Universität Karlsruhe (TH)
Institut für Angewandte und Numerische Mathematik

Skriptum zur Vorlesung

Einstieg in die Informatik mit Java

Dr. Gerd Bohlender

Inhaltsverzeichnis

1 Einleitung

1.1 Bemerkungen zum Skriptum

Diese Kurzfassung enthält nur den Tafelanschrieb. Teile der Vorlesung wurden auf Folien bzw. mit dem Beamer gehalten.

1.2 Ziele der Vorlesung

- Vermittlung der Programmiersprache Java,
- Bildung von Grundlagen,
- Vorstellung einiger Algorithmen aus der Informatik und Mathematik,
- Erlangen praktischer Fähigkeiten durch ein Programmierpraktikum.

1.3 Geschichte und Ziele von Java

- Projekt *Green* bei Sun Microsystems (James Gosling u. a., z. Bsp. Hoff, Shaio, Starbuck) seit 1991 für Gerätesteuerungen (Kabel–TV–Anschlussbox, Videorecorder, . . .).
- Nachfolger *Oak* wurde aufgrund eines Namenskonfliktes umbenannt in *Java*, Einsatz als Sprache im Internet seit 1995.
- Ziele sind Internetapplikationen, d. h. Programme werden beim Aufruf einer Webseite übertragen und ausgeführt.

1.4 Spracheigenschaften

- Portabel auf Quelltextebene (*standardisierte* Sprache/Bibliotheken),
- portabel auf Codeebene (maschinenunabhängiger *Bytecode*, Interpreter),
- einfache Sprache, auf Basis bekannter Sprache (C++),
- sicher (bewusste Einschränkungen),
- multi-threaded,
- verteilte Anwendungen (Internet),
- einfaches Erstellen von Windows–Programmen.

1.5 Ein erstes Java–Programm

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

(a) **Kompileraufruf:**
Syntax:

`javac` *Dateiname*

Beispiel: Quelltext ist unter dem Dateinamen `Hallo.java` gespeichert. Kompilierung erfolgt dann durch Eingabe von

```
javac Hallo.java
```

an der Kommandozeile.
Durch die Kompilierung wird eine Datei erstellt, die den gleichen Namen trägt, jedoch die Endung `.class` besitzt. Im vorliegenden Beispiel wird somit die Datei `Hallo.class` erstellt.

(b) **Ausführen des Java–Programms:**
Syntax:

`java` *Dateiname*

Beispiel: Datei `Hallo.java` wurde erfolgreich kompiliert und es wurde folglich eine Datei `Hallo.class` erstellt. Dann kann das Programm durch die Eingabe von

```
java Hallo
```

an der Kommandozeile des Betriebsystems ausgeführt werden.
Achtung: Das Programm wird nur ausgeführt, wenn der Dateiname ohne Erweiterung `.class` eingegeben wird!

1.6 Versionen der Sprache

JavaSoft hat bisher sechs Hauptversionen der Sprache Java veröffentlicht:

- Java 1.0 — eine kleine webausgerichtete Version, die von allen wichtigen Browsern unterstützt wird.
- Java 1.1 — das Release von 1997. Es beinhaltet Verbesserungen im Bereich der Benutzerschnittstellen, ein überarbeitetes Eventhandling und führte die Komponententechnologie namens *JavaBeans* ein.

- Java 2 mit SDK 1.2 — eine deutlich erweiterte Version, die 1998 erschien. Sie brachte neue Features bei der graphischen Benutzeroberfläche, der Datenbankanbindung und viele andere Verbesserungen.
- Java 2 mit SDK 1.3 — die Version des Jahres 2000. Die neuen Kernfeatures umfassen verbesserte Multimediafähigkeiten, leichte Zugänglichkeit und schnelle Kompilierung.
- Java 2 mit SDK 1.4 — erschien 2002 mit Erweiterungen und verbesserter Performance.
- Java 2 mit SDK 1.5 — erschien 2004.

2 Grundelemente eines Java–Programms

2.1 Kommentare

```
//  Kommentar bis zum Zeilenende
/*  Kommentar evtl. ueber mehrere Zeilen */
/** Kommentar evtl. ueber mehrere Zeilen
    (Dokumentationskommentare) */
```

Achtung: Kommentare können nicht beliebig geschachtelt werden und dürfen nicht in Namen und Wortsymbolen stehen!

Fehlerhafte Schachtelung:

```
/*
   a = b + c;              /* Addition */
   System.out.println(a);  /* Ausgabe  */
*/
```

Mögliche Schachtelung:

```
/*
   a = b + c;              // Addition
   System.out.println(a);  // Ausgabe
*/
```

Dokumentationskommentare: Dokumentationskommentare werden wirksam, wenn man die Java–Datei mit dem im JDK enthaltenen Programm `javadoc` bearbeitet. `javadoc` legt mehrere `html`–Dateien an, die zusammen mit den `class`–Dateien als Dokumentation ausgeliefert werden können. Folgende "Tags" werden dabei berücksichtigt:

Tag	Bedeutung	Anwendung
`@see`	Verweis auf andere Stelle	Klasse, Methode, Variable
`@author`	Name des Autors	Klasse
`@version`	Versionsnummer	Klasse
`@param`	Name u. Beschreibung von Parametern	Methode
`@return`	Beschreibung des Funktionswertes	Methode
`@exception`	Name u. Beschreibung von Ausnahmen	Methode

2.2 Bezeichner für Klassen, Methoden, Variablen

- *Bezeichner* (auch geläufig als *Namen*) bestehen aus beliebig langen Folgen von Unicodebuchstaben und Ziffern.
- Ein Bezeichner beginnt mit einem Buchstaben, es folgen Buchstaben oder Ziffern.
- `A` bis `Z`, `a` bis `z`, `_` und `$` sind Unicodebuchstaben, `0` bis `9` sind Unicodeziffern.
- Nicht erlaubt sind alle Wortsymbole und die Literale `true`, `false` und `null`.

Beispiele: `Anfang`, `Ende`, `x1`, `_425`

Achtung: Java unterscheidet in der Groß– und Kleinschreibung! Dadurch sind folgende Bezeichner erlaubt: `Int`, `INT`
Dagegen verboten, da ein reserviertes Wortsymbol: `int`

Konventionen:
Für die Bezeichner von Klassen, Methoden, Variablen und Konstanten wurden die folgenden Vereinbarungen getroffen:

- Klassen: Anfangsbuchstabe groß, Rest klein, Wortanfänge groß,
  z. Bsp. `HalloWelt`,
- Methoden und Variablen: Anfangsbuchstabe klein, sonst identisch mit Konventionen für Klassennamen,
  z. Bsp. `main`, `xWert`, `piMalDaumen`,
- Konstanten: lauter Grossbuchstaben,
  z. Bsp. `MAX`, `DIM`.

2.3 White–Space–Zeichen

Unter dem Begriff *White–Space–Zeichen* werden folgende Zeichen zusammengefaßt:

- Leerzeichen,
- Zeilenendezeichen,
- Tabulator,
- Seitenvorschub.

White–Space–Zeichen erhöhen die Lesbarkeit des Quelltextes und werden zum Trennen von Wortsymbolen, Bezeichnern, usw. benötigt (z. Bsp. `public class Hallo`). Ansonsten besitzen sie keine Wirkung.

Achtung: White–Space–Zeichen dürfen nicht innerhalb von Namen und Wortsymbolen stehen!

2.4 Wortsymbole

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

Achtung: Die Literalkonstanten `true`, `false` und `null` dürfen ebenfalls nicht als Bezeichner verwendet werden!

2.5 Interpunktionszeichen

( ) { } [ ] ; , .

2.6 Operatoren

Insgesamt benutzt Java 36 Operatoren:

=	<	>	!=	? :	==
>>>=	<=	>=	&&	++	−−
+	−	*	/	|	^
%	<<	>>	+=	−=	*=
/=	&=	|=	^=	%=	<<=
>>=	!	>>>	&	||	~

2.7 `import`–Anweisungen

`import`–Anweisungen werden verwendet, um Bibliotheken, die in separaten Klassen ausgelagert sind, der eigenen Klasse zugänglich zu machen.

Beispiele:

```
import Prog1Tools.IOTools;   für die Eingabe,
import java.awt.*;           für graphische Elemente.
```

Achtung: `import`–Anweisungen müssen vor der Definition der Klasse erfolgen!

2.8 Form eines Programms

Wesentliches Konzept von Java ist die objektorientierte Programmierung. Daher muss die `main`–Methode in einer Klasse definiert sein. Für Applikationen muss genau eine Methode der Form `public static void main (String[] args)` definiert sein, dies ist das Hauptprogramm.

Typischer Aufbau eines Programms:

```
import–Anweisungen
public class Klassenname {
    Definitionen globaler Variablen und Methoden
    public static void main (String[] args) {
        Definition lokaler Variablen
        Anweisungen
    }
    weitere Definitionen
}
```

Typische Anweisungen sind Wertzuweisungen der Form $Variable = Ausdruck$ und die Ausgabe von Zeichenketten auf dem Bildschirm.

3 Vordefinierte Datentypen

3.1 Ganzzahlige Typen

Die Darstellung und Genauigkeit der vordefinierten Datentypen ist in Java genau festgelegt.

Typ	Speicherbedarf	Wertebereich
`byte`	8 bit	von -128 bis 127
`short`	16 bit	von -32768 bis 32767
`int`	32 bit	von -2.147.483.648 bis 2.147.483.647
`long`	64 bit	von -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`char`	16 bit	von 0 bis 65.535 (von '\u0000' bis '\uffff')

Achtung: `char` zählt zu den ganzzahligen Typen! Dies ist der einzige vorzeichenlose numerische Datentyp.

3.2 Boolscher Typ

Boolsche Variablen, d. h. Variablen des Typs `boolean`, können nur die beiden Wahrheitswerte `true` und `false` annehmen.

3.3 Gleitkommatypen

Floating point numbers heißen in ihrer deutschen Übersetzung *Gleitkommazahlen* und sind nach dem IEEE–Standard 754 implementiert.

Typ	Speicherbedarf	Wertebereich (ca.)
`float`	32 bit	von 3.4028234663852886E+038 bis 1.4012984643248171E-045
`double`	64 bit	von 1.7976931348623157E+308 bis 4-9406564584124654E-324

Achtung: Obwohl dieser Datentyp in der deutschen Übersetzung das Wort Komma enthält, so werden die Nachkommastellen von Gleitkommazahlen in Java trotzdem mit einem Punkt getrennt!
Tritt "Uberlauf" ein, so wird mit $\pm$ `Infinity` weitergerechnet!
Bei einem sinnlosen Ergebnis (z. Bsp. Null durch Null) wird `NaN` ("Not a Number") eingesetzt.

3.4 Referenztypen

Referenztypen sind das Gegenteil von den bisher genannten *Grundtypen/elementaren Typen* und werden in späteren Kapiteln ausführlich behandelt.

Beispiele: Klassen, Felder, Strings.

3.5 `void`–Typ

Dieser Typ ist ein Pseudotyp und tritt vor allem bei Methoden auf.
Besitzt eine Methode keinen Rückgabewert, so wird `void`, d. h. "nichts" als Ergebnis zurückgegeben.

Beispiel: `public static void main (String[] args) {}`

3.6 Implizite und explizite Typumwandlungen

(a) **implizite Typumwandlungen:**
Implizite Typenumwandlungen treten dann auf, wenn ein kleinerer Zahlenbereich in einen größeren Zahlenbereich abgebildet wird. Solche Umwandlungen werden automatisch ausgeführt.

Beispiele: von `byte` nach `short`, von `short` nach `int`, von `int` nach `long`.

(b) **explizite Typumwandlungen:**
Soll ein größerer in einen kleineren Zahlenbereich umgewandelt werden, so muss dies dem Kompiler explizit mitgeteilt werden. Die Mitteilung erfolgt durch eine vorangestellte Klammerung des gewünschten Datentyps.

Beispiel: Um den ganzzahligen Anteil einer Gleitkommazahl zu bestimmen, muss diese explizit in einen ganzzahligen Datentyp umgewandelt werden:

```
double a = 5.35;
int b    = (int) a;
```

Achtung: Ohne obige explizite Typumwandlung wird bei der Kompilierung ein Fehler ausgegeben!

4 Literalkonstanten

Literalkonstanten besitzen den Wertebereich des entsprechenden Datentyps.

4.1 Ganzzahlige Konstanten

Darstellung	**Beschreibung**
dezimal	Ziffernfolge, die nicht mit `0` beginnt.
oktal	`0` gefolgt von oktaler Ziffernfolge (`0` ... `7`).
hexidezimal	`0x` gefolgt von hexadezimaler Ziffernfolge (`0` ... `9`, `a`, ... `f`, `A`,..., `F`).

Beispiel: Darstellung der Zahl Zehn:

$$\begin{aligned} 10 &= 1 \cdot 10^1 + 0 \cdot 10^0 = 10_{dez}, \\ 012 &= 1 \cdot 8^1 + 2 \cdot 8^0 = 10_{dez}, \\ 0xa &= 10 \cdot 16^0 = 10_{dez}. \end{aligned}$$

Die Suffixe `l`, `L` sorgen für eine Umwandlung in eine Konstante vom Datentyp `long` und erhöhen damit den Speicherbedarf auf 64 Bit.

4.2 Gleitkommakonstanten

Eine Gleitkommazahl besteht aus:

- Mantisse (Ziffernfolge, ggfs. Dezimalpunkt)
- Exponentenangabe (`e` bzw. `E`, ggfs. Vorzeichen)

Achtung: Die Buchstaben `e`, `E` bedeuten Exponentieren zur Basis 10.

Beispiele: `1.23e5` bedeutet $1,23 \cdot 10^5$, `.1e-3` bedeutet $0,1 \cdot 10^{-3} = 10^2$.

Die Suffixe `f`, `F` legen eine Konstante vom Typ `float` an, die Suffixe `d`, `D` entsprechend eine Konstante vom Typ `double`, z. Bsp. `1.2f`, `1.2d`.

4.3 Zeichenkonstanten

Zeichenkonstanten werden in Java in Hochkommata geschrieben.

Beispiele: `'a'`, `'1'`.

Achtung: Unicode–Escapes sind dabei ebenfalls erlaubt!

Escape–Sequenzen:

Zeichen	Unicode	Bezeichung	Wirkung
\b	'u0008'	Backspace	Bewegt Cursor einen Schritt nach links.
\r	'u000d'	Carriage Return	Bewegt Cursur an Zeilenanfang.
\f	'u000c'	Formfeed	Seitenvorschub, neue Seite
\t	'u0009'	Horizontal Tab	Tabulator
\n	'u000a'	Line Feed	Zeilenvorschub, neue Zeile
\\	'u005c'	Backslash	
\'	'u0027'	Anführungszeichen	
\"	'u0022'	"Gänsefüßchen"	

ASCII–Tabelle:

hex		$00	$10	$20	$30	$40	$50	$60	$70
	dez	00	16	32	48	64	80	96	112
$00	00	NUL	DLE		0	@	P	‘	p
$01	01	SOH	DC1	!	1	A	Q	a	q
$02	02	STX	DC2	■	2	B	R	b	r
$03	03	ETX	DC3	#	3	C	S	c	s
$04	04	EOT	DC4	$	4	D	T	d	t
$05	05	ENQ	NAK	%	5	E	U	e	u
$06	06	ACK	SYN	&	6	F	V	f	v
$07	07	BEL	ETB	’	7	G	W	g	w
$08	08	BS	CAN	(	8	H	X	h	x
$09	09	HT	EM	)	9	I	Y	i	y
$0A	10	LF	SUB	*	:	J	Z	j	z
$0B	11	VT	ESC	+	;	K	[	k	{
$0C	12	FF	FS	,	<	L	\	l	|
$0D	13	CR	GS	–	=	M	]	m	}
$0E	14	SO	RS	.	>	N	^	n	~
$0F	15	SI	US	/	?	O	_	o	DEL

Beispiel: Bei der Verwendung des Unicode–Zeichensatzes wird durch

```
System.out.println ('\u0051');
```

das Zeichen `Q` ausgegeben.

4.4 Zeichenketten

- Für *Zeichenketten* (engl. *Strings*) ist in Java der Typ `String` vorgesehen.
- Im Gegensatz zu Zeichenkonstanten werden Zeichenketten in doppelte Hochkommata gesetzt.
- Die Verwendung der Unicode– und Escape–Sequenzen ist identisch zu ihrer Verwendung bei Zeichenkonstanten.
- Durch Einfügen von `\n` in die Zeichenkette kann die Ausgabe auf dem Bildschirm über mehrere Zeilen erfolgen.
- Lange Zeichenketten können mit Hilfe der *Stringkonkatenation* + in mehrere Teilzeichenketten aufgespalten werden.

Beispiel: `"Dies ist ei" + "n String."`

Achtung: `"x"` ist nicht vom Typ `char`, sondern eine Zeichenkette und damit vom Typ `String`! Tritt die gleiche Zeichenkette mehrmals auf, so wird diese nur einmal gespeichert.

Beispiel:

```
String a = "Einmal ist genug!";
String b = "Einmal ist genug!";
```

Beispiel:

```
public class Beispiel {
   public static void main (String [] args) {
      double celsius, fahrenheit;
      celsius    = 30.0;
      fahrenheit = 1.8*celsius + 32.0;
      System.out.println ("Temperatur in Fahrenheit = "
                          + fahrenheit);
   }
}
```

4.5 Boolsche Konstanten

Boolsche Konstanten sind die beiden aus Abschnitt 3.2 bereits bekannten Wahrheitswerte `true` (bedeutet wahr) und `false` (bedeutet falsch).

4.6 `null`–Referenz

Nullreferenzen treten im Zusammenhang mit Objekten auf.

Achtung: Aufgrund der Unterscheidung zwischen Klein– und Großbuchstaben müssen die beiden boolschen Konstanten und die Nullreferenz in Kleinbuchstaben angegeben werden!

5 Variablen und symbolische Konstanten

5.1 Variablendeklaration

Die *Deklaration* der Variablen erfolgt durch eine Typangabe gefolgt von einem Namen. Dadurch wird der dem Typ entsprechende Speicherplatz reserviert. Variablen können, wie es ihr Name bereits vermuten lässt, im weiteren Verlauf des Programms verändert werden.

Syntax:

Typangabe Namensliste; `// Namen durch Komma getrennt`

Beispiele: `char ch; int i, j, k; double d;`

Achtung: Im Gegensatz zu C++ erfolgt keine Trennung von Deklaration und Definition!

5.2 Initialisierung von Variablen

Variablen werden durch Zuweisung eines Wertes initialisiert.
Einmal initialisierte Variablen können durch weitere Wertzuweisungen verändert werden.

Beispiele:

```
double wert = 123.45;
int dim     = 10;
int i, j = 10, k;  // i und k sind nicht initialisiert!
wert        = dim;
```

5.3 Symbolische Konstanten

Das zusätzliche Wortsymbol `final` bewirkt, daß der Wert der "Variablen" nicht verändert werden darf. Daraus folgt, daß symbolische Konstanten immer initialisiert werden müssen.

Beispiele:

```
final int n        = 5;
final char newline = '\n';
```

Achtung: `const` wird zur Zeit von Java nicht unterstützt!

6 Ausdrücke

6.1 Die wichtigsten arithmetischen Ausdrücke

6.1.1 Arithmetische Operatoren

einstellig:	+ −
zweistellig:	+ − * / %

Achtung: Sind beide Operanden ganzzahlig, so handelt es sich bei / (vgl. Pascal: `div`) um die ganzzahlige Division, d. h. Nachkommastellen werden abgeschnitten! Dies gilt auch bei einem negativen Ergebnis!

Der Rest bei der Division zweier ganzer Zahlen (vgl. Pascal: `mod`) kann mit Hilfe des Operators % bestimmt werden. Es gilt: `a % b = a - (a/b)*b`.

Beispiele:

```
a = 5.0/3.0;  // ergibt a = 1.666, ebenso: 5.0/3 und 5/3.0
b = 5/3;      // ergibt b = 1
c = 5 % 3;    // ergibt c = 2
d = (-5)/3    // ergibt d = -1
e = (-5) % 3  // ergibt e = -2
```

Achtung: Der "Restoperator" kann auch auf Gleitkommazahlen angewandt werden!

Beispiel:

```
f = 3.5 % 1.1;  // ergibt f = 0.2
```

6.1.2 Inkrement– und Dekrementoperatoren

- Der Inkrementoperator ++ erhöht den Operanden um eins (vgl. Pascal: `succ`, `inc`).
- Der Dekrementoperator -- vermindert den Operanden um eins (vgl. Pascal: `pred`, `dec`).

Präfix– und Postfixvarianten:
Beide Operatoren können sowohl vor als auch nach dem Operanden stehen. Ihre Wirkung auf den Operanden ist dabei gleich, allerdings unterscheiden sich die beiden Varianten im Wert des Ausdrucks:
Bei der Präfixvariante wird zuerst der Wert des Operanden um eins verändert (erhöht bzw.

vermindert) und erst danach der Wert des Ausdrucks zugewiesen, bei der Postfixvariante geschieht dies in genau umgekehrter Reihenfolge.

Beispiel:

```
i = 1; g = ++i;  // ergibt g = 2 und i = 2.
i = 1; h = i++;  // ergibt h = 1 und i = 2.
```

6.1.3 Zuweisungsoperator

Der Zuweisungsoperator = steht zwischen einer Variablen (linke Seite) und einem Ausdruck (rechte Seite). Der Ausdruck wird ausgewertet und der Wert anschließend der Variablen zugewiesen. Da es sich um einen Operator handelt, erhält der gesamte Ausdruck ebenfalls den ermittelten Wert.

Achtung: Bei der Zuweisung handelt sich in Gegensatz zu Pascal um einen Operator!
Mehrfachzuweisungen sind erlaubt (`a = b = 1;`)!

6.1.4 Mathematische Standardfunktionen

Mathematische Standardfunktionen sind in Java in der Klasse

```
java.lang.Math
```

enthalten. Diese Klasse braucht nicht zu importiert werden, sie wird standardmäßig geladen. Allerdings muss bei der Verwendung von Methoden und Konstanten jeweils der Klassenname `Math` vorangestellt werden.

Beispiele: Sinusfunktion: `Math.sin(x)`, Konstante π: `Math.PI`

Methoden der Klasse `java.lang.Math` **(JDK 1.3):**

Methode	Funktion	Argumenttyp	Ergebnistyp
`abs(x)`	Absolutbetrag	`int`	`int`
		`long`	`long`
		`float`	`float`
		`double`	`double`
`acos(x)`	Arkuskosinus	`double`	`double`
`asin(x)`	Arkussinus	`double`	`double`
`atan(x)`	Arkustangens	`double`	`double`
`atan2(x,y)`	Konvertiert kartesische	`x`: `double`	`double`
	in polare Koordinaten.	`y`: `double`	
`cos(x)`	Kosinus	`double`	`double`
`ceil(x)`	Nächste grössere ganze Zahl	`double`	`double`
`exp(x)`	Exponentialfunktion	`double`	`double`
`floor(x)`	Nächste kleinere ganze Zahl	`double`	`double`
`log(x)`	Natürlicher Logarithmus	`double`	`double`
`max(x,y)`	Maximum	`int`	`int`
		`long`	`long`
		`float`	`float`
		`double`	`double`
`min(x,y)`	Minimum	`int`	`int`
		`long`	`long`
		`float`	`float`
		`double`	`double`
`pow(x,y)`	`x` hoch `y`	`x`: `double`	`double`
		`y`: `double`	
`random()`	G. v. Zufallszahl aus $[0, 1)$		`double`
`rint(x)`	Nächste ganze Zahl	`double`	`double`
`round(x)`	Rundet zur	`float`	`int`
	nächsten ganzen Zahl	`double`	`long`
`sin(x)`	Sinus	`double`	`double`
`sqrt(x)`	Wurzel	`double`	`double`
`tan(x)`	Tangens	`double`	`double`
`toDegrees(x)`	Konvertiert	`double`	`double`
	Bogen– in Gradmaße		
`toRadians(x)`	Konvertiert	`double`	`double`
	Grad– in Bogenmaße		

6.1.5 Vergleichsoperatoren

Vergleichsoperatoren liefern boolsche Werte als Ergebnistyp zurück.

Operator	Bedeutung
==	gleich
!=	ungleich
<=	kleiner gleich
>=	grösser gleich
<	kleiner
>	grösser

6.2 Kombinierte Zuweisungsoperatoren

Kombinierte Zuweisungsoperatoren verkürzen die Scheibweise von Zuweisungen der Art:

```
a = a ○ b;
```

Steht also links wie rechts vom Zuweisungsoperator = dieselbe Variable, kann der Ausdruck auch verkürzt geschrieben werden als:

```
a ○= b;
```

Der Zuweisungsoperator lässt sich mit den folgenden Operatoren kombinieren:

*=	-=	/=	&=
|=	^=	%=	<<=
>>=	>>>=		

Beispiel:

```
g=1, h=1;
g  = g + 5; // ergibt g = 6.
h += 5;     // ergibt h = 6.
```

Achtung: Die Wirkung auf `g` und `h` ist die gleiche, die Anzahl der Auswertungen von `g` und `h` sind dagegen nicht gleich. `g` wird hier zweimal ausgewertet, `h` hingegen nur einmal (wichtig bei Nebeneffekten)!

6.3 Logische Operatoren

Operator	Bedeutung
`&` `&&`	*Logisches Und* Wahr, falls beide Operanden wahr sind.
`|` `||`	*Logisches Oder* Wahr, falls mindestens einer der Operanden wahr ist.
`!`	*Logische Negation* Wahr, falls der Operand falsch ist.
`~`	*Exklusives Oder* Wahr, falls die Operanden verschieden sind.

Die Operatoren `&&` und `||` heißen *Kurzschlußoperatoren*. Sie unterscheiden sich von den Operatoren `&` und `|` durch die Art der Auswertung des zugrundeliegenden Ausdrucks:
Kann aus der linken Seite schon das Ergebnis bestimmt werden, so wird die rechte Seite nicht mehr ausgewertet.

Beispiel:

```
int i, j;
if ((j!=0) && (i/j < 5))  // i/j wird bei j=0
                          // nicht ausgewertet!
if ((j!=0) & (i/j < 5))   // i/j wird bei j=0
                          // ausgewertet! Fehler!
```

6.4 Weitere Operatoren

Operator	Bedeutung
`a?b:c`	*Konditionaloperator*: Verkürzte Form für `if ... else ...` Ist `a` wahr, so ist das Ergebnis `b`, sonst `c`.
`&  |  ^  ~`	*Bitoperatoren*: Bitweises Und, Oder und bitweise Negation
`a << b`	*Schiebeoperator*: `a` wird um `b` Bits nach links verschoben.
`a >> b`	*Schiebeoperator*: `a` wird um `b` Bits nach rechts verschoben, dabei wird links mit einem Vorzeichenbit aufgefüllt.
`a >>> b`	*Schiebeoperator*: `a` wird um `b` Bits nach rechts verschoben, dabei wird mit Nullen aufgefüllt.

Achtung: Grund für die Existenz des Operators >>> ist, daß es in Java keine `unsigned`–Typen gibt!

6.5 Klassifizierung von Operatoren

Operatoren werden anhand der folgenden Kriterien klassifiziert:

(1) Anzahl der Operanden (*unär*, *binär*, *tenär*),

(2) Priorität der Operatoren (siehe auch Abschnitt 6.7),

(3) Assoziativität: Auswertung erfolgt in Normalfall von links nach rechts, bei unären Operatoren und Zuweisungen von rechts nach links,

(4) Präfix– oder Postfixvariante (nur für ++ und −− relevant).

Achtung: Operanden werden im Gegensatz zu Operatoren immer von links nach rechts ausgewertet!

Beispiel:

```
int k=2, l=3, m=4;
k = l = m;           // k,l,m haben alle den Wert 4.
```

Der obige Ausdruck wird wie folgt ausgewertet:

(1) `k`,

(2) `l`,

(3) `m`,

(4) rechte Zuweisung,

(5) linke Zuweisung.

Achtung: Reihenfolge der Auswertungen ist bei Seiteneffekten bedeutsam!

6.6 Typumwandlungen

(a) Umwandlung in "größere" Typen geht automatisch.

`byte` → `short` → `int` → `long` → `float` → `double`
`char` → `int`

Achtung: Von `long` nach `float` können Rundungsfehler auftreten!

(b) Umwandlungen in "kleinere" Typen können zu Fehlern führen und müssen deshalb explizit dem Kompiler mitgeteilt werden (siehe auch Abschnitt 3.6). Dies ist mit Hilfe des Cast–Operators möglich:

`double` → `float` → `long` → `int` → `short`
`int` → `char` → `byte`
`short` ↔ `char`

Syntax:

(*Typname*) *Ausdruck*;

Beispiel:

```
float f;
short s = (short) f;
```

Achtung: Es existieren keine Konversionen von oder nach `boolean`, allerdings ist folgendes möglich:

Beispiel:

```
boolean b, int i;
b = i != 0;  // wandelt Null in false, Eins in true um.
i = b?1:0;   // wandelt false in Null, true in Eins um.
```

Aus Effizienzgründen wird nicht in den Datentypen `byte`, `short` und `char` gerechnet, sondern in `int`.
Bei gemischten Ausdrücken wird im größeren Typ gerechnet.

Beispiel:

```
byte a, b;
byte c = (byte) (a + b); // Cast, da (a+b) vom Typ int
```

6.7 Priorität der Operatoren

Priorität	Operator	Assoz.	Bedeutung
15	`.`, `[]`, `()`	L	Komponentenzugriff bei Klassen, Feldern und Methodenaufruf
14	`++`, `--`, `+`, `-`, `!`, `~` `new`	R	Unäre Operatoren Instanzbildung
13	(*Typ*)	R	Explizite Typkonvertierung
12	`*`, `/`, `%`	L	Multiplikative Operatoren
11	`+`, `-`	L	Additive Operatoren
10	`<<`, `>>`, `>>>`	L	Schiebeoperatoren
9	`<`,`>`,`<=`, `>=` `instanceof`	L	Vergleichsoperatoren
8	`==`, `!=`	L	Vergleichsoperatoren
7	`&`	L	Und–Operator (bitweise, logisch)
6	`^`	L	Exklusives Oder (bitweise, logisch)
5	`|`	L	Oder–Operator (bitweise, logisch)
4	`&&`	L	(Logisches Und)–Operator (Kurzschluß–Auswertung)
3	`||`	L	(Logisches Oder)–Operator (Kurzschluß–Auswertung)
2	`?:`	R	Konditionaloperator
1	`=`, `+=`, `*=`, usw.	R	Zuweisungsoperatoren

Die höchste Priorität wird mit der Zahl 15 gekennzeichnet, die niederste mit der Zahl Eins. Operatoren mit gleicher Priorität werden von links nach rechts ("Assoz." = L = linksassoziativ) bzw. von rechts nach links ("Assoz." = R = rechtsassoziativ) ausgewertet. Ausßerdem kann die Reihenfolge der Operationen immer durch Klammern `()` festgelegt werden.

7 Anweisungen

Anweisungen sind die eigentliche "Befehle", die vom Computer ausgeführt werden sollen.

7.1 Ausdrucksanweisung

Syntax:

Ausdruck;

Beispiele:

```
a = 1;
a++;
y = Math.sin(x);
```

Achtung: Die folgenden Beispiele sind syntaktisch korrekt, aber sinnlos:

```
1;
1 + 2 + 3;
```

7.2 Einfache Ausgabeanweisung

Syntax:

`System.out.println(` *Ausdruck* `);`

Der Ausdruck muss nicht unbedingt vom Typ `String` sein, sondern kann zunächst auch von einem anderen Typ sein, der dann in eine Zeichenkette konvertiert wird.

Achtung: Zeichenketten können mit Hilfe der *Stringkonkatenation* + verknüpft werden. Allerdings besteht dabei die Gefahr der Verwechslung mit der Addition!

Beispiele:

```
System.out.println ("Ergebnis:"+1+2);   // ergibt 12
System.out.println ("Ergebnis:"+(1+2)); // ergibt 3
System.out.println (1+':'+2);           // ergibt 61
System.out.println (""+1+':'+2);        // ergibt 1:2
System.out.println (1+":"+2);           // ergibt 1:2
```

- `System.out.println ();` beendet nach Ausgabe die Zeile, die nächste Ausgabe erfolgt in einer neuen Zeile (vgl. Pascal: `writeln`).
- `System.out.print ();` beendet die Zeile nicht, die nächste Ausgabe erfolgt in der gleichen Zeile (vgl. Pascal: `write`).

7.3 Einfache Eingabeanweisung

Eingabeanweisungen sind in Java nur schwer zu realisieren. Deshalb wird in der Vorlesung eine Klasse zur Verfügung gestellt, die das Einlesen von der Konsole ermöglicht. Dazu muss zu Beginn eines jeden Quelltextes mit Hilfe der `import`–Anweisung die Klasse `Prog1Tools.IOTools` bekannt gemacht werden.
Zusätzlich muss einmalig das Verzeichnis, in dem sich die Klasse befindet, den Java–Suchpfaden hinzugefügt werden (vgl. Webseiten der Vorlesung).

Beispiele: Einlesen einer ganzen und einer Gleitkommazahl.

```
import Prog1Tools.IOTools;
...
int a    = IOTools.readInteger (" a = ");
double b = IOTools.readDouble (" b = ");
```

7.4 Verbundanweisung

Syntax:

Anweisung 1 ... *Anweisung n*

Mittels des geschweiften Klammerpaars `{ }` können mehrere Anweisungen zu einer *Verbundanweisung* zusammengefaßt werden.
Die Anweisungen werden dabei in der angegebenen Reihenfolge ausgeführt.

Beispiel:

```
{ x = 1; y = 2; z = 3; }
```

Achtung: Zu einer Ausdrucksanweisung gehört immer ein Semikolon!

7.5 Bedingte Anweisung

(a) **Einseitige bedingte Anweisung:**

Syntax:

`if (` *Bedingung* `)` *Anweisung1*

(b) **Zweiseitige (doppelseitige) bedingte Anweisung:**

Syntax:

`if (` *Bedingung* `)` *Anweisung1* `else` *Anweisung2*

Bei der Bedingung in den runden Klammern handelt es sich um einen boolschen Ausdruck. Ergibt der Wert des Ausdrucks `true`, so wird die erste Anweisung (*Anweisung1*) ausgeführt. Ergibt der Wert des Ausdrucks `false`, so wird bei der einseitigen bedingten Anweisung nichts, bei der zweiseitigen bedingten Anweisung die zweite Anweisung (*Anweisung2*) ausgeführt.

Beispiel: Bestimmung des Maximums zweier ganzen Zahlen `x` und `y`.

```
int x = 21; int y = 12;
if (x>y) max = x; else max = y;  // ergibt: max = 21
```

Achtung: Möchte man anstelle einer einzigen Anweisung mehrere Anweisungen in den `if`– bzw. `else`–Zweigen ausführen, so verwendet man dafür eine Verbundanweisung!

Existieren keine Klammern, so wird jedes `else` immer dem nächsten davorstehenden `if` zugeordnet. Diese Zuordnung kann durch eine geeignete Klammerung geändert werden.

Beispiel:

```
int x = 5, y = 6; z = 4;
if (x>y)
   if (x>z) max = x;
   else max = z;        // Zuordnung zum zweiten if
...
if (x>y)
   {if (x>z) max = x;}
   else max = z;        // Zuordnung zum ersten if
```

7.6 Auswahlanweisung

Syntax:

```
switch ( Ausdruck ) {
case konstanter Ausdruck1: Anweisungsfolge1
case konstanter Ausdruck2: Anweisungsfolge2
...
default: Anweisungsfolge
}
```

Der Ausdruck darf vom Typ `byte`, `short`, `int` oder `char` sein, bei den konstanten Ausdrücken muss es sich um voneinander verschiedene zuweisungskompatible Konstanten handeln.
Stimmt der Ausdruck mit einem konstanten Ausdruck überein, so wird bei der zugehörigen Anweisungsfolge fortgefahren und die folgenden Anweisungsfolgen werden sequentiell abgearbeitet.
Der `default`–Zweig ist optional. Dieser wird ausgeführt, falls kein konstanter Ausdruck paßt.

Achtung: Die einzelnen Fälle müssen sich nicht ausschließen! Um die sequentielle Abarbeitung zu verhindern, muss den Anweisungsfolgen i. allg. eine `break`–Anweisung (siehe auch Abschnitt 7.10) folgen!

Beispiel:

```
switch (x) {
case  1:
case  2: x *= 4;
case  3: x *= 8;
case  4: x *= 16; break;
default: x *= 2;
}
```

Für `x=1` ergibt sich der Wert `512`, für `x=2` der Wert `1024`, für `x=3` der Wert `384`, für `x=4` der Wert `64` und für alle anderen Werte von `x` jeweils der Wert `2*x`.

7.7 `for`–Schleife

Syntax:

`for` (*Initialisierung*; *Ausdruck*; *Update*) *Anweisung*

- Alle drei Teile (Initialisierungs–, Ausdrucks–, Update–) sind optional. Allerdings müssen in den runden Klammern insgesamt immer zwei Semikolons vorhanden sein.
- Der Initialisierungsteil wird einmal ausgeführt und dient zur Deklaration von lokalen Variablen. Er kann eine Liste von Ausdrücken zur Initialisierung von Zählern enthalten (alle mit Komma getrennt).
- Der Ausdrucksteil ist vom Typ `boolean`. Solange sich der Wert `true` ergibt, wird die Anweisung und anschließend der Updateteil wiederholt. Die Schleife wird beendet, wenn der Ausdruck `false` ist.

- Der Updateteil enthält eine Liste von Ausdrücken (alle mit Komma getrennt), welche der Reihe nach ausgeführt werden.

Achtung: Fehlt der Ausdrucksteil, so gilt die Bedingung immer als erfüllt. Es ergibt sich eine Endlosschleife, die mit einer `break`–Anweisung verlassen werden muss!

Beispiele:

```
int i;
for (i=0; i<10; ++i) System.out.println (i);
    // 0 1 ... 9, Var. i gilt ueber Schleifen-Ende hinaus.
for (int j=0; j<10; ++j) System.out.println (j);
    // 0 1 ... 9, Variable j gilt nur bis Schleifen-Ende.
for (int j=2, k=3; j+k<27; j+=1; k+=3)
    System.out.println (j); // verwirrend, aber korrekt.
for (int j=0; j<100; j+=2) System.out.println (j);
    // 0 2 4 ... 98
for (int j=9; j>=0; --j) System.out.println (j);
    // 9 8 ... 0
```

Achtung: Das nachstehende Beispiel ist nur mit Vorsicht zu geniessen!

```
for (double x=0; x<=1; x+=0.1) System.out.println (x);
```

Durch die inkrementelle Erhöhung in Schritten von $0,1$ kann es zu Rundungsfehlern kommen. Es ist hier deshalb nicht klar, ob die Zahl Eins erreicht wird oder nicht!

7.8 `while`–Schleife

Syntax:

`while (` *Bedingung* `)` *Anweisung*

Äquivalent:

`for ( ;` *Bedingung* `; )` *Anweisung*

Ist die Bedingung in den runden Klammern wahr, so wird die Anweisung solange wiederholt, bis die Bedingung falsch ist.

Achtung: Bei der Anweisung kann es sich natürlich wieder um eine Verbundanweisung handeln!

Beispiel: Berechnung der Quersumme einer natürlichen Zahl

```
import Prog1Tools.IOTools;

public class Quersumme {
   public static void main (String [] args) {
      int n, Summe = 0;
      n = IOTools.readInteger("n = ");
      while (n>0) {
         Summe += n % 10;
         n     /= 10;
      }
      System.out.println("Quersumme = "+Summe);
   }
}
```

7.9 do–Schleife

Syntax:

do *Anweisung* while (*Bedingung*);

Im Prinzip wie die while–Schleife, allerdings wird hier zuerst die Anweisung ausgeführt und dann die Bedingung geprüft. Das bedeutet, daß bei einer do–Schleife im Gegensatz zu einer while–Schleife die Anweisung immer mindestens einmal ausgeführt wird.

Achtung: Die Bedingung hat genau die gegenteilige Wirkung wie in Pascal bei repeat ... until.

Beispiel: Vergleich von do– und while–Schleife.

```
int n = 5; m = 5;
do n /= 2; while (n>=10);  // ergibt n=2
while(m>=10) m /= 2;       // ergibt m=5
```

Beispiel: Einlesen einer positiven Zahl.

```
do p = IOTools.readInteger (" p = "); while (p<=0);
```

7.10 break–Anweisung

Syntax:

```
break;
```

Die break–Anweisung steht innerhalb der Auswahl–Anweisung oder innerhalb von for–, while– und do–Schleifen. Diese werden bei Erreichen der break–Anweisung sofort verlassen.
Es existiert auch eine Variante mit zugehöriger Marke, d. h. beim Erreichen der break–Anweisung wird die markierte Anweisung verlassen. Diese Variante kann auch bei anderen Anweisungen (z. Bsp. Verbundanweisungen) eingesetzt werden.

Beispiel:

```
draussen:                       // gesetzte Marke
for (int i=0; i<10; ++i)        // aeussere Schleife
   for (int j=0; j<100; ++j) { // innere Schleife
      if (...) break;           // verlaesst innere Schleife
      if (...) break draussen; // verlaesst die mit "draussen"
                                // markierte aeussere Schleife
   }
```

Achtung: Dieser Programmierstil ist verwirrend und macht den Quelltext schnell unübersichtlich und fehleranfällig!

7.11 continue–Anweisung

Syntax:

```
continue;
```

Die continue–Anweisung steht nur innerhalb von for–, while– und do–Schleifen. Dabei wird bei Erreichen der continue–Anweisung der aktuelle Schleifendurchlauf sofort beendet und der nächste begonnen.
Analog zur break–Anweisung existiert auch hier eine Variante mit zugehöriger Marke, an die das Programm bei Erreichen der continue–Anweisung "springt", d. h. der aktuelle Schleifendurchlauf der markierten Schleife wird abgebrochen und der nächste begonnen.

Beispiele:

```
for (int i=0; i<10; ++i) {
   int a = IOTools.readInteger (" a = ");
   int b = IOTools.readInteger (" b = ");
   if (b==0) continue;
   System.out.println (a/b);
}
```

```
aussen:
for (int i=1; i<=4; ++i) {
   for (int j=1; j<=4; ++j) {
      if (j==2) continue;
```

```
            if (i==2) continue aussen;
            System.out.println (i*j); // Ausgabe: 1 3 4 3 9 12 4 12 16
        }
    }
```

Achtung: Wie die Verwendung der `break`–Anweisung ist auch die Verwendung der `continue`–Anweisung sehr verwirrend! Mit beiden Anweisungen sollte, wenn überhaupt, nur sehr sparsam umgegangen werden!

8 Felder

Häufig werden viele gleichartige Daten zusammen abgespeichert.

Beispiele: Vektoren, Matrizen, Tabellen, Monitorbilder als eine Matrix von Pixeln.

- Die Einträge in einem Feld heissen *Elemente* oder *Komponenten*.

Beispiel: Elemente a_0 bis a_{n-1}:

a_0	a_1	a_2	...	$a_n - 1$

- Alle Komponenten sind vom gleichen Typ (*Komponententyp*).
- Felder sind in Java Objekte, die dynamisch erzeugt werden, und sind im Prinzip nur eindimensional, können aber als Komponententyp einen anderen Feldtyp besitzen.

8.1 Vereinbarung von Feldern

Syntax:

$Komponententyp$ `[ ]` $Bezeichner$;

Alternativ ist auch noch die C++ Syntax möglich:

$Komponententyp\ Bezeichner$ `[ ];`

Achtung: Anders als in Pascal und C++ wird durch die Deklaration noch kein Speicher für die Komponenten reserviert, es existiert bislang nur eine Referenz auf das Feld!

Beispiele:

```
int[] a;       // math. Vektor mit ganzzahligen Komp.
double[] b, c; // math. Vektoren mit Komp. vom Typ double
double[][] m;  // Matrix: Feld von Feldern
```

8.2 Erzeugen von Feldern

Felder können auf die folgenden Weisen erzeugt werden:

(1) Mit dem Operator `new` bei der Initialisierung,

(2) durch eine Initialisiererliste bei der Deklaration,

(3) durch Zuweisung eines mit `new` erzeugten Feldes,

(4) durch eine Kombination aus (2) und (3).

Beispiele:

```
int[] a = new int[3];  // Fall (1)
int[] b = {1, 2, 3};   // Fall (2)
int[] c;
...
c = new int[3];        // Fall (3)
int[] d;
...
d = new byte[] {1,2,3} // Fall (4)
```

Achtung: Im Gegensatz zu C++ darf die Initialisiererliste auch Variablen enthalten! Im Fall (4) müssen die eckigen Klammern leer sein!

Die Länge eines Feldes kann mit einer ganzzahligen Variablen (allerdings nicht vom Typ `long`) angegeben werden.

Achtung: Das heißt natürlich nicht, daß sich die Feldlänge bei einer Änderung der Variablen ebenfalls ändert!

Beispiel:

```
int dim = 5;
int[] e = new int[dim];   // Feld e hat 5 Komp.
dim     = 10;             // Feld e hat immer noch 5 Komp.
```

8.3 Zugriff auf Feldkomponenten

Ist `i` ein `int`–Ausdruck, so kann mit

```
f[i]
```

auf das `i`-te Feldelement von `f` zugegriffen werden.

Beispiel:

```
int[] f = { 1, 2, 3 };
```

f[0]	f[1]	f[2]
1	2	3

Damit ergibt sich für die Komponente `f[0]` der Wert 1, für die Komponente `f[1]` der Wert 2 und für die Komponente `f[2]` der Wert 3.

Achtung: Die Indizierung der Feldelemente beginnt in Java immer bei Null! Für das obige Beispiel bedeutet dies, daß die Komponente `f[3]` nicht existiert!
Im Gegensatz zu C++ werden Zugriffe auf nichtexistente Feldelemente geprüft und führen zu einem Laufzeitfehler!

Die Länge eines Feldes kann durch

```
f.length
```

bestimmt werden.

Beispiel: Vektoraddition.

```
import Prog1Tools.IOTools;
public class AddVek {
   public static void main (String[] args) {
      double[] a, b;
      int dim = IOTools.readInteger ("Dimension = ");
       a = new double[dim];
       b = new double[dim];
       for (int i=0; i<a.length; ++i) {
          a[i] = IOTools.readDouble ("a [" +i+ "]:");
          b[i] = IOTools.readDouble ("b [" +i+ "]:");
       }
      for (int i=0; i<a.length; ++i)
         System.out.println("Summe["+i+"]:"+(a[i]+b[i]));
   }
}
```

8.4 Mehrdimensionale Felder

Mehrdimensionale Felder sind genau genommen eindimensionale Felder, deren Komponenten selbst wieder Felder sind. Wie schon bei eindimensionalen Feldern gibt es auch hier wieder mehrere Möglichkeiten, mehrdimensionale Felder zu initialisieren. Die Geläufigste ist die Erzeugung mittels des `new`–Operators. Daneben findet gelegentlich auch die Erzeugung mit Hilfe einer Initialisiererliste Anwendung.

Beispiele:

```
double[][][] nd_feld = new double[1][2][3];
double[][] quadrat   = { { 1, 2 }, { 3, 4 } };
double[][] dreieck   = { { 1 }, { 2, 3 }, { 4, 5, 6 } };
```

Achtung: Wie im dritten Beispiel erkennbar, darf die Zeilenlänge durchaus auch unterschiedlich sein!

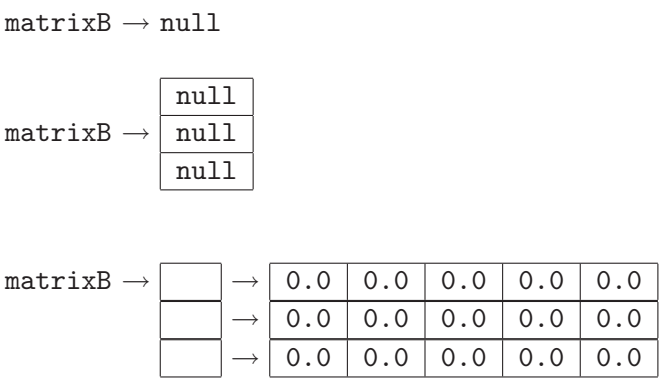
Die Dimensionsangaben dürfen ganzzahlige Ausdrücke sein (jedoch nicht vom Typ `long`), die Angabe der ersten Dimension ist notwendig, die weiteren können entfallen.

Beispiele:

```
double[][] matrixA = new double[3][];
double[][] matrixB = new double[3][5];
```

Im ersten Beispiel wird nur ein Vektor von Referenzen auf die Zeilen angelegt, die Zeilen selber existieren noch nicht! Beim zweiten Beispiel geschieht im Detail folgendes:

```
double[][] matrixB;
matrixB = new double[3][];
for (int i=0; i<3; ++i) matrixB[i] = new double[5];
```



8.5 Felder als Objekte, Referenzen

Im Gegensatz zu den Grundtypen (Zahlen, Zeichen und boolsche Typen) werden Felder wie alle Objekte als Referenzen behandelt.
Das bedeutet, daß eine Zuweisung über den Zuweisungsoperator = nur eine neue Referenz (einen *Aliasnamen* für das Feld) erzeugt, jedoch keine echte Kopie.

Achtung: Bei Kopien mit Hilfe des Zuweisungsoperators wird bei Grundtypen der Wert kopiert, bei Referenztypen nur die Referenz, nicht der Wert!

```
int   i = 1;
int   j = i;  // ergibt i=1, j=1
      i = 2;  // ergibt i=2, j=1
int[] a = {1};
int[] b = a;  // ergibt a[0]=1, b[0]=1
   a[0] = 2;  // ergibt a[0]=2, b[0]=2
```

`i` und `j` sind zwei unabhängige Variablen, `a` und `b` sind lediglich zwei unabhängige Referenzen.

Abhilfe schafft die Methode

```
clone()
```

Achtung: `clone()` bearbeitet nur die erste Ebene, bei mehrdimensionalen Feldern muss `clone()` mehrfach aufgerufen werden!

8.5.1 Kopieren von Feldern mit `System.arraycopy()`

Syntax:

`System.arraycopy (`*Quelle*, *Quellindex*, *Ziel*, *Zielindex*, *Laenge*`);`

Beispiel:

```
int[] a = {0, 1, 2, 3, 4, 5};
int[] b = new int[10];       // b[0]=0, ..., b[9]=0
System.arraycopy (a,2,b,4,3); // b[4]=2, b[5]=3, b[6]=4
```

b →

0	0	0	0	2	3	4	0	0	0

Achtung: Felder werden wie alle Objekte automatisch initialisiert!

8.5.2 Vergleich von Feldern

Mit dem logischen Vergleich `a==b` werden nur die Referenzen verglichen, nicht jedoch der Inhalt der Felder. Dasselbe gilt bei der Verwendung der Methode `equals()`.

Beispiel:

```
int[] a = {1};
int[] b = {1};
if (a==b) ...        // Bedingung ergibt false
if (a.equals(b)) ... // Bedingung ergibt false
```

9 Zeichenketten

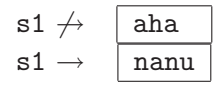
Im Gegensatz zu C sind Zeichenketten in Java Objekte und keine `char`–Felder. Dabei kann zwischen konstanten und variablen Zeichenketten unterschieden werden. Konstante Zeichenketten sind Objekte der Klasse `String`, variable Zeichenketten sind Objekte der Klasse `StringBuffer`.

9.1 Erzeugen von Zeichenketten

Zeichenketten lassen sich durch eine direkte Zuweisung oder durch Aufruf des Konstruktors der Klasse `String` erzeugen.

Beispiele:

```
String s1 = "aha";
String s1 = "nanu";
String s2 = new String ("aha");
```



- Zeichenfelder lassen sich nur durch Übergabe an den Konstruktor der Klasse `String` in Zeichenketten umwandeln. Eine direkte Zuweisung ist nicht erlaubt.
- Beliebige Typen können mit Hilfe der Methode `toString()` oder mit Hilfe der Stringkonkatenation in Zeichenketten umgewandelt werden.

Beispiele:

```
char[] c_array = {'a', 'h', 'a'};
String s3 = new String (c);  // ergibt "aha"
int i     = 123;
String s4 = "" + i;          // ergibt "123"
```

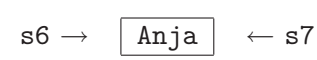
Achtung: Die Umwandlung von Zeichenketten in ganze Zahlen geschieht über spezielle Hüllklassen!

9.2 Operatoren für Zeichenketten

- Die Stringkonkatenation + verknüpft zwei Zeichenketten miteinander.
- Die Verknüpfung ist auch mit Hilfe des kombinierten Zuweisungsoperators += möglich.
- Ein Test auf Gleichheit bzw. Ungleichheit der Referenzen kann mit Hilfe von == bzw. != durchgeführt werden. Es findet dabei kein inhaltlicher Vergleich statt. Trotzdem kann dieser gutgehen, da der Kompiler nach Möglichkeit die gleiche Zeichenkette nur einmal abspeichert.

Beispiele:

```
String s5  = "Willi";
       s5 += "am";  // ergibt s5="William"
String s6  = "Anja";
       s7  = "Anja";
```



9.3 Methoden zur Bearbeitung von Zeichenketten

Die folgenden Methoden erfordern allesamt die objektorientierte Schreibweise:

Syntax:

Variable.*Methode* (*Argumente*);

Methode	Beschreibung	Argumenttyp	Ergebnistyp
`equals(s)`	Vergleicht Zeichenketten inhaltlich miteinander.	`String`	`boolean`
`length()`	Ermittelt Länge der Zeichenkette.		`int`
`charAt(i)`	Greift ein Zeichen raus.	`int`	`char`
`substring(i,j)`	Greit Teilzeichenkette raus	`i: int` `j: int`	`String`
`replace(c1,c2)`	Ersetzt Zeichen `c1` durch Zeichen `c2`.	`c1: char` `c2: char`	`String`
`indexOf(s)`	Liefert Index eines Zeichens oder einer Teilzeichenkette bzw. -1, falls nicht vorhanden.	`char, String`	`int`
`compareTo(s)`	Vergleicht Zeichenketten in lexikalischer Reihenfolge.	`String`	`int`

Beispiele:

```
String  s8 = "Alex";
String  s9 = "Alex";
if (s8.equals(s9)) ...            // ergibt true
```

```
String  s10 = "Achtung";
int    dim = s10.length();          // ergibt dim=7
String s11 = s10.substring(3,5);    // ergibt s11="tu"
String s12 = "blabla";
String s13 = s12.replace('a','u'); // ergibt s13="blublu"
int      i = s13.indexOf('b');      // ergibt i=0
int      j = s13.indexOf("blu");    // ergibt j=0
String s14 = "Anita";
if (s8.compareTo(s14)<0) ...        // ergibt false
```

Es gibt noch viele weitere Methoden (siehe auch Webseiten der Vorlesung und Java–Dokumentation).

10 Methoden

Methoden bilden das Analogon zu den aus Pascal, C und C++ bekannten *Funktionen* und dienen dazu, mehrfach vorkommenden Quelltext zusammenzufassen und Programme übersichtlicher zu gestalten.

In diesem Abschnitt werden zunächst nur statische Methoden, d. h. nicht objektorientierte Methoden, behandelt, die aus diesem Grunde mit dem Modifizierer `static` markiert werden.

10.1 Methodendefinition

Syntax:

Modifizierer Typspezifikation Methodenname
(*formale Parameterliste*) {
Deklarationen und Anweisungen
}

Die formale Parameterliste setzt sich jeweils aus der Typspezifikation gefolgt vom zugehörigen Bezeichner zusammen und wird durch Kommata getrennt:
Typ 1 Name 1, ..., *Typ n Name n*

- Die formale Parameterliste darf leer sein, die runden Klammern hingegen müssen stehen.
- Für jeden Parameter muß eine eigene Typspezifikation existieren.
- Feldtypen, Zeichenketten und beliebige andere Objekte sind als Parameter und Ergebnisse zugelassen. Bei Feldern muss die korrekte Anzahl von eckigen Klammern, jedoch ohne die Dimension, zusammen mit dem Basistyp angegeben werden.
- Die Rückgabe des Ergebnisses erfolgt über

  `return` *Ausdruck*;

  Der Ausdruck muß dabei zuweisungskompatibel zum Typ der Methode sein.
- Mit der `return`–Anweisung wird die Methode abgebrochen und das angegebene Ergebnis zurückgeliefert. Insgesamt muß ein `return` durchlaufen werden, sonst tritt ein Fehler auf.

  Achtung: Dies ist eine beliebte Fehlerquelle bei der Verwendung von bedingten und verzweigten Anweisungen!

- Eine Schachtelung von Methoden ist im Gegensatz zu Pascal und C++ nicht erlaubt. Methoden müssen stets auf äußerster Ebene in einer Klasse definiert werden.
- Wird kein Ergebnis benötigt, so ist der Ergebnistyp `void` und die `return`–Anweisung lautet schlicht:

  ```
  return;
  ```

  Diese Anweisung muß nicht explizit angegeben werden, sondern wird am Ende der Methode automatisch ausgeführt.

Beispiel: Methode zur Berechnung von x^n mit n ganzzahlig

```
static double power (double x, int n) {
   int m;
   double y = 1.0;
   if (n>=0) m = n;             // Absolut-Betrag von n
   else m = -n;
   for (int i; i<n; ++i) y*=x;  // Berechnung von x^n
   if (n>=0) return y;
   else return 1.0/y;
}
```

Der Aufruf erfolgt dann im Hauptprogramm über:

```
double y = power (a,5);
```

Die Methode `Math.pow()` (vgl. Abschnitt 6.1.4) bestimmt ebenfalls x^y, wobei y dann auch eine Gleitkommazahl sein darf.

10.2 Parameterübergabe, Methodenaufruf

Syntax:

Methodenname (*aktuelle Parameterliste*)

- Die aktuelle Parameterliste muß genauso viele Parameter besitzen wie die formale Parameter–Liste.
- Im Gegensatz zu Pascal und C++ werden alle Parameter einer Methode als *Werteparameter* übergeben.
- Die aktuellen Parameter werden von links nach rechts ausgewertet.
- Beim Aufruf der Methode wird das aktuelle Argument ausgewertet und das Ergebnis in einer lokalen Variablen mit dem Namen des formalen Parameters abgelegt, d. h. es wird stets mit einer Kopie der übergebenen Daten gearbeitet.

- Weiterhin muß der Methodenaufruf in einem Ausdruck des entsprechenden Typs erfolgen.
- Ist die formale (und damit auch die aktuelle) Parameterliste leer, so müssen die runden Klammern trotzdem stehen.
- Der aktuelle Parameter läßt sich bei einer Übergabe als Werteparameter nicht verändern. Für dieses Vorhaben gibt es in Java Referenztypen.

Beispiel:

```
static int f (int i, int j) {    // Definition von f
   return (i+j);
}
...                              // im Hauptprogramm:
int i = 1;
System.out.println (f (i++;i));  // Aufruf: f(1,2) ergibt 3
```

10.3 Referenztypen bei Methoden

Objekte, also auch Felder und Zeichenketten, sind Referenztypen. Auch sie werden per Wertaufruf übergeben, d. h. die Referenz wird in eine lokale Variable kopiert. Folglich kann die aktuelle Größe im aufrufenden Programm nicht verändert werden.

Achtung: Die Werte, auf die die Referenz verweisen, werden hingegen nicht kopiert und können deshalb verändert werden!

Beispiel:

```
static void demo1 (int[] x) {
   x[0]++;
}
...                   // im Hauptprogramm:
int[] y = new int[] {1, 2};
demo1 (y);            // Aufruf der Methode demo1
                      // x und y zeigen beide auf gleiche
                      // Speicheradressen! x=y={2,2}
```

Achtung: Dies entspricht aus den folgenden beiden Gründen nur teilweise einem Referenzaufruf von Pascal (`var`) oder C++ (`&`):

(1) Es besteht keine Wahlmöglichkeit. Grundtypen sind immer Werte, Objekte immer Referenzen.

(2) Auch bei Referenztypen kann das Objekt selber (d. h. die Referenz) nicht verändert werden, nur seine Komponenten.

Beispiel:

```
static void demo2 (int[] x) {
   x = new int[] {4, 5, 6};  // x komplett neues Objekt
   System.out.println (x[0]);
}
...                          // im Hauptprogramm:
int[] y = new int[] {1,2};
demo2 (y);                   // Aufruf der Methode demo2
System.out.println (y[0]);   // ergibt y[0]=1, x[0]=4
```

10.4 Überladen von Methoden

In Java können wie in C++ mehrere Methoden mit gleichem Namen in einer Klasse existieren.

Beispiel:

```
double max (double x, double y) { ... }
int    max (int    x, int    y) { ... }
```

- Die Methoden müssen unterschiedliche Parameterlisten besitzen.
- Der Ergebnistyp ist frei wählbar. Allerdings ist es nicht erlaubt, daß sich die Methoden nur in ihrem Ergebnistyp unterscheiden.
  → *Signatur*: Name und Parameterliste sind entscheidend.
- Im Gegensatz zu C++ gibt es keine separate Deklaration der Methoden.
- Lokale Variablen dürfen nicht den gleichen Namen wie ein formaler Parameter besitzen.
- Die aktuelle und formale Parameterliste müssen nicht die gleichen Bezeichner enthalten.

10.5 Hauptprogrammparameter

Das Hauptprogramm bei Java (Applets sind hiervon ausgenommen) ist eine Methode der Form

```
public static void main (String[] args) {
                 // args ist ein Feld von Zeichenketten.
```

- Argumente werden an der Kommandozeile mit Leerzeichen getrennt.
- Die Umleitung der Standardausgabe vom Bildschirm in eine Datei erfolgt durch

  > *Dateiname*

  entsprechend erfolgt die Eingabe über

  < *Dateiname*

  Achtung: Diese Angaben zählen nicht zur Argumentliste!

- Im Gegensatz zu C++ kommt der Programmname ebenfalls nicht in der Argumentliste vor.

Beispiele:

```
java MeinPrg Hallo Welt  // args[0]=Hallo, args[1]=Welt
java MeinPrg < Daten.dat > Ausgabe.txt Hallo Welt
                         // args[0]=Hallo, args[1]=Welt
```

10.6 Rekursion

Es werden bei Methoden zwei Rekursionsarten unterschieden:

(1) Bei der *direkten Rekursion* ruft sich die Methode selber auf,

(2) bei der *indirekten Rekursion* rufen sich zwei oder mehrere Methoden gegenseitig auf, d. h. Methode `a()` ruft Methode `b()` und umgekehrt.

- Lokale Variablen werden bei jedem rekursiven Aufruf erneut angelegt.
- Es handelt sich bei der Rekursion um ein sehr leistungsfähiges Werkzeug, allerdings muß darauf geachtet werden, daß die Rekursion nach endlich vielen Schritten endet.
- In vielen Fällen ist die Iteration mit einer Schleife effizienter.

Beispiel: Berechnung von x^n, $n \in \mathbb{N}$

$$x^n = \begin{cases} x^{n-1} \cdot x, & \text{falls } n > 0, \\ 1, & \text{sonst.} \end{cases}$$

Beispiel: Berechnung der *Fibonacci–Zahlen*

$$\begin{aligned} f_0 &= 1, \\ f_1 &= 1, \\ f_i &= f_{i-1} + f_{i-2} \quad \text{für } i \geq 2. \end{aligned}$$

```
public class Fibonacci {
   static int fibonacci (int i) {
      if (i<=1) return 1;          // nicht-rek. Teil
      else  return (fibonacci(i-1)+fibonacci(i-2));
                                   // rek. Teil
   }
   public static void main (String[] args) {
      for (int i=0; i<=100; ++i)
         System.out.println (fibonacci (i));
   }
}
```

Achtung: Ein großer Vorteil liegt hier in der direkten Umsetzung der mathematischen Formel! Allerdings ist dies in diesem konkreten Fall sehr ineffizient, da viele Werte mehrfach berechnet werden!

11 Objektorientierte Programmierung

11.1 Die Philosophie

Paradigmen:

(1) *Generalisierung*: Gemeinsame Strukturen von Objekten werden in Superklassen zusammengefasst.

(2) *Vererbung*: Eigenschaften einer Superklasse werden automatisch an eine Subklasse vererbt.

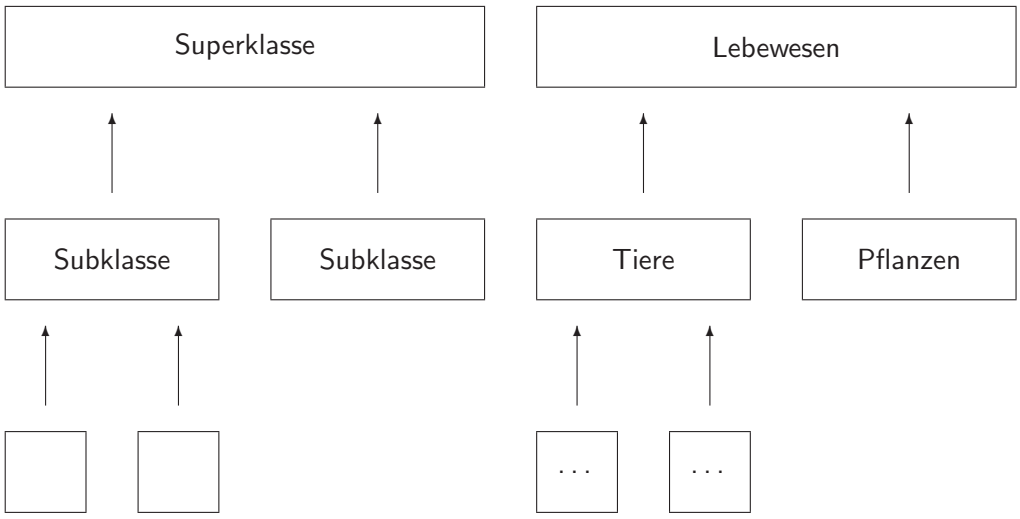
(3) *Daten-Kapselung*: Variablen und Methoden werden in einem Objekt zusammengefasst. Die interne Struktur wird vor dem Benutzer versteckt, der Zugriff erfolgt nur über genau definierte Schnittstellen.

(4) *Polymorphie*: Eine Methode kann in verschiedenen Subklassen individuell angepasst werden.

Vorteile:

- Es entsteht weniger Quelltext.
- Es bestehen weniger Fehlermöglichkeiten.
- Das Programm erhält eine bessere Strukturierung.
- Der Quelltext ist besser wartbar.

Beispiele:



11.2 Definition von Klassen

Syntax:

`class` *Bezeichner* {*Elementliste*}

Die Elementliste besteht aus *Datenelementen* (d. h. Variablen) und *Methoden* (d. h. Funktionen), die diese Daten berarbeiten (vgl. in Pascal: `record` und in C/C++: `struct`, `class`).

Zur Datenkapselung kann die Sichtbarkeit der Elemente mit Hilfe von vorangestellten *Modifizierern* geregelt werden.

nichts	Standard-Rechte, auch *friendly* genannt. Element kann u. a. in der Klasse selber sowie in allen Klassen des gleichen Pakets verwendet werden.
`public`	Element kann von jeder Klasse verwendet werden.
`private`	Element kann nur von der eigenen Klasse verwendet werden.
`protected`	später ...

- Datenelemente werden aus Gründen der Datenkapselung meistens mit dem Modifizierer `private` versehen.
- Die Klasse selber darf nur Standardrechte oder `public`–Rechte besitzen.
- In einer Datei dürfen mehrere Klassen definiert werden, jedoch darf nur eine davon `public`–Rechte besitzen. Diese Klasse muss den gleichen Namen wie die Datei tragen.

Beispiel: Nachfolgend wird der neue Datentyp `Point` definiert.

```
class Point {
   private double x,y; // Schutz vor unberecht. Zugriff
   public double getx () { return x; }
   public double gety () { return y; }
   void move (double dx, double dy) { x += dx; y += dy; }

   public static void main (String[] args) {
      Point p = new Point (); // Instanz der Klasse Point
   }                          // Aufruf des Standardkonstruktors
}                             // --> siehe Kapitel 11.4
```

Achtung: Unterscheidung zwischen Klasse und Instanz!

Klasse	Datentyp, es wird noch kein Speicher für Komponenten reserviert.
Instanz	Variable des Datentyps, es wird für Komponenten Speicher reserviert ($\longrightarrow$ *Instanzvariablen*, *Instanz–Methoden*).

Eine Ausnahme bilden Komponenten mit dem Modifizierer `static`. Diese werden für die gesamte Klasse nur einmal angelegt, unabhängig davon, ob keine, eine oder mehrere Instanz(en) der Klasse gebildet wurden (⟶ *Klassenvariablen*, *Klassenmethoden*).

Achtung: Im Gegensatz zu C++ ist keine separate Deklaration und Definition der Methoden möglich!

11.3 Zugriff auf Elemente

Elementvariablen und Elementmethoden werden auch unter dem Begriff *Komponenten* zusammengefasst.

Innerhalb der Klasse ist der Zugriff auf alle Komponenten erlaubt, und zwar einfach durch Angabe des entsprechenden Bezeichners. Wird der Bezeichner durch eine gleichnamige lokale Variable verdeckt, kann mit

`this.`*Bezeichner*

auf die Komponente zugegriffen werden. `this` ist dabei eine Referenz auf die momentan betrachtete Instanz.

Beispiel: Hinzufügen einer weiteren Methode zur Klasse `Point`

```
...
void moveWeit () {
   move (1000,1000);  // Aufruf der Methode move
}
```

Außerhalb der Klasse, d. h. in einer anderen Klasse, ist der Zugriff nur auf sichtbare Komponenten erlaubt. Diese sind folglich mit öffentlichen Zugriffsrechten bzw. für den Fall, daß sich die Klasse im gleichen Verzeichnis befindet, mit Standardrechten ausgestattet.

Syntax:

Klassenname.*Komponenten*	bei Klassenkomponenten
Instanzname.*Komponenten*	bei Instanzkomponenten

Beispiel: Erweiterung der Klasse `Point`

```
...                          // im Hauptprogramm:
System.out.println ("x-Koodinate von p" + p.getx () );
System.out.println (p.x);  // syntaktisch richtig,
                           // aber Zugriff verboten!
```

11.4 Konstruktoren

Bislang gibt es keine Möglichkeit, Werte für die Variablen `x` und `y` vorzugeben. Eine einfache Abhilfe schafft eine weitere Methode `init()`.

Beispiel: Hinzufügen der Methode `init()` zur Klasse `Point`

```
public void init (double x, double y) {
   // ggfs. pruefen, ob x und y im zulaessigen Bereich sind.
   this.x = x;
   this.y = y;
}
```

Nachteil: Die Methode `init` muss für jedes neue Objekt aufgerufen werden! Besser und komfortabler ist die Verwendung eines *Konstruktors*.

- Ein Konstruktor ist eine spezielle Methode ohne Ergebnistyp (auch nicht `void`), die den gleichen Namen wie die Klasse trägt.
- Konstruktoren können *überladen* werden, d. h. es dürfen mehrere Konstuktoren existieren.
- Konstruktoren ohne Argumente werden *Standardkonstruktoren* genannt. Werden keine Konstruktoren definiert, so erzeugt der Kompiler automatisch den Standardkonstruktor, andernfalls nicht.
- Konstruktoren können gegenseitig mitels `this` aufgerufen werden. Dieser Aufruf ist allerdings in jedem Konstruktor nur als erste Anweisung erlaubt.

Beispiel:

```
public Point (double x, double y) { // 1.Konstruktor
   this.x = x;
   this.y = y;
}
public Point () { // 2.Konstruktor = Standardkonstruktor
   x = 0;         // alternativ ueber Aufruf
   y = 0;         // des 1.Konstruktors: this (0,0);
}
public static void main (String[] args) { // im Hauptprogramm
   Point q = new Point (1,2);             // Aufruf des 1.Konstr.
   Point r = new Point ();                // Aufruf des 2.Konstr.
}
```

Mittels `new` wird der benötigte Speicher beschafft, und die angegebenen Anweisungen im Konstruktor ausgeführt (evtl. noch weitere Operationen).

11.5 Garbage Collection — `finalize`

Im Gegensatz zu Pascal (`new`, `delete`) und C++ (`new`, `delete`, Konstruktor, Destruktor) gibt es in Java im Wesentlichen kein Gegenstück zum Konstruktor. Der Speicher von nicht mehr referenzierten Objekten wird früher oder später automatisch freigegeben, im schlimmsten Fall erst am Programmende $\longrightarrow$ *Garbage collection*.

Syntax:

```
System.gc ();
```

Beispiel:

```
String s = ''blub'';
...
s        = null;      // Ref. auf ''blub'' geht verloren,
                      // sofern keine weitere Referenz ex.
```

Beispiel:

```
class Demo { };
...
Demo d = new Demo (); // Instanz der Klasse Demo
...
Demo d = new Demo (); // neue Instanz, alte geht vorloren.
```

Achtung: Selbst beim expliziten Aufruf des *Garbage collectors* muss der Speicher nicht komplett freigegeben werden!

Sind weitere Aufräumarbeiten wie das Zählen lebender Objekte, Abbau der Internetverbindung usw. nötig, dann kann man eine Methode `finalize()` definieren. Diese wird aufgerufen, bevor ein Objekt vom *Garbage collector* freigegeben wurde.

```
protected void finalize () throws Throwable {
   ...
}
```

Lebenszyklen von Klassen und Instanzen

(1) Ein Klasse wird "geladen" (z. B. von der Festplatte in den Hauptspeicher), sobald sie benötigt wird und wieder entfernt, sobald sie nicht mehr benötigt wird.

(2) Instanzen werden i. allg. mit `new` erzeugt. Sobald keine Referenz mehr auf die Instanz existiert, kann sie vom Java–System über die automatische *Garbage collection* wieder entfernt werden. Dies kann allerdings später oder auch gar nicht geschehen (siehe oben).

12 Vererbung

12.1 Grundlagen

Durch *Vererbung* werden von einer allgemeinen Klasse (*Superklasse*, *Basisklasse*) spezielle Klassen (*Subklassen*, *abgeleitete Klassen*) abgeleitet. Diese erben die Komponenten der Superklasse und besitzen ggfs. weitere Methoden und Variablen.

Syntax:

```
class Subklasse extends Superklasse {
    eigene Methoden und Variablen
}
```

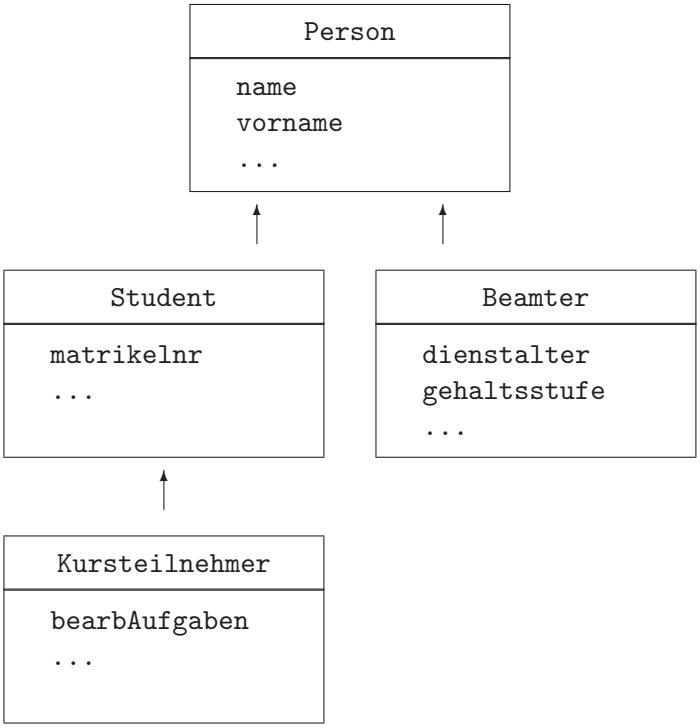
- Im Gegensatz zu C++ ist keine *Mehrfachvererbung* möglich, d. h. die Subklasse kann nur von einer Superklasse abgeleitet werden. Allerdings ist bei *Schnittstellen* Mehrfachvererbung wieder erlaubt.

- Es darf über mehrere Stufen hinweg vererbt werden. Man spricht dann von *indirekten* Sub– und Superklassen, andernfalls von *direkten*.

- Alle Klassen ohne Angabe von `extends` sind abgeleitet von der Superklasse `Object`.

- Konstruktoren, `static`–Initialisierer und `private`–Komponenten werden nicht vererbt.

Beispiel:



Neben den in Kapitel 11 schon erwähnten Modifizierern existiert zusätzlich der Modifizierer `protected`.
Der Zugriff auf die entsprechende Komponente ist dabei im gleichen Paket (d. h. i. allg. gleichen Verzeichnis) und in allen Subklassen erlaubt.

Achtung: Zugriffsrechte beziehen sich auf die Klasse, nicht auf die Instanzen und werden zur Übersetzungszeit geprüft, nicht erst zur Laufzeit!

12.2 Verdeckte Variablen

Gleichnamige Variablen in Sub– und Superklasse mit gleichem oder verschiedenem Typ sind erlaubt. Die Variablen aus der Superklasse bleiben erhalten, sie werden lediglich verdeckt, d. h. der direkte Zugriff aus der Subklasse heraus ist nicht möglich. Abhilfe schafft der Zugriff mit **Syntax:**

Klassenname`.`*Variablenname*	bei Klassenvariablen
Instanzname`.`*Variablenname*	bei Instanzvariablen und direkter Superklasse
`((`*Superklasse*`) this).`*Variablenname*	in beiden Fällen und auch bei indirekter Superklasse

Achtung: Zwischen Methodennamen und Variablennamen besteht kein Problem. Diese dürfen sowohl innerhalb einer Klasse als auch in Sub– und Superklasse den gleichen Namen tragen!

Beispiel:

```
class Super {
   int x = 1;
   int x () { return 2: }
}

class Sub extends Super {
  int x = 3;
  int x () { return 4; }
  void info () {
     System.out.println ("" + x + x()
                         + super.x + super.x());
  }
  ...  // im Hauptprogramm:
  Sub Instanz = new Sub ();
  Sub.info ();  // ergibt 3412
}
```

Achtung: Anstatt `super.x` kann man auch `((Super) this).x` verwenden!

12.3 Verdeckte Methoden

- Methoden können auch im Zusammenhang mit Vererbung *überladen* werden, d. h. sie tragen den gleichen Namen, aber eine andere Signatur.
- Hat in der Subklasse eine Methode den gleichen Namen und die gleiche Signatur wie in der Superklasse, so wird bei einer Objektreferenz automatisch die Methode der Subklasse verwendet (*Polymorphie*). Der Zugriff auf die Methode erfolgt dann über

  `super.`*Methodenname*

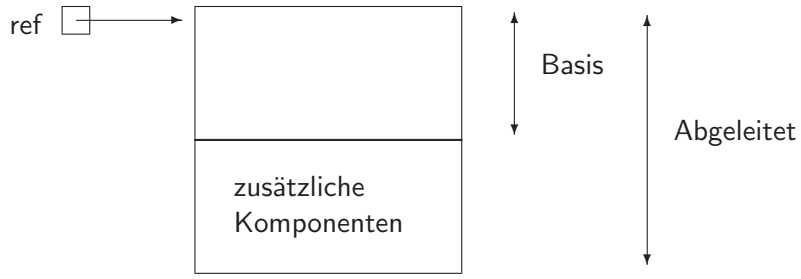
  Achtung: Außerhalb der Subklasse ist ein Zugriff nicht möglich, da der Typ von Klassen dynamisch zur Laufzeit bestimmt wird und somit eine Referenz der Superklasse auch auf die Subklasse verweisen darf!

Beispiel:

```
class Basis {
   void info () { System.out.println ("Basis"); }
}

class Abgeleitet extends Basis {
   void info () { System.out.println ("abgeleitet"); }
   ... // im Hauptprogramm:
   Abgeleitet a = new Abgeleitet ();
```

```
   a.info ();                          // abgeleitet
   ((Basis)a).info ();                 // auch: abgeleitet
   Abgeleitet b = new Basis ();        // Fehler!
   Basis      b = new Abgeleitet ();   // ok!
   b.info ();                          // abgeleitet
}
```



Damit gelten offenbar die folgenden Regeln für Konversionen zwischen Basisklasse und abgeleiteter Klasse:

(1) Ein Objekt der abgeleiteten Klasse kann automatisch in ein Objekt der Basisklasse konvertiert werden, da es alle notwendigen Komponenten enthält (*Up–Cast*).

(2) Dagegen kann ein Objekt der Basisklasse nicht automatisch in ein Objekt der abgeleiteten Klasse konvertiert werden, da Komponenten fehlen.
Ist es aber "in Wirklichkeit" ein Objekt der abgeleiteten Klasse, dann geht eine explizite Konversion mittels eines *Down–Cast*.

Beispiele:

```
Basis      b = new Basis ();
Abgeleitet a = b;              // Fehler!
Abgeleitet a = (Abgeleitet) b; // Fehler!
Abgeleitet a = new Abgeleitet ();
Basis      b = a;         // ok. implizite
                          // Konversion der Referenz
Abgeleitet c = (Abgeleitet) b; // ok. b referenziert
                          // Objekt vom Typ Abgeleitet
```

Beispiel:

```
class Basis {
   void info () { System.out.println ("Basis"); }
}

class Abgeleitet extends Basis {
   void info () { System.out.println ("abgeleitet"); }
}
```

```
class Haupt {
   public void static main (String [] args) {
      Basis [] a = {new Abgeleitet ();
                    new Basis ();
                    new Abgeleitet (); };
      for (int i=0; i<a.length; i++)
         a[i].info (); // ergibt: abgeleitet
                       //         Basis
                       //         abgeleitet
   }
}
```

Der Typ der Klasse, zu der die Methode `info()` gehört, wird erst zur Laufzeit bestimmt (*späte Bindung*).
Bei Methoden mit den Wortsymbolen `final`, `static` oder `private` erfolgt die Bestimmung bereits zur Kompilerlaufzeit (*frühe Bindung*) $\longrightarrow$ keine Polymorphie.

12.4 Konstruktoren und Vererbung

Konstruktoren werden nicht vererbt, allerdings kann der Konstruktor der direkten Superklasse aufgerufen werden.

Syntax:

`super (`*Parameterliste*`);`

- Der Aufruf muss als erste Anweisung im Konstruktor der Subklasse stehen.
- Ohne explizitem Aufruf wird am Anfang der Standardkonstruktor der Superklase aufgerufen, d. h. der Konstruktor der Subklasse wird ergänzt um

  ```
  super ();
  ```

  Das bedeutet, daß jeder Konstruktor implizit mit dem Aufruf von `super()` beginnt. Bei der Superklasse wird der Konstruktor der Klasse `Object` aufgerufen.

Beispiel:

```
class Basis {
   int x;
   Basis () { x = 5; } // Standardkonstruktor
}

class Abgeleitet extends Basis {
   int y;
    Abgeleitet () {  // Standardkonstruktor
```

```
        super ();      // Standardkonstruktor der
        y = 6;         // Superklasse
    }
    ...                // im Hauptprogramm:
    Abgeleitet a = new Abgeleitet ();
    System.out.println (a.x);  // ergibt 5
    System.out.println (a.y);  // ergibt 6
}
```

Achtung: Nicht erlaubt ist der Aufruf der Konstruktors der Superklasse innerhalb von bedingten Anweisungen, da dann der Aufruf nicht an erster Stelle erfolgt!

Ablauf der Instanziierung:

```
class Vater { int x = 1; }
class Sohn extends Vater { int y = 2; }
...
Sohn s = new Sohn ();
```

(1) Die Referenz wird angelegt und mit `null` initialisiert.

(2) Es wird Speicher für zwei Werte vom Typ `int` auf dem *Heap* (Freispeicher) beschafft.

(3) Die Komponenten `x` und `y` werden mit Null initialisiert.

(4) Den Komponenten `x` und `y` werden die Werte 1 bzw. 2 zugewiesen, und es werden ggfs. weitere Initialisiererblöcke ausgeführt.

(5) Der Konstruktor (hier: Standardkonstruktor) der Klasse `Sohn` wird aufgerufen. Dieser ruft als erstes den Standardkonstruktor der Basisklasse `Vater` und der wiederum den Standardkonstruktor der Klasse `Object` auf.

(6) Die Variable `s` erhält eine Referenz auf das neu erzeugte Objekt.

12.5 Klassen und `final`

Die Angabe von `final` ist bei Klassen, Methoden und Instanzen möglich.
Dies ist bei Klassen und Methoden sinnvoll, die nicht weiter spezialisiert werden können. Es erhöht die Effizienz, da eine frühe Bindung erfolgen kann. Das bedeutet, daß Methoden eventuell auch *inline* erzeugt werden (kein Aufruf, keine Parameterübergabe usw.).
Desweiteren bietet es einen Schutz vor *trojanischen Pferden*, die die Funktionalität der Klasse ersetzen könnten.

(1) Klassen mit dem Wortsymbol `final` dürfen nicht vererbt werden.

(2) Methoden mit dem Wortsymbol `final` dürfen nicht vererbt werden, d. h. bei einer Methode mit gleicher Signatur in der abgeleiteten Klasse tritt ein Fehler auf, dagegen ist eine Methode mit einer anderen Signatur erlaubt.

(3) Instanzen mit dem Wortsymbol `final` sind konstant und dürfen folglich nicht verändert werden.

Achtung: Die Komponenten der Instanz sind damit allerdings nicht automatisch konstant!

Beispiel:

```
class Punkt {
   int x, y;
   final void verschiebe (int dx, int dy) {
      x += dx;
      y += dy;
   }
}
class FarbPunkt extends Punkt {
   Color c;
   void verschiebe (Punkt p) {        // ok!
      x += p.x;                       // andere Signatur
      y += p.y;
   }
   void verschiebe (int dx, int dy) { // Fehler!
      x += dx;                        // gleiche Signatur
      y += dy;
   }
... // im Hauptpogramm:
   final Punkt p = new Punkt ();
   p    = new Punkt ();               // Fehler!
   p    = null;                       // Fehler!
   p.x  = 7;                          // Fehler!
}
```

12.6 Zugriffsrechte und Vererbung

Die Zugriffsrechte von Methoden dürfen beim Überschreiben in der abgeleiteten Klasse nicht reduziert werden. Die Reihenfolge lautet dabei:

`private` $\longrightarrow$ `default` $\longrightarrow$ `protected` $\longrightarrow$ `public`

Achtung: Die Umkehrung der Reihenfolge ist verboten, da aufgrund der Polymorphie ein Objekt der Basisklasse in Wirklichkeit zur abgeleiteten Klasse gehören kann! In diesem Fall müssen polymorphe Methoden zugreifbar bleiben!
Bei Variablen und überladenen Methoden existiert diese Einschränkung nicht!

Beispiel:

```
class Mutter {
   private void a () { }
```

```
    void b () { }
    protected void c () { }
    public void d () { }
}
class Tochter extends Mutter {
    // Ueberschreiben von Methoden der Superklasse Mutter
}
```

Folgende Varianten sind beim Überschreiben der Methoden aus der Superklasse `Mutter` möglich:

- `a()`: kein Überschreiben möglich, allerdings ist eine eine Neudefinition mit beliebigen Rechten möglich.
- `b()`: Überschreiben mit `default`, `protected` oder `public}` möglich.
- `c()`: Überschreiben mit `protected` oder `public` möglich.
- `d()`: Überschreiben mit `public` möglich.

12.7 Abstrakte Methoden und Klassen

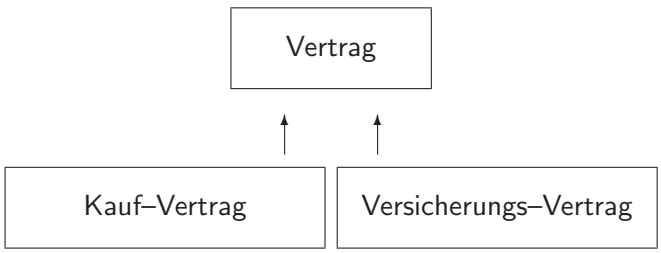
Syntax:

`abstract class` *Klasenname*
`abstract` *Modifizierer*

Achtung: Abstrakte Methoden dürfen keinen Rumpf besitzen!

- Ist eine Methode einer Klasse abstrakt, so muss auch die Klasse selber abstrakt sein.
- Von abstrakten Klassen dürfen keine Instanzen gebildet werden. Die Klassen dienen nur zur Vorlage bei der Vererbung.
- Die abgeleitete Klasse kann abstrakt, muß aber nicht, falls alle abstrakten Methoden durch konkrete Implementierungen überschrieben werden.

Beispiel:



Eine abstrakte Klasse kann auch als *Protokollklasse* aufgefaßt werden, sie definiert nur ein *Interface*, keine Implementierung.

Ähnlich zu abstrakten Klassen verhalten sich die beiden folgenden Fälle:

- Sind alle Konstruktoren `private`, so können ebenfalls keine Instanzen gebildet werden,
- Schnittstellen.

Literaturverzeichnis

[1] **J. Bishop**, *Java lernen*,
Addison Wesley, 2001, EUR 19,95

[2] **R. Cadenhead und L. Lemay**, *Java 2 in 21 Tagen*,
Markt & Technik Verlag, 2001, EUR 44,95

[3] **M. Campione und K. Walrath**, *The Java Tutorial*,
Addison-Wesley
`http://java.sun.com/docs/books/tutorial/information/download.htm`

[4] **G. Cornell und C. S. Horstmann**, *Core Java 2, Band 1 — Grundlagen*,
Sun Microsystems, 1999, EUR 49,95

[5] **M. Ernest, P. Heller und S. Roberts**, *Complete Java 2 Certification, Study Guide*,
Sybex, 2000, EUR 65 (?)

[6] **J. Goll, P. Müller und C. Weiß**, *Java als erste Programmiersprache*,
Teubner, 2001, 3.Auflage, EUR 36,00

[7] **R. Jesse**, *Java 2*,
bhv Verlag, 5.Auflage, EUR 16,95

[8] **M. Kopp und G. Wilhelms**, *Java professionell*,
mitp, 2000, EUR 50,62

[9] **G. Krüger**, *GoTo Java 2*,
Addison-Wesley, 2000, 2. Auflage, EUR 49,95
`http://www.javabuch.de`

[10] **D. Louis und P. Müller**, *Jetzt lerne ich Java*,
Markt & Technik Verlag, EUR 24,95

[11] **D. Ratz, J. Scheffler und D. Seese** *Grundkurs Programmieren in Java*
Hanser Verlag, 2.Auflage, EUR 29,90

[12] **M. Schader und L. Schmidt-Thieme**, *Java, Eine Einführung*,
Springer, 2000, 3. Auflage, EUR 37,95