Algorithmen und Datenstrukturen
2. Grundlagen der Programmierung (Teil 1): Variablen, Datentypen, Operatoren, Ausdrücke

Prof. Dr.-Ing. Marc Stamminger

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

DEPARTMENT INFORMATIK
LEHRSTUHL FÜR PROGRAMMIERSYSTEME
(INFORMATIK 2)

Worum geht es in dieser Lehreinheit?

- Um selbst Programme in einer Programmiersprache wie Java erstellen zu können, bedarf es einigen Handwerkszeugs.
- Um während des Programmablaufs Daten speichern zu können, z. B. um Zwischenergebnisse zu merken, benötigen wir *Variablen*.
- Da rechnerintern alle Daten durch Folgen aus 0 und 1 gespeichert werden, sind Variablen *Datentypen* zugeordnet, um solche Bitfolgen bspw. als ganze Zahl oder als Zeichen interpretieren zu können. Wir lernen einige solcher Datentypen und deren Repräsentation kennen.
- Datentypen sind *Operatoren* zugeordnet, bspw. kann man ganze Zahlen addieren oder vergleichen. Aus der Verknüpfung von Werten, Variablen und Operatoren lassen sich sogenannte *Ausdrücke* konstruieren, die bei der Programmierung großen Stellenwert haben. Dabei werden auch Fragen zur Typkonvertierung und -sicherheit erörtert.

Lernziele: Was sollen Sie am Ende dieser Lehreinheit können?

- Syntaxdiagramme lesen und zu einer gegebenen Aufgabenstellung erstellen können
- Durchlauf durch ein Syntaxdiagramm anhand gegebenen Programmstücks beschreiben können
- Variablen in Java-Notation deklarieren und ihnen Werte zuweisen können
- Zulässigkeit von z. B. Deklaration- und Wertzuweisungs-anweisungen überprüfen können
- Datentyp eines Ausdrucks bestimmen können
- Wert eines Ausdrucks bestimmen können
- mathematische Ausdrücke (Formeln) in Java-Notation überführen können
- faule und strikte Auswertung von Ausdrücken am Beispiel ausführen können

Gliederung der Lehreinheit

Problembeschreibung und Lösungsdarstellung am Computer

Problem → Wie kommt man möglichst komfortabel von einer Problembeschreibung zu einer ausführbaren Lösung? → am Computer ausführbare Lösung

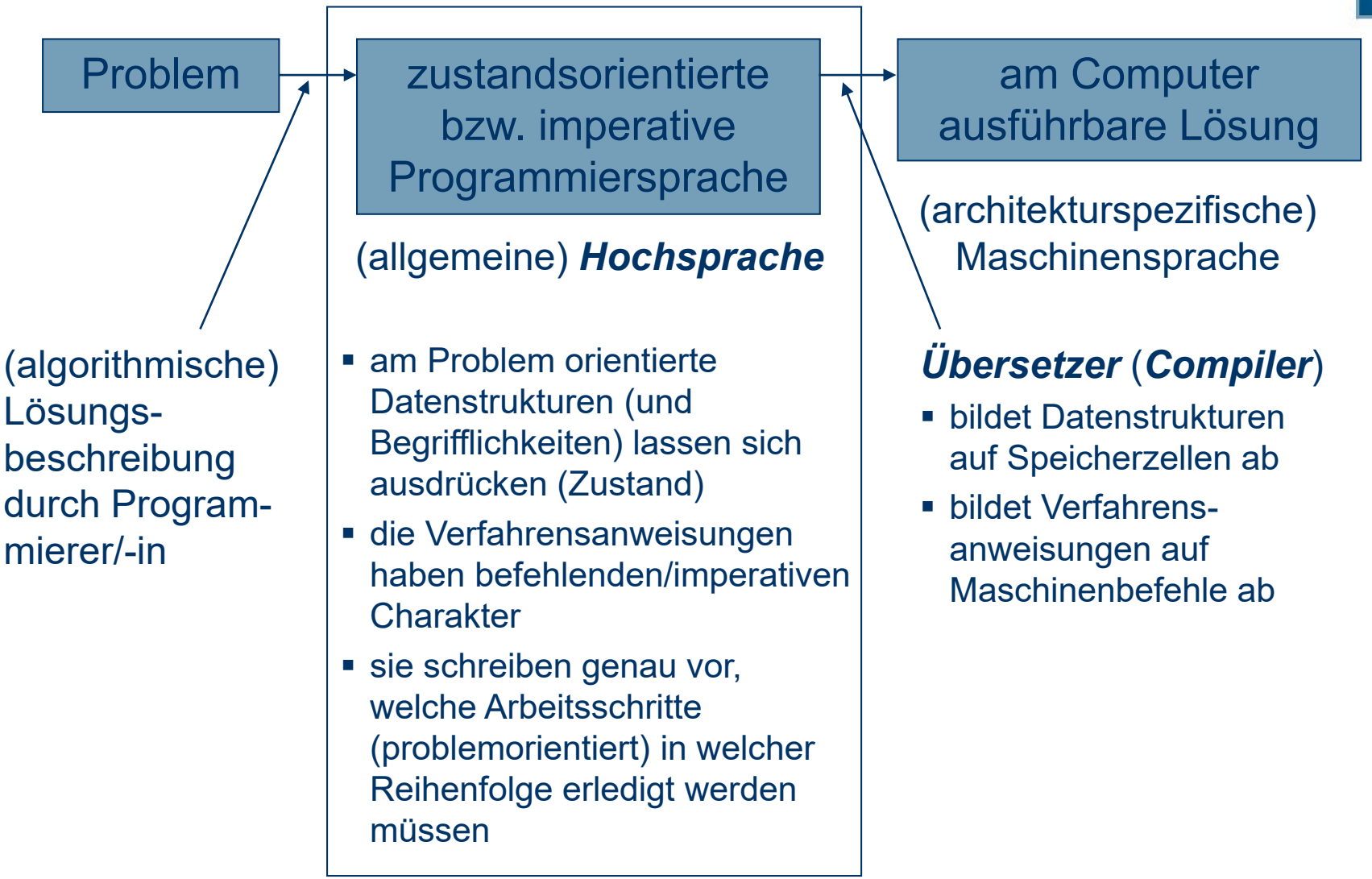
z. B. Minimum in Folge von ganzen Zahlen suchen

Algorithmus
- formuliert in Begrifflichkeiten der Problemdomäne
- wenn … kleiner als … ist, dann …
- solange … ist, führe aus …

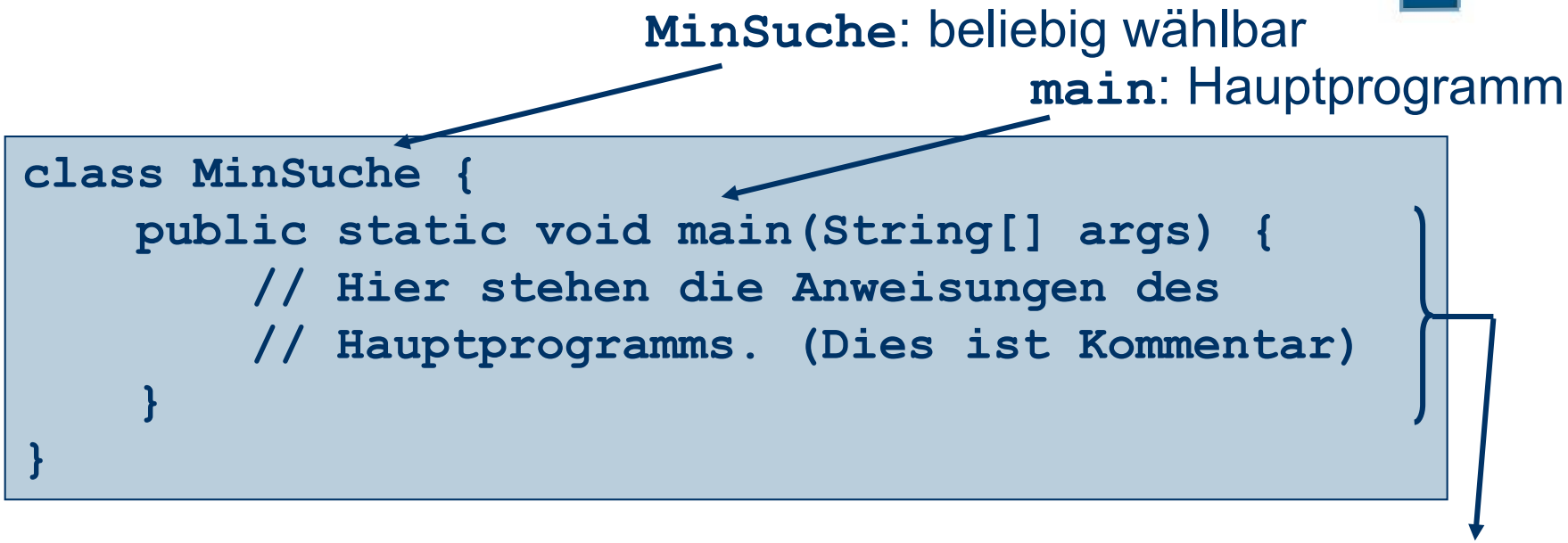
ausführbares Maschinenprogramm
- unten: Sicht auf ein Maschinenprogramm
- Repräsentation von Maschinenbefehlen im Speicher (spezifisch je Rechnerarchitektur)
- vom Menschen nur mit erheblichem Aufwand lesbar

```
00000000   77   90  192    0   52    0  248    0   64    0
00000010  110    4  110  164  154    6    0   64    0    0
00000020  103   11    0    0   28    0    0    0  151    0
00000030    0    0   65    1    0    0   79    1    0    0
```

Hochsprachen zur komfortablen Lösungsbeschreibung



Minimales Java-Programm



`class`, **`public`**, **`static`**, **`void`**:

- Vertreter der reservierten ***Schlüsselwörter*** der Programmiersprache Java
- dürfen ***nicht*** mit anderer Bedeutung belegt werden, insbesondere nicht als Bezeichner, z. B. Variablennamen, verwendet werden

Hinter **`class MinSuche`** folgt ***Block*** mit eigentl. ***Programm*** (sog. ***Rumpf*** der Klasse): in **`{...}`** enth. ***Anweisungen***.

Schlüsselwörter von Java

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

Übersetzung von Java-Programmen

Hochsprache

Sprache basierend auf der operationalen Semantik realer Rechner (Java-Bytecode)

Umgebung zur Ausführung des übersetzten Programms (Java-Bytecode-Interpreter, „Java Runtime Environment“)

- Bei der Übersetzung (***zur Übersetzungszeit***) eines Java-Programms wird es in gewissem Rahmen auf seine Korrektheit geprüft, z. B. syntaktische Struktur der Programms.
- Bei der Ausführung (***zur Laufzeit***) finden weitere Prüfungen statt, die erst beim Programmlauf bekannt werden können, z. B. Division durch 0 aufgrund einer falschen Benutzereingabe.
- Daher ist es wichtig, Übersetzungszeit und Laufzeit zu unterscheiden.

Konventionen für Java-Quelltextdateien und Dateinamen

- Groß- und Kleinschreibung sind relevant
- Anweisungen werden mit Semikolon `;` abgeschlossen
- ***Dateinamen von Java-Quelltextdateien***
  - ergeben sich aus dem Klassennamen, der als Wort hinter dem Schlüsselwort **`class`** steht, ergänzt um die Endung **`.java`**
  - also für das vorherige Beispiel:
    - **`MinSuche.java`**
    - **`javac MinSuche.java`** zum Übersetzen, erzeugt:
- ***Bytecode-Dateinamen***
  - ergeben sich aus dem Namen der Java-Quelltextdatei gemäß der Konvention, dass die Endung **`.java`** durch **`.class`** ersetzt wird
  - also für das vorherige Beispiel:
    - **`MinSuche.class`**
    - **`java MinSuche`** zum Ausführen

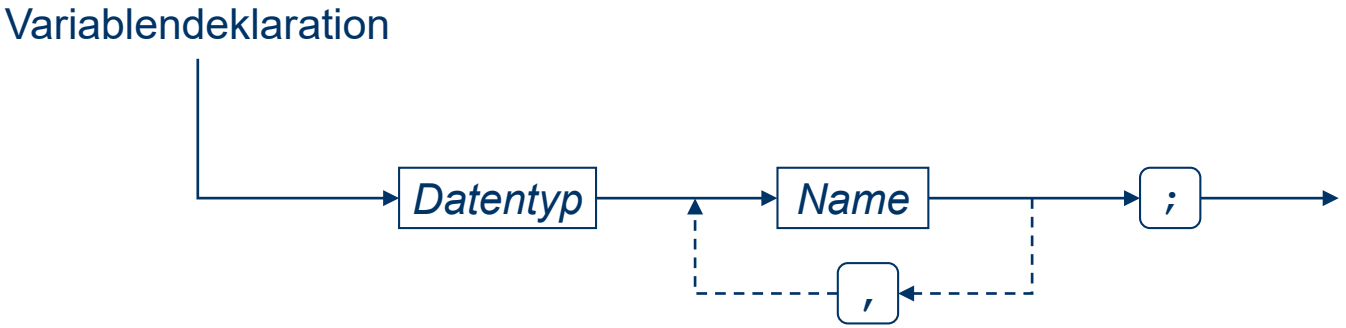
Gliederung der Lehreinheit

Variable

- bezeichnet Speichereinheit/-stelle zur Aufnahme von Datenwerten (z. B. (Zwischen-) Ergebnisse einer Berechnung)
- hat Namen und ***Datentyp***, der angibt, welcher Typ von Werten in ihr gespeichert werden kann
- muss vor ihrer ersten Verwendung ***deklariert*** (auch: vereinbart) worden sein
- Bei der ***Deklaration*** ist der Typ der Daten anzugeben, der in der Variable zu speichern ist.
- Beispiele für Datentypen (weitere folgen später):
  - **`int`**: ganze Zahlen, z. B. -3215, 0, 150, 395
  - **`double`**: Gleitpunktzahlen, z. B. -2.9e-7 (= $-2.9 \cdot 10^{-7}$), 0.0, 49.86, 3.14e5 (= $3.14 \cdot 10^{5}$)
  - **`char`**: Zeichen, z. B. 'a', 'Z', '7', '?'
  - **`String`**: Zeichenkette, z. B. "Algorithmen und Datenstrukturen", "Das Ergebnis der Berechnung ist: "

Variablendeklaration (I)

- legt Datentyp und Namen (Bezeichner) einer Variablen fest
- hat im einfachsten Fall die Form **`<Datentyp> <Name>;`**



- Beispiele:

```
int[] a;
int merker;
int i;
int n;
char z;
double d;
```

- alternative Schreibweise:

```
int merker, i, n;
```

Variablendeklaration (II)

- Bei Übersetzung des Programms wird Variable dadurch im Speicher erzeugt und entsprechender Speicherplatz reserviert.
- Erforderlicher Speicherplatz ergibt sich aus dem Datentyp.
- ist die Voraussetzung dafür, dass eine Variable in einem Programm verwendet werden kann.
- Da intern jede Information (z. B. Zahl, Zeichen) als Bitstring (aus 0 und 1) repräsentiert wird (wie, folgt gleich), dient Datentyp dazu, Speicherinhalt korrekt zu ***interpretieren***.

Benennung von Variablen

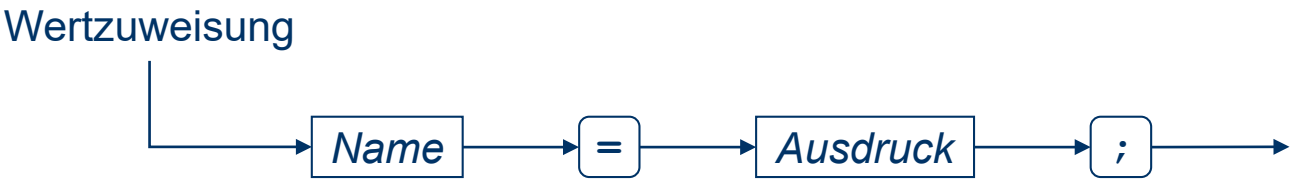
- ***Bezeichner*** (z. B. Variablenname) können aus kleinen und großen lateinischen Buchstaben, Ziffern, `_` und `$` bestehen. Keine Leerzeichen.
- Reservierte Wörter/Schlüsselwörter der Programmiersprache Java, wie z. B. **`class`** oder **`if`**, sind als Bezeichner ausgeschlossen.
- Konventionen/Programmierstil:
  - Variablennamen sollten mit Kleinbuchstaben beginnen.
  - Von der Verwendung des **`$`**-Zeichens wird abgeraten.
  - Namen sollten selbsterklärend sein (sog. ***mnemonische Namen***) und die Lesbarkeit eines Programms fördern (z. B. **`summe`** statt **`s`**).
  - Bei zusammengesetzten Substantiven sollte jedes neue Wort aus Gründen der Lesbarkeit mit einem Großbuchstaben beginnen.
- Beispiele:
  - gültige Variablenbezeichner: **`saldo`**, **`summeEinnahmen`**
  - *unerwünschter Variablenbezeichner:* ***`Summe`*** (Großschreibung)
  - *ungültige Variablenbezeichner:* ***`public`*** (Schlüsselwort), ***`1_anz`*** (Ziffer am Anfang), ***`summe Einnahmen`*** (Leerzeichen enthalten)

Anmerkungen zu den Hinweisen zum Programmierstil

- Motivation dafür:
  - Im Durchschnitt fallen 80% der Kosten eines Programms während der Wartung an.
  - Kaum ein Programm wird über seine ganze Lebensdauer vom ursprünglichen Autor betreut.
- Typische Problemquellen sind:
  - Verwendung von Techniken, die zwar syntaktisch korrekt, aber extrem schwer lesbar und damit fehleranfällig sind.
  - Mangelnde optische Gliederung von Programmtexten (z. B. durch geeignete Einrückungen von Programmteilen oder Leerzeilen zwischen inhaltlich zusammengehörigen Blöcken)
- Sog. ***Codier-Regeln*** dienen der Verbesserung der Lesbarkeit von Programmtexten und damit der Senkung der Wartungskosten.

Wertzuweisung an eine Variable (I)

- ***Wertzuweisung*** ist eine Operation, die dazu dient, Wert in Speichereinheit zu schreiben, die durch Variable bezeichnet ist *(**schreibender Zugriff**)*
- Sie hat die Form: **`<Name> = <Ausdruck>;`**



- Beispiele:

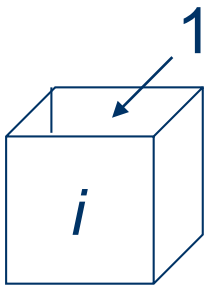
```
i = 1;
merker = a[0];
z = 'p';
name = "Meier";
```

(Lies: „Der Variable *i* wird der Wert 1 zugewiesen.“)

Behältermodell

`i = 1;` (Lies: „Der Variable *i* wird der Wert 1 zugewiesen.“)

Hilfreiche Vorstellung ***(Behältermodell)***:



durch die Zuweisung wird „das Element 1 in den Behälter mit der Beschriftung *i* gelegt“

zuvor dort gespeicherter Wert geht verloren, er wird ***überschrieben***

Wertzuweisung an eine Variable (II)

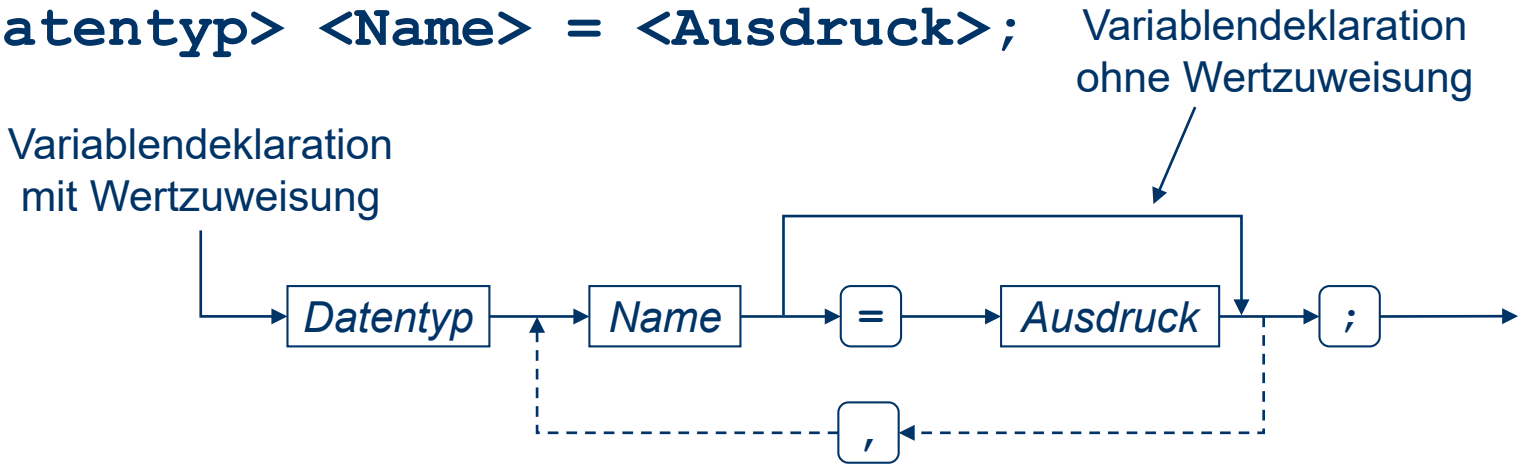
- Links vom ***Wertzuweisungsoperator***, dem „=“-Zeichen, steht Name der Variablen, rechts der Wert der zugewiesen werden soll.
- Wert kann auch ***Ausdruck*** (z. B. `(a+3)/5`, Details folgen) sein.
- Wert einer Variablen wird verwendet, indem Variablenname dort eingesetzt wird, wo ihr Wert benötigt wird *(**lesender Zugriff**).*
- Beispiel:

```
int a, b; // Deklaration der Variablen a und b
a = 2;    // Variable a erhaelt den Wert 2
b = a + 5;// Variable a wird gelesen (2), Addition
          // wird ausgewertet (7) und Ergebnis wird
          // der Variablen b zugewiesen
```

Wertzuweisung an eine Variable (III)

- auch möglich (und auch sinnvoll), einer Variablen bereits bei ihrer Deklaration einen Wert zuzuweisen
- Form:

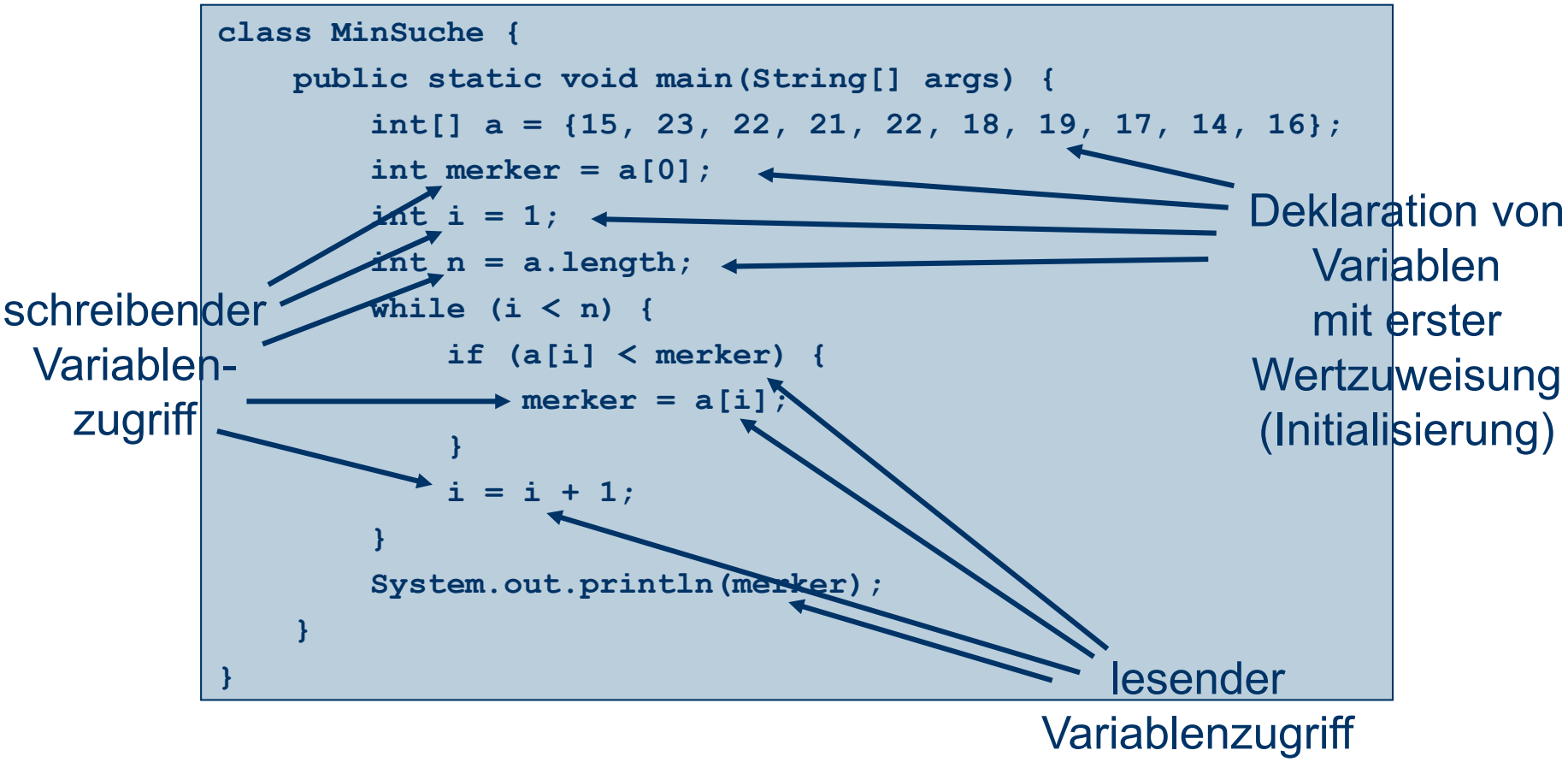
```
<Datentyp> <Name> = <Ausdruck>;
```



- Beispiele:

```
int[] a = {15, 23, 22, 21, 22, 18, 19, 17, 14, 16};
int merker = a[0];
char z = 'x';
int n = a.length;
```

Java-Programm `MinSuche`



Mögliche Fehlerquelle

- ***Vorwärtsverweise*** auf Variablen, die im Programmtext später deklariert werden, sind ***NICHT erlaubt***.
- Variablen können erst dann in einem Programm verwendet werden, wenn sie ZUVOR deklariert wurden.
- Fehlerbeispiel:

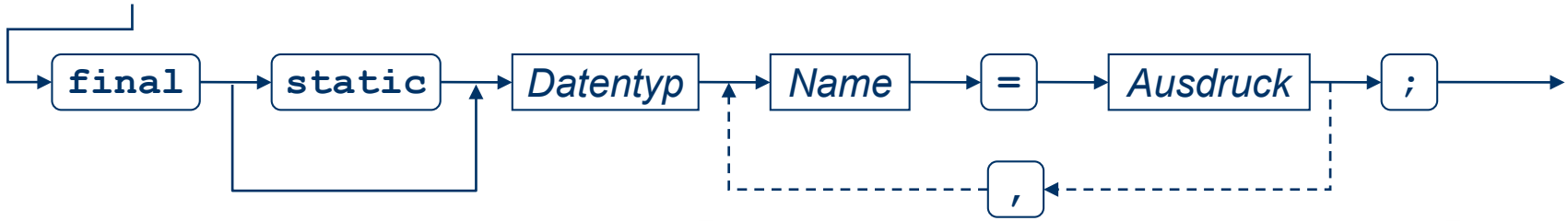
```
int i = j;  // fuehrt zu Fehlermeldung
int j = 1;
```

(Symbolische) Konstante (I)

- Wert, der sich zur Laufzeit eines Programms nicht ändern kann
- verwendbar wie Variablen, aber nur lesender Zugriff
- zwei Arten von Konstanten
  - statische (globale) Konstanten, überall im Programm verfügbar
  - (lokale) Konstanten, nur in bestimmtem Programmteil verfügbar (folgt später)
- Beispiele:

```
final static double PI = 3.1414;
final int MAXKUNDENNR = 1_000_000;
```

Konstantendeklaration mit Wertzuweisung

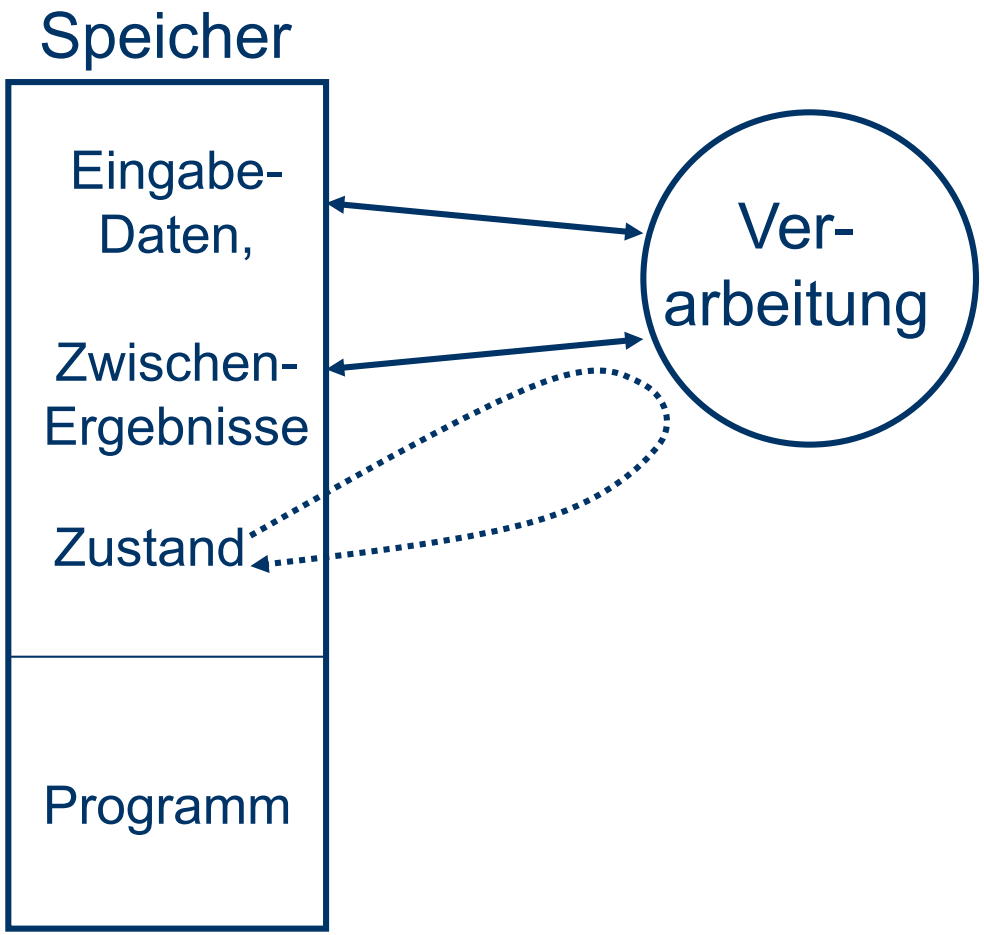


(Symbolische) Konstante (II)

- Zweck
  - häufig verwendete, konstante Werte in einem Programm als benannte Konstante deklarieren
  - wenn Wert geändert werden muss, reicht es aus, die Änderung einmal bei der Deklaration vorzunehmen und nicht an allen Stellen, an denen die Konstante verwendet wird
  - erleichtert die Änderbarkeit und Wartbarkeit eines Programms
- Konventionen
  - Konstanten werden am Anfang eines Blocks deklariert.
  - Bezeichner von Konstanten werden in GROSSBUCHSTABEN geschrieben.
  - Bei zusammengesetzten Bezeichnern Unterstrich _ zur Worttrennung verwenden.
  - Beispiele: **`PI`**, **`MIN_WIDTH`**, **`MAX_LENGTH`**, **`MAX_KUNDEN_NR`**

Gliederung der Lehreinheit

Programmgesteuerter Rechner (Babbage, 1833)

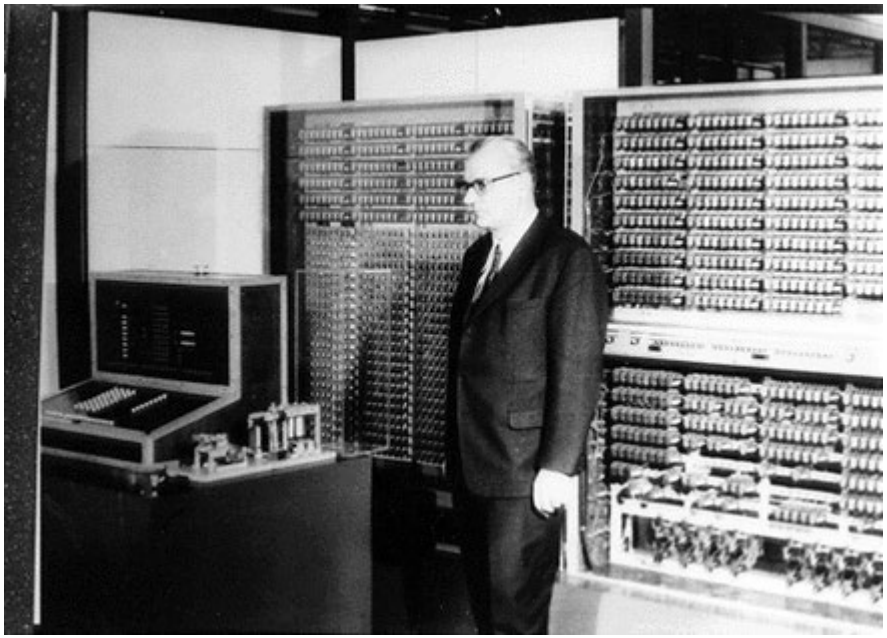


Idee der programm-gesteuerten Rechner: die Verarbeitung nicht ein für allemal festlegen, sondern die Verarbeitungsschritte durch ein austauschbares Programm variabel bestimmen

Programm extern (auf Lochkarten)

Von der Zuse Z3 zum Von-Neumann-Rechner

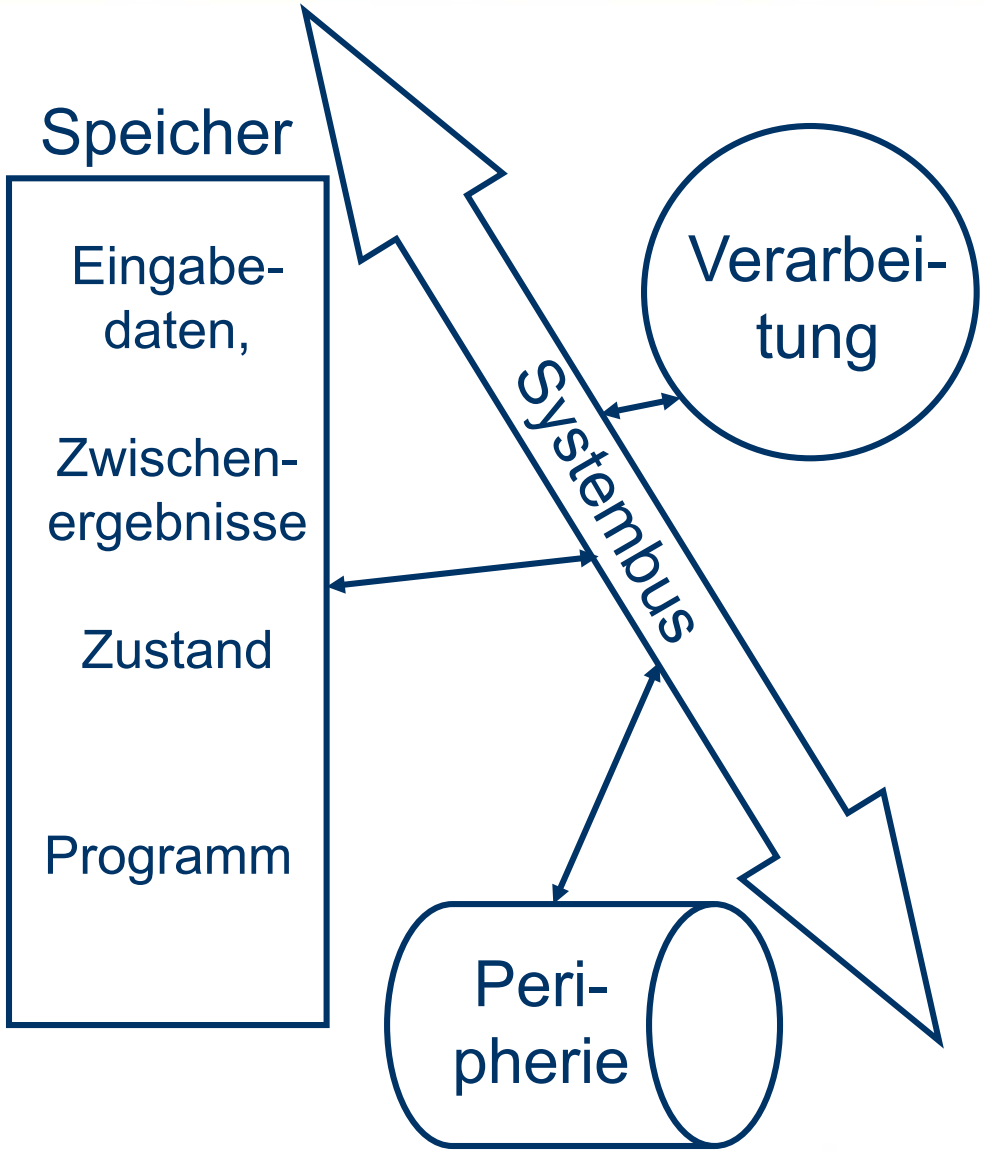
- Konrad Zuse (1910-1995) baute 1941 die Z3, den ersten elektromechanischen, programmgesteuerten Rechner der Welt



Nachbau der Z3 befindet sich im Deutschen Museum

- Die Mark II, die ENIAC und die Colossus folgten ab 1943.
- Von-Neumann baute ab 1946 den ersten elektronischen Rechner (Röhren statt Relais) nach gleichem Prinzip.

Von-Neumann-Rechner



Verarbeitung ist schneller als Speicherzugriff (10-100x)
Abhilfe:

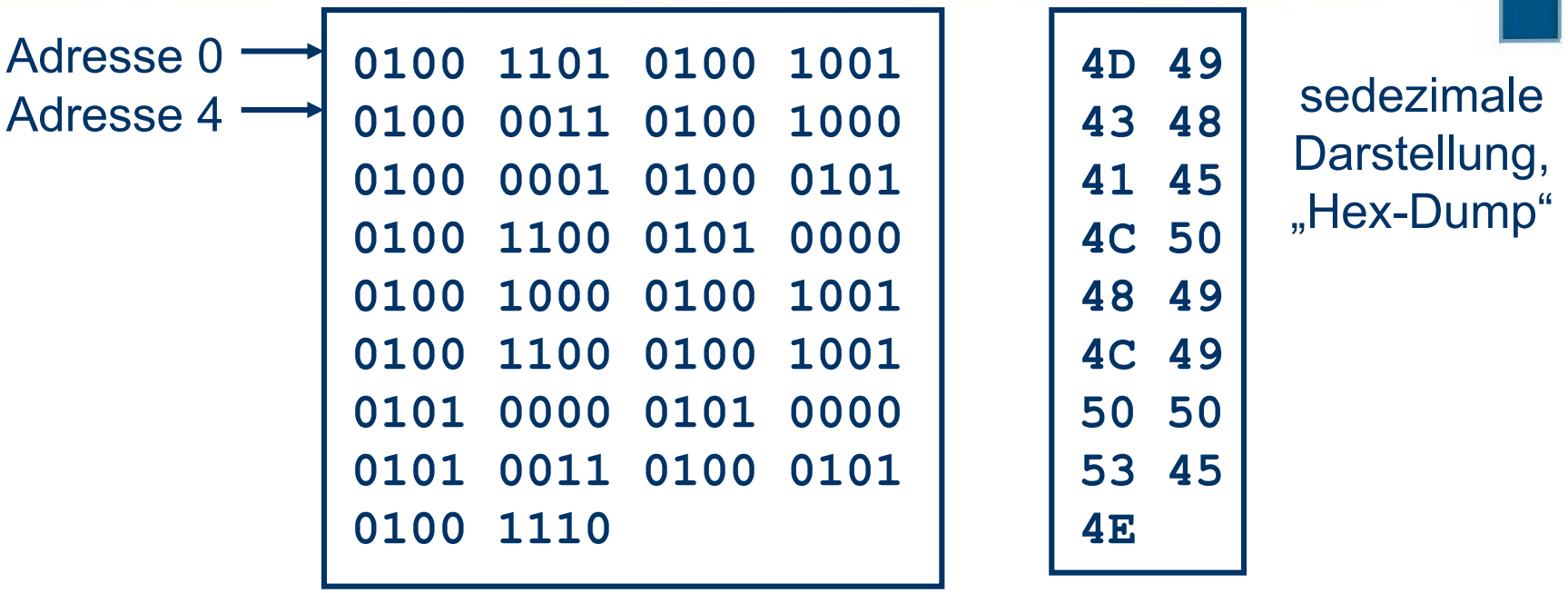
- Register: Speicherplätze in Verarbeitungseinheit
- Pufferspeicher (Cache)
- separate Daten- und Befehlsbusse

Im Speicher: Daten und Programm

Peripherie:

- Ein-/AusgabeGeräte (Drucker, Maus, Bildschirm)
- Band-, Plattenspeicher
- …

Speicher, konzeptionell



- nur mit Interpretationsvorschrift ist Menschen klar, was die Werte im Speicher bedeuten, z. B.
  - Binärdarstellung von Zahlen
  - binär codierter Zeichenvorrat (z. B. das Zeichen mit der Nr. 65)
  - „Nummern“ von Maschinenbefehlen
- alles das ist nicht „benutzerfreundlich“, deshalb Binärrepräsentation „verborgen“

Sedezimalsystem (auch: Hexadezimalsystem)

```
0100 1101 0100 1001
0100 0011 0100 1000
0100 0001 0100 0101
0100 1100 0101 0000
0100 1000 0100 1001
0100 1100 0100 1001
0101 0000 0101 0000
0101 0011 0100 0101
0100 1110
```

```
4D 49
43 48
41 45
4C 50
48 49
4C 49
50 50
53 45
4E
```

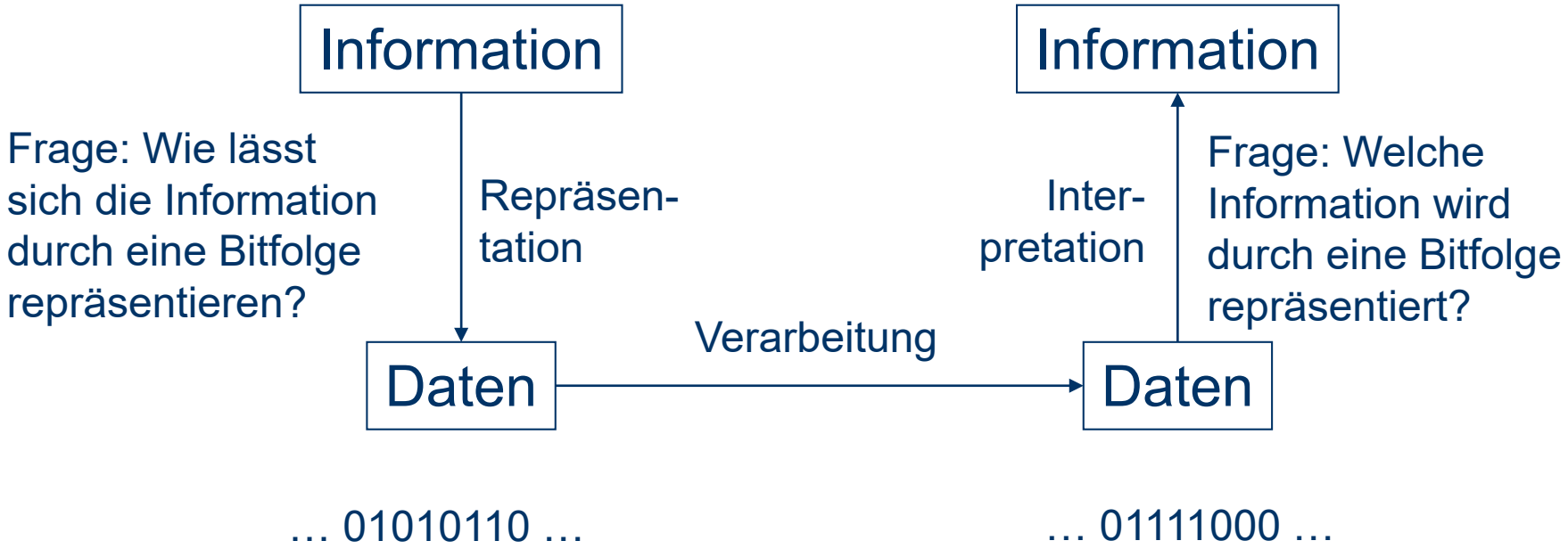
dezimal	dual	sedezimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	*0100*	*4*
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	*1101*	*D*
14	1110	E
15	1111	F

Information und Daten

… die Zeichenkette: "Die Vorlesung Algorithmen und Datenstrukturen …" … das Zeichen: 'x' …

die Gleitpunktzahl: 7.5

die Ganzzahl: 32156



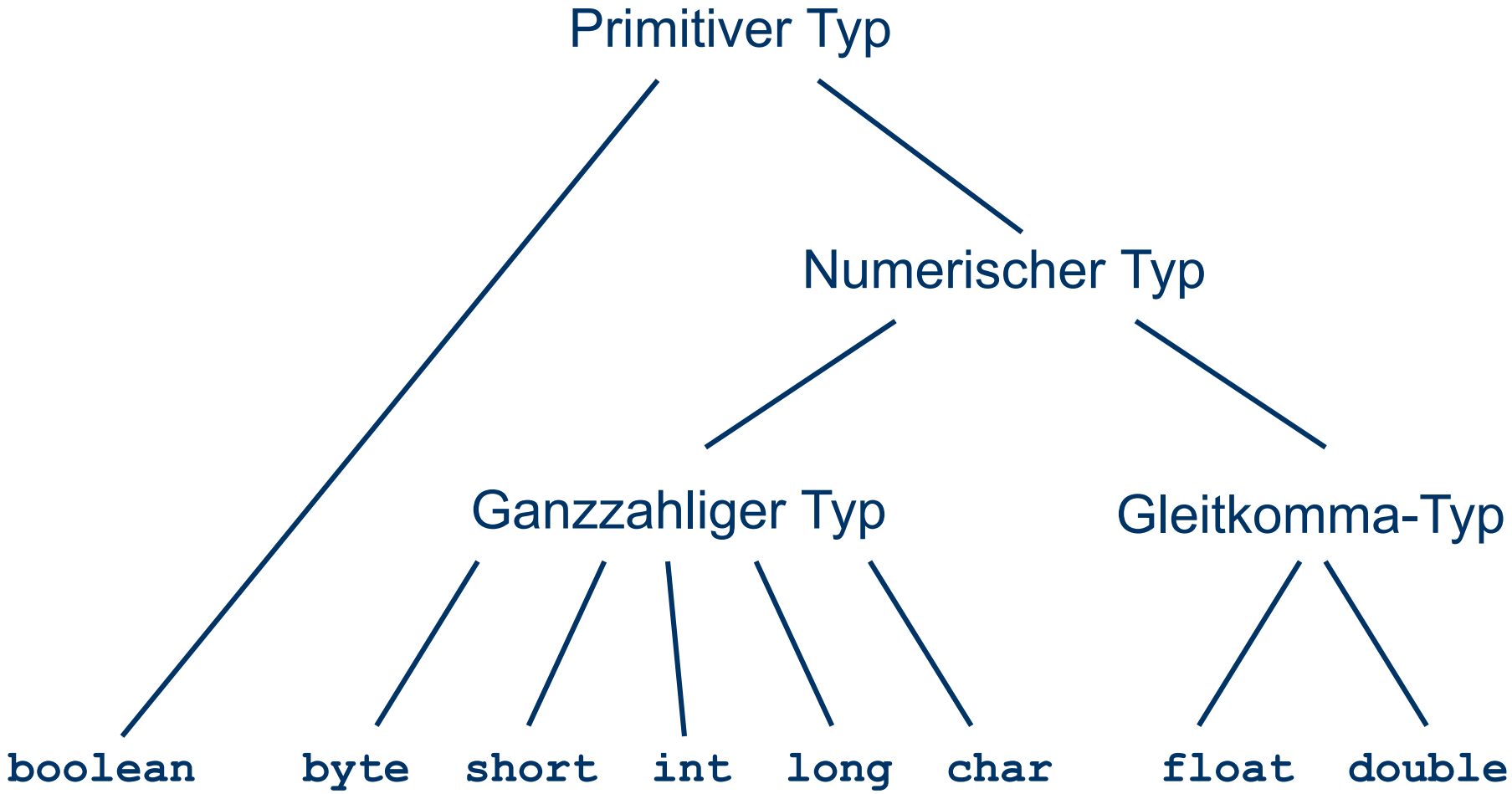
Interpretation von Daten

- Binärfolge gewinnt erst durch Interpretationsvorschrift an Bedeutung.
- Beispiel:
  - 100 1101 bedeutet *M* oder 77
  - Dateien auf Festplatte sind nur 0/1-Folgen; durch Endung des Dateinamens wird in der Regel die Bedeutung der Folgen festgelegt, z. B. **`test.pdf`** ist ein pdf-Dokument.
  - Ändern der Dateiendung (z. B. von **`.pdf`** nach **`.html`**) macht in der Regel die Datei unbrauchbar.
  - Linux-Kommando **`file`** versucht, den Typ einer Datei zu erkennen.

Datentyp

- ***Datentyp**:* Menge von Daten gleicher Art
- Beispiele für ***direkt verfügbare Datentypen*** in Java:
  - ganze Zahlen: **`byte`**, **`short`**, **`int`**, **`long`**
  - Fließkommazahlen: **`float`**, **`double`**
  - Zeichen: **`char`**
  - boolescher Wert: **`boolean`**
- Diese Datentypen werden als ***primitive Datentypen*** bezeichnet.
- Daneben: direkt verfügbare, ***zusammengesetzte Datentypen***, z. B. ***Arrays*** und ***Zeichenketten***
- Auch möglich: selbst ***neue Datentypen*** deklarieren, Details später in der Lehreinheit zur objektorientierten Modellierung und Programmierung.

Primitive (Daten-) Typen in Java: Überblick



Bit und Bitfolgen

- ***Bit***
  - kleinstmögliche Einheit der Information
  - Informationsmenge in einer Antwort auf eine Frage, welche zwei Möglichkeiten zulässt, z. B. ja – nein, wahr – falsch, hell – dunkel, …
  - Man benutzt dazu die Zeichen 0 und 1.
  - erforderlich, weil Information technisch dargestellt werden muss, z. B. 0: ungeladen – 1: geladen, 0: 0 Volt – 1: 5 Volt, 0: unmagnetisiert – 1: magnetisiert
  - Bit: Maßeinheit, bit: Einheit(szeichen) in Größenangaben
- ***Bitfolgen***, z. B. 10
  - erforderlich, wenn mehr als zwei Alternativen, z. B. 00: Süd, 01: West, 10: Nord, 11: Ost
  - Länge der Bitfolge steigt mit Zahl der Alternativen
  - Es gibt genau 2^N mögliche Bitfolgen der Länge N.
  - Spezialfall $N = 8$: 1 Byte = 8 Bits mit Einheitszeichen B

Ganze Zahlen (I)

Datentyp	Beschreibung	Wertebereich
`byte`	vorzeichenbehaftete Ganzzahlen von 8 bit Länge	$-2^7 = -128 \leq$ `byte` $\leq 127 = 2^7 - 1$
`short`	vorzeichenbehaftete Ganzzahlen von 16 bit Länge	$-2^{15} = -32\,768 \leq$ `short` $\leq 32\,767 = 2^{15} - 1$
`int`	vorzeichenbehaftete Ganzzahlen von 32 bit Länge	$-2^{31} = -2\,147\,483\,648 \leq$ `int` `int` $\leq 2\,147\,483\,647 = 2^{31} - 1$
`long`	vorzeichenbehaftete Ganzzahlen von 64 bit Länge	$-2^{63} = -9\,223\,372\,036\,854\,775\,808 \leq$ `long` `long` $\leq 9\,223\,372\,036\,854\,775\,807 = 2^{63} - 1$

Ganze Zahlen (II)

- Grund für die Verfügbarkeit mehrerer Datentypen:
  durch Auswahl des Datentyps Speicherplatz sparen,
  wenn der Wertebereich für die Anwendung ausreichend ist.
- Bei der Zahlangabe kann der Datentyp **`long`** durch Anhängen von **`L`** kenntlich gemacht werden (kleines **`l`** auch möglich, aber wegen Lesbarkeit lieber vermeiden).
- Beispiele:

```
int i = 0;
long x = 24L;
long y = 47_110_815L;
```

Literal: Zeichenfolge zur Darstellung eines Wertes eines Basistyps

die Ganzzahl 47.110.815 (seit Java7) _ zur besseren Lesbarkeit zwischen Ziffern in Literalen erlaubt.

- Frage: Was passiert, wenn der Wertebereich überschritten wird?

```
byte b = 125;
b += 3; // Kurzschreibweise für b = b + 3;
System.out.println(b); // Ausgabe: -128
```

Binärdarstellung natürlicher Zahlen

- gegeben: eine Folge von Bits

Position	*n-1*	*n-2*	...	*2*	*1*	*0*
Wert	z_{n-1}	z_{n-2}	...	z_2	z_1	z_0

Wortbreite n Bits

$z_i \in \{0,1\}$

- Diese ***Binärdarstellung*** entspricht dem dezimalen Wert $z = \sum z_i \, 2^i$.
- Beispiel:

1	0	0	1	0	1	
2^5	2^4	2^3	2^2	2^1	2^0	Wertigkeit der Position
32 +	0 +	0 +	4 +	0 +	1 =	37

Binärdarstellung negativer Zahlen: Vorzeichendarstellung

Frage: Wie stellt man negative Zahlen dar?

Intuitiver Ansatz: Absolutwert plus Vorzeichenbit (hier: 0: +, 1: -)
(sog. ***Vorzeichendarstellung***)

0000 = +0	1000 = -0
0001 = +1	1001 = -1
0010 = +2	1010 = -2
0011 = +3	1011 = -3
0100 = +4	1100 = -4
0101 = +5	1101 = -5
0110 = +6	1110 = -6
0111 = +7	1111 = -7

Nachteile:

- 2 unterschiedliche Repräsentationen für 0: +0, -0
- Rechnen wird kompliziert:

$$\begin{array}{rcl} -2 & = & 1010 \\ +\ +5 & = & +\ 0101 \\ \hline +3 & \neq & 1111\ (= -7) \end{array}$$

andere Darstellung erforderlich!

Zweierkomplement-Darstellung

- Fall $N = 4$:
  - mit 4 Bits kann man einen Bereich von $2^4 = 16$ ganzen Zahlen abdecken
  - Bereich frei wählbar, z. B. -8 bis +7
  - von 0 beginnend aufwärts zählen bis obere Grenze +7, dann von unterer Grenze -8 bis -1

1000 = -8	1100 = -4	0000 = 0	0100 = 4
1001 = -7	1101 = -3	0001 = 1	0101 = 5
1010 = -6	1110 = -2	0010 = 2	0110 = 6
1011 = -5	1111 = -1	0011 = 3	0111 = 7

- Wahl des Bereiches von -8 … +7 hat den Vorteil, dass das *erste Bit* wieder *Vorzeichenbit* ist.
- Rechnen mit Zweierkomplement in Vorl. *„Grundl. der techn. Inf.“*

Ganze Zahlen (III)

- Seit Java7 können Literale ganzer Zahlen auch in Binärschreibweise verwendet werden.
- Bei der Zahlangabe muss dann diese Schreibweise mit einem vorangestellten **`0b`** oder **`0B`** kenntlich gemacht werden.
- Beispiele:

```
// A 32-bit 'int' value (4711):
int posI = 0b1001001100111;

// A 32-bit 'int' value (-4711):
int negI = 0b11111111111111111110110110011001;

// A 64-bit 'long' value (4711081547110815). Note the "L" suffix:
long l = 0B0000000000010000101111001011010000100110000111111001100110011111L;

System.out.println(posI + "     " + negI + "     " + l);
```

Ausgabe des Programmfragments:

```
4711     -4711     4711081547110815
```

-Folien

- Sie sind eine sehr heterogene Gruppe bzgl. Ihres Vorwissens.
- Knobelfolien sollen „die Experten“ vor dem Einschlafen bewahren.
- Anfänger:
  - Zunächst ignorieren.
  - Später durcharbeiten, sobald Ihre Programmierfertigkeiten ausreichen!

Wer suchet, der findet

- Was wird ausgegeben?

```
public class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
                b < Byte.MAX_VALUE; b++) {
            if (b == 0x90)
                System.out.println("Treffer!");
        }
    }
}
```

`0x90` ist vom Typ `int`
10010000_2=128+16=144.
144 nicht in [-128;127]

- Vorschläge:
  - Einmal "Treffer"
  - Mehrmals "Treffer"
  - etwas anderes

Nichts!

→Vermeide Vergleiche von unterschiedlichen Typen!

Schau genau! (I)

- Was wird ausgegeben?

```
public class SchauGenau {
    public static void main(String[] args) {
        System.out.print(12345 + 54321);
        System.out.print(" ");
        System.out.print(01234 + 43210);
    }
}
```

Kleines `l` sieht der `1` ähnlich.
→ Lieber großes `L` schreiben!

Führende 0 leitet Oktalzahl ein.
$01234_8 = 668_{10}$
→ Vermeiden!

- Vorschläge:
  - 17777 44444
  - 17777 43878
  - 66666 44444
  - 66666 43878

Es sind nicht alles `int`-Werte

Einschub: Aufzählungstypen (engl. enumeration types) (I)

- sind gedacht für **`int`**-Variablen, die nicht jeden beliebigen Wert annehmen dürfen, sondern auf eine begrenzte Anzahl von Werten beschränkt sind
- Diese Werte werden über Namen angesprochen.
- Typische Kandidaten für solche Aufzählungen sind die Tage einer Woche, die Himmelsrichtungen, die Monate eines Jahres oder die Einheiten einer Währung.
- Beispiel zur Definition von Aufzählungstypen: Wochentage

```
enum Wochentag {
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
    FREITAG, SAMSTAG, SONNTAG
}
```

- Hinweis: da die Ausprägungen solcher Datentypen Konstanten sind, werden diese gemäß Konvention in Großbuchstaben geschrieben.

Einschub: Aufzählungstypen (engl. enumeration types) (II)

- Beachte: die Deklaration solcher Aufzählungstypen muss zum jetzigen Zeitpunkt immer innerhalb des Rumpfes der Klasse, die das Programm enthält, jedoch außerhalb der Deklaration von **`main`** erfolgen.

```
class OrientationTest {

    enum Orientation {
        NORTH, SOUTH, WEST, EAST
    }

    public static void main(String[] args) {
        Orientation dir = Orientation.NORTH;
        System.out.println(dir);
    }
}
```

Den einzelnen Literalen wird der Name des Typs mit einem Punkt vorangestellt.

Ausgabe des Programms (letzte Zeile): NORTH

Gleitkommazahlen (auch: Fließkommazahlen) (I)

- Ziel: approximative Darstellung einer reellen Zahl
  - ***Festkommazahl***: begrenzte Ziffernfolge der Länge n, Komma an festgelegter Stelle und damit $1 \leq k < n$ Vorkommastellen und $n - k$ Nachkommastellen. Gibt es in Java nicht.
  - ***Gleitkommazahl*** (***Fließkommazahl***, engl. floating point number): Darstellung der Zahl x mit zwei Werten, der Mantisse m (mit $1 \leq m < 10$) und dem Exponenten e: $x = \mp m \cdot 10^e$

Datentyp	Beschreibung	Wertebereich
`float`	Gleitpunktzahlen von 32 bit gemäß IEEE 754-1985	▪ kleinste `float`-Zahl >0: 1.40239846e-45 ▪ größte `float`-Zahl >0: 3.40282347e+38
`double`	Gleitpunktzahlen von 64 bit gemäß IEEE 754-1985	▪ kleinste `double`-Zahl >0: 4.9406564584124654e-324 ▪ größte `double`-Zahl >0: 1.7976931348623157e+308

Gleitkommazahlen (auch: Fließkommazahlen) (II)

- Beispiele:

```
float f1 = 0.0f;
float f2 = 1.7F;  //ohne f oder F ist es double-Wert
double d1 = -1.71e-19d;
double d2 = 2.718e17;
float pi = 3.14159_26535_89793_23846_26433_83279F;
double d = 123_456_789.012_345E-123;
```

- Hinweis: anstelle des Dezimalkommas wird in Java (und den meisten anderen Programmiersprachen) ein Punkt verwendet.

Wechselgeld (I)

- Was wird ausgegeben?

```
public class WechselGeld {
    public static void main(String[] args) {
        System.out.println(2.00 - 1.10);
    }
}
```

- Vorschläge:
  - Laufzeitfehler/Exception
  - 0
  - 0.9
  - eine andere Zahl

Es wird ausgegeben:
`0.899999999999...99`

Wechselgeld (II)

- Was wird ausgegeben?

```
public class WechselGeld {
    public static void main(String[] args) {
        System.out.println(2.00 - 1.10);
    }
}
```

- Problem: 1.1 ist *nicht exakt als* **`double`** *darstellbar.*
- **`float`** und **`double`** sind *immer gerundete Zahlen* und für absolut exakte Berechnungen ungeeignet.
- Behebungsmöglichkeiten:
  - **`new BigDecimal("2.00").subtract(new BigDecimal("1.10"))`**
  - In Cent rechnen und dafür Datentyp **`int`** verwenden.

→ Primitive Datentypen wohlüberlegt wählen!

Zeichen

- Rechner werden auch zur Textverarbeitung eingesetzt, Programme müssen Textausgaben für Interaktion mit Benutzer bereitstellen.
- Programmiersprachen stellen daher auch Datentypen zur Speicherung und Verarbeitung von Zeichen zur Verfügung.
- Datentyp **`char`** repräsentiert Menge der Zeichen.
- Literale: Zeichenkonstante wird durch Angabe des Zeichens in Hochkommata repräsentiert.
  Z. B. das Zeichen a als `'a'`, oder das Komma durch `','`
- Beispiele:

```
char z = 'x';
char komma = ',';
char grossesM = 77; //das Zeichen mit der Nr. 77
                    //siehe ASCII- bzw. Unicode-Tabelle
                    //folgt gleich
```

Binärdarstellung von Zeichen: ASCII-Code (7 bit)

ASCII-Code zur
Codierung von Zeichen
(128 Zeichen darstellbar)

Sedezimalsystem:
mit 4 Bits sind Zahlen 0-15 darstellbar

0110 1011

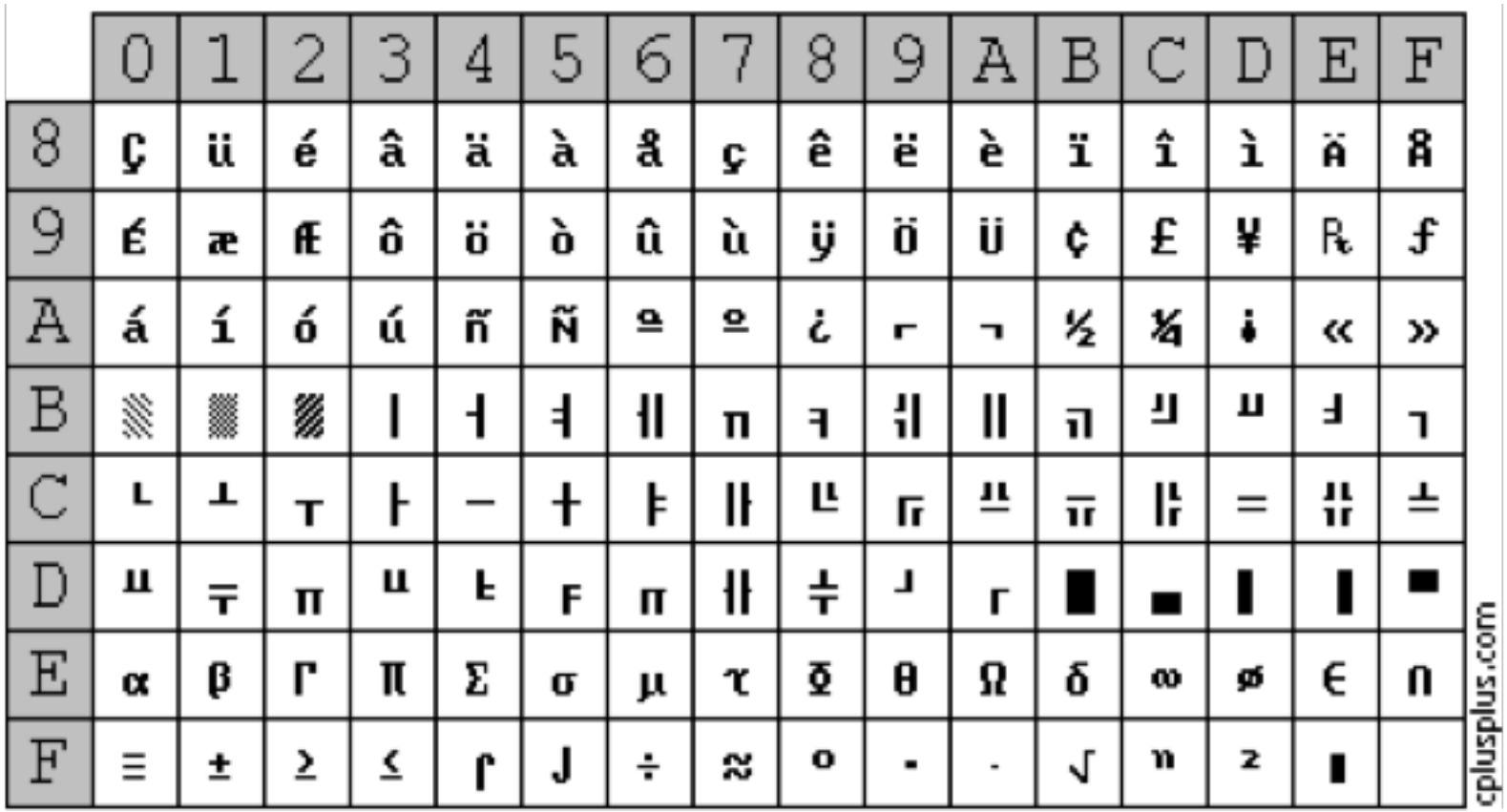
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ursprünglicher Zweck: Datenübertragung;
erste Zeichen (ASCII 0 bis ASCII 31) und
ASCII 127 dienen Signalisierungs- und
Steuerungszwecken

4D = 100 1101
= 77 (dezimal)

Binärdarstellung von Zeichen: Erweiterter ASCII-Code (8 Bits) – OEM Extended ASCII

Erweiterter ASCII-Code verwendet 8 Bits und erweitert den einfachen ASCII-Code um 128 weitere Zeichen:



Binärdarstellung von Zeichen: Erweiterter ASCII-Code (8 Bits) – ANSI Extended ASCII (Windows)

Erweiterter ASCII-Code verwendet 8 Bits und erweitert den einfachen ASCII-Code um 128 weitere Zeichen:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	□	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	□	□
9	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	□	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

cplusplus.com

Aufgrund der Vielfalt normierte die International Organisation for Standardization (ISO) verschiedene ASCII-Erweiterungen, diese als ISO 8859.

Binärdarstellung von Zeichen: Unicode (I)

- Problem: Vielfalt der ASCII-Erweiterungen
- Ziel: Zusammenfassen sämtlicher relevanter Zeichen verschiedener Kulturkreise in einem universellen Code
- sog. ***Basic multi-lingual plane of Unicode***, 16 Bit-Codierung, enthält Platz für 65536 Zeichen
- **`char`** kann ein solches 16-Bit-Zeichen aufnehmen; für Zeichen anderer „planes“ werden zwei **`char`**-Werte benötigt.
- enthält landesspezifische Zeichen, wie z. B. ä, ö, æ, ç, ebenso wie kyrillische, japanische oder tibetanische Schriftzeichen
- Details: www.unicode.org

Binärdarstellung von Zeichen: Unicode (II)

Code Charts | Look up by character code: Go | Home | Site Map | Search

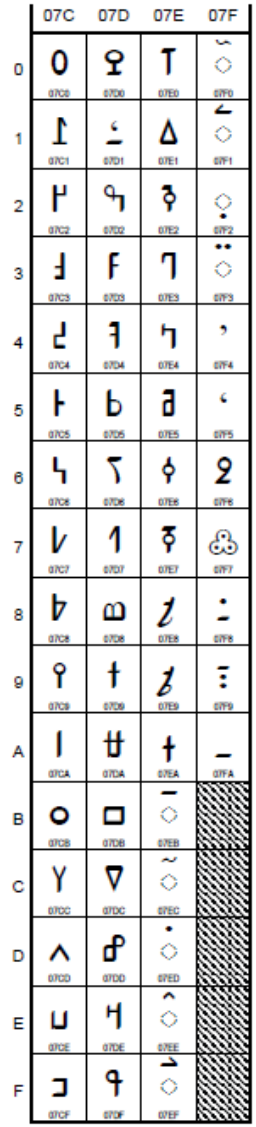
Unicode 6.0 Character Code Charts

SCRIPTS | SYMBOLS | NOTES

Related links: Name index Help & links

Scripts

European Scripts	African Scripts	South Asian Scripts	East Asian Scripts
Armenian	Bamum	Bengali	Bopomofo
Armenian Ligatures	Bamum Supplement	Brahmi	Bopomofo Extended
Coptic	Egyptian Hieroglyphs (1MB)	Devanagari	CJK Unified Ideographs (Han) (28MB)
Coptic in Greek block	Ethiopic	Devanagari Extended	CJK Extension-A (6.3MB)
Cypriot Syllabary	Ethiopic Supplement	Gujarati	CJK Extension B (30MB)
Cyrillic	Ethiopic Extended	Gurmukhi	CJK Extension C (2.8MB)
Cyrillic Supplement	Ethiopic Extended-A	Kaithi	CJK Extension D
Cyrillic Extended-A	N'Ko	Kannada	(see also Unihan Database)
Cyrillic Extended-B	Osmanya	Kharoshthi	CJK Compatibility Ideographs (.5MB)
Georgian	Tifinagh	Lepcha	CJK Compatibility Ideographs Supplement
Georgian Supplement	Vai	Limbu	CJK Radicals / KangXi Radicals
Glagolitic	**Middle Eastern Scripts**	Malayalam	CJK Radicals Supplement
Gothic	Arabic	Meetei Mayek	CJK Strokes
Greek	Arabic Supplement	Ol Chiki	Ideographic Description Characters
Greek Extended	Arabic Presentation Forms-A	Oriya	Hangul Jamo
Latin	Arabic Presentation Forms-B	Saurashtra	Hangul Jamo Extended-A
Latin-1 Supplement	Aramaic, Imperial	Sinhala	Hangul Jamo Extended-B
Latin Extended-A	Avestan	Syloti Nagri	
Latin Extended-B	Carian	Tamil	
Latin Extended-C	Cuneiform (1MB)	Telugu	



Quelle: www.unicode.org

Binärdarstellung von Zeichen: Unicode – Spezialzeichen

Für manche Zeichen des Unicodes gibt es besondere Schreibweisen:

Spezialzeichen (ASCII-Abkürzungen)	Unicode	Ersatzdarstellung
Rückschritt („backspace“, BS)	\u0008	\b
horizontaler Tabulator („TAB“, HT)	\u0009	\t
neue Zeile („line feed“, LF)	\u000a	\n
Seitenvorschub („form feed“, FF)	\u000c	\f
Wagenrücklauf („carriage return“, CR)	\u000d	\r
doppeltes Anführungszeichen	\u0022	\"
einfaches Anführungszeichen	\u0027	\‘
Rückstrich („backslash“)	\u005c	\\

erste Zeichen stimmen mit ASCII-Code überein

sog. ***Escape-Zeichen***: Zeichenkombinationen, die Sonderfunktionen ausführen

Nur weil ich es kann! (I)

- Was wird ausgegeben?

```
\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020\u0020
\u0063\u006c\u0061\u0073\u0073\u0020\u0055\u0067\u006c\u0079
\u007b\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020
\u0020\u0020\u0020\u0020\u0073\u0074\u0061\u0074\u0069\u0063
\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028
\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0020
\u0020\u0020\u0020\u0020\u0061\u0072\u0067\u0073\u0029\u007b
\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074
\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u0020
\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u0077\u0022\u002b
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u007d\u007d
```

Nur weil ich es kann! (II)

- Mit „normalem“ Zeichensatz:

```
public
class Ugly
{public
    static
void main(
String[]
    args){
System.out
.println(
"Hello w"+
"orld");}}
```

→Unicode nur verwenden, wenn es wirklich zwingend nötig ist, wenn also ein Zeichen ansonsten nicht dargestellt werden kann!

Wahrheitswerte

- Bedingung in Alternative oder Schleife kann wahr oder falsch sein, „wahr“ und „falsch“ sind sogenannte Wahrheitswerte.
- Literale: Wahrheitswerte **`true`** („wahr“, 1) und **`false`** („falsch“, 0)
- Java: Datentyp **`boolean`**
- Bezeichnung geht auf französischen Mathematiker George Boole (1815-1864) zurück, der sich mit Mathematik von Wahrheitswerten befasst hat.
- Beispiel:

```
boolean minimumGefunden = true;
```

Arrays (auch: Reihungen)

- ***Array***

  = zusammengesetzter Datentyp

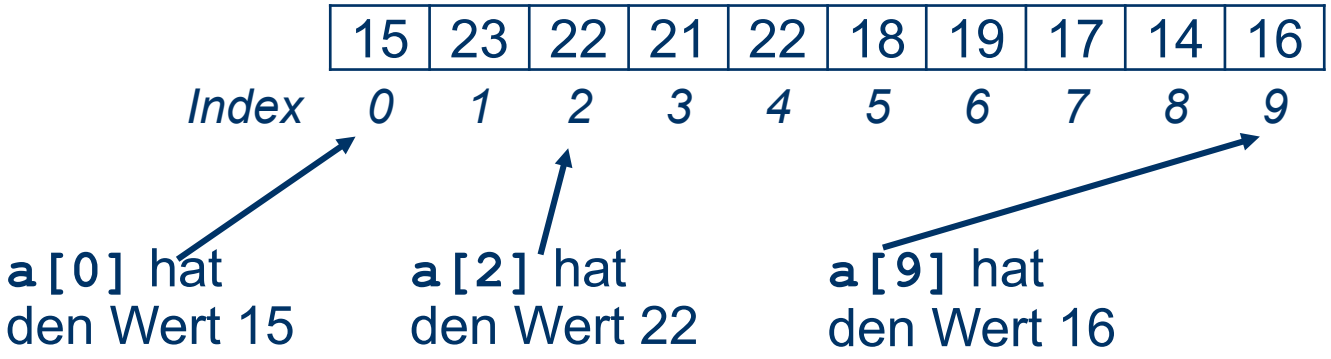
  = endliche Folge von Werten dieses Datentyps

- Beispiel:

  Deklaration einer Variable vom Datentyp **`int`**-Array, evtl. mit Initialisierung.

```
int[] b;
int[] a = { 15, 23, 22, 21, 22, 18, 19, 17, 14, 16 };
```

`[]` nach dem Typ, nicht nach Variablennamen!



- Beachte: Elemente sind ***beginnend ab 0*** durchnummeriert.
- **`a.length`** ergibt Elementanzahl (hier 10).

Erzeugung von Arrays

- Die Größe (= Anzahl der Elemente) wird bei Erzeugung des Arrays festgelegt und kann dann nicht mehr geändert werden.
- ***Variante 1***: Deklaration und erste Wertzuweisung

  ```
  int[] a = {15, 23, 22, 21, 22, 18, 19, 17, 14, 16};
  ```

  erzeugt ein **`int`**-Array der Länge 10 mit Wert-Belegung
- ***Variante 2***: Deklaration und Erzeugung eines Arrays

  ```
  int[] a = new int[10];
  ```

  erzeugt ein **`int`**-Array der Länge 10
- ***Variante 3***: erst Deklaration einer Array-Variablen, dann Erzeugung

  ```
  int[] a;              // von a ist nur Name erklärt
  a = new int[10];      // für a ist Speicherplatz reserviert
  ```

  erzeugt ein **`int`**-Array der Länge 10
- nicht initialisierte Arrays werden mit Standardwerten belegt:
  - 0 bei **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**
  - **`false`** bei **`boolean`**; **`null`** bei Referenzen (folgt später)

Java-Programm `MinSuche`

```
class MinSuche {
    public static void main(String[] args){
        int[] a = {15, 23, 22, 21, 22, 18, 19, 17, 14, 16};
        int merker = a[0];
        int i = 1;
        int n = a.length;
        while (i < n) {
            if (a[i] < merker)
                merker = a[i];
            i = i + 1;
        }
        System.out.println(merker);
    }
}
```

Anzahl der Elemente des Arrays

Deklaration einer Array-Variablen mit erster Wertzuweisung

Zugriff auf Array-Elemente:
`a[0]` (erstes Element),
`a[1]` (zweites Element), ...,
`a[a.length-1]` (letztes Element)

Arbeiten mit Arrays

- Deklaration und Initialisierung:

```
int[] a = { 15, 23, 22, 21, 22, 18, 19, 17, 14, 16 };
```

	15	23	22	21	22	18	19	17	14	16
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

- lesender Zugriff:

```
int y = a[1] + a[9]; // y = 39
```

- schreibender Zugriff:

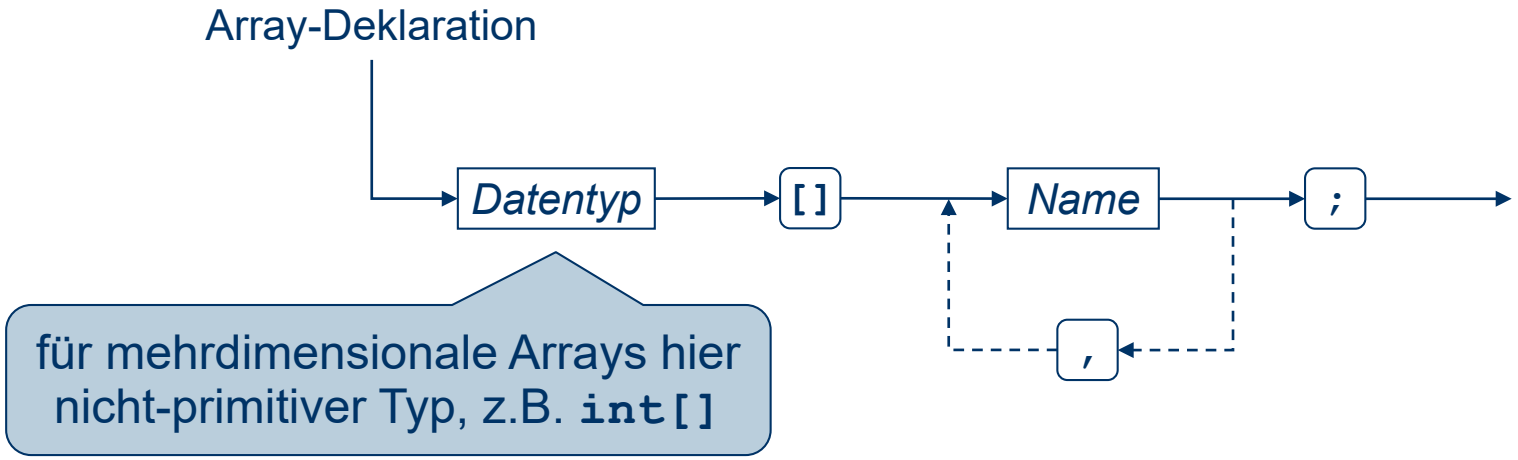
```
a[2] = 3;
a[4] = a[5] / 2 + 1;
```

	15	23	*3*	21	*10*	18	19	17	14	16
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Mehrdimensionale Arrays

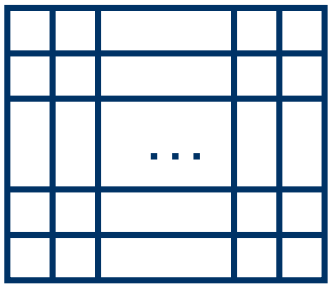
- Der für die Array-Elemente verwendete Typ kann beliebig sein: primitiver Typ, zusammengesetzter Typ (z. B. wiederum ein Array-Typ), selbst erstellter Typ (Klasse, folgt später).
- Daher sind ***mehrdimensionale Arrays*** (Array von Arrays) möglich.
- Deklarationsbeispiele:

```
int[][] mx;                 // Array aus int-Arrays
double[][] rechngsDat;      // Array aus double-Arrays
```



„Rechteckige“ mehrdimensionale Arrays (I)

- gibt es in den meisten Programmiersprachen
- sind dort meist „rechteckige“ mehrdimensionale Datenfelder (C, C++, Fortran, ...)
- gut geeignet, um z. B. feste tabellarische Strukturen von Daten zu verwalten



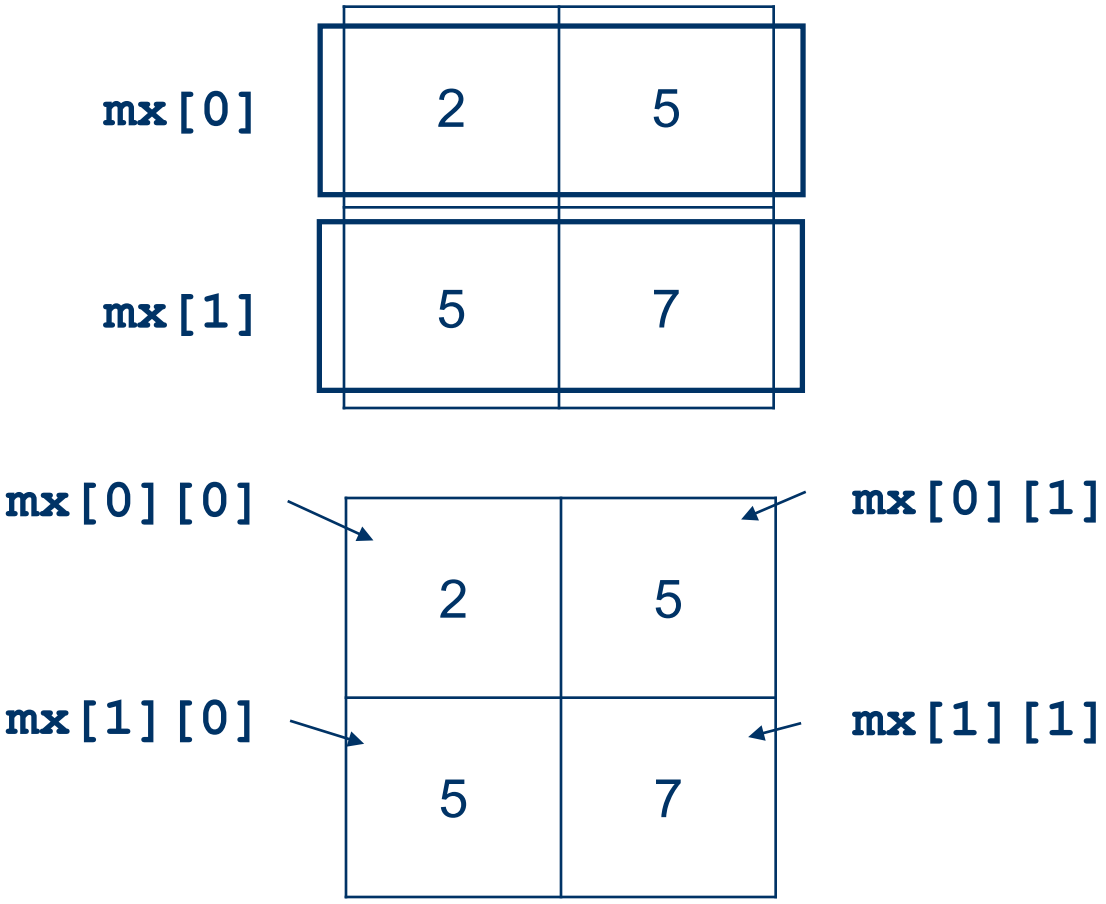
C/C++: Bei linearer Abbildung in den Hauptspeicher kann aus dem Index (i, j) und dem Platzbedarf des Grundtyps die Speicheradresse jedes Elements ermittelt werden.

- Varianten der Erzeugung „rechteckiger“ zweidimensionaler Arrays:

```
int[][] tab1  = {{2, 5}, {5, 7}};  // Variante 1
int[][] tab2 = new int[2][2];      // Variante 2
int a = 1, b = 1;
int[][] tab3;                      // Variante 3: erst Dekl.,
tab3 = new int[a+b][2];            // dann Speicherplatz
```

„Rechteckige“ mehrdimensionale Arrays (II)

- Beispiel: `int[][] mx = {{2, 5}, {5, 7}};`



Zeichenfolgen (auch: Strings)

- ***Zeichenfolge*** (auch: ***String***) setzt sich aus Folge von Zeichen zusammen, also z. B. **`"Peter Mueller", "a+b", "abcd"`**
- Menge aller Zeichenfolgen definiert Datentyp, der in Java (und anderen Sprachen) ***String*** genannt wird.
- Literale: Zeichenfolge wird in Java als Folge der Zeichen in Anführungszeichen geschrieben.
- Deklarationsbeispiel: **`String name = "Peter Mueller";`**
- Repräsentation von Zeichenfolgen im Speicher: man codiert (Unicode-) Zeichen (Ziffern, Buchstaben, Sonderzeichen, s. o.) in Bitfolgen.

Gliederung der Lehreinheit

Operatoren und Ausdrücke

- ***Ausdrücke*** sind Formeln zum Berechnen von Werten, deren ***Operanden*** mittels ***Operatoren*** verknüpft werden.
- Unterscheidung zwischen verschiedenen Typen von Ausdrücken, abhängig davon, welchen Wert sie am Ende liefern, z. B. ***arithmetische Ausdrücke***, ***boolesche Ausdrücke***.

Operator	Ausdruck	Beschreibung
+	x + y	Addition
-	x – y	Subtraktion
*	x * y	Multiplikation
/	x / y	Division
%	x % y	Modulo, Restwert

Operator	Ausdruck	liefert `true`, wenn
>	x > y	x größer als y
>=	x >= y	x größer oder gleich y
<	x < y	x kleiner als y
<=	x <= y	x kleiner oder gleich y
==	x == y	x gleich y
!=	x != y	x ungleich y

binäre Operatoren: Operator hat jeweils zwei Operanden x und y

auch andere Typen, z. B. **`boolean`**

Unäre/binäre „Strichrechung“ auf numerischen Typen

- ***Addition*** von `a` und `b` (zwei Operanden → ***binärer Operator***)

```
int a = 0, b = 2;
int a = a + b;   // a hat den Wert 2 und b den Wert 2
```

- ***(Post-/Pre-) Inkrementierung*** (ein Operand → ***unärer Operator***)

  `a++`: Inkrementiere `a` um 1 (***nach*** der Auswertung von `a`)

```
int a = 0, b;    // a hat den Wert 0, b uninitialisiert
b = a++;         // b hat den Wert 0, a den Wert 1
```

  `++a`: Inkrementiere `a` um 1 (***vor*** der Auswertung von `a`)

```
int a = 0, b;    // a hat den Wert 0, b uninitialisiert
b = ++a;         // b hat den Wert 1, a den Wert 1
```

- ***(Post-/Pre-) Dekrementierung*** `a--`, `--a` entsprechend
- ***Vorzeichenwechsel*** (ein Operand → ***unärer Operator***)

  `-a`: Änderung des Vorzeichens von `a`

```
int a = -1, b;   // a hat den Wert -1, b uninitialisiert
b = -a;          // b hat den Wert 1, a den Wert -1
```

Hinweise zum Programmierstil

- Bei unären Operatoren keine Leerzeichen verwenden:

```
int a = -3; // ok
a++;        // ok
a  ++;      // zwar syntaktisch ok,
            // aber schlechter lesbar
```

- Um binäre Operatoren immer Leerzeichen verwenden:

```
a = 3 + 5;
```

Eins plus Eins macht...

- Was wird ausgegeben?

```
public class Increment {
    public static void main(String[] args) {
        int j = 0;
        for (int i = 0; i < 100; i++) {
            j = j++;
        }
        System.out.println(j);
    }
}
```

- Vorschläge:
  - ☐ 100
  - ☐ 1
  - ☐ 99
  - ☐ etwas anderes
  - ☐ 0

0!

```
int tmp = j;
j = j + 1;
j = tmp;
```

Gemeint war vermutlich nur: `j++`

→Beschreibe eine Variable nie mehr als einmal pro Anw.

Arithmetische Ausdrücke

- ***Arithmetische Ausdrücke*** ähneln Formeln aus Zahlenwerten in der Mathematik, die u. a. die üblichen Operatoren (+, -, *, /) verwenden und geklammert sein können.
- Beispiele: `i + 1, 3 * (2 + i), (5 + 2) / -3`
- ***Auswertung eines Ausdrucks*** erfolgt „von links nach rechts“ unter Berücksichtigung der Regeln „Klammern zuerst“, „Punktrechnung vor Strichrechnung“ und sog. Bindungsregeln (Details folgen).
- Beispiel: `(3 + 9) + 4 * (5 + 3)` wird ausgewertet zu `44`

- möglich: ***Verknüpfung von Wertzuweisung und Ausdruck***
- Beispiel:

```
int a = (3 + 9) + 4 * (5 + 3);
int b = 3, c = b * (5 + 3);
```

Boolesche Ausdrücke (I)

- ***Boolesche Ausdrücke*** sind Formeln, in denen Operanden durch boolesche Operatoren verknüpft werden, und die als Ergebnis einen Wahrheitswert „richtig“ (`true`) oder „falsch“ (`false`) liefern.

- Operanden können sein:
  - die Wahrheitswerte `true` oder `false`,
  - Vergleiche zwischen arithmetischen Ausdrücken oder
  - boolesche Variablen.

- Beispiele

`3 == 7`	hat den Wert `false`
`3 != 7`	hat den Wert `true`
`a[i] < merker`	hat den Wert `true` oder `false` abhängig von `a[i]` und `merker`

Boolesche Ausdrücke (II)

- Logische Verknüpfungen erfolgen mit ***booleschen Operatoren***.
- Die bekanntesten booleschen Operatoren sind
  `!` (nicht), `&&` (und), `||` (oder)

`x`	`!x`
true	false
false	true

`x`	`y`	`x && y`
false	false	false
false	true	false
true	false	false
true	true	true

`x`	`y`	`x || y`
false	false	false
false	true	true
true	false	true
true	true	true

Boolesche Ausdrücke (III)

- Boolescher Ausdruck mit booleschen Operatoren und Vergleichen:
  `(3 < 7) && (3 == 7)` wird ausgewertet zu `false`
  `((3 == 7) || (3 != 7)) && (2 <= 2)` wird ausgew. zu `true`
- Verknüpfung von Wertzuweisung und Ausdruck:

```
boolean w = ((3 == 7) || (3 != 7)) && (2 <= 2);
```

  bewirkt, dass `w` den Wert `true` zugewiesen bekommt.
- Verwendung von booleschen Variablen in booleschen Ausdrücken:

```
boolean v = (3 == 7);          // v erhaelt den Wert false
boolean w = (v || (2 <= 2));// w erhaelt den Wert true
```

- Boolesche Operatoren auch in Mathematik in ***Aussagenlogik***:
  - Symbole: $\neg$ (nicht) $\wedge$ (und) $\vee$ (oder)
  - `(3 < 7) && (3 == 7)` entspricht dann: $(3 < 7) \wedge (3 = 7)$

Faule vs. strikte Auswertung

- Die Operatoren `&&` und `||` heißen auch ***bedingte logische Operatoren***, da sie ihren rechten Operanden nur dann auswerten, wenn dies wirklich nötig ist. Sie sind in diesem Sinne „faul“.
  - In `((b = false) && (c = true))` wird also die Zuweisung zu `c` nicht ausgeführt, da der Wert der Zuweisung zu `b false` ist.
    Da der linke Operand von `&&` schon falsch ist, kann der Gesamtausdruck nicht mehr wahr werden, die Auswertung des rechten Operanden wird eingespart.
- Im Gegensatz dazu evaluieren die ***symmetrischen Bitoperatoren*** `&`, `|` und `^` stets (strikt) beide Operanden.
- Beispiel:
  - Durch faule Auswertung führt folgender Ausdruck nicht zur Division durch Null: `(x != 0) && ( (1 - x) / x > 1)`
  - Durch strikte Auswertung gibt es hier *bei x = 0* einen *Fehler*:
    `(x != 0) & ( (1 - x) / x > 1)`

Nur 25% ungerade Zahlen?

- Wo steckt der Fehler?

```
public static boolean isOdd(int i) {
    return i % 2 == 1;
}
```

- Bei negativem `i` ist das Resultat von `%` negativ (oder 0).

- Behebungsmöglichkeit:

```
public static boolean isOdd(int i) {
    return i % 2 != 0;
}
```

Ausdrücke (allgemein)

- Jede als vom Typ *T* deklarierte Variable ist ein Ausdruck vom Typ *T*.

  Beispiel:

  ```
  int anzSMS, anzMin;
  ```

  `anzSMS` und **`anzMin`** sind Ausdrücke vom Typ **`int`**.
- Jeder konstante Wert vom Typ *T* ist ein Ausdruck vom Typ *T*.

  Beispiel:

  `12.7` ist ein Ausdruck vom Typ **`double`**.
- Sind $a_1, \ldots, a_n$ Ausdrücke der Typen $T_1, \ldots, T_n$ und ist $f: T_1 \times \ldots \times T_n \rightarrow T$ eine Operation, so ist $f(a_1, \ldots, a_n)$ ein Ausdruck vom Typ *T*.

  Beispiel:

  `(295 + anzSMS * 4 + anzMin * 5) / 100.0` ist ein Ausdruck vom Typ **`double`**

  Achtung: der Klammerausdruck (Typ **`int`**) wird im Beispiel automatisch in den Typ **`double`** konvertiert (sog. automatische ***Typumwandlung***, Details folgen)

Hinweis zur Programmzeilenformatierung

- Lange Ausdrücke sollten im Sinne der Programmlesbarkeit geeignet formatiert werden.
- Die Formatierung sollte zusammengehörige Teilausdrücke sichtbar machen
- statt:

```
longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longName6;
```

- besser:

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longName6;
```

Weitere Grundoperationen für primitive Typen

- logische Bitoperatoren (Hinweis: &, |, ^ auch für **`boolean`**)
  - & (bitweises Und): 0101 & 1011 => 0001 („nur wenn beide 1“)
  - | (bitweises Oder): 0101 | 1011 => 1111 („wenn min. eine Zahl 1 ist“)
  - ^ (XOR, bitweises Entweder-Oder): 0101 ^ 1011 => 1110 („nur wenn beide Zahlen unterschiedlich“, „die eine oder die andere Stelle 1 ist“)
  - ~ (bitweises Komplement): 0110 => 1001 (bitweise Invertierung)
- Bit-Verschiebung
  - << (Verschiebung nach links, left shift)

    15 << 3: (15 =) 00001111 wird zu 01111000 (= 120)
  - >> (Verschiebung nach rechts, right shift sign; beachtet Vorzeichen, von links altes Vorzeichen-Bit einschieben)

    15 >> 3: (15 =) 00001111 wird zu 00000001 (= 1)
  - >>> (Verschiebung nach rechts ohne Berücksichtigung der Vorzeichen, right shift no-sign; ignoriert Vorzeichen, von links neue Nullen einschieben)

    -1 >>> 3: (-1 =) 11111111 wird zu 00011111 (= 31)

Kurzschreibweise: Grundoperatoren kombiniert mit Zuweisung

=	Zuweisung
+=	Inkrementierung, z. B. `alter += 80` (in etwa `alter = alter + 80`)
-=	Dekrementierung, z. B. `alter -= 80` (in etwa `alter = alter - 80`)
*=	Skalierung, z. B. `alter *= 2` (in etwa `alter = alter * 2`)
/=	Division, z. B. `alter /= 2` (in etwas `alter = alter / 2`)
%=	Restebildung, z. B. `alter %= 10` (in etwa `alter = alter % 10`)
|=	bitweises Oder
&=	bitweises Und
^=	bitweises Entweder-Oder (Exklusives Oder)
<<=	Verschieben nach links, z. B. `alter <<= 2` (in etwa `alter = alter << 2`)
>>=	Verschieben nach rechts (mit Vorzeichenerhalt), z. B. `alter >>= 2` (in etwa `alter = alter >> 2`)
>>>=	Verschieben nach rechts (ohne Vorzeichenerhalt) z. B. `alter >>>= 2` (in etwa `alter = alter >>> 2`)

Auswertungsreihenfolge für Ausdrücke

- innerhalb von Ausdrücken: links vor rechts
- dabei: Berücksichtigung von Klammern und folgender Präzedenz-/Vorrangregeln
- Präzedenz in Java (mit expliziter Klammerung zu umgehen):

Postfix-Operatoren	[] . (params) expr++ expr--
unäre Operatoren	++expr --expr +expr -expr ! ~
Erzeugung oder Typumwandlung	new (type) expr
Multiplikationsoperatoren	* / %
Additionsoperatoren	+ -
Verschiebeoperatoren	<< >> >>>
Vergleichsoperatoren	< > <= >= instanceof (später mehr)
Gleichheitsoperatoren	== !=
Bitoperator Und	&
Bitoperator exklusives Oder	^
Bitoperator inklusives Oder	|
logisches Und	&&
logisches Oder	||
Fragezeichenoperator	?
Zuweisungsoperatoren	= += -= *= /= %= >>= <<= >>>= &= ^= |=

Hinweise zum Programmierstil

- Nicht jedem Leser eines Programms sind die zuvor benannten Präzedenzregeln bekannt.
- Verwenden Sie deshalb im Zweifelsfalle im Sinne einer besseren Lesbarkeit Ihrer Programme lieber zu viele Klammern.
- Beispiel:

  statt: `(a == b && c == d)`

  besser: `((a == b) && (c == d))`

Operatoren für Zeichenketten

- Konkatenation von Zeichenketten und Vergleich:

```
boolean istGleich;
istGleich = "hello" + "students" == "hellostudents";
// istGleich: true
```

- **aber:** Ausblick auf „Objektorientierte Modellierung und Programmierung“; Erklärung solcher Phänomene folgt dort

```
String s1 = "hello";
String s2 = "students";
String s3 = s1 + s2;  // s3: "hellostudents"
istGleich = s1 + s2 == s3;
// istGleich: false
```

- Länge der Zeichenkette:

```
int laenge_s3 = s3.length(); // laenge_s3 = 13
```

Wer zuletzt lacht ... (I)

- Was wird ausgegeben?

```
public class LetzterLacher {
    public static void main(String[] args) {
        System.out.print("H" + "a");
        System.out.print('H' + 'a');
    }
}
```

- Vorschläge:
  - Laufzeitfehler/Exception
  - Übersetzungsfehler
  - HaHa
  - Etwas anderes

Es wird ausgegeben:
`Ha169`

Wer zuletzt lacht ... (II)

- Was wird ausgegeben?

```
public class LetzterLacher {
    public static void main(String[] args) {
        System.out.print("H" + "a");
        System.out.print('H' + 'a');
    }
}
```

- `'H'` und `'a'` sind Literale vom Typ **`char`**.
- Der **+**-Operator führt daher eine Addition der Zeichenwerte durch (72 + 97 = 169) und keine Zeichenkettenkonkatenation.

→ Doppelte und einfache Anführungszeichen wohlüberlegt nutzen!

→ Implizite Typumwandlung genau verstehen!

Wer zuletzt lacht ... (III)

- Keine Lösung:

```
System.out.print((String)'H' + (String)'a');
```

- Behebungsmöglichkeiten:

```
System.out.print("" + 'H' + 'a');
```

```
StringBuffer sb = new StringBuffer();
sb.append('H');
sb.append('a');
System.out.print(sb);
```

Gleicher als gleich? (I)

- Was wird ausgegeben?

```
public class GleicherAlsGleich {
   public static void main(String[] args) {
      String s1 = "FAU";
      String s2 = "FAU";
      String s3 = new String("FAU");
      String s4 = new StringBuilder()
         .append("FAU").toString();
      if (s1 == s2)    System.out.println("s1 == s2");
      if (s1 == "FAU") System.out.println("s1 == \"FAU\"");
      if (s1 == s3)    System.out.println("s1 == s3");
      if (s1 == s4)    System.out.println("s1 == s4");
}  }
```

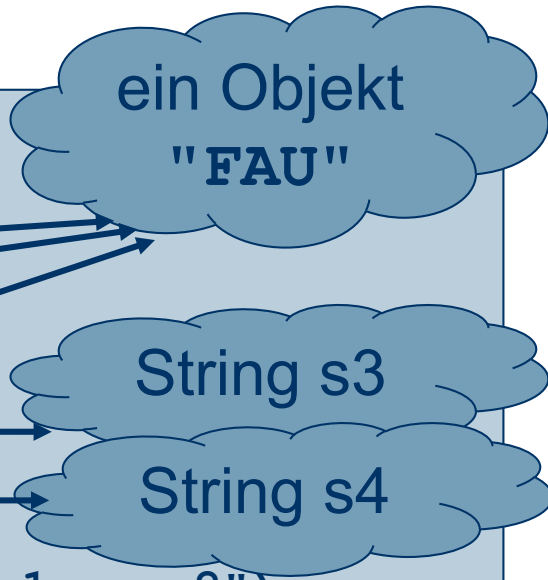
- Es wird nur ausgegeben:

```
s1 == s2
s1 == "FAU"
```

Gleicher als gleich? (II)

- Strings sind Objekte.
- Übersetzungszeitkonstanten werden zusammengefasst.
- == ist Referenzvergleich.

```
public class GleicherAlsGleich {
   public static void main(String[] args) {
      String s1 = "FAU";
      String s2 = "FAU";
      String s3 = new String("FAU");
      String s4 = new StringBuilder()
         .append("FAU").toString();
      if (s1 == s2)     System.out.println("s1 == s2");
      if (s1 == "FAU") System.out.println("s1 == \"FAU\"");
      if (s1 == s3)     System.out.println("s1 == s3");
      if (s1 == s4)     System.out.println("s1 == s4");
} }
```



→ Vergleich besser mit `s1.equals(s3)`

George Orwells Farm der Tiere (I)

- Was wird ausgegeben?

```
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "
                           + pig == dog);
    }
}
```

- Vorschläge:
  - **`Animals are equal: true`**
  - **`Animals are equal: false`**
  - Etwas anderes

`dog` ist keine String-Konstante und zeigt daher auf ein anderes Objekt als **`pig`**. Daher wohl **`... false.`**

Leider falsch!
Ausgegeben wird *nur* **`false`**.

George Orwells Farm der Tiere (II)

- Was wird ausgegeben?

```
final String pig = "length: 10";
final String dog = "length: " + pig.length();
System.out.println("Animals are equal: "
                   + pig == dog);
```

- Da + stärker bindet als == wird das Vergleichsergebnis ausgegeben.
- Behebungsmöglichkeit:

```
System.out.println("Animals are equal: "
                   + (pig == dog));
```

→ Zeichenketten mit **`equals()`** vergleichen!

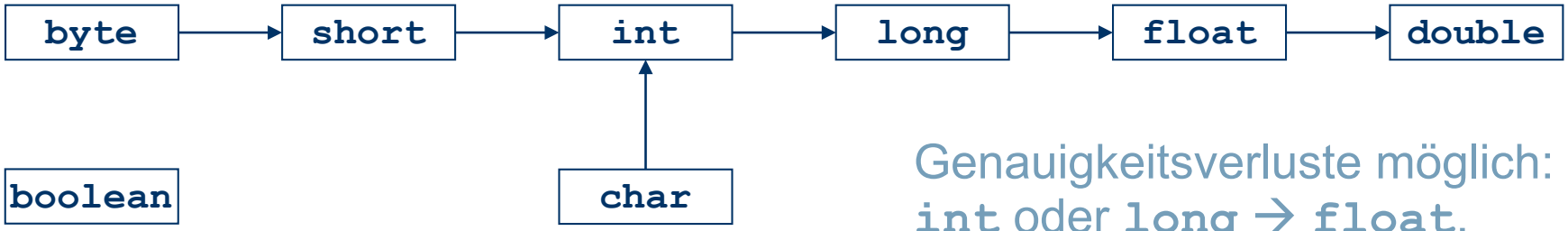
→ Bindungsstärken genau kennen!

→ Bei String-Konkatenation sicherheitshalber () um nicht-triviale Argumente.

Gliederung der Lehreinheit

Typkonvertierung (engl. type casting)

- In Mathematik erlaubt: `10 + 0.9`, da ganze Zahlen Teilmenge der reellen Zahlen.
- In Programmen möglich: Teilberechnungen führen zu Ganzzahl bzw. Fließkommaergebnis, die abschließend durch eine Operation verknüpft werden sollen.
- Lösung: ***Typkonvertierung*** (engl. ***type casting***)
- Erweiterungsbeziehung zwischen primitiven Datentypen:



Genauigkeitsverluste möglich:
`int` oder `long` → `float`,
`long` → `double`

- Datentyp an der Pfeilsitze ist Erweiterung des Typs am Pfeilanfang.
- in Pfeilrichtung: ***automatische erweiternde Konvertierung***

Automatische (auch: implizite) Typkonvertierung

- implizite Umwandlung in Ausdrücken, um definierten Operator anzuwenden, wenn Wertebereich nicht verkleinert wird („in Pfeilrichtung“)

```
int a = 0;
float b = 2;     // 2 (int) wird implizit zu float
boolean c;
c = (a == b);    // a wird implizit in float gewandelt
                 // da Wert von a ungleich b ist, wird
                 // c ausgewertet zu false
```

- bei `++`, `--` auf **`byte`**, **`short`**, **`char`** wird Wert zu **`int`**

Explizite Typkonvertierung

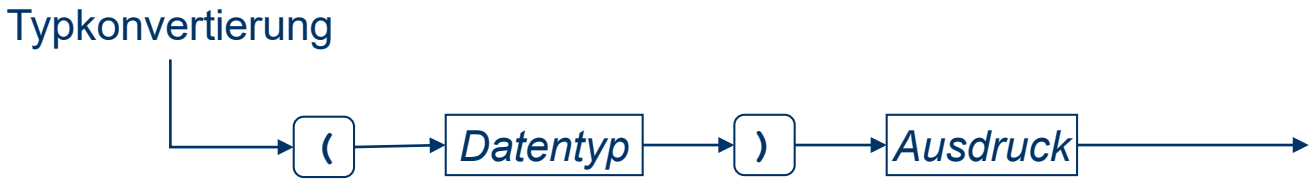
- problematischer: Konvertierung gegen Pfeilrichtung
- Wie soll Gleitkommazahl 2.5 in Ganzzahl konvertiert werden?
- Zur ausdrücklich gewünschten ***expliziten Typkonvertierung*** stellt Java einen ***Typkonvertierungsoperator*** zu Verfügung (eventuell: Genauigkeitsverlust)
- zwei Operanden: **`(<Zieldatentyp>) <Wert>`**

```
double x = 2.0;
int y = (int) x;          // y = 2
int z = (int) (x / 3);    // x / 3 wird ausgew. zu 0.66..
```

- Anwendung des (**`int`**)-Operators führt zur Rundung hin zur 0, d. h. **`z`** wird der Wert 0 zugewiesen.
- Weglassen des Konvertierungsoperators führt hier zu Fehlermeldung
- allgemein:
  1. bei **`byte`** bis **`long`** höherwertige Bits wegstreichen
  2. **`float`** und **`double`** erst (so gut es geht) Richtung 0 „runden“ nach **`long`**, dann ggf. von dort weiter mit Regel 1

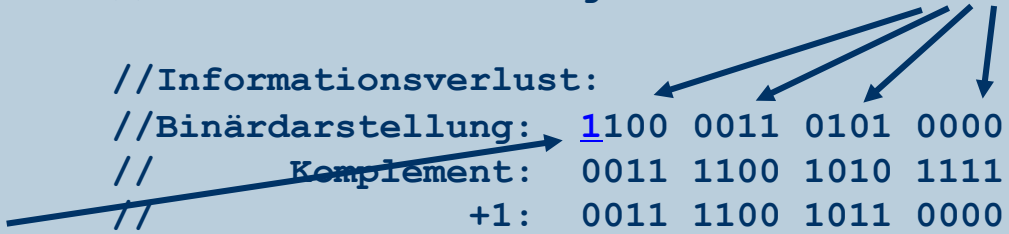
Typkonvertierung

- ***Typkonvertierung*** bedeutet die Überführung des Werts eines Datentyps in einen Wert eines anderen Datentyps.
- Datentyp kann somit in anderen Datentyp konvertiert werden, wenn letzterer Erweiterung, d. h. Obermenge des ersten ist.
- Sofern Zieldatentyp Erweiterung des zu konvertierenden Datentyps ist, erfolgt Konvertierung bedarfsweise automatisch.
- Typkonvertierung kann auch durch Anwendung eines Typkonvertierungsoperators ausgeführt werden (explizite Typkonvertierung).



Beispiele für implizite und explizite Typkonvertierung (I)

```
double pi = 3.14f;              //3.14 implizit geweitet float → double
int i = 8;

double k = pi * i;              //i implizit geweitet int → double
                                //k = 25.12

int k1 = (int) (pi * i);        //25.12 explizit auf int verkleinert
                                //Informationsverlust: k1 = 25

int k2 = ((int) pi) * i;        //pi wird auf int verkleinert (3)
                                //Informationsverlust: k2 = 24

long l = (int) k * 2000;        //k wird auf int verkleinert (25)
                                //l = 50000
                                //Sedezimaldarstellung: 000000000000C350

short s = (short) l;            //Informationsverlust:
                                //Binärdarstellung:  1100 0011 0101 0000
                                //      Komplement:  0011 1100 1010 1111
                                //              +1:  0011 1100 1011 0000
                                //s = -15536
```



short, int,... sind vorzeichenbehaftet; führende 1 führt daher zu Interpretation als negative Zahl

Stiller Alarm (I)

- Was wird ausgegeben?

```
public class GrosseDivision {
    private static final long MILLIS_PRO_TAG
        = 24 * 60 * 60 * 1000;
    private static final long MICROS_PRO_TAG
        = 24 * 60 * 60 * 1000 * 1000;

    public static void main(String[] args) {
        System.out.println(
            MICROS_PRO_TAG / MILLIS_PRO_TAG);
    }
}
```

- Vorschläge:
  - ☐ 1000
  - ☐ 5
  - ☐ 5000
  - ☐ Ausnahme/Exception

Es wird 5 ausgegeben!

Stiller Alarm (II)

```
private static final long MICROS_PRO_TAG
        = 24 * 60 * 60 * 1000 * 1000;
```

= 86.400.000.000 > Integer.MAX_VALUE = 2.147.483.647

- Rechts vom = sind nur **`int`**-Werte, daher Rechnung in **`int`**.
  Bei der letzten Multiplikation mit 1000 *läuft der Zahlenbereich* über.
- Und zwar um 20 * (Integer.MAX_VALUE-Integer.MIN_VALUE)
  = 20 * 4.294.967.296.
  Das in **`int`** darstellbare Ergebnis ist 500.654.080.

```
500_654_080 / 80_400_000
```

= 5.79 also ganzzahlig 5.

- Behebungsmöglichkeit:

```
private static final long MICROS_PRO_TAG
        = 24L * 60 * 60 * 1000 * 1000;
```

→Bei Arbeit mit großen Zahlen an stille Überläufe denken!

Typsicherheit

- jede Variable, jede Konstante und jedes Literal besitzt einen ***Typ***:
  - Festlegung des gültigen Wertebereichs
  - Festlegung der anwendbaren Operationen
- ***Typsicherheit***: auf Operanden können nur die ihrem Typ entsprechenden Operationen angewandt werden.
- In ***typsicheren Programmiersprachen*** kann der Übersetzer für jeden Operanden zu jeder Zeit den zugehörigen Typ ermitteln und damit feststellen, ob eine Operation anwendbar ist:
  - wenn nicht, dann liefert der Übersetzer eine Fehlermeldung.
  - Somit lassen sich anhand/mittels der Typen manche Fehler ermitteln, bevor ein Schaden angerichtet wird; damit verbesserte Korrektheit von Programmen (statische Typsicherheit)

Beispiel für Typfehler

```
...
a = 5 + true;
...
```

Binärer +-Operator ist nur für numerische Typen (und z. B. auch für Strings) erklärt.
Übersetzer stellt fest, dass **`true`** ein Wahrheitswert und damit kein numerischer Wert ist → Fehlermeldung

Name der Quellcode-Datei und Zeilennummer des Fehlers

Übersetzerlauf:

```
> javac Test.java

...
Test.java:4: operator + cannot be applied to int, boolean
          a = 5 + true;
                  ^
...
```

Stelle des Fehlers

Quellcode-Zeile

(Statischer) Typ eines Ausdrucks

- Der Typ eines Ausdrucks kann bereits zur Übersetzungszeit bestimmt werden und leitet sich im Wesentlichen aus den Typen der Teilausdrücke und des angewendeten Operators ab.
- Beispiel:

```
int a = 0;
float b = 2;
boolean c;

c = a == b;  // Typ des Ausdrucks a == b ist boolean,
             // a wird implizit in float gewandelt (s.o.),
             // alles ok
c = a * b;   // Typ des Ausdrucks a * b ist float,
             // a wird implizit in float gewandet (s.o.),
             // Fehler bei der Zuweisung!
c = c * c;   // Fehler, da * nicht auf boolean anwendbar.
```

(Statischer) Typ einer Zuweisung

- Die Typen der linken und der rechten Seite einer Zuweisung müssen zusammenpassen.
- Nicht nur Ausdrücke haben einen Typ, sondern auch Zuweisungen. Zuweisungen liefern als Wert das jeweilige Ergebnis der rechten Seite.
- Beispiel:

```
int i, j;
i = j = 0;                  // i und j wird der Wert 0 zugewiesen,
                            // da j = 0 den Wert 0 liefert
                            // Kettenzuweisung
d = (a = b + c) + r         // eingebettete Zuweisung
```

- Hinweis: Kettenzuweisungen und eingebettete Zuweisungen gelten aufgrund der schlechten Lesbarkeit als ***schlechter Programmierstil***.