundlinie",textx+breite2+50,Zeile2-mitte);
   g.drawLine(textx,Zeile2+unterlaenge,textx+breite2+50,
              Zeile2+unterlaenge);
   g.drawString("Unterlänge          [getDescent()]",textx+breite2+50,
                Zeile2+unterlaenge);

// getHeight()
   g.drawLine(textx-15,Zeile1,textx-15,Zeile2);
   g.drawLine(textx-15,Zeile1,textx-20,Zeile1+10);
   g.drawLine(textx-15,Zeile1,textx-10,Zeile1+10);
   g.drawLine(textx-15,Zeile2,textx-20,Zeile2-10);
   g.drawLine(textx-15,Zeile2,textx-10,Zeile2-10);
   g.drawString("getHeight()",textx-80,(Zeile1+Zeile2)/2+5);

// Textlänge von s1 anzeigen
   g.drawLine(textx,Zeile1,textx,Zeile1-50);
```

```
      g.drawLine(textx+breite1,Zeile1,textx+breite1,Zeile1-50);
      g.drawString("stringWidth(s1)",textx+breite1/2-40,Zeile1-40);

    // Textlänge von s2 anzeigen
      g.drawLine(textx,Zeile2-oberlaenge,textx,Zeile2+50);
      g.drawLine(textx+breite2,Zeile2-oberlaenge,textx+breite2,
                 Zeile2+50);
      g.drawString("stringWdith(s2)",textx+breite2/2-40,Zeile2+35);
      g.drawLine(textx,Zeile2+40,textx+breite2,Zeile2+40);
      g.drawLine(textx,Zeile2+40,textx+10,Zeile2+35);
      g.drawLine(textx,Zeile2+40,textx+10,Zeile2+45);
      g.drawLine(textx+breite2,Zeile2+40,textx+breite2-10,Zeile2+35);
      g.drawLine(textx+breite2,Zeile2+40,textx+breite2-10,Zeile2+45);

      // Zeilenabstand anzeigen
      g.drawString("Zeilendurchschuss [getLeading()]="+fm.getLeading(),
                   textx,Zeile2+80);

      // Text ausgeben
      g.setFont(f);
      g.drawString(s1,textx,Zeile1);
      g.drawString(s2,textx,Zeile2);
   }

   public static void main(String[] argv)
   {  System.out.println("Bitte warten. Frame wird erzeugt.");
      Frame f=new Schriften();
      f.setSize(620,300);
      System.out.println("Bitte warten. Grafik wird geladen");
      f.show();
   }
}
```

Bei entsprechender Eingabe für die Schriftart und -größe erscheint folgender Rahmen:



15 Applets

15.1 Vorbemerkungen

Applets sind ähnlich aufgebaut wie die zuvor besprochenen Grafikprogramme. Insbesondere können alle grafischen Methoden benutzt werden.

Beispiel: Appletversion des Java-Programms **einfachsteGrafik**

```
import java.awt.*;
import java.applet;
public class einfachstesApplet extends Applet
{ public void paint (Graphics g)
  {     g.drawString("Beispieltext",20,40);
  }
}
```

Unterschiede Applets-Grafikprogramme:

- ein Applet ist eine Subklasse der Klasse **Applet** und nicht der Klasse **Frame**
- eine Methode **main** ist nicht erforderlich. Das Applet wird durch den Browser gestartet; dabei werden die in **paint** angegebenen Anweisungen durchgeführt
- eine Ereignisbehandlung zum Schließen des Fensters ist nicht erforderlich; man kann des Fenster des Browsers schließen. Folglich ist auch keine Anmeldung einer solchen Ereignisbehandlung erforderlich.
- der Aufruf des Applets geschieht durch den Browser durch das Öffnen einer HTML-Seite, die mindestens folgendes HTML-Tag enthält :
  <APPLET CODE="XXX.class" WIDTH="200" HEIGHT="300"> </APPLET>
  wobei **XXX** für den Namen des jeweiligen Applets steht. Die Angaben zur Höhe und Breite geben die Größe des Appletfensters in Punkten an.

15.2 Initialisierung und Finalisierung

Genau wie beim Erzeugen eines neuen Objekts (**new ...**) müssen in der Regel beim Start eines Applets bestimmte Initialisierungen vorgenommen werden. Konstruktoren stehen hierfür nicht zur Verfügung; diese Aufgabe übernimmt die Methode **init()**.

Beispiel: Initialisierung der Appletversion von **Schriften.java**:

```
public void init()
{Schriften=getToolkit().getFontList();
 textx=180;Zeile1=100;
 s1="Üben";s2="bringt Erfolg.";
 Schrift=1;
 groesse=40;
}
```

init() wird beim Laden des Applets ausgeführt.

Zwei weitere Methoden zur Initialisierung bzw. Finalisierung stehen zur Verfügung:

- **start()**
  wird beim ersten Start nach **init()** und bei jedem Wiederbetreten der Appletseite aufgerufen

- **`stop()`**
  wird beim Verlassen der Appletseite aufgerufen.

Werden im Applet Aktionen durchgeführt, deren Weiterführung beim Verlassen der Appletseite nicht sinnvoll ist (z.B. Animationen), so können diese in der Methode **`stop`** angehalten werden, um Ressourcen zu sparen. Beim Wiederbetreten der Appletseite werden diese Aktionen durch **`start`** wieder gestartet.

- **`destroy()`**
  wird beim Beenden des Applets aufgerufen
  Diese Methode wird benutzt, um Aktionen auszuführen, die beim Beenden des Applets erforderlich sind wie z.B. das Freigeben von Netzwerkverbindungen, die beim Start des Applets aufgebaut wurden.

Zusammenfassung:

- Das Hauptprogramm eines Applets ist die Methode **`paint`**.
- Beim Start eines Applets wird zunächst **`init`** und dann **`start`** aufgerufen.
- Beim Beenden eines Applets wird zunächst **`stop`** und dann **`destroy`** aufgerufen.
- Beim Verlassen einer Appletseite wird **`stop`** aufgerufen.
- Beim Wiederbetreten einer Appletseite wird **`start`** aufgerufen.

15.3 Parameterübergabe an Applets

Ähnlich wie die Übergabe von Kommandozeilenparameter an Applikationen lassen sich auch aus einer HTML-Seite durch Erweiterung des Applet-Tags Parameter an Applets übergeben.

Beispiel:

```
<APPLET CODE="Klassenname", WIDTH=Breite, HEIGTH=Höhe>
<PARAM NAME=ParName1 VALUE=Wert1>
<PARAM NAME=ParName2 VALUE=Wert2>
....
</APPLET>
```

Im Java-Programm wird dann der Parameter der HTML-Seite mit der Methode

```
String getParameter(String ParName)
```

gelesen. Diese Methode liefert immer einen String, der bei Bedarf in einen anderen Datentyp konvertiert werden muss. Wird der angegebene Parametername im zugehörigen Applet-Tag nicht gefunden, so wird der Wert **`null`** zurückgeliefert.

Beispiel:

HTML-Datei:

```
<html>
<applet code="SchachbrettApplet.class" width=300 height=300>
<PARAM NAME=Groesse VALUE=20>
<PARAM NAME=Anzahl VALUE=10>
</applet>
</html>
```

Java-Datei:

```
import java.awt.*;
import java.applet.*;
public class SchachbrettApplet extends Applet
{  // Farbsetzungen fuer die Felder
   private Color farbe1=Color.red, farbe2=Color.black;

   // linke obere Ecke des Bretts
   private int offsetx=50, offsety=100;

   // Groesse des Bretts

   private int anzahl=8;

   // Groesse der Felder
   private int gr=30;

   public void init()
   { String par;
     // Groesse der Felder
     if ((par=getParameter("Groesse"))!=null)
        gr=Integer.parseInt(par);
     // Anzahl der Reihen und Spalten
     if ((par=getParameter("Anzahl"))!=null)
        anzahl=Integer.parseInt(par);

    }

     public void paint(Graphics g)
     { this.setBackground(Color.white);
       g.drawString("Schachbrett",50,70);
       for (int i=0;i<anzahl;i++)
           for (int k=0;k<anzahl; k++)
               { if ((i+k)%2==0)
                    g.setColor(farbe1);
                 else
                    g.setColor(farbe2);
                 g.fillRect(offsetx+i*gr,offsety+k*gr,gr,gr);
               }
      }
   }
```

16 Literatur

Zu Teil 1 :

Gulbins, Jürgen ; Obermayr, Karl
AIX UNIX: {System V.4} ; Begriffe, Konzepte, Kommandos
Springer, 1996

Rembold, Ulrich ; Blume, Christian
Einführung in die Informatik für Naturwissenschaftler und Ingenieure
Hanser, 1991

Zu Teil 2 :

Arnow, David; Weiss, Gerald
Java – An Object-Oriented Approach
Addison-Wesley 1998

Bell, Douglas; Parr, Mike
Java for Students
Prentice Hall Europe 1998

Deitel, H.M.; Deitel P.J.
Java – How To Program
Prentice Hall International Inc. 1998

Dieterich, E.-W.
Java
Oldenbourg 1999

Flanagan, David
Java in a Nutshell
O'Reilly 1998

Gosling, James; Joy, Bill; Steele, Guy
The Java Language Specification
Addison-Wesley 1996

Schader, Martin; Schmidt-Thieme, Lars
Java™
Springer 1999

Java – Begleitmaterial zu Vorlesungen/Kursen
Regionales Rechenzentrum für Niedersachsen / Universität Hannover 1998

Java Dokumentation
Sun Microsystems, Inc.
(erhältlich über Internet unter http://www.javasoft.com/products/jdk/1.1/docs.html)

Java Tutorial
Sun Microsystems, Inc.