Algorithmen am Beispiel „Java“ I

www.java.com/de

Entwicklungsumgebung:
JDK 6.1.0

empfohlener Editor :
http://notepad-plus.sourceforge.net/de/site.htm

Download Java SE Runtime Environment (deutsch):
http://www.soft-ware.net/add.asp?url=http%3A%2F%2Fjava-runtime-environment.soft-ware.net%2Fdownload2.asp%3Fcode%3Djbibb%26pro%3Dp02231

---

1. Sprachmerkmale

am Beispiel des Collatz-Algorithmus
umgangssprachlich: Falls x gerade → teile durch 2, sonst → verdreifache x und erhöhe um 1

Programm mit Java als Formalismus, um präzise einen Algorithmus zu formulieren:

```
/*************************** Collatz.java ****************************************/

import AlgoTools.IO;                          // Klasse Algotools soll benutzt werden

/** Berechnet Collatz-Funktion, d.h.
* Anzahl der Iterationen der Funktion g: N -> N
* bis die Eingabe auf 1 transformiert ist
* mit g(x) = x/2 falls x gerade, 3*x+1 sonst
*/

public class Collatz
{

public static void main(String [] argv)
{

int x, z;                                     // 2 Variablen mit int (Ganzzahl) vereinbaren
x = IO.readInt("Bitte eine Zahl: ");          // lies einen Wert ein [Klasse IO, Methode readlnt]
z = 0; // setze z auf 0

while (x != 1)                                // solange x ungleich 1 ist
{
   if (x % 2 == 0)                            // falls x gerade ist
   x = x / 2;                                 // halbiere x
   else                                       // andernfalls
   x = 3 * x + 1;                             // verdreifache x und add. 1
   z = z+1;                                   // erhöhe z um eins
}
IO.println("Anzahl der Iterationen: " + z);   // gib z aus [Klasse: IO, Methode: printl]

}

}
```

Alternative Schreibweise

```
x = x % 2 == 0 ? x/2 : 3*x + 1; // bedingter Ausdruck mit ? und : (entweder-oder)
                                // wenn Bedingung vor ? wahr, dann weise der linken
                                // Seite den Ausdruck vor : zu, sonst den hinter ':'
```

Zeichen

While	solange
If	wenn
Else	sonst
%	[Modulo-Operator: Rest, der bei ganzzahliger Division entsteht]
==	Gleichheit [mathematische Identität]
=	Zuweisung [keine mathematische Gleichheit]
!=	ungleich
( )	steht für Bedingungen [was in den Klammern steht, soll geprüft werden]
;	schließt Befehl ab
println	sorgt dafür, dass etwas in neuer Zeile ausgegeben wird
print	sorgt dafür, dass etwas innerhalb einer Zeile erscheint
{ }	fasst mehrere Anweisungen zusammen
+	kann arithmetisch gelten, aber auch als string-Verknüpfung

Java → objektorientierte Programmiersprache: Wir können mit Java Objekte erzeugen, an denen „Methoden" dranhängen (andere Programmiersprachen: „Prozedur").

Def. Methode: Ansammlung von Befehlen, denen man einen Namen (z.B. „**main**" = Hauptmethode) gibt, unter dem man dann die Folge der Anweisungen abarbeiten kann.

Damit der Compiler alles versteht, müssen in Java alle **Methoden** in eine **Klasse** gepackt werden:
z.B. public class Collatz

`public`	`static`	`void`	**`main`**	`(String`	`[]`	`argv)`
öffentlich hängt nicht an einem Objekt	statische Methode	liefert keinen Wert zurück	**Methode**, die beginnt	Zeichen	array	

Kommentararten

//	Dies sind die Vorzeichen für einen Kommentar
/*	Das sind Klammerzeichen für einen Kommentar ***/**
//*	Gibt an, was eine Klasse tut /

Ausführung

1. Quelltext im Editor speichern:
   Dateiname muss immer heißen wie die Klasse heißt, und zwar mit Anhang „*.java"!
2. „Javac" aufrufen, Quelldatei „Collatz.java" kompilieren (Bytecode „Collatz.class" wird angelegt)
3. „Java Collatz" ausführen.

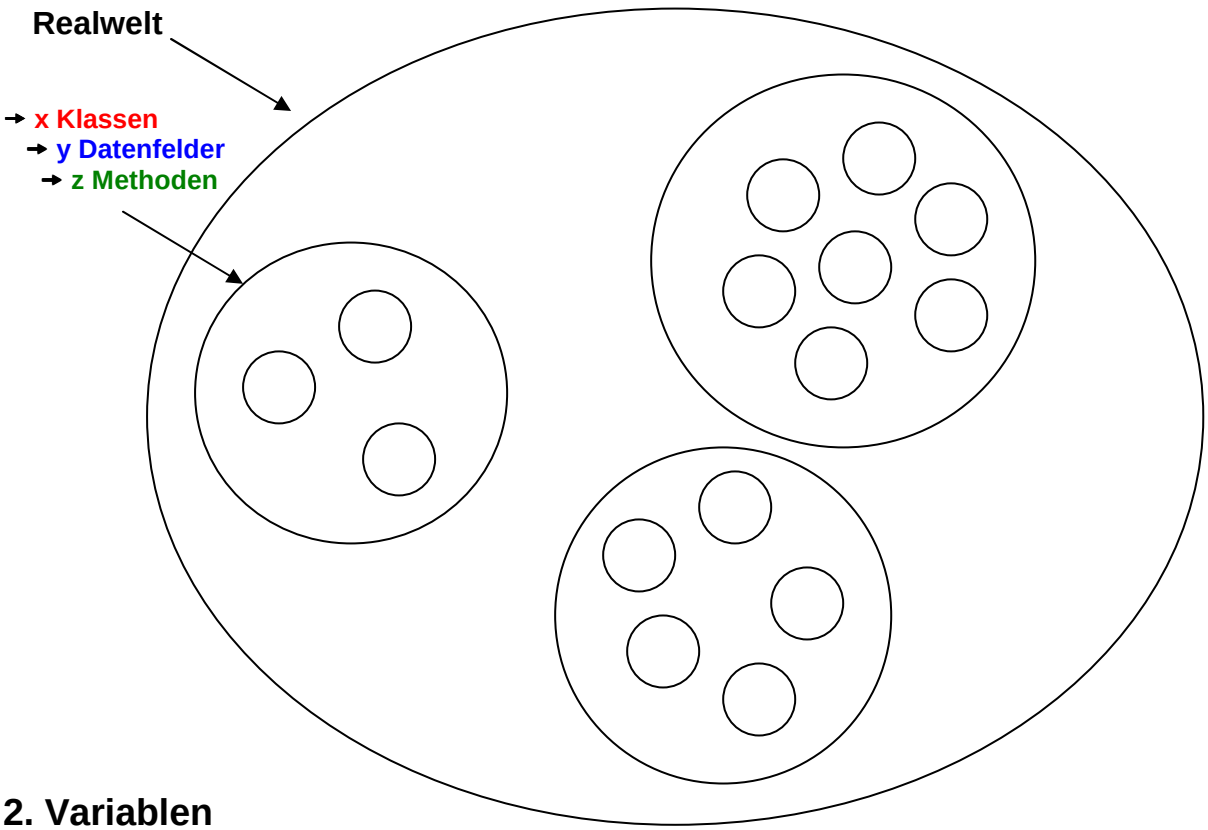
Zusammenfassung

Java ist objektorientiert (s. auch Skript „Java" II: S. 1 ff.), das heißt:
Die **Modellierung der Realwelt** erfolgt durch **Objekte,** zusammengefasst in **Klassen**.
Zu einer Klasse und ihren Objekten gehören **Datenfelder** mit entsprechenden Daten.
Auf ihnen operieren **Methoden,** das sind Anweisungsfolgen, die ihren Zustand zu manipulieren.
Der Collatz-Algorithmus als Java-Programm besteht aus der Definition der Klasse Collatz mit der Methode main.

Nach Übersetzung des Programms in den maschinenunabhängigen Bytecode wird die Methode main gestartet, sobald die sie umschließende Klasse geladen wurde. Der Quelltext besteht aus durch Wortzwischenräume (Leerzeichen, Tabulatoren, Zeilen- und Seitenvorschubzeichen) getrennten Token.

Zur Verbesserung der Lesbarkeit werden Kommentare eingestreut. Der Dokumentationsgenerator ordnet den durch /** . . . */ geklammerten Vorspann der nachfolgenden Klasse zu. Die von den Schlüsselwörtern verschieden gewählten Bezeichner beginnen mit einem Buchstaben, Unterstrich ( ) oder Dollarzeichen ($). Darüber hinaus dürfen im weiteren Verlauf des Bezeichners auch Ziffern verwendet werden.

Zur Vereinfachung der Ein-/Ausgabe verwenden wir die benutzerdefinierte Klasse AlgoTools.IO mit den beiden Methoden readInt und println.



2. Variablen

Datentypen für Variablen (s. S. 9)

z.B.: int x, y, z → drei Variablen anlegen mit Datentyp „integer“

Variablen sind benannte Speicherstellen, deren Inhalte gemäß ihrer vereinbarten Typen interpretiert werden. Java unterstützt folgende “eingebaute” Datentypen (genannt einfache Datentypen):

• boolean	entweder true oder false	8	Bit
• char	16 Bit Unicode	16	Bit
• byte	vorzeichenbehaftete ganze Zahl	8	Bit
• short	vorzeichenbehaftete ganze Zahl	16	Bit
• int	vorzeichenbehaftete ganze Zahl	32	Bit
• long	vorzeichenbehaftete ganze Zahl	64	Bit
• float	Gleitkommazahl	32	Bit
• double	Gleitkommazahl	64	Bit

Zum Begriff „Gleitkommazahlen“: Ein Computer kann nicht problemlos eine unendliche Zahl anzeigen, bzw. berechnen. Das würde ja bedeuten, dass er unendlich lang diese Zahl ausdrucken müsste oder er

unendlich viel Speicher besitzen müsste. Aus diesem Grund beschränkt man sich auf eine bestimmte Anzahl an Nachkommastellen, eben in Abhängigkeit von der Kapazität des jeweiligen Datentyps.

3. Kontrollstrukturen

In strukturierten Anweisungen regeln Kontrollstrukturen den dynamischen Ablauf der Anweisungsfolge eines Programms durch Bedingungen, Verzweigungen und Schleifen. Und zwar unter Verwendung von **Operatoren**, die auf das Ergebnis „wahr“ oder „falsch“ hinauslaufen.

	Operator	**Mathematik**	**Java**
Vergleiche	größer	>	>
	kleiner	<	<
	größer oder gleich	$\geq$	>=
	kleiner oder gleich	$\leq$	<=
	gleich	=	==
	ungleich	$\neq$	!=
Verknüpfungen (Boole)	Und	$\wedge$	&, &&
	Oder	$\vee$	|, ||
	Nicht	$\neg$	!

Tabelle mit Wahrheitswerten, verknüpft durch logische Operatoren

→ Boole’sche O.: Verknüpfung von zwei Wahrheitswerten (a, b) zu einem dritten
→ Kombinationen von true (1), false (0) zu 1 oder 0

Wenn bei einem 1. Boole’schen Wahrheitswert a = 0 (falsch) herausgekommen ist und bei einem 2. Boole’schen Wahrheitswert b = 0 (falsch), dann folgt für die logische Verknüpfung beider mittels && entweder 1 oder 0

a	**b**	**a && b**	**a || b**	**a ∧ b**	**a ! b**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

[(a:2+3=6)→**0** && (b:4+5=10) →**0**)] → 0 // [a∧b(0) && a∧b(0)] → 0
[(a:2+3=6)→**0** && (b:4+5=9) →**1**)] → 0
[(a:2+3=5)→**1** && (b:4+5=10) →**0**)] → 0
[(a:2+3=5)→**1** && (b:4+5=9) →**1**)] → 1

Boole'scher Ausdruck mit Modulo (s.o.)
(mod gibt den Rest aus der Division zweier ganzer Zahlen an, z.B. 7 mod 2 = 1)
Ermittlung gerader Zahlen in Java-Syntax: → wenn kein Rest (R = 0) bleibt, dann ist x gerade
{
IF x % 2 == 0 // if ((x mod 2) == 0), dann ist x gerade
…} // Falls x gerade, dann …
Wenn es wahr ist, dass x % 2 == 0, dann folgt für den Ausdruck der Wahrheitswert 1 (→ „true")

Bedingte Anweisung (Beispiel mit Boole'schen Operatoren)

```
/*************************** Bedingung.java ************************************/

public class Bedingung
{
public static void main (String [] argv)
{
int x = -5, m;                  /* definiere zwei Integer-Variablen, davon x
                                   als -5 initialisiert */
if (x < 0) x = -x;              /* prüft eingetipptes x auf Vorzeichen:
                                   wenn x = 0 oder positiv → keine Veränderung
                                   wenn x = negativ, dann multipliziere x mit -1 */
m = IO.readInt("Bitte Monatszahl: ");                    // Eingabeaufforderung
  if ((3 <= m) && (m <= 5)) IO.println("Frühling");      // fünf Tests!
  else if ((6 <= m) && (m <= 8)) IO.println("Sommer");
  else if ((9 <= m) && (m <= 11)) IO.println("Herbst");
  else if (m==12 || m==1 || m==2) IO.println("Winter");
  else IO.println("unbekannte Jahreszeit");
}
}
```

Fallunterscheidung 1 (switch/case-Anweisung)

```
/*************************** Fall.java ************************************/

import AlgoTools.IO;
public class Fall
{
public static void main (String [] argv)
{
int zahl = 42;                          // initialisiert mit „42"
int monat = 11;                         // vorgegebene Monatszahl „11"
switch (zahl % 10)                      // berechnet den ganzzahligen Rest → Div:10
{
case 0: IO.println("null "); break;     // ‚break' unterbricht Durchlauf aller Fälle
case 1: IO.println("eins "); break;
case 2: IO.println("zwei "); break;
case 3: IO.println("drei "); break;
case 4: IO.println("vier "); break;
case 5: IO.println("fuenf "); break;
case 6: IO.println("sechs "); break;
case 7: IO.println("sieben"); break;
case 8: IO.println("acht "); break;
case 9: IO.println("neun "); break;
}
switch(monat) {                         // verzweige in Abhängigkeit vom Monat
case 3: case 4: case 5: IO.println("Frühling"); break;
case 6: case 7: case 8: IO.println("Sommer "); break;
case 9: case 10: case 11: IO.println("Herbst "); break;
case 12: case 1: case 2: IO.println("Winter "); break;
default: IO.println("unbekannte Jahreszeit");
}
}
}
```

Fallunterscheidung 2 (switch/case-Anweisung)

```
/************************** Fall2.java ************************************/

import AlgoTools.IO;
public class Fall2
{
public static void main (String [] argv)
{
int m = IO.readInt("Bitte einen Monatsnamen  „);
IO.print("Es handelt sich um  „);                    // „print“ für direkte Anzeige
Switch m                                             // „Println“ für neue Zeile
(                                                    // ‚switch“ für Verzweigung
case 3:case 4: case 5: IO.println("Frühling"); break;
case 6:case 7: case 8: IO.println("Sommer"); break;
case 9:case 10: case 11: IO.println("Herbst"); break;
case 12:case 1: case 2: IO.println("Winter"); break;
default: IO.println("unbekannte Jahreszeit");       // Ausnahme
}
}
}
```

Schleife 1 (while-Schleife, do-while-Schleife, break, continue, for-Schleife)

```
/************************** Schleife.java ***************************************/

import AlgoTools.IO;
*/
public class Schleife
{
public static void main (String [] argv)
{
int i, x=10, y=2, summe;                             // 4 Integer-Variablen

while (x > 0) {                                      // solange x groesser als 0
x--;                                                 // erniedrige x um eins
y = y + 2;                                           // erhöhe y um zwei
}
do {
x++;                                                 // erhöhe x um eins
y += 2;                                              // erhöhe y um 2
}
while (x < 10);                                      // solange x kleiner als 10
while (true) {                                       // auf immer und ewig
x /= 2;                                              // teile x durch 2
if (x == 1) break;                                   // falls x=1 verlasse Schleife
if (x % 2 == 0) continue;                            // falls x gerade starte Schleife
x = 3*x + 1;                                         // verdreifache x und erhoehe um 1
}
IO.println("Bitte Zahlen eingeben. 0 als Abbruch");
summe = 0;                                           // initialisiere summe
x = IO.readInt();                                    // lies x ein
while (x != 0) {                                     // solange x ungleich 0 ist
summe += x;                                          // erhöhe summe
x = IO.readInt();                                    // lies x ein
}
IO.println("Die Summe lautet " + summe);
do {
x=IO.readInt("Bitte 1<= Zahl <=12");                 // lies x ein
} while (( x < 1) || (x > 12));                      // solange x unzulaessig
for (i=1; i<=10; i++) IO.println(i*i,6);             // drucke 10 Quadratzahlen
}
}
```

Schleife 2 (Berechnung der Fakultät mit for-, while- und do-while-Schleifen)

```
/************************** Fakultaet.java ************************************/

import AlgoTools.IO;
public class Fakultaet                          //** n! := 1 fuer n=0; * 1*2*3* ... *n sonst
{
public static void main (String [] argv)
{
int i, n, fakultaet;                            // 3 Integer-Variablen
n = IO.readInt("Bitte Zahl: ");                 // fordere Zahl an
fakultaet = 1;                                  // berechne n! mit for-Schleife
for (i = 1; i <= n; i++)
fakultaet = fakultaet * i;
IO.println(n + " ! = " + fakultaet);
fakultaet = 1;                                  // berechne n! mit while-Schleife
i = 1;
while (i <= n) {
fakultaet = fakultaet * i;
i++; }
IO.println(n + " ! = " + fakultaet);
fakultaet = 1;                                  // berechne n! mit do-while-Schleife
i = 1;
do {
fakultaet = fakultaet * i;
i++; }
while (i <= n);
IO.println(n + " ! = " + fakultaet);
}
}
```

Schleife 3 (Berechnung des GGT while-Schleifen)

```
/************************** GGT.java ******************************************/

import AlgoTools.IO;                            // Berechnung des GGT von x und y
public class GGT
{
public static void main (String [] argv)
{
int teiler, a, b, x, y, z;                      // 6 Integer-Variablen
IO.println("Bitte zwei Zahlen: ");
a=x=IO.readInt(); b=y=IO.readInt();             // lies 2 Zahlen ein
teiler = x;                                     // beginne mit einem Teiler

while ((x % teiler != 0) ||                     // solange x nicht aufgeht
(y % teiler != 0))                              // oder y nicht aufgeht
teiler--;                                       // probiere Nächstkleineren
IO.println("GGT = " + teiler);

while (a != b)                                  // solange a ungleich b
if (a > b) a = a - b;                           // subtrahiere die kleinere
else b = b - a;                                 // Zahl von der größeren
IO.println("GGT = " + a);

while (y != 0)                                  // solange y ungleich 0
{
z = x % y;                                      // ersetze x durch y
x = y;                                          // und y durch x modulo y
y = z;
}
IO.println("GGT = " + x);
}
}
```

4. Datentypen (s.S. 3)

Mit dem Deklarieren weist man einer Variable einen Datentyp zu, legt aber auch fest:
→ Operationen → z.B. Addition, Multiplikation usw.
→ Wertebereich → z.B. ganze Zahlen bei Integer
→ Konstantenbezeichner → Wertzuweisung, z.B. **x** = 42; **x** = -42; **x** = 0x2A [→ 42 als Hex-Wert]
→ **Kodierung** → Kodierung durch Bit-Sequenzen im Binärsystem

Datentyp z.B.	Länge	Wertebereich: → ganze Zahlen darstellbar aufgrund von 8, 16, 32, 64 Bit
byte	8 Bit	-128 …… 127
short	16 Bit	-32768 …….32767
int	32 Bit	-2147483648 ……2147483647
long	64 Bit	-9223372036854775808…...9223372036854775807

„digits" (d_i): dn_{-1} dn_{-2} ….. d_2, d_1, d_0 (→ Sequenz von Nullen und Einsen)
Zeichen: ….. 1 1 1
Wert: ….. 4 2 1 (→ gewichtet nach Position der jeweiligen 2er-Potenz)

a) Duale Kodierung natürlicher Zahlen (positiv, neutral):

Binärwert → Dezimalwert (x = z.B. $1101_{[2]} = 13_{[10]}$) x = $\sum_{i=0}^{n-1} d_i \cdot 2^i$

1. 13/2 = [Rest1] → ungerade: **1**
2. 12/2 = 6 → gerade: **0**
3. 6/2 = 3 → ungerade: **1**
4. 3/2 = 1 → ungerade: **1**

Summierung der gewichteten 2er-Potenzen,
(hier: 1101)
jedes d_i wird mit 2^i multipliziert:
$d_3 \times 2^3$ | $d_2 \times 2^2$ | $d_1 \times 2^1$ | $d_0 \times 2^0$
8 4 0 1 = $13_{[10]}$

Dezimalwert → Dualwert (Algorithmus)

```
while (x != 0)                  // Eingabe: z.B. 13
{
if (x%2 == 0) IO.print('0');
else IO.print('1');
x = x/2;    }                   // Resultat: 1101
```

b) Duale Kodierung ganzer Zahlen [positiv, neutral, negativ (→ 2er-Komplement)]
Achtung! → vorgegebene Zahl der Bits: 4 Stellen, wobei die 1. Stelle (0 oder 1) für das Vorzeichen reserviert ist!

d_3 d_2 d_1 d_0	Dezimalwert
0 1 1 1	7
0 1 1 0	6
0 1 0 1	5
0 1 0 0	4
0 0 1 1	3
0 0 1 0	2
0 0 0 1	1
0 0 0 0	0
1 1 1 1	-1
1 1 1 0	-2
1 1 0 1	-3
1 1 0 0	-4
1 0 1 1	-5
1 0 1 0	-6
1 0 0 1	-7
1 0 0 0	-8

Umfang ganzer Zahlen bei z.B. 8 Stellen (1. Bit regelt Vorzeichen!):

1000 0000…..0111 1111
-128…….+127

Ermittlung der Kodierung einer negativen Zahl

Problem: Kodiere -x
Lösung: 1. Kodiere x
2. Invertiere x
3. Addiere dual 1

Beispiel:

gegeben negative Zahl −x **-4**

Problem: finde d_i zu x = 4 **0100**
Lösung: invertiere Bits **1011**
addiere dual 1 **+1**
1100

2er-Komplement: Addition/Subtraktion → 3 + (-5) = 3 - 5

```
  0011   3
+ 1011  -5
= 1110  -2

  0010   2
+ 1011  -5
  1101  -3
```

Achtung, Überlauf!

```
  0110  6
+ 0111  7
  1101 -3   falsch!
```

Trick zur Überprüfung:
1. Verdoppele das Vorzeichen-Bit, aber nur für die Dauer der Rechnung
(*Zahl* ist im Speicher mit 4 Bits, zusätzliches 5. *Bit* nur im Rechenwerk)
2. Verknüpfe
3. Falls Vorzeichen-Bits identisch ➜ Ergebnis korrekt, sonst: nicht korrekt

```
  00111  7          00011   3
+ 00001  1        + 11011  -5
  01000  8          11110  -2
```

Ergebnis undefiniert Ergebnis wahr

c) Kodierung von Gleitkommazahlen (*float*, *double*)

Gleitkommazahlen sind der übliche Weg, um Kommazahlen, also reelle Zahlen, für den Computer darzustellen.

Rationelle Zahlen wie z.B. 3,5 sind unproblematisch. Schwieriger wird es bei Kommazahlen, die kein Ende, also unendlich viele Nachkommstellen haben, wie z.B. bei 10 geteilt durch 3 = 3,33333…

Weil ein Computer keine unendliche Zahl anzeigen bzw. berechnen kann, beschränkt man sich auf eine bestimmte Anzahl an Nachkommastellen, z.B. beim Geldwert auf 3,33 Euro.

Für den Computer werden Gleitkommazahlen durch **Vorzeichen**, **Mantisse** und **Exponent** beschrieben und erreichen damit deutlich größere Absolutwerte als Integerzahlen:

Für eine gegebene Zahl **x** werden Mantisse **m** und Exponent **e** gesucht mit der Eigenschaft

$$\mathbf{x = m \cdot 2^e}$$

Der Wert der Gleitkommazahl ergibt sich also aus dem Produkt von Mantisse und der Zweierpotenz des Exponenten. Da es für ein gegebenes x mehrere zueinander passende Mantissen und Exponenten gibt, wählt man die normalisierte Form mit einer Mantisse:

z.B. 12 $= 6 \times 2^1$
$= 3 \times 2^2$
$= 1{,}5 \times 2^3$
$= 0{,}75 \times 2^4$

Mantisse soll „normalisiert“ sein: $1 \leq m < 2$

Da eine normalisierte Mantisse immer mit einer 1 vor dem Komma beginnt, reicht es aus, ihren Nachkommawert als reduzierte Mantisse $f = m - 1.0$ abzuspeichern.

Bei einer **Kodierung** für 32 Bits
mit dem Datentyp **float** sind vorgesehen für

das	**Vorzeichen:**	1 Bit
den	**Exponenten:**	8 Bits
die	**Mantisse:**	23 Bits

Java verwendet zur Kodierung von Vorzeichen, Mantisse und Exponent den IEEE-Standard 754-1985.

d) Kodierung logischer Werte (*boolean*)

Der Typ boolean dient zum Speichern der logischen Werte bzw. Operationen im Sinne von wahr und falsch. Die einzigen Konstantenbezeichner lauten *true* (1) und *false* (0).

e) Kodierung von Zeichen (*char*)

Der Datentyp *Character* beschreibt in der EDV Datenelemente des Typs „Zeichen“, d.h. alle Zeichen im 16 Bit breiten Unicode-Zeichensatz. Jedes Zeichen wird kodiert durch eine 2 Byte breite Zahl, z.B. der Buchstabe A hat die Nummer 65 (dezimal) = 00000000 01000001 (binär). Die niederwertigen 7 Bits stimmen mit dem ASCII-Zeichensatz (American Standard Code for Information Interchange) überein.

5. Arrays

Mehrere Variablen desselben Typs können zu einem Feld („Array“) zusammengefasst werden:

Statt Aufzählung der Variablen z.B. **m1** für Januar, **m2** für Februar, **m3** für März usw.
→ Zusammenfassung der 12 Variablen zu einem Namen (Index)

Zusammenfassen lassen sich in diesem Sinn:

- Ziffern
- Daten
- Zeichen
- Wahrheitswerten
- Indizes
- Zustände