Einführung in die Programmierung

Vorlesungsteil 2
Elementare Datentypen, Variablen, Arithmetik, Typecast

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

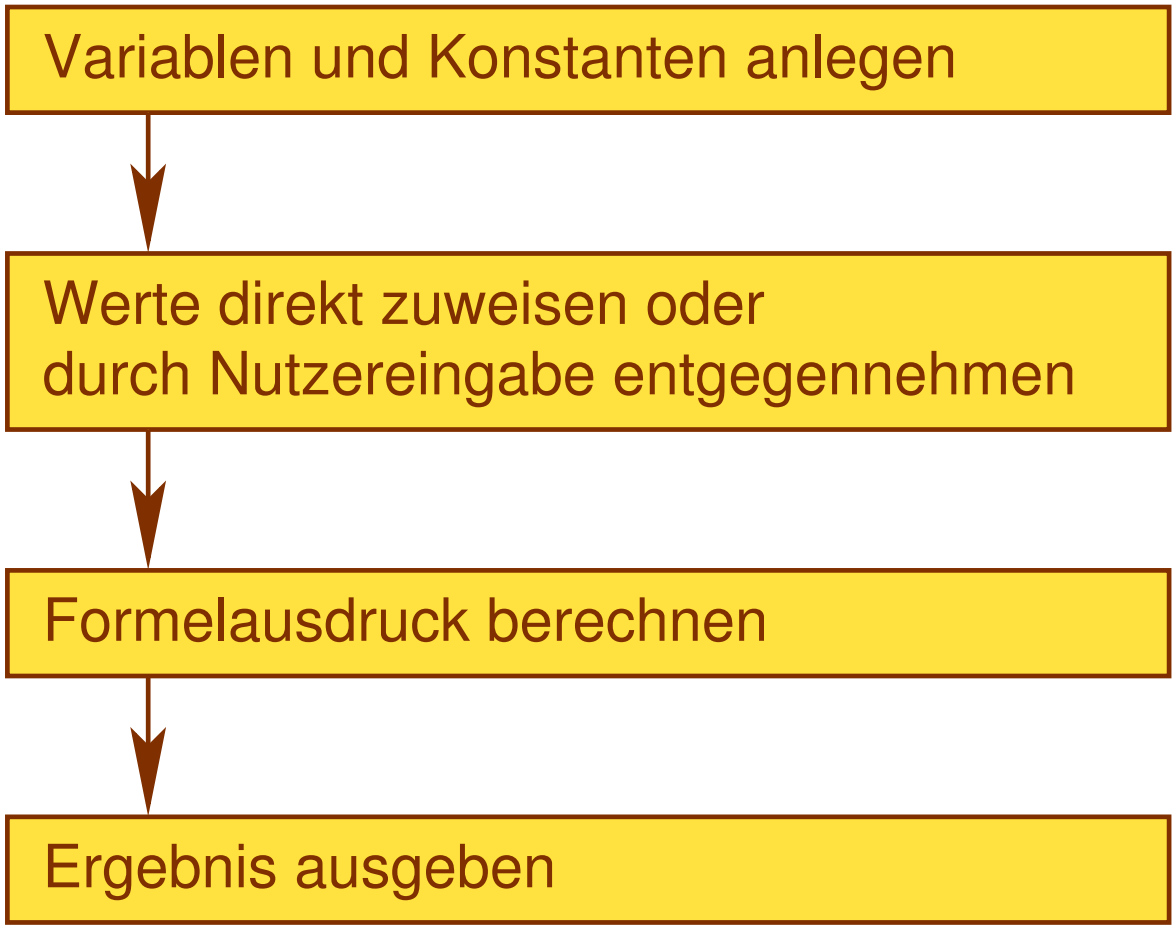
Sommersemester 2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Heute lernen wir, wie man in Java Programme zur Berechnung arithmetischer Ausdrücke schreibt.

(Bild: www.lehrfilme.eu)

Programmablauf

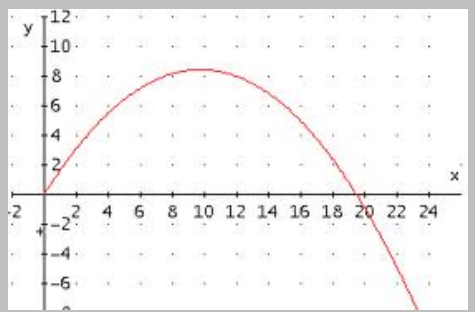


Vorlesung Einführung in die Programmierung mit Java

1. **Einführung und erste Schritte** . . . . . Installation Java-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . Java als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen, Schleifen und logische Ausdrücke programmieren
4. **Methoden selbst programmieren** . . . . . Methoden als wiederverwendbare Funktionen, Werteübernahme und -rückgabe
5. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
6. **Objektorientiert programmieren** . . . . . Klassen, Objekte, Attribute, Methoden, Sichtbarkeit, Vererbung, Polymorphie
7. **Felder und Graphen** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Netzwerken
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeichenketten, Dateiarbeit, Ausnahmen** . . . . . Texte analysieren, ver-/entschlüsseln, Dateien lesen/schreiben, Fehler behandeln
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Beispiel: Schräger Wurf vom Boden zum Boden

Idealisiert als Wurfparabel betrachtet



$$\text{wurfweite} = \frac{v_0^2}{g} \cdot \sin(2 \cdot \phi)$$

v_0: Anfangsgeschwindigkeit
ϕ: Abwurfwinkel
g: Erdbeschleunigung

Beispiel: Schräger Wurf – Java-Quelltext

SchraegerWurf.java

```
import java.util.Scanner; //benoetigte Bibliotheken einbinden

public class SchraegerWurf {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in); //Bibliotheksklasse Scanner zum
                                         //Handling von Tastatureingaben
    final double G = 9.81; //Erdbeschleunigung als Konstante G hinterlegt
    double winkel = 45;    //Abwurfwinkel in Grad
    double v0;             //Anfangsgeschwindigkeit

    double phi = winkel * Math.PI / 180; //Winkel im Bogenmasz (radiant)
    double weite;

    System.out.print("Schraeger Wurf\n\nAnfangsgeschwindigkeit m/s: ");
    v0 = Double.parseDouble(sc.next()); //Tastatureingabe entgegennehmen
                                        //in double-Wert konvertieren
                                        //und der Variablen v0 zuweisen

    weite = v0 * v0 / G * Math.sin(2 * phi);
    System.out.println("Wurfweite bei Abwurfwinkel "+winkel+" Grad in m: "+weite);
  }
}
```

Compilieren mit **`javac SchraegerWurf.java`**

Ausführen mit **`java SchraegerWurf`**

Die 51 Schlüsselwörter von Java

Java als kompakte Programmiersprache

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

Die 51 Schlüsselwörter von Java

Heute und nächste Woche lernen wir davon kennen . . .

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

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ . . . . . Menge der natürlichen Zahlen

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ . . . . . Menge der natürlichen Zahlen
  $\mathbb{R}$ . . . . . Menge der reellen Zahlen

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

- Manche *Operationen* nur auf ausgewählten Zahlenbereichen sinnvoll bzw. definiert:

  $\lceil\ \rceil$ ..... Aufrunden nur außerhalb ganzer Zahlen sinnvoll

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

- Manche *Operationen* nur auf ausgewählten Zahlenbereichen sinnvoll bzw. definiert:

  $\lceil\ \rceil$ ..... Aufrunden nur außerhalb ganzer Zahlen sinnvoll
  $<$ ..... Vergleich auf „kleiner“ nicht zwischen komplexen Zahlen

Ein (kleiner) Exkurs in die Mathematik

- Aus der Mathematik kennen Sie *Zahlenbereiche*:

  $\mathbb{N}$ ..... Menge der natürlichen Zahlen
  $\mathbb{R}$ ..... Menge der reellen Zahlen
  $B = \{0, 1\}$ ..... selbst definiert

- *Variablen* nehmen *Werte* aus zugrunde liegendem Zahlenbereich an:

  $n \in \mathbb{N}$ ..... Gleichung $4 \cdot n = 1$ hat keine Lösung
  $x \in \mathbb{R}$ ..... Gleichung $4 \cdot x = 1$ hat Lösung $\frac{1}{4}$

- Manche *Operationen* nur auf ausgewählten Zahlenbereichen sinnvoll bzw. definiert:

  $\lceil\ \rceil$ ..... Aufrunden nur außerhalb ganzer Zahlen sinnvoll
  $<$ ..... Vergleich auf „kleiner“ nicht zwischen komplexen Zahlen
  $-$ ..... Subtraktion $3 - 5$ nicht in $\mathbb{N}$, aber auf ganzen Zahlen

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“

Der Sinn verschiedener Zahlenbereiche

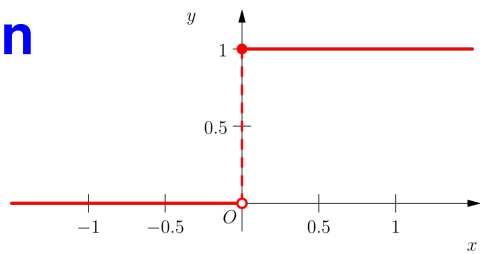
- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“
  - Ist es sinnvoll, jemandem eine Viertelvase zuzusprechen?

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“
  - Ist es sinnvoll, jemandem eine Viertelvase zuzusprechen?
- **Beschreibungen so einfach wie möglich halten, um (Anwendungs)Fehlern entgegenzuwirken**

Der Sinn verschiedener Zahlenbereiche

- **Einheitliche Operationswirkung im Zahlenbereich**
  - Berechnungsvorschrift gilt für beliebige Werte innerhalb des Zahlenbereiches, aber nicht darüber hinaus
  - Vorgehen bei Division auf natürlichen Zahlen weicht ab von Division auf reellen Zahlen, wenn ein Divisionsrest entsteht
- **Mathematische Beschreibung nah am Sachverhalt**
  - „Ein Erbe besteht ausschließlich aus 97 gleichartigen wertvollen Porzellanvasen. Von den drei Erbberechtigten bekommt einer die Hälfte und die anderen beiden je ein Viertel des Nachlasses.“
  - Ist es sinnvoll, jemandem eine Viertelvase zuzusprechen?
- **Beschreibungen so einfach wie möglich halten, um (Anwendungs)Fehlern entgegenzuwirken**
  - Schwellenwertfunktion



In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `Ü`, `$` und viele mehr bildet einen Typ.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `Ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten bilden ebenfalls einen Typ.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `Ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten bilden ebenfalls einen Typ.
- Um einen Variablenwert eines Typs abspeichern zu können, steht aber nur endlich viel Speicherplatz zur Verfügung.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `Ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten bilden ebenfalls einen Typ.
- Um einen Variablenwert eines Typs abspeichern zu können, steht aber nur endlich viel Speicherplatz zur Verfügung.
- Es gilt, sinnvolle Kompromisse zwischen *Speicherplatzbedarf*, *Wertevorrat* und *Aufwand* zur Ausführung der Operationen zu finden.

In der Programmierung greifen wir die Idee der Zahlenbereiche auf und verallgemeinern sie zu *Typen*.

- Die Menge aller eingebbaren Zeichen wie `a`, `Ü`, `$` und viele mehr bildet einen Typ.
- Alle (in ihrer Länge begrenzten) Zeichenketten bilden ebenfalls einen Typ.
- Um einen Variablenwert eines Typs abspeichern zu können, steht aber nur endlich viel Speicherplatz zur Verfügung.
- Es gilt, sinnvolle Kompromisse zwischen *Speicherplatzbedarf*, *Wertevorrat* und *Aufwand* zur Ausführung der Operationen zu finden.
- Aus diesen Überlegungen resultiert eine Vielzahl *elementarer (primitiver) Datentypen* in Java.

Grundbegriffe

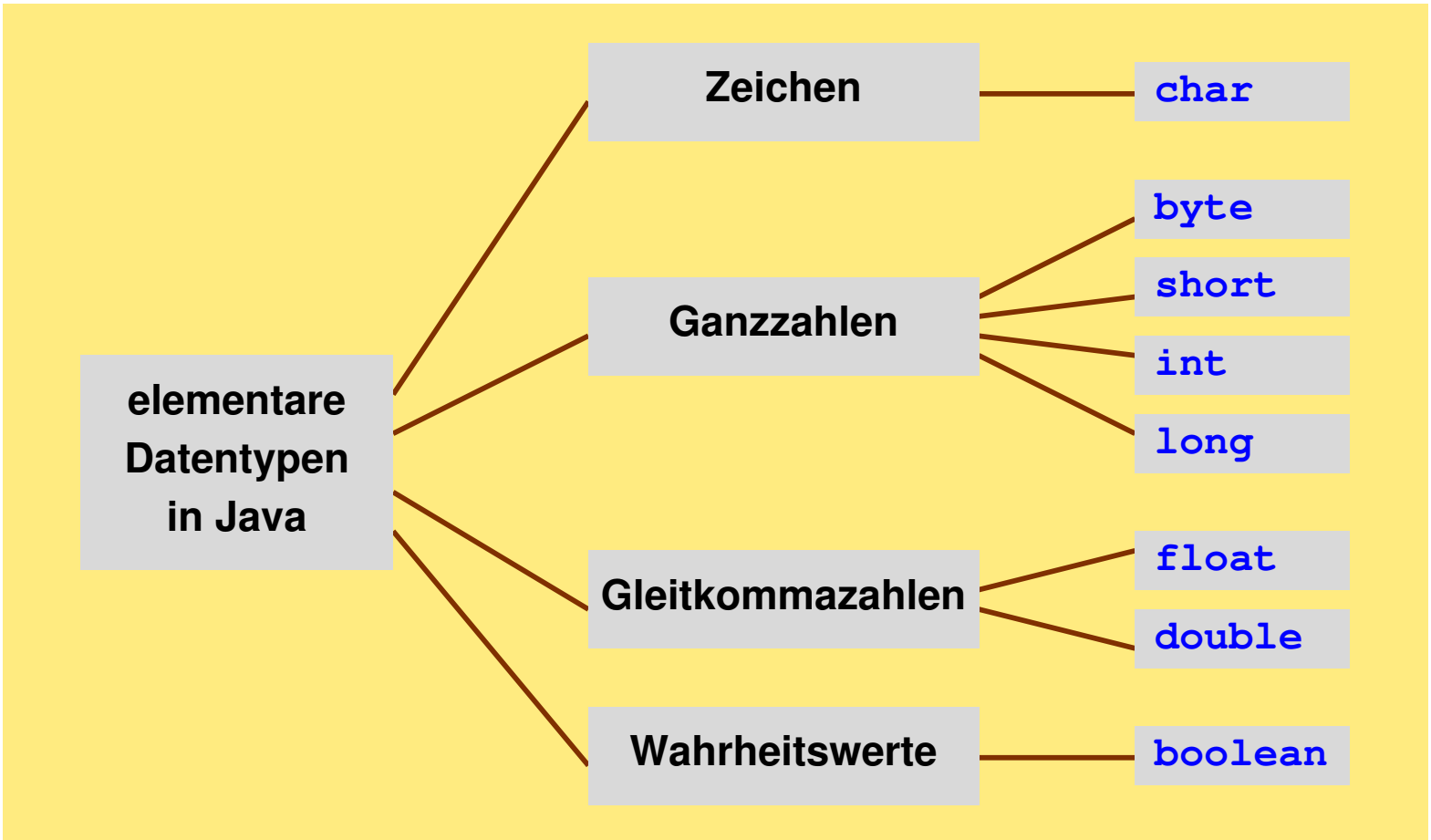
Werte sind im Allgemeinen klassische algebraische Elemente wie *Zahlen*, *Zeichen*, *Symbole* und *Zeichenketten (Strings)*.

Grundbegriffe

Werte sind im Allgemeinen klassische algebraische Elemente wie *Zahlen*, *Zeichen*, *Symbole* und *Zeichenketten (Strings)*.

Typen bezeichnen eine *Menge gleichartiger Werte*. Für jeden Typ sind bestimmte Operationen definiert. Zudem legt der Typ einheitlich fest, wie jeder seiner Werte als Bitmuster kodiert im Speicher abgelegt wird und wieviel Speicherplatz dafür nötig ist.

Elementare Datentypen in Java – Übersicht



Merke: Zeichenkettentyp `String` zählt nicht zu elementaren Typen

Elementare Datentypen kennenlernen

Zu elementaren Datentypen schauen wir uns im Folgenden an:

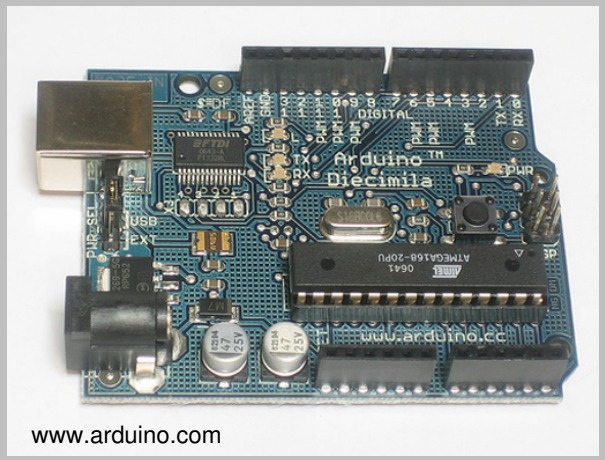
- Wie sehen Werte dieses Typs aus?
- Wie legt man Variablen für Werte dieses Typs an?
- Wieviel Speicherplatz belegt jeder Wert dieses Typs?
- Welchen Wertebereich besitzt dieser Typ?
- Welche Operationen sind für diesen Typ vordefiniert?
- Wie werden Werte dieses Typs als Bitmuster kodiert?
- Gibt es Besonderheiten und falls ja, welche?

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.

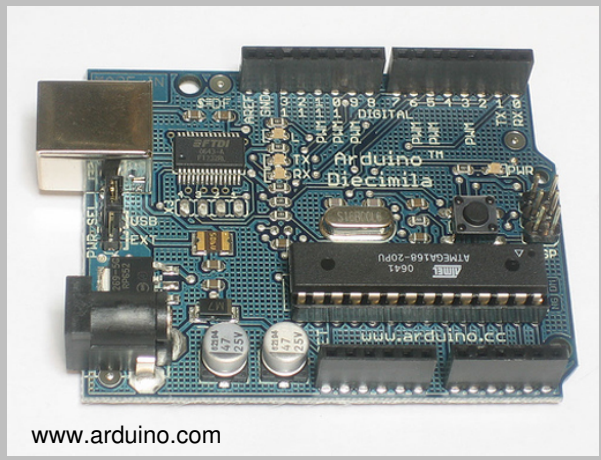


Microcontroller–Speicherdump nach Crashtest

0	1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1
1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0
1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.



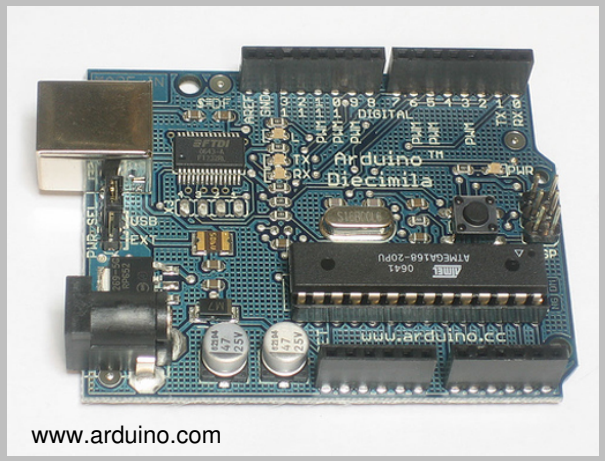
Microcontroller–Speicherdump nach Crashtest

0	1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1
1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0
1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1

```
int v_max = messwert_maximalgeschw();
float a_max = messwert_max_insassenbeschleunigung();
char bremsstufe = messwert_bremsstufe();
```

Wie werden Werte im Speicher abgelegt?

Kodierung in Bitmuster hängt vom Typ ab.

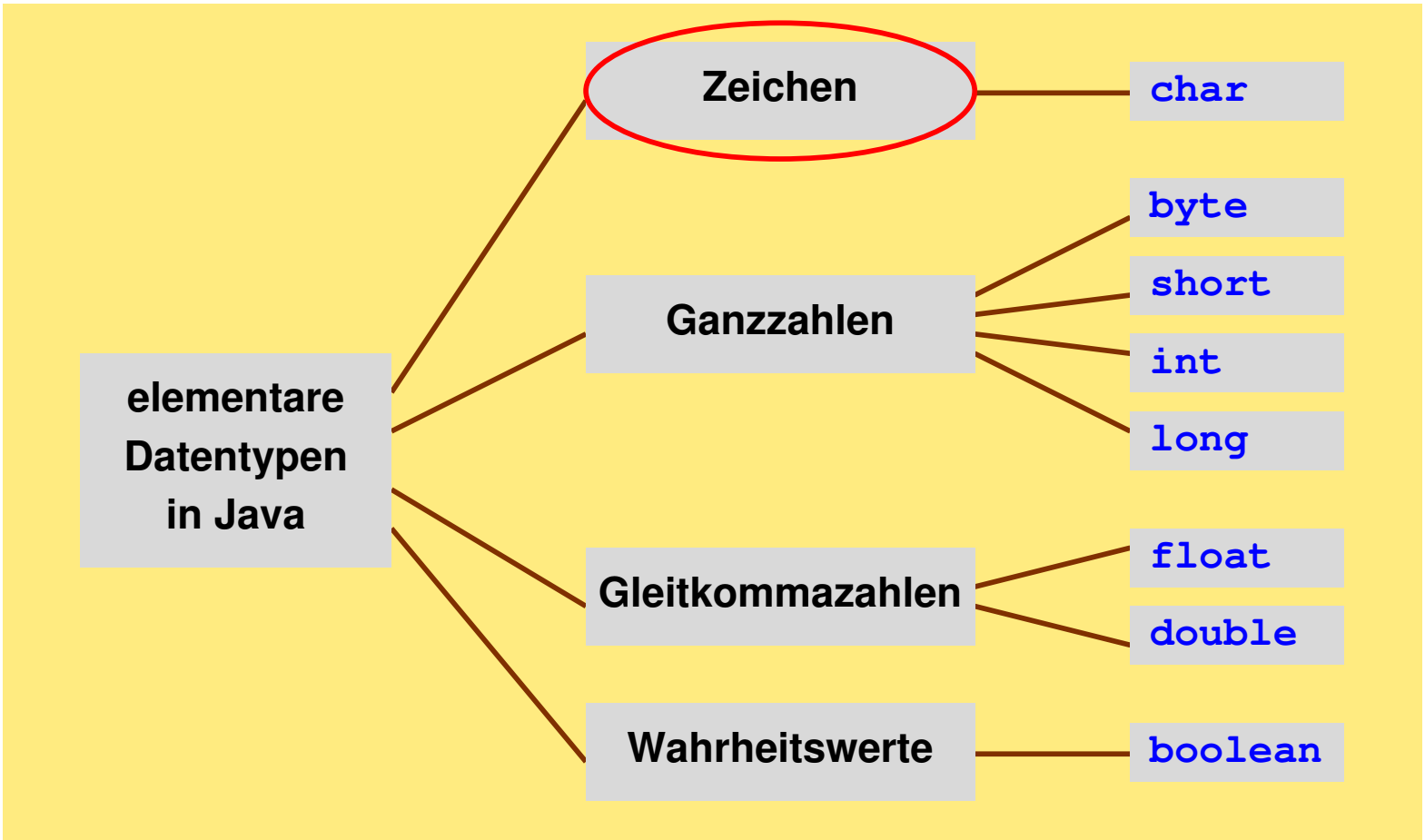


Microcontroller–Speicherdump nach Crashtest

0	1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1			
1	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1	1	1	0	0	1	0			
1	1	0	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1			

```
int v_max = messwert_maximalgeschw();
float a_max = messwert_max_insassenbeschleunigung();
char bremsstufe = messwert_bremsstufe();
```

Elementare Datentypen in Java – Übersicht



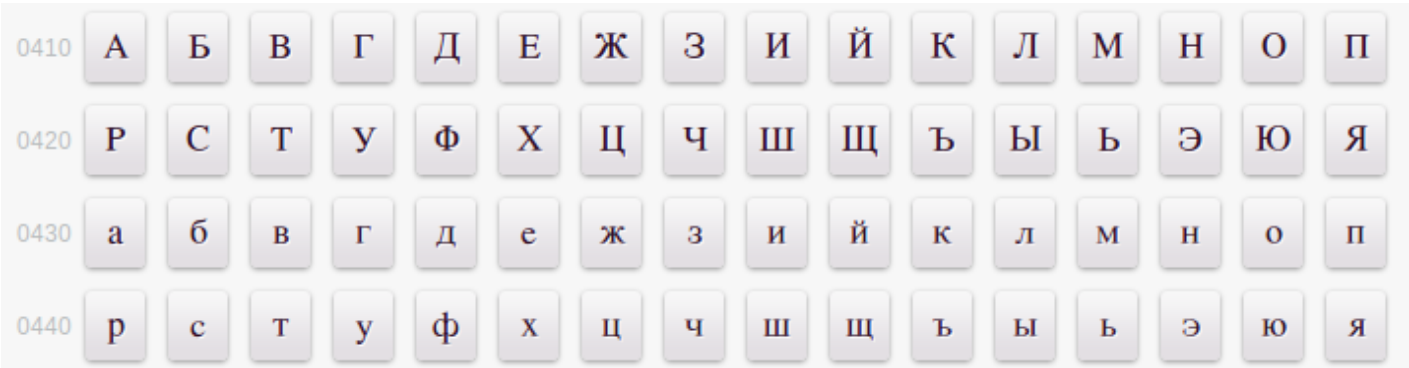
Zeichen

- **Verfügbarer elementarer Typ**

`char`	16-Bit-Unicode-Zeichen	`'A'`, `'\n'`, `'\u0411'`

- **Wertevorrat**

  65536 Zeichen entsprechend Tabelle (Auszug)



www.unicode-table.com

- **Wertenotation**
  - Zeichen in Hochkommas (siehe oben)
  - Jedem Zeichen feste Zahl zugeordnet, z.B. `'A'` die 65
  - Zeichen und zugeordnete Zahl gleichwertig nutzbar
  - Spiegelt „alphabetische Ordnung“ wider, z.B. `'A'` $<$ `'B'`
  - sinnvolle Operatoren: `==`, `!=`, `<`, `>`, `<=`, `>=`

Zeichen

Die ersten 128 Zeichen (Unicode-Nummern 0 bis 127) stimmen mit der älteren 8-Bit-Zeichenkodierung ASCII überein.

Die ASCII-Tabelle ist faktisch der Anfangsteil der Unicode-Tabelle.

Dort findet man die englischen Buchstaben, Ziffern und häufig benutzte druckbare Sonderzeichen.

	32	0	48	@	64	P	80	`	96	p	112
!	33	1	49	A	65	Q	81	a	97	q	113
"	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
'	39	7	55	G	71	W	87	g	103	w	119
(	40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[	91	k	107	{	123
,	44	<	60	L	76	\	92	l	108	|	124
-	45	=	61	M	77	]	93	m	109	}	125
.	46	>	62	N	78	^	94	n	110	~	126
/	47	?	63	O	79	_	95	o	111	□	127

American Standard Code for Information Interchange (ASCII)

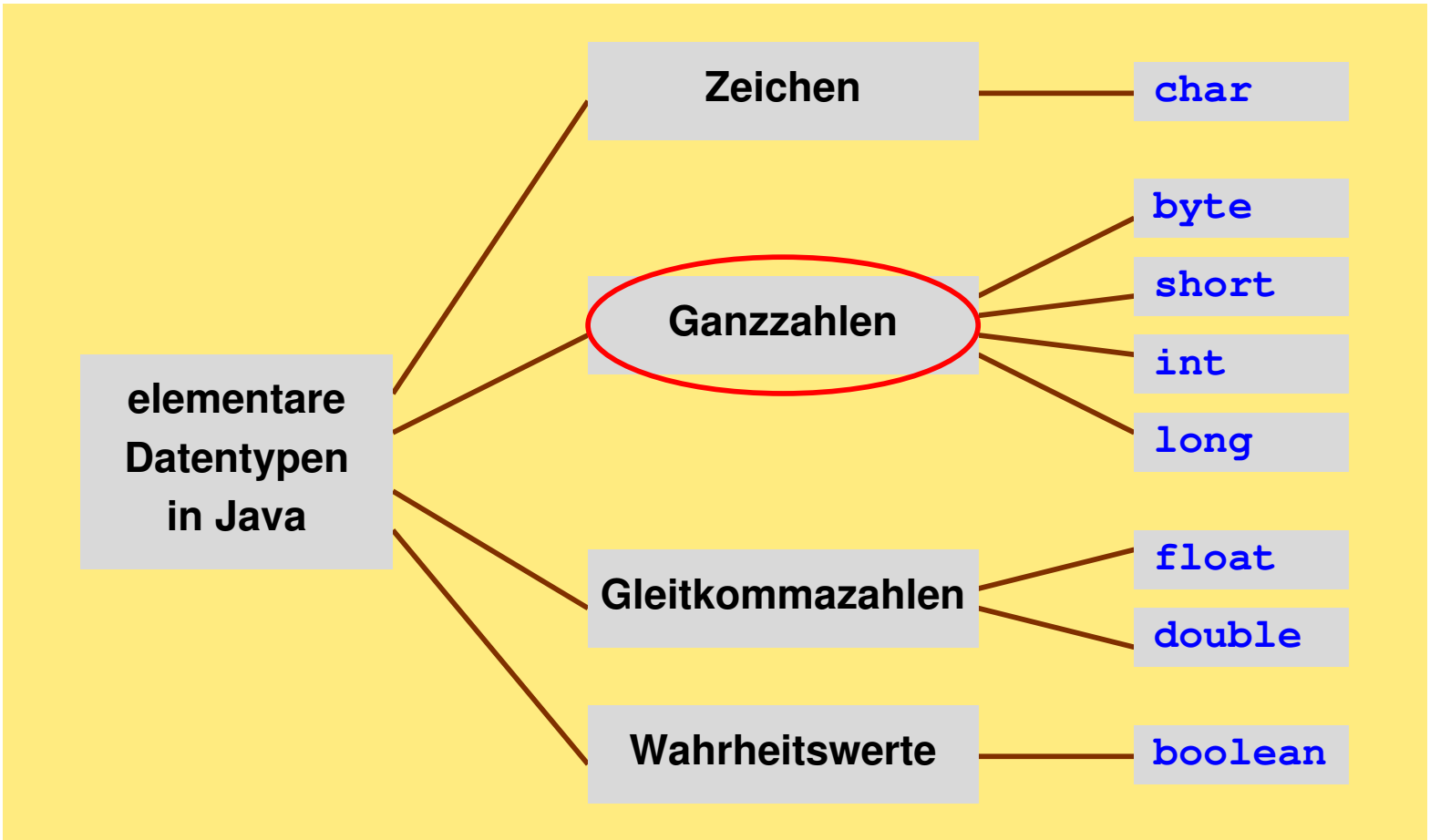
Nicht druckbare Zeichen (Escape-Sequenzen)

Nonprinting or hard-to-print characters		
Name of character	**Written in C**	**ASCII Value**
alert	`\a`	7
backslash	`\\`	92
backspace	`\b`	8
carriage return	`\r`	13
double quote	`\"`	34
form feed	`\f`	12
horizontal tab	`\t`	9
newline	`\n`	10
null character	`\0`	0
single quote	`\'`	39
vertical tab	`\v`	11

- Escape character (\)
- Escape sequence (e.g. `\n`)

$\Longrightarrow$ dürfen z.B. in **`print`**-Ausgaben verwendet werden.

Elementare Datentypen in Java – Übersicht



Ganzzahltypen

Typ	Größe in Byte	Größe in Bit	Wertebereich
`byte`	1	8	$-128 \ldots 127$
`short`	2	16	$-32768 \ldots 32767$
`int`	4	32	$-2^{31} \ldots 2^{31} - 1$
`long`	8	64	$-2^{63} \ldots 2^{63} - 1$

Größeneinheiten für Datenspeicherung

1K	2^{10}	1 024	Kilo	griech. „tausend“
1M	2^{20}	1 048 576	Mega	gr. „groß“
1G	2^{30}	1 073 741 824	Giga	gr. „riesig“
1T	2^{40}	1 099 511 627 776	Tera	gr. „ungeheuerlich“

Ganze Zahlen

- **Wertenotation in verschiedenen Zahlensystemen**
  - **Dezimal** (Basis 10): Standard, ohne führende Nullen . . . 42
  - **Binär** (Basis 2): `0b` voranstellen . . . . . `0b101` für 5
  - **Hexadezimal** (Basis 16): `0x` voranstellen . . . . . `0x1A` für 26
  - **Oktal** (Basis 8): `0` voranstellen . . . . . `022` für 18 (!!!)
  - Bei Vorgabe von Werten des Typs `long` stellt man ein `L` oder `l` (kleines L) hintenan, z.B. `4711L`

Ganze Zahlen

- **Wertenotation in verschiedenen Zahlensystemen**
  - **Dezimal** (Basis 10): Standard, ohne führende Nullen . . . 42
  - **Binär** (Basis 2): `0b` voranstellen . . . . . `0b101` für 5
  - **Hexadezimal** (Basis 16): `0x` voranstellen . . . . . `0x1A` für 26
  - **Oktal** (Basis 8): `0` voranstellen . . . . . `022` für 18 (!!!)
  - Bei Vorgabe von Werten des Typs `long` stellt man ein `L` oder `l` (kleines L) hintenan, z.B. `4711L`
- **Operatoren**
  - `==, !=, <, >, <=, >=, +, -, *, /, %, ++, --`
  - modulo-Operator `%` liefert den Rest bei ganzzahliger Division, z.B. `7 % 4` ist `3`

Ganze Zahlen

- **Wertenotation in verschiedenen Zahlensystemen**
  - **Dezimal** (Basis 10): Standard, ohne führende Nullen . . . 42
  - **Binär** (Basis 2): `0b` voranstellen . . . . . `0b101` für 5
  - **Hexadezimal** (Basis 16): `0x` voranstellen . . . . . `0x1A` für 26
  - **Oktal** (Basis 8): `0` voranstellen . . . . . `022` für 18 (!!!)
  - Bei Vorgabe von Werten des Typs `long` stellt man ein `L` oder `l` (kleines L) hintenan, z.B. `4711L`
- **Operatoren**
  - `==, !=, <, >, <=, >=, +, -, *, /, %, ++, --`
  - modulo-Operator `%` liefert den Rest bei ganzzahliger Division, z.B. `7 % 4` ist `3`
- **Bitmuster-Kodierung im Speicher**
  - *Zweierkomplement* (entspricht bei Ganzzahlen ≥ 0 der *einfachen Binärdarstellung*, für Ganzzahlen < 0 beginnt das Bitmuster stets mit 1)

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$
- binäres Stellenwertsystem: Basis 2, Ziffern 0 und 1
  $41_{(10)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101001_{(2)}$

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$
- binäres Stellenwertsystem: Basis 2, Ziffern 0 und 1
  $41_{(10)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101001_{(2)}$

Umrechnung dezimal $\longrightarrow$ binär

$$
\begin{array}{rcrl}
41 & :2 = & 20 & \text{Rest } \mathbf{1} \\
20 & :2 = & 10 & \text{Rest } \mathbf{0} \\
10 & :2 = & 5 & \text{Rest } \mathbf{0} \\
5 & :2 = & 2 & \text{Rest } \mathbf{1} \\
2 & :2 = & 1 & \text{Rest } \mathbf{0} \\
1 & :2 = & 0 & \text{Rest } \mathbf{1}
\end{array}
\Bigg\uparrow
$$

Einfache Binärdarstellung

- dezimales Stellenwertsystem: Basis 10, Ziffern 0 bis 9
  $41 = 4 \cdot 10^1 + 1 \cdot 10^0$
- binäres Stellenwertsystem: Basis 2, Ziffern 0 und 1
  $41_{(10)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101001_{(2)}$

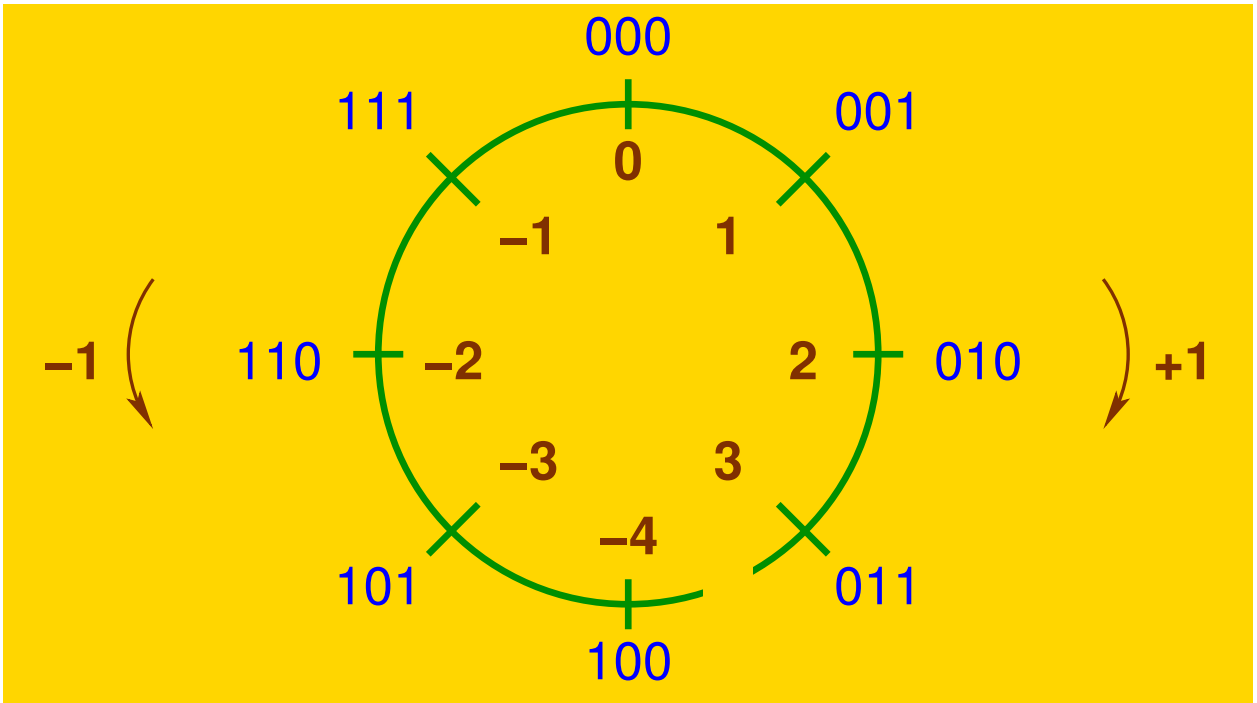
Umrechnung dezimal $\longrightarrow$ binär

$$
\begin{array}{rcrl}
41 & :2 & = & 20 \quad \text{Rest } \mathbf{1} \\
20 & :2 & = & 10 \quad \text{Rest } \mathbf{0} \\
10 & :2 & = & 5 \quad \text{Rest } \mathbf{0} \\
5 & :2 & = & 2 \quad \text{Rest } \mathbf{1} \\
2 & :2 & = & 1 \quad \text{Rest } \mathbf{0} \\
1 & :2 & = & 0 \quad \text{Rest } \mathbf{1}
\end{array}
\Bigg\uparrow
$$

Falls erforderlich, in Binärdarstellung links Nullen auffüllen
z.B. 32-Bit-Integerwert 41 ist das Bitmuster:
00000000 00000000 00000000 00101001

Ganzzahlen mit Vorzeichen im Zweierkomplement

Idee: Für beliebige Ganzzahlen soll gelten $a - b = a + (-b)$, wobei $+$ und $-$ die Addition bzw. Subtraktion auf Bitebene sind. Dadurch sind arithmetische Operationen besonders effizient ausführbar.



Bildung des n-Bit-Zweierkomplements

- in Binärdarstellung wird $-z$ durch $2^n - z$ ersetzt
- d.h. Invertieren aller Bits in der Binärdarstellung von z und Addition von 1

Beispiel

Repräsentation von -13 im 8-Bit-Zweierkomplement

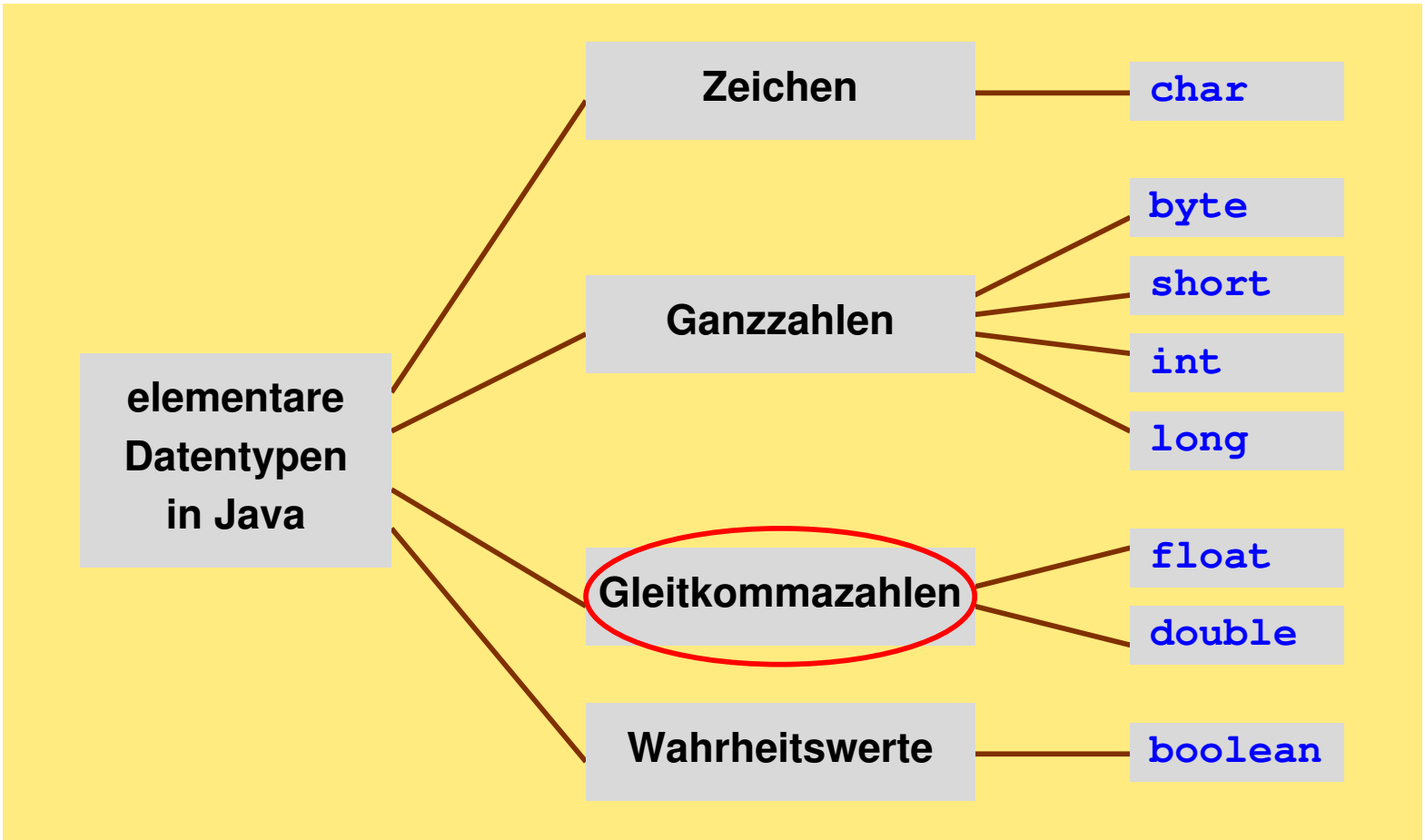
	0000	1101	$z = 13_{(10)} = 1101_{(2)}$
	1111	0010	Inverses
$+$	0000	0001	Addition von 1
	1111	0011	$-z$ im Zweierkomplement

Zweierkomplement für `short`-Werte

16 Bit, d.h. Wertebereich $-2^{15} \ldots 2^{15} - 1$, also $-32768 \ldots 32767$

0111	1111	1111	1111	→	32767
0111	1111	1111	1110	→	32766
...					...
0000	0000	0000	0010	→	2
0000	0000	0000	0001	→	1
0000	0000	0000	0000	→	0
1111	1111	1111	1111	→	-1
1111	1111	1111	1110	→	-2
...					...
1000	0000	0000	0001	→	-32767
1000	0000	0000	0000	→	-32768

Elementare Datentypen in Java – Übersicht



Gleitkommazahlen

Typ	**Größe in Byte**	**Wertebereich (Näherungsangabe)**	**Genauigkeit (Dezimal-stellen)**
`float`	4	$-3,4 \cdot 10^{38} \ldots 3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308} \ldots 1,7 \cdot 10^{308}$	ca. 15

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38} \ldots 3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308} \ldots 1,7 \cdot 10^{308}$	ca. 15

Große Zahlen zum Vergleich

- 10^{22} Byte: *Gesamtkapazität* der bisher weltweit produzierten digitalen *Speichermedien* (10 Zettabyte)

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38} \ldots 3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308} \ldots 1,7 \cdot 10^{308}$	ca. 15

Große Zahlen zum Vergleich

- 10^{22} Byte: *Gesamtkapazität* der bisher weltweit produzierten digitalen *Speichermedien* (10 Zettabyte)
- 10^{23} Moleküle: *menschlicher Körper* aus etwa 10^{14} Zellen

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38} \ldots 3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308} \ldots 1,7 \cdot 10^{308}$	ca. 15

Große Zahlen zum Vergleich

- 10^{22} Byte: *Gesamtkapazität* der bisher weltweit produzierten digitalen *Speichermedien* (10 Zettabyte)
- 10^{23} Moleküle: *menschlicher Körper* aus etwa 10^{14} Zellen
- 10^{100} wird *Googol* genannt, Namensursprung von Google

Gleitkommazahlen

Typ	Größe in Byte	Wertebereich (Näherungsangabe)	Genauigkeit (Dezimal-stellen)
`float`	4	$-3,4 \cdot 10^{38} \ldots 3,4 \cdot 10^{38}$	ca. 7
`double`	8	$-1,7 \cdot 10^{308} \ldots 1,7 \cdot 10^{308}$	ca. 15

Große Zahlen zum Vergleich

- 10^{22} Byte: *Gesamtkapazität* der bisher weltweit produzierten digitalen *Speichermedien* (10 Zettabyte)
- 10^{23} Moleküle: *menschlicher Körper* aus etwa 10^{14} Zellen
- 10^{100} wird *Googol* genannt, Namensursprung von Google
- 10^{200} Elementarteilchen: nach heutiger Schätzung *Universum*

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* . . . . . `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85E-12` für $1,85 \cdot 10^{-12}$

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85E-12` für $1,85 \cdot 10^{-12}$
  - Bei Vorgabe von Werten des Typs `float` stellt man ein `f` hintenan, z.B. `9.81f` oder `1.85E+12f`

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85E-12` für $1,85 \cdot 10^{-12}$
  - Bei Vorgabe von Werten des Typs `float` stellt man ein `f` hintenan, z.B. `9.81f` oder `1.85E+12f`
- **Operatoren und Eigenschaften**
  - `==, !=, <, >, <=, >=, +, -, *, /,`
    viele Funktionen aus Bibliotheksklasse `Math`

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85E-12` für $1,85 \cdot 10^{-12}$
  - Bei Vorgabe von Werten des Typs `float` stellt man ein `f` hintenan, z.B. `9.81f` oder `1.85E+12f`
- **Operatoren und Eigenschaften**
  - `==, !=, <, >, <=, >=, +, -, *, /,`
    viele Funktionen aus Bibliotheksklasse `Math`
  - Beim Rechnen häufig *numerische Ungenauigkeiten*
    (z.B. $0.1 + 0.1 + 0.1$ kann ergeben 0.2999999 statt 0.3)
  - Deshalb Gleitkommazahlen möglichst nicht auf Gleichheit ($==$) oder Ungleichheit ($!=$) vergleichen, sondern ε-Toleranzbereich definieren

Gleitkommazahlen

- **Wertenotation**
  - immer **dezimal**
  - **Dezimalkomma** ist stets der *Punkt* ..... `3.14` für $3,14$
  - Vorkommanullen dürfen weggelassen werden `.75` für $0,75$
  - *Exponentennotation* ..... `1.85E-12` für $1,85 \cdot 10^{-12}$
  - Bei Vorgabe von Werten des Typs `float` stellt man ein `f` hintenan, z.B. `9.81f` oder `1.85E+12f`
- **Operatoren und Eigenschaften**
  - `==, !=, <, >, <=, >=, +, -, *, /,`
    viele Funktionen aus Bibliotheksklasse `Math`
  - Beim Rechnen häufig *numerische Ungenauigkeiten*
    (z.B. $0.1 + 0.1 + 0.1$ kann ergeben 0.2999999 statt 0.3)
  - Deshalb Gleitkommazahlen möglichst nicht auf Gleichheit ($==$) oder Ungleichheit ($!=$) vergleichen, sondern ε-Toleranzbereich definieren
- **Bitmuster-Kodierung im Speicher**
  - nach Standard IEEE754 $\longrightarrow$ in der Hörsaalübung detailliert

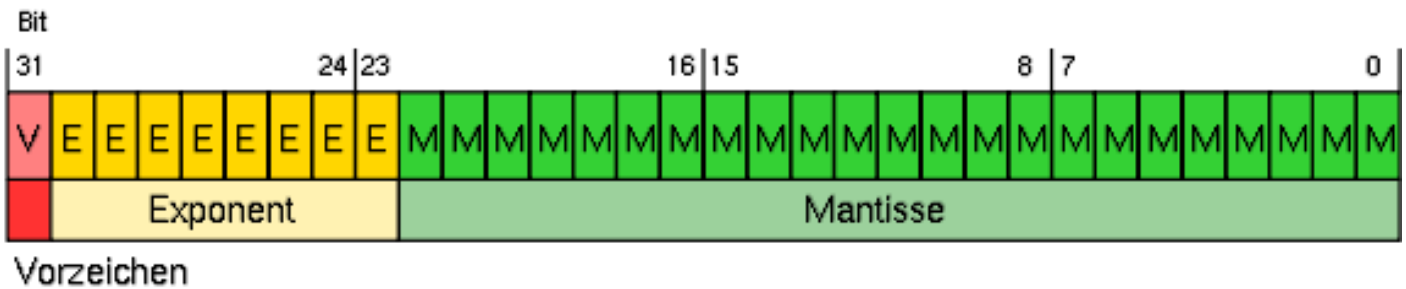
Gleitkommaformat nach IEEE754

$$\text{zahl} = (-1)^v \cdot z(m) \cdot 2^{z(e)}$$

Kodierung der drei Komponenten:

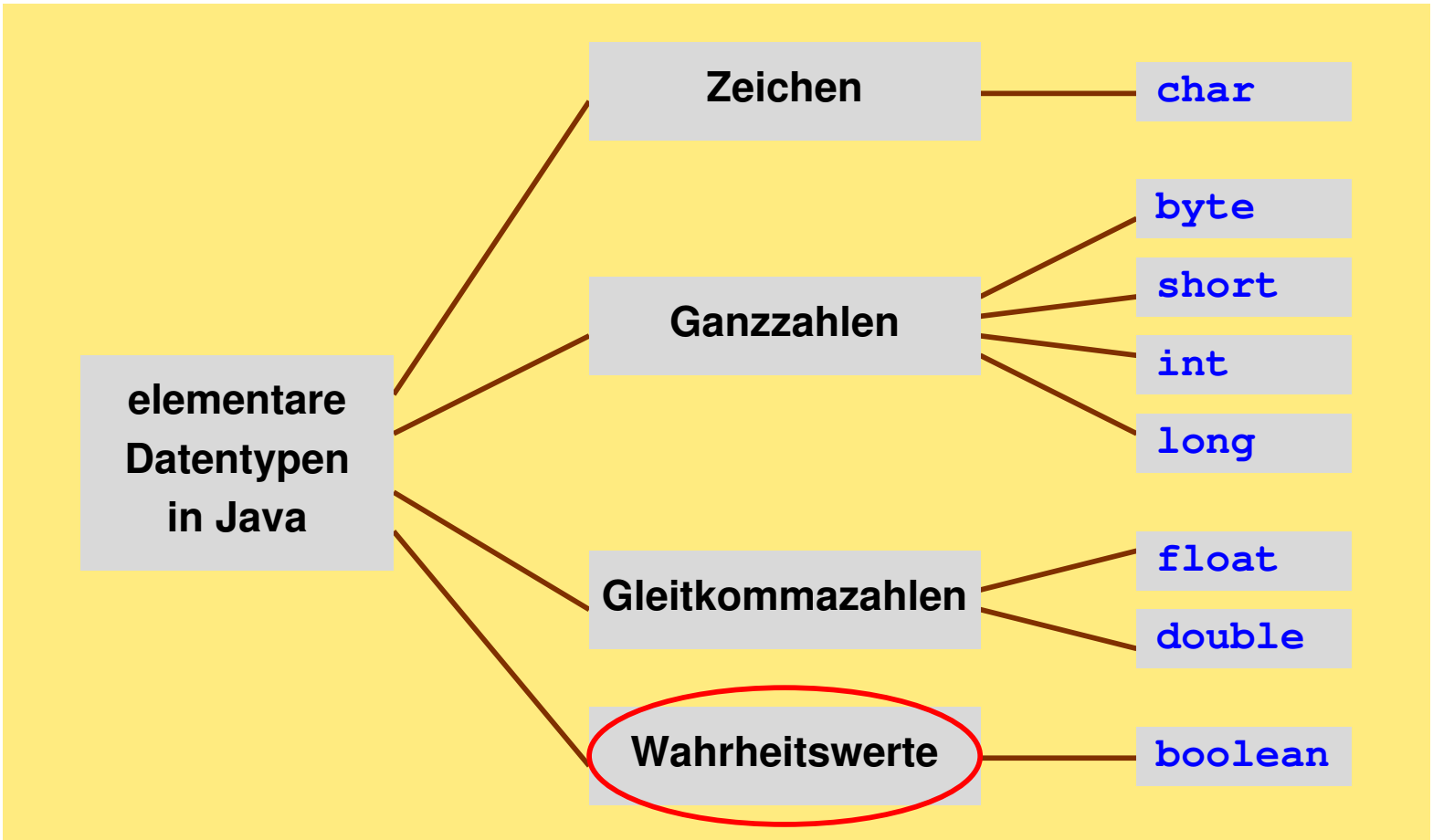
Vorzeichen v ..... 1 Bit
Exponent e ..... k Bits
Mantisse m ..... l Bits

k, l und die Berechnung von $z(m)$ und $z(e)$ sind prozessorabhängig



Beispiel `float`: 32 Bit (4 Byte)

Elementare Datentypen in Java – Übersicht



Wahrheitswerte

- **Verfügbarer elementarer Typ**

`boolean`	1 Bit	`true`, `false`

Wahrheitswerte

- **Verfügbarer elementarer Typ**

`boolean`	1 Bit	`true`, `false`

- **Wertenotation**
  - `true` (*wahr*) oder `false` (*falsch*) direkt vorgeben

Wahrheitswerte

- **Verfügbarer elementarer Typ**

`boolean`	1 Bit	`true`, `false`

- **Wertenotation**
  - `true` (*wahr*) oder `false` (*falsch*) direkt vorgeben
  - Wahrheitswerte entstehen stets als Ergebnis von Vergleichsoperationen
    `(5 < 8)` ergibt ..... `true`

Wahrheitswerte

- **Verfügbarer elementarer Typ**

`boolean`	1 Bit	`true`, `false`

- **Wertenotation**
  - `true` (*wahr*) oder `false` (*falsch*) direkt vorgeben
  - Wahrheitswerte entstehen stets als Ergebnis von Vergleichsoperationen

    `(5 < 8)` ergibt ..... `true`

    `(7 == 4)` ergibt ..... `false`

Wahrheitswerte

- **Verfügbarer elementarer Typ**

`boolean`	1 Bit	`true`, `false`

- **Wertenotation**
  - `true` (*wahr*) oder `false` (*falsch*) direkt vorgeben
  - Wahrheitswerte entstehen stets als Ergebnis von Vergleichsoperationen

    `(5 < 8)` ergibt ..... `true`

    `(7 == 4)` ergibt ..... `false`

    `(3.14 + 2.72 >= 5)` ergibt ..... `true`

Wahrheitswerte

- **Verfügbarer elementarer Typ**

`boolean`	1 Bit	`true`, `false`

- **Wertenotation**
  - `true` (*wahr*) oder `false` (*falsch*) direkt vorgeben
  - Wahrheitswerte entstehen stets als Ergebnis von Vergleichsoperationen

    `(5 < 8)` ergibt ..... `true`

    `(7 == 4)` ergibt ..... `false`

    `(3.14 + 2.72 >= 5)` ergibt ..... `true`
- **Wichtige Operatoren**
  - `==` (Äquivalenz), `!=` (Antivalenz), `!` (Negation), `&&` (logisches Und), `||` (logisches Oder)

Kodierungsformate elementarer Datentypen in Bitketten

Typ	Größe	Format
Ganzzahlen		
`byte`	8 Bit	Zweierkomplement
`short`	16 Bit	Zweierkomplement
`int`	32 Bit	Zweierkomplement
`long`	64 Bit	Zweierkomplement
Fließkommazahlen		
`float`	32 Bit	IEEE 754
`double`	64 Bit	IEEE 754
Weitere Datentypen		
`boolean`	1 Bit	`true`, `false`
`char`	16 Bit	16-Bit-Unicode

C. Ullenboom. Java ist auch eine Insel. Rheinwerk-Verlag, 2014

Definitionen

Eine **Variable** ist ein Platzhalter (Behälter, *Speicherbereich*), der nacheinander verschiedene Werte (in Java: gleichen Typs) annehmen kann, die über einen eindeutigen *Namen* (Bezeichner) zugänglich gemacht werden.

```
double kontostand = .0;

int anzahlStudenten = 24;

long anzahlZugriffe = 87654321L;

char geschlecht = 'm';

boolean erledigt = true;

float temperatur = 20.75f;

kontostand = 1.5E+6;
```

Definitionen

Eine **Konstante** ist ein Name, der bei seiner Deklaration mit einem Wert verbunden wird. Diesen Wert behält die Konstante während ihrer gesamten Lebensdauer unverändert bei. In Java werden Konstanten mit dem Schlüsselwort `final` eingeführt.

```
final double PI = 3.14159265;

final float G = 9.81f; //f kodiert Wert direkt als float

final int V = 299793218;

V = 0; //Compiler meldet hier einen Fehler
```

Definitionen

Literale sind explizit im Programmtext angegebene *Konstantenwerte* oder *Konstantenausdrücke*.

```
final double G = 9.81;
double phi = 45 * 3.14159265 / 180;
```

- `G` ist eine *Konstante* vom Typ `double` mit dem Wert `9.81`.
- `phi` ist eine *Variable* vom Typ `double`.
- Die Werte `9.81` und `45 * 3.14159265 / 180` sind jeweils *Literale*.
- Variablen dürfen auch ohne Wertzuweisung angelegt werden, es wird das vorhandene Bitmuster des ausgefassten Speicherbereiches als Wert (meist 0) interpretiert.

Variablen und Konstanten anlegen („deklarieren“)

```
int i, k, zaehler = 1, x_0, anzahl;
double phi = zaehler, dt = 0.1, alpha;
```

- Mehrere Variablen desselben Typs dürfen durch Komma getrennt hintereinander deklariert werden.
- Zur Wertzuweisung dürfen auch schon bekannte Variablen- oder Konstantennamen genutzt werden, soweit sie **typverträglich** sind.
- Der Typ einer deklarierten Variablen lässt sich (in Java) während ihrer Lebensdauer nicht mehr verändern.

Namen von Variablen und Konstanten

Bezeichner (Namen)

- beginnen mit einem Buchstaben
- dürfen enthalten: engl. Buchstaben, Ziffern, Unterstriche _, Sonderzeichen (seit Java-Version 1.7)
- dürfen (theoretisch) beliebig lang sein
- Groß- und Kleinbuchstaben werden unterschieden.
- Java-Schlüsselwörter (wie `class` oder `false`) nicht zugelassen
- Reservierte Wörter (wie `main`) sollten nicht verwendet werden

Namen von Variablen und Konstanten

Bezeichner (Namen)

- beginnen mit einem Buchstaben
- dürfen enthalten: engl. Buchstaben, Ziffern, Unterstriche _, Sonderzeichen (seit Java-Version 1.7)
- dürfen (theoretisch) beliebig lang sein
- Groß- und Kleinbuchstaben werden unterschieden.
- Java-Schlüsselwörter (wie `class` oder `false`) nicht zugelassen
- Reservierte Wörter (wie `main`) sollten nicht verwendet werden

Konventionen

- Aussagekräftige Namen verwenden
- Variablennamen mit Kleinbuchstaben schreiben
- In zusammengesetzten Wörtern beginnen neue Wortteile mit Großbuchstaben (`anzahlStudenten`)
- Konstanten mit Großbuchstaben schreiben

Grundoperationen

Operator	Beispiel	Wirkung
`+`	`a + b`	Addiert **a** und **b**
`-`	`a - b`	Subtrahiert **b** von **a**
`*`	`a * b`	Multipliziert **a** und **b**
`/`	`a / b`	Dividiert **a** durch **b**
`%`	`a % b`	Liefert den Rest bei der ganzzahligen Division **a / b**

- Operation `%` (Divisionsrest) nur auf Ganzzahltypen definiert
- Division `/` auf Ganzzahltypen schneidet Nachkommastellen ab
- Division durch 0 führt zum Programmabsturz
- Punktrechnung vor Strichrechnung
- Gleichrangige Operatoren von links nach rechts abgearbeitet
- Runde Klammern wie in der Mathematik zulässig

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.
- Bei ungeschickter Programmierung kann es leicht zur *Wertebereichsüberschreitung* kommen

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.
- Bei ungeschickter Programmierung kann es leicht zur *Wertebereichsüberschreitung* kommen
- Bei Ganzzahltypen leider keine Fehlermeldung, es entstehen dann *falsche Ergebnisse*

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

$$\binom{n}{6} = \frac{n!}{(n-6)! \cdot 6!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5)}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6}$$

- Es gibt 13 983 816 Kombinationen bei 6 aus 49.
- Bei ungeschickter Programmierung kann es leicht zur *Wertebereichsüberschreitung* kommen
- Bei Ganzzahltypen leider keine Fehlermeldung, es entstehen dann *falsche Ergebnisse*

Berechnungsidee, so dass keine zu großen Zwischenergebnisse entstehen:

$$\frac{n \cdot (n-1)}{2} \cdot \frac{n-2}{3} \cdot \frac{n-3}{4} \cdot \frac{n-4}{5} \cdot \frac{n-5}{6}$$

Beispiel: Anzahl Kombinationen „6 aus n“

Wieviele Lotto-Tipps 6 aus n gibt es?

```
public class Komb6ausn {
  public static void main(String[] args) {

    // Anzahl verschiedener Lotto-Tipps bei "6 aus 49" bzw. "6 aus n"

    long n = 49L; //Italien: 90, Ungarn: 45, Litauen: 30

    long komb = n * (n-1) / 2 * (n-2) / 3 * (n-3) / 4 * (n-4) / 5 * (n-5) / 6;

    System.out.println("Es gibt " + komb + " Kombinationen 6 aus " + n);

  }
}
```

Der Zuweisungsoperator = und seine Funktionsweise

```
public class Xz {
  public static void main(String[] args) {
    int x = 5;
    int z = 3;

    x = x + 2;
    z = x * z;

    System.out.println("x: " + x + " z: " + z);
  }
}
```

- Variablen können während Programmabarbeitung ihren Wert verändern.
- Rechts vom Zuweisungsoperator = haben die Variablen ihren alten Wert, daraus wird der neue Wert berechnet und dann der Variablen links vom = zugewiesen.

Abkürzende Schreibweisen

```
public class Xz2 {
  public static void main(String[] args) {
    int x = 5;
    int z = 3;

    x += 2;
    z *= x;

    System.out.println("x: " + x + " z: " + z);
  }
}
```

- `<Variable> = <Variable> <ArithOp> <Operand>;` kann gleichwertig kürzer geschrieben werden durch `<Variable> <ArithOp>= <Operand>;`
- `<ArithOp>` ist beliebige arithmetische Operation
- `<Operand>` beliebige typverträgliche Variable oder Literal

Inkrementieren $++$ und Dekrementieren $--$

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen

Inkrementieren ++ und Dekrementieren --

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)

Inkrementieren ++ und Dekrementieren --

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`

Inkrementieren ++ und Dekrementieren −−

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`
- Postfix- und Präfixform unterscheiden sich in ihrer Wirkung, wenn sie in Formelausdrücke eingebettet sind („freakig-kompakte Quelltexte“)

Inkrementieren ++ und Dekrementieren −−

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`
- Postfix- und Präfixform unterscheiden sich in ihrer Wirkung, wenn sie in Formelausdrücke eingebettet sind („freakig-kompakte Quelltexte“)
- `a = b * (x++) - c;` entspricht der Abarbeitung

```
a = b * x - c; x++;
```

Inkrementieren ++ und Dekrementieren −−

- Häufig muss in Programmen ein Ganzzahlwert **um** 1 **erhöht** (*inkrementiert*) oder **um** 1 **erniedrigt** (*dekrementiert*) werden, z.B. beim Zählen
- Statt `x = x + 1;` oder `x += 1;` kann man dann noch kürzer schreiben `x++;` (*postfix*) oder `++x;` (*präfix*)
- Analog statt `x = x - 1;` oder `x -= 1;` entsprechend `x--;` oder `--x;`
- Postfix- und Präfixform unterscheiden sich in ihrer Wirkung, wenn sie in Formelausdrücke eingebettet sind („freakig-kompakte Quelltexte“)
- `a = b * (x++) - c;` entspricht der Abarbeitung `a = b * x - c; x++;`
- `a = b * (++x) - c;` entspricht der Abarbeitung `x++; a = b * x - c;`

Erste Bibliotheksmethoden: Arithmetik

Bibliothek/Package: Sammlung von Algorithmen und Strukturen, z.B. Mathematik, Ein- und Ausgabe, nützliche Datenstrukturen, ...

Erste Bibliotheksmethoden: Arithmetik

Bibliothek/Package: Sammlung von Algorithmen und Strukturen, z.B. Mathematik, Ein- und Ausgabe, nützliche Datenstrukturen, . . .

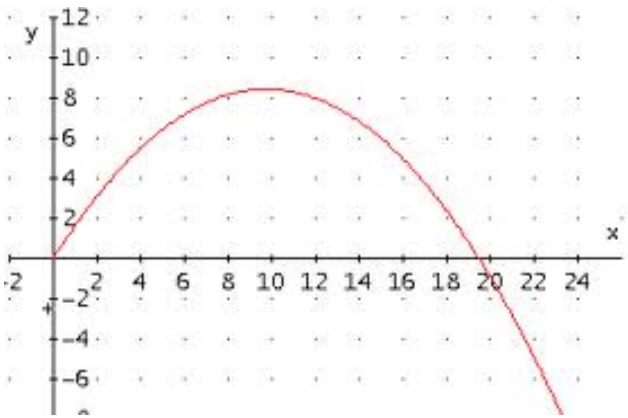
Wir betrachten das Package **`java.lang`** mit der Klasse **`Math`**, die mathematische Funktionen enthält.

`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.
`static double`	**`ceil`**`(double a)` Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer.
`static double`	**`floor`**`(double a)` Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer.
`static double`	**`cos`**`(double a)` Returns the trigonometric cosine of an angle.
`static double`	**`sin`**`(double a)` Returns the trigonometric sine of an angle.
`static double`	**`pow`**`(double a, double b)` Returns the value of the first argument raised to the power of the second argument.

zahlreiche weitere Methoden verfügbar wie **`sqrt`**, **`log`**, **`asin`**, . . .

Einfache Berechnung einer Wurfweite mit Java

```
public class Wurf {
   public static void main(String[] args){
      System.out.println("Schiefer Wurf");
      final double G = 9.81;
      double v0 = 10;
      double winkel = 45;
      double phi = winkel * (Math.PI / 180);
      double weite = v0 * v0 / G * Math.sin(2 * phi);
      System.out.println("Anfangsgeschwindigkeit = " + v0 + "m/s.");
      System.out.println("Winkel = " + winkel + " Grad.");
      System.out.println("Wurfweite = " + weite + "m.");
    }
}
```



(Pseudo)Zufallszahlen erzeugen (Wuerfel.java)

```
public class Wuerfel {
  public static void main(String[] args) {
    // dreimal hintereinander wuerfeln
    System.out.println("1. Wurf: " + ((int) (Math.random() * 6) + 1));
    System.out.println("2. Wurf: " + ((int) (Math.random() * 6) + 1));
    System.out.println("3. Wurf: " + ((int) (Math.random() * 6) + 1));
  }
}
```

- **`Math.random()`** liefert eine Pseudozufallszahl zwischen 0 und 1, aber stets kleiner als 1, als **`double`**-Wert
- Multiplikation mit 6 skaliert in Intervall $[0, 6)$
- Durch Voranstellen von **`(int)`** wird der **`double`**-Wert in einen ganzzahligen **`int`**-Wert umgewandelt durch Abschneiden aller Nachkommastellen. Resultierende Ganzzahlwerte liegen zwischen 0 und 5
- Abschließende Addition von 1 verschiebt in Ganzzahl-Intervall $[1, 6]$

Einfache Bildschirmausgabe mit `print` und `println`

```
public class UnformatierteAusgabe {
  public static void main(String[] args) {
    double phi = 45 * Math.PI / 180;

    System.out.print("Ausgabe ohne ");
    System.out.println("Zeilenumbruch.");
    System.out.println(phi);
    System.out.println("Winkel phi: " + phi);
  }
}
```

```
Ausgabe ohne Zeilenumbruch.
0.7853981633974483
Winkel phi: 0.7853981633974483
```

- In der Bibliotheksklasse `System.out` stehen die Methoden `print` und `println` zur *unformatierten* Bildschirmausgabe
- Beide Methoden zeigen die übergebene Zeichenkette an, wobei `println` noch einen *Zeilenumbruch* am Ende anfügt.
- Zusätzliche Zeilenumbrüche können in der übergebenen Zeichenkette durch `\n` gesetzt werden.
- Variablenwerte elementarer Datentypen werden zur Ausgabe automatisch in eine Zeichenkette umgewandelt.
- Die Zeichenketten-Operation + hängt Zeichenketten aneinander („*Konkatenieren*“). Bitte nicht mit Zahlenaddition verwechseln

Formatierte Bildschirmausgabe mit `printf`

```
public class FormatierteAusgabe {
  public static void main(String[] args) {
    int grad = 45;
    double phi = grad * Math.PI / 180;

    System.out.printf("Winkel Grad: %d  Bogenmasz: %f\n", grad, phi);
  }
}
```

```
Winkel Grad: 45  Bogenmasz: 0,785398
```

- **`printf`** steht für *formatierte* Ausgabe
- Argumente sind der *Formatstring* begrenzt durch `"  "`, danach folgen durch Kommas getrennt die auszugebenden Variablen
- Im Formatstring *Platzhalter* für jede Variable, eingeleitet durch `%`. Reihenfolge der Platzhalter von links nach rechts entspricht Reihenfolge der Variablen
- Bezeichnungen der Platzhalter fest vorgegeben nach Variablentyp und Formatierungsoption
- Steuerzeichen (Escape-Sequenzen) wie `\n` zulässig

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)
Beispiel: `-1234.56789`

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)
Beispiel: `-1234.56789`
`%.2f` ..... zwei Nachkommastellen

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)

Beispiel: `-1234.56789`

`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)
Beispiel: `-1234.56789`
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: `-1.23456789e+3`

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)
Beispiel: `-1234.56789`
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: `-1.23456789e+3`
`%s` ..... Zeichenkette („string“)

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)
Beispiel: `-1234.56789`
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: `-1.23456789e+3`
`%s` ..... Zeichenkette („string“)
`%b` ..... Wahrheitswert („boolean“)

Ein- und Ausgabe von Variablenwerten

Im Formatstring von **`printf`** werden Platzhalter für Variablenwerte wie folgt zugeordnet:

`%d` ..... Ganzzahl (byte, short, int, long) dezimal („digits“)
`%c` ..... einzelnes Zeichen (char) im Unicode („character“)
`%f` ... Gleitkommazahl (float, double) in Einzelstellennotation („float“)
Beispiel: `-1234.56789`
`%.2f` ..... zwei Nachkommastellen
`%5.3f` ... Normierung: 5 Vor- und 3 Nachkommastellen
`%e` ..... Gleitkommazahl in Exponentialschreibweise („exponential“)
Beispiel: `-1.23456789e+3`
`%s` ..... Zeichenkette („string“)
`%b` ..... Wahrheitswert („boolean“)

seltener genutzt:

`%x` ..... Ganzzahl hexadezimal („heX“)
`%%` ..... Ausgabe des Prozentzeichens

Tastatureingaben mittels `Scanner` entgegennehmen

(Vieleckseiten.java)

```
import java.util.Scanner; //benoetigte Bibliotheken einbinden

public class Vieleckseiten {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
      //Bibliotheksklasse Scanner zum Handling von Tastatureingaben
      //Neues Objekt mit Namen sc der Scannerklasse angelegt
      //System.in waehlt Tastatur (Konsole) fuer Eingabezeichenstrom

    double a;
    int n;

    System.out.print("Regelmaessiges n-Eck\nAnzahl Seiten: ");
    n = Integer.parseInt(sc.next());
      //Tastatureingabe zunaechst als Zeichenkette entgegengenommen
      //Zeichenkette anschliessend in einen int-Wert umgewandelt

    System.out.print("Seitenlaenge: ");
    a = Double.parseDouble(sc.next());
      //Naechste Eingabe entgegennehmen und in double-Wert umwandeln

    System.out.printf("%d Seiten mit der Laenge %f\n", n, a);
  }
}
```

Programmiertechnischer Hintergrund Tastatureingabe

```
import java.util.Scanner;
```

- Während die Bibliotheksklassen `Math`, `System.out` und andere automatisch eingebunden werden, muss man die Bibliotheksklasse `Scanner` ausdrücklich *importieren*, sonst sind die Methoden daraus nicht zugänglich.

Programmiertechnischer Hintergrund Tastatureingabe

```
import java.util.Scanner;
```

- Während die Bibliotheksklassen `Math`, `System.out` und andere automatisch eingebunden werden, muss man die Bibliotheksklasse `Scanner` ausdrücklich *importieren*, sonst sind die Methoden daraus nicht zugänglich.

```
Scanner sc = new Scanner(System.in);
```

- Es können gleichzeitig mehrere Eingabekanäle geöffnet sein (z.B. Tastatur, Datei1, Datei2, ...).
- Um diese unterscheiden zu können, legt man für jeden Kanal (Eingabezeichenstrom) ein gesondertes *Objekt* (Exemplar) der Klasse `Scanner` an unter Nutzung des Schlüsselwortes `new`. Objektname ist frei wählbar, hier: `sc`
- Beim Anlegen des Objektes wird die Quelle des Eingabezeichenstroms festgelegt. `System.in` steht dabei für die Standardeingabe von Tastatur (Konsole).

Wrapper-Klassen für elementare Datentypen

```
a = Double.parseDouble(sc.next());
```

- Auf das Objekt `sc` wird die Methode `next()` angewendet, die die *nächsten* eingegebenen Zeichen bis Entertaste oder bis zum nächsten Leerzeichen als *Zeichenkette* entgegennimmt.
- Zeichenkette muss in den gewünschten elementaren Datentyp *konvertiert* werden. Dazu gibt es für nahezu jeden elementaren Datentyp eine sogenannte Wrapper-Klasse (engl. für „Umschlag“) mit entsprechender Methode, hier `parseDouble`

Klasse	Konvertierungsmethode	Rückgabetyp
java.lang.Boolean	parseBoolean(String s)	boolean
java.lang.Byte	parseByte(String s)	byte
java.lang.Short	parseShort(String s)	short
java.lang.Integer	parseInt(String s)	int
java.lang.Long	parseLong(String s)	long
java.lang.Double	parseDouble(String s)	double
java.lang.Float	parseFloat(String s)	float

Die Wrapper-Klassen werden automatisch eingebunden.

Typumwandlung – Welche Ausgabe erwarten Sie hier?

```java
public class Typumwandlung {
  public static void main(String[] args) {
    int x = 2;
    double y = 3;

    double z = y / x;
    System.out.println(z);
  }
}
```

Typumwandlung – Welche Ausgabe erwarten Sie hier?

```
public class Typumwandlung {
  public static void main(String[] args) {
    int x = 2;
    double y = 3;

    double z = y / x;
    System.out.println(z);
  }
}
```

- Mitunter kommt es vor, dass Variablen unterschiedlichen Typs im gleichen Ausdruck verarbeitet werden (sollen).
- Daraus resultiert Notwendigkeit, Variablen- und/oder Konstantenwerte möglichst *werterhaltend* von einem Typ in einen anderen umzuwandeln.
- Ist der Wertebereich des Zieltyps gleich oder größer (z.B. `int` nach `double`), geht keine Information verloren.
- Solche Konvertierungen werden automatisch und problemlos ausgeführt (*impliziter Typecast*)
- Bei arithmetischen Operationen werden die Operanden vor Operationsausführung in den wertebereichsgrößten der beteiligten Typen umgewandelt.

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
        public static void main(String[] args) {
            int i = 10;
            double d = i;
            System.out.println(d);
        }
}
```

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
    public static void main(String[] args) {
        int i = 10;
        double d = i;
        System.out.println(d);
    }
}
```


int wird *implizit* nach **double** konvertiert.

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
        public static void main(String[] args) {
            double d = 10.0;
            int i = d;
            System.out.println(i);
        }
}
```

Anders ist die Situation, wenn bei einer Typkonvertierung Information verloren gehen kann.

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
    public static void main(String[] args) {
        double d = 10.0;
        int i = d;
        System.out.println(i);
    }
}
```

"Type mismatch: cannot convert from double to int"

Der Compiler erkennt dies und bricht das Compilieren mit einer Fehlermeldung ab.

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
    public static void main(String[] args) {
        double d = 10.0;
        int i = (int) d;
        System.out.println(i);
    }
}
```

double muss *explizit* nach **int** konvertiert werden.

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
        public static void main(String[] args) {
            double d = 9.99999;
            int i = (int) d;
            System.out.println(i);
        }
}
```

Als Programmierer kann man Typecasts im Quelltext erzwingen, indem man den Zieltyp in runde Klammern vor die Variable oder das Literal schreibt. Dieser *explizite Typecast* hat eine höhere Priorität (bindet stärker) als alle arithmetischen Operatoren.

Typumwandlung (Typecast, Typkonvertierung)

```
public class MyTypesTest {
        public static void main(String[] args) {
            double d = 9.99999;
            int i = (int) d;
            System.out.println(i);
        }
}
```

Ausgabe: 9

(Nachkommastellen werden immer abgeschnitten)

Typumwandlung (Typecast, Typkonvertierung)

Implizite Typumwandlung

vom wertebereichskleineren in wertebereichsgrößeren Typ wird vom Compiler automatisch durchgeführt.

Von Typ ...		...nach Typ
`byte`	$\rightarrowtail$	`short, char, int, long, float, double`
`char, short`	$\rightarrowtail$	`int, long, float, double`
`int`	$\rightarrowtail$	`long, float, double`
`long`	$\rightarrowtail$	`float, double`
`float`	$\rightarrowtail$	`double`
«alle»	$\rightarrowtail$	`String`

Tabelle 2.9. Harmloses Casting (automatisch durch den Compiler)

Explizite Typumwandlung

- muss im Programm angegeben werden,
  z.B. `int i = (int)3.14;`
- kann zu Informationsverlust führen

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` ...“

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` . . .“

Numerische Ungenauigkeiten minimieren

- Zählen in Ganzzahlschritten mit Ganzzahltypen stets präzise, aber mit Gleitkommatypen können sich Ungenauigkeiten einschleichen und aufschaukeln
- Ganzzahloperationen ebenfalls absolut genau, aber in vergleichsweise kleinem Wertebereich

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` . . .“

Numerische Ungenauigkeiten minimieren

- Zählen in Ganzzahlschritten mit Ganzzahltypen stets präzise, aber mit Gleitkommatypen können sich Ungenauigkeiten einschleichen und aufschaukeln
- Ganzzahloperationen ebenfalls absolut genau, aber in vergleichsweise kleinem Wertebereich

Speicherschonende Programmierung

- Wenn ein Schalter z.B. nur 8 unterscheidbare Stufen kennt, braucht man dafür wahrlich keine 64 Bit im Speicher

Wofür ist (expliziter) Typecast sinnvoll?

„Nehmen wir für Zahlen doch einfach immer `double` . . .“

Numerische Ungenauigkeiten minimieren

- Zählen in Ganzzahlschritten mit Ganzzahltypen stets präzise, aber mit Gleitkommatypen können sich Ungenauigkeiten einschleichen und aufschaukeln
- Ganzzahloperationen ebenfalls absolut genau, aber in vergleichsweise kleinem Wertebereich

Speicherschonende Programmierung

- Wenn ein Schalter z.B. nur 8 unterscheidbare Stufen kennt, braucht man dafür wahrlich keine 64 Bit im Speicher

Möglichst zeiteffiziente Operationsausführung

- Gleitkommaoperationen sind zeitaufwendiger als Ganzzahloperationen

Prioritäten von Operatoren in Java (Auswahl)

hoch	`()`	Methodenaufruf und Klammerung
	`+ -` `++ --`	Vorzeichen Inkrement, Dekrement
	`(Typ)`	Typecast
	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`< <= > >=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
niedrig	`=`	Zuweisung

- Innerhalb einer Anweisung werden Operatoren mit hoher Priorität vor solchen mit niedriger Priorität abgearbeitet
- Gleichrangige Operatoren werden (bis auf wenige Ausnahmen wie Zuweisung `=`) von links nach rechts abgearbeitet

Prioritäten von Operatoren in Java (Auswahl)

hoch	`()`	Methodenaufruf und Klammerung
	`+ -` `++ --`	Vorzeichen Inkrement, Dekrement
	`(Typ)`	Typecast
	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`< <= > >=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
niedrig	`=`	Zuweisung

- Innerhalb einer Anweisung werden Operatoren mit hoher Priorität vor solchen mit niedriger Priorität abgearbeitet
- Gleichrangige Operatoren werden (bis auf wenige Ausnahmen wie Zuweisung `=`) von links nach rechts abgearbeitet
- `float v = 0.5f * (int) (9.81 - Math.sqrt(9));`
  ergibt .....

Prioritäten von Operatoren in Java (Auswahl)

hoch	`()`	Methodenaufruf und Klammerung
	`+ -` `++ --`	Vorzeichen Inkrement, Dekrement
	`(Typ)`	Typecast
	`* / %`	Multiplikation, Division, Modulo
	`+ -`	Addition, Subtraktion
	`< <= > >=`	Vergleichsoperatoren
	`== !=`	Vergleichsoperatoren
niedrig	`=`	Zuweisung

- Innerhalb einer Anweisung werden Operatoren mit hoher Priorität vor solchen mit niedriger Priorität abgearbeitet
- Gleichrangige Operatoren werden (bis auf wenige Ausnahmen wie Zuweisung `=`) von links nach rechts abgearbeitet
- `float v = 0.5f * (int) (9.81 - Math.sqrt(9));`
  ergibt ..... `3.0`