Java: Grundlagen der Sprache: Lexikalische Elemente

Lexik und Lexeme

Im Lexikon findet man:

- Lexem (griechisch) das, Sprachwissenschaft: kleinste semantische Einheit, Träger der lexikalischen Bedeutung; das Lexem tritt als Einzelwort (z. B. Wald), als Teil eines Wortes (z. B. wald- in waldig) und als Wortverbindung auf (z. B. Waldbrand).

- Lexik die, der Wortschatz einer Sprache.

  Die Lexik einer Programmiersprache bestimmt die textuellen Grundbausteine der Programme. Solche Bausteine sind etwa Schlüsselwörter und Bezeichner. Sie werden z. B. durch Aufzählung oder reguläre Ausdrücke angegeben. Lexeme einer Programmiersprache können aus mehr als einem Zeichen bestehen.

Eingabezeichen

- Ein Java-Programm besteht aus einer Folge von Unicode-Zeichen. Im Unicode sollen alle weltweit verwendeten Schriftzeichen und Symbole erfasst werden.

- Der Coderaum von Unicode ermöglichte ursprünglich 65.536 Zeichen (Code points). Bald aber stellte sich dieses als unzureichend heraus. Der Coderaum wurde auf 1.114.112 Zeichen erweitert. Bestimmte Zeichen stehen gemäß dem Standard nicht für die Codierung von Zeichen zur Verfügung. Der Coderaum ermöglicht 1.111.998 Codepunkte. Er wird noch lange nicht vollständig ausgenutzt.

- Der Unicode weist jedem Zeichen eine Zahl zu. Diese werden in der Form U+*yyyy* geschrieben, wobei *yyyy* der so genannte Kode-Wert ist. Der Kode-Wert ist eine hexadezimale Zahl. Beispielsweise ist U+0041 der Kode-Wert von *A*. Die hexadezimale Zahl 41 besitzt im Dezimalsystem den Wert $4 \cdot 16^1 + 1 \cdot 16^0 = 65$.

Eingabezeichen

- Ein Unicode-Transformationsformat (UTF) bildet Unicode-Zeichen auf Folgen von Bytes ab.
- Beispiele von UTF-8 sehen wir uns in der Übung an.
- Neben `String`- und `char`-Typen werden auch Literale und Bezeichner durch Unicode-Zeichen realisiert. Folgerung: Variablen- und Klassennamen können mit nationalen Zeichen und anderen Symbolen versehen werden.

Kommentare

Java kennt drei Arten von Kommentaren:

- Einzeilige Kommentare beginnen mit `//` und enden mit der aktuellen Zeile.
- Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`.
- Dokumentationskommentare beginnen mit `/**` und enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken. Dokumentationskommentare stehen vor dem Element, das sie beschreiben. Die Dokumentation wird mit dem Programm `javadoc` erzeugt. Sie besteht aus Html-Dateien.

Bezeichner (identifier, name)

- Bezeichner spezifizieren die Namen von Klassen, Methoden, Variablen und Paketen. Ein Bezeichner kann beliebig lang sein. Alle Stellen sind signifikant.

- Bezeichner beginnen mit einem Buchstaben, Dollar-Zeichen oder Unterstrich. Dann dürfen weitere Buchstaben, Ziffern, Dollarzeichen oder Unterstriche folgen. Bezeichner dürfen weder Schlüsselwörter noch `true`, `false` oder `null` sein.

- Klassennamen sollten mit einem Großbuchstaben beginnen. Methodennamen fangen mit einem Kleinbuchstaben an. Haben sie mehrere Komponenten, ist die erste oft ein Verb. Weitere Komponenten beginnen dann mit einem Großbuchstaben: `hasMoreElements`. Für Variablennamen gelten die Konventionen wie für Methoden. Paketnamen bestehen ausschließlich aus Kleinbuchstaben.

Literale

Literale bezeichnen konstante Werte. Beispiele sind

- ganze Zahlen (23, -234),
- Gleitkommazahlen (3.14),
- Wahrheitswerte (`true`, `false`),
- einzelne Zeichen in Hochkommata (`'a'`),
- Zeichenketten in Anführungszeichen (`"Dies ist eine Zeichenkette"`) und
- das Null-Literal `null`.

Schlüsselwörter und weitere Zeichen

- Gewisse Wörter dürfen in Java nicht als Bezeichner verwendet werden. Diese Wörter heißen Schlüsselwörter oder reservierte Wörter. Beispiele hierfür sind `boolean`, `if`, `else` oder `while`.

- Trennzeichen:
  Leerzeichen, Zeilenendezeichen (`ENTER`-Taste), Tabulatorzeichen (`TAB`-Taste)

- Operatoren: `+`, `-`, `...`

- Interpunktionszeichen: `, . ; ( ) { } [ ]`

Java: Grundlagen der Sprache: Datentypen und Variable

Struktur eines Java-Programms

- Ein Java-Programm besteht aus einer oder mehreren Klassen. Klassen setzen das objektorientierte Paradigma von Algorithmen um (s. Einführung).

- Ausgeführt wird die `main`-Methode einer Klasse. Die `main`-Methode ist nicht an ein Objekt gebunden (`static`):

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

- Eine weitere Möglichkeit sind Applets („little applications“). Sie werden wir später behandeln.

Java-Datentypen

- **Primitive Datentypen:**
  - boolean
  - char
  - *Numerische Typen:*
    - Ganzzahlige Typen: byte, short, int, long
    - Gleitkomma-Typen: float, double

- **Referenztypen** (vor- und selbstdefinierte Typen), Beispiele:
  - Array (Feld)
  - String (Zeichenkette)
  - Wrapper (primitive Datentypen als Referenztypen)
  - Enumeration (Aufzählung)

Datentypen

Jeder Datentyp setzt sich aus den folgenden Bestandteilen zusammen:

1. Die Menge der Werte, d. h. der Wertebereich.

2. Die Darstellungen für die Werte.
   Man unterscheidet zwischen der Darstellung im Programmtext (Literale) und der im Speicher.

3. Die Methoden, die für diesen Typ definiert sind.

Primitive Datentypen

Typ	Länge	Wertebereich	Standardwert
boolean	1	true, false	false
char	2	Unicode-Zeichen	\u0000
byte	1	$-2^7 \ldots 2^7-1$	0
short	2	$-2^{15} \ldots 2^{15}-1$	0
int	4	$-2^{31} \ldots 2^{31}-1$	0
long	8	$-2^{63} \ldots 2^{63}-1$	0
float	4	$\pm 3.4 \cdot 10^{38}$	0.0
double	8	$\pm 1.8 \cdot 10^{308}$	0.0

Der Datentyp `boolean`

- Literale des Datentyps `boolean` sind `true` und `false`.
- Weitere Literale existieren nicht.

Der Datentyp `char`

- `char`-Literale werden in einfachen Hochkommata geschrieben. Daneben gibt es `String`-Literale, die in doppelten Hochkommata stehen.

- Besondere Zeichen:

`\b`	Backspace	`\t`	Tabulator
`\n`	Newline	`\f`	Seitenumbruch (Formfeed)
`\"`	Doppeltes Anführungszeichen	`\‘`	Einfaches Anführungszeichen
`\\`	Backslash		

- Außerdem können Unicode-Zeichen in der Form `\uxxxx` angegeben werden, wobei `xxxx` eine Folge von bis zu 4 hexadezimalen Ziffern ist. So steht `\u0020` beispielsweise für das Leerzeichen (vgl. ASCII-Zeichensatz). Entsprechend können Unicode-Zeichen in der Form `\nnn` für eine Oktalzahl `nnn` angegeben werden.

Ganzzahlige Typen

- Ganzzahlige Literale können in Dezimal-, Oktal-, Binär- oder Hexadezimalform geschrieben werden. Ein oktaler Wert beginnt mit dem Präfix `0`, ein binärer Wert mit `0b`, ein hexadezimaler mit `0x`.

- Dezimale Literale dürfen nur aus den Ziffern `0` bis `9`, oktale aus den Ziffern `0` bis `7`, binäre aus den Ziffern `0` und `1` und hexadezimale aus den Ziffern `0` bis `9` sowie den Buchstaben `A` bis `F` bzw. `a` bis `f` bestehen.

- Positive Zahlen können wahlweise auch mit einem + beginnen. Durch Voranstellen des Zeichens – werden negative Zahlen dargestellt.

- Ganzzahlige Literale sind grundsätzlich vom Typ `int`, sofern nicht ein `L` bzw. `l` hinten angehängt wird. In diesem Fall sind sie vom Typ `long`.

Ganzzahlige Literale

Beispiel:

$$43 = (101011)_2 = (53)_8 = (2B)_{16}$$

Darstellung in Java:

```
int dezimal =       43;
int binaer  = 0b101011;
int oktal   =      053;
int hexadez =     0x2B;
```

Underscore in Literalen

Lange Ziffernfolgen sind manchmal schwierig zu lesen:

```
1234567891011213
```

Leichter ist es, wenn im Literal ein Trennzeichen enthalten ist:

```
12_345_678_910_111_213
```

12 Billiarden 345 Billionen 678 Milliarden 910 Millionen 111 Tausend 213

In Java dürfen Unterstriche als Trennzeichen innerhalb von Ziffernfolgen verwendet werden, ohne den Wert des Literals zu ändern:

```
long i = 12_345_678_910_111_213
```

Einschränkung: Der Unterstrich darf nicht überall stehen, zum Beispiel nicht am Anfang oder Ende eines Literals, nach `0x` oder nach einem Dezimalpunkt:

```
_12345    12345_    0x_053    3._141592.
```

Gleitkomma-Typen

- Gleitkomma-Literale werden immer in Dezimalnotation aufgeschrieben. Sie bestehen aus einem Vorkommateil, einem Dezimalpunkt, einem Nachkommateil, einem Exponenten und einem Suffix.

- Es muss mindestens der Dezimalpunkt, der Exponent oder der Suffix vorhanden sein. Vorkomma- oder Nachkommateil dürfen fehlen, aber nicht beide. Der Exponent, der durch `E` oder `e` eingeleitet wird, darf fehlen. Vorkommateil und Exponent können wahlweise durch + oder − eingeleitet werden.

- `F` oder `f` als Suffix zeigt an, dass es sich um einen `Float`-Wert handelt. Entsprechend steht `D` bzw. `d` für einen `Double`-Wert. Ist kein Suffix vorhanden, ist der Wert vom Typ `Double`.

- Beispiele: 3.14 2f 3e3 .5f 6.

Gleitkomma-Typen

Konstante	verfügbar für	Bedeutung
`MAX_VALUE`	float, double	größter positiver Wert
`MIN_VALUE`	float, double	kleinster positiver Wert
`NaN`	float, double	Not-a-Number
`NEGATIVE_INFINITY`	float, double	negativ unendlich
`POSITIVE_INFINITY`	float, double	positiv unendlich

- `NaN` tritt z. B. als Ergebnis einer Division durch 0 auf.

- `NEGATIVE_INFINITY` ist eine Zahl, die kleiner als jede andere Zahl ist. Entsprechend ist `POSITIVE_INFINITY` größer als alle anderen Zahlen.

Variable

Variable dienen dazu, Daten im Hauptspeicher abzulegen, ggf. zu verändern und wieder zu lesen. In Java gibt es drei Variablentypen:

- Instanzvariable (Attribute) werden im Rahmen einer Klasse definiert und zusammen mit einem Objekt angelegt.
- Klassenvariable werden ebenfalls innerhalb einer Klasse definiert, existieren aber – unabhängig von Objekten – für jede Klasse genau einmal. Klassenvariable werden mit dem Modifikator `static` definiert.
- Lokale Variable werden innerhalb eines Blocks oder einer Methode verwendet und sind nur darin sichtbar.

Variable

- Variable gehören immer zu einem bestimmten Datentyp.
- Die Deklaration erfolgt durch

  ```
  Typname Variablenname
  ```

  Variablendeklarationen dürfen an beliebiger Stelle erfolgen.
- Variablen können bei der Deklaration initialisiert werden.
- Der Wert einer Variablen kann durch eine Zuweisung oder durch einen Inkrement- bzw. Dekrement-Operator verändert werden.

Variable

```
class Variable {
   public static void main(String[] args) {
     int a;                    // Deklaration von a
     a = 1;                    // Zuweisung
     char b = 'x';             // Deklaration mit Initialisierung
     System.out.println(a);
     double c = 3.1415;
     System.out.println(b);
     System.out.println(c);
     boolean d = false;
     System.out.println(d);
   }
}
```

Variable

```
class VariablenTypen {
   int a = 1;                                    // Instanzvariable
   static int b = 2;                             // Klassenvariable
   public static void main(String[] args) {
      int c = 3;                                 // Lokale Variable
      int a = 4;                                 // Lokale Variable
      VariablenTypen x = new VariablenTypen();
      System.out.println(x.a);
      System.out.println(b);
      System.out.println(c);
      System.out.println(a);
   }
}
```

Lebensdauer und Sichtbarkeit

- Instanzvariable: Instanzvariable werden zum Zeitpunkt des Erzeugens einer neuen Instanz einer Klasse angelegt. Sie sind innerhalb der ganzen Klasse sichtbar, sofern sie nicht von gleichnamigen lokalen Variablen überdeckt werden. Jedes Objekt kann mit `this` auf alle seine Attribute zugreifen. Instanzvariable leben solange das zugehörige Objekt lebt.

- Klassenvariable: Die Regeln für die Sichtbarkeit von Klassenvariablen entsprechen denen von Instanzvariablen. Sie leben während des gesamten Programms.

- Lokale Variable: Lokale Variable sind von der Stelle der Deklaration bis zum Ende der Methode sichtbar. Die Lebensdauer endet mit dem Ende des Methodenaufrufs. Lokale Variable dürfen nicht durch weiter innen liegende andere lokale Variable überdeckt werden. Dies gilt nicht für Instanzvariable und Klassenvariable.

Arrays

- Ein Array ist eine Reihung von Elementen eines festen Grundtyps.
- Die Größe eines Arrays kann zur Übersetzungs- oder Laufzeit festgelegt, danach aber nicht mehr geändert werden. Arrays sind semidynamisch.
- Arrays sind Objekte. Sie besitzen daher Instanzvariable und Methoden.

Arrays

Arrays werden in zwei Schritten erzeugt und einer Array-Variablen zugewiesen:

1. Deklaration einer Array-Variablen:

```
int[] a;
boolean[] b;
```

2. Erzeugen eines Arrays und Zuweisen an die Array-Variable:

```
a = new int[5];
b = new boolean[3];
```

Arrays

- Deklaration und Zuweisung können auch zusammengefasst werden:

```
int[] a = new int[5];
boolean[] b = new boolean[3];
```

- Es ist möglich, Arrays literal zu initialisieren:

```
int[] a = {1,2,3,4,5};
```

oder

```
int[] a = new int[] {1,2,3,4,5};
```

Arrays

```
class Array {
   public static void main(String[] args) {
      int[] a;
      a = new int[10];
      for (int i = 0; i < a.length; i = i+1) {
         a[i] = i*i*i;
      }
      for (int i = 0; i < a.length; i = i+1) {
         System.out.println(""+ i + "^3 = " + a[i]);
      }
   }
}
```

Arrays

```
0^3 = 0
1^3 = 1
2^3 = 8
3^3 = 27
4^3 = 64
5^3 = 125
6^3 = 216
7^3 = 343
8^3 = 512
9^3 = 729
```

Achtung: Der Operator ^ besitzt in Java eine andere Bedeutung, s. unten.

Arrays

- Auf das `i`-te Element eines Arrays `a` wird durch `a[i]` zugegriffen.
- Das durch

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

  deklarierte Feld besitzt n Elemente: a[0], ..., a[n-1].
- Jedes Array besitzt die Instanzvariable `length`.
- Ein Array-Index ist stets vom Typ `int`.
- Indexausdrücke werden während der Laufzeit auf Einhaltung der Grenzen überprüft.

Arrays

```
public class ArrayMehrDim {
   public static void main(String[] args) {
      int[][] a = new int[2][3];
      a[0][0] = 1;
      a[0][1] = 2;
      a[0][2] = 3;
      a[1][0] = 4;
      a[1][1] = 5;
      a[1][2] = 6;
      System.out.println(""+a[0][0]+a[0][1]+a[0][2]);
      System.out.println(""+a[1][0]+a[1][1]+a[1][2]);
   }
}
```

Arrays

- Mehrdimensionale Arrays werden als Arrays von Arrays betrachtet und als solche behandelt.

- Es ist möglich, nicht-rechteckige Arrays zu erzeugen:

```
int[][] a = { {0},
              {1,2},
              {3,4,5},
              {6,7,8,9}
            };
```

  Das Feld `a` besitzt die 4 Elemente `a[0]`, ..., `a[3]`. Jedes dieser Elemente ist wiederum ein Feld, auf dessen Elemente durch `a[i][j]` zugegriffen werden kann. Die Länge von `a[i]` kann durch `a[i].length` ermittelt werden.

Strings

- In Java werden Zeichenfolgen durch die Klasse `String` repräsentiert. Ein string, d. h. ein Objekt der Klasse `String`, kann als Reihung von Elementen des Typs `char` gesehen werden.

- String-Literale sind Zeichenfolgen, die in " eingeschlossen werden: `"abc"`

- Die Deklarationen

  ```
  String str = "abc";
  ```

  und

  ```
  char[] data = {'a', 'b', 'c'};
  String str = new String(data);
  ```

  sind äquivalent. Feldelemente können geändert werden, Strings nicht.

Strings

- Strings sind konstant. Nach Erzeugung kann ein String nicht mehr verändert werden.

- Die Klasse `String` definiert eine Vielzahl von Methoden zur Manipulation und zur Bestimmung der Eigenschaften von Zeichenfolgen.

- Die wichtigste Operation ist die String-Konkatenation. Sie wird durch den Operator + realisiert.

Referenztypen

- Referenztypen bilden neben den primitiven Datentypen die zweite wichtige Klasse von Datentypen.
- Zu den Referenztypen gehören Objekte, Arrays und Strings. Die vordefinierte Konstante `null` bezeichnet die leere Referenz.
- Für Arrays und Strings existieren – wie gesehen – Literale, für andere Objekte nicht.
- Auf Aufzählungstypen und Wrapper-Typen gehen wir später ein.

Referenztypen

- Werte der Referenztypen werden indirekt gespeichert. Der Compiler greift auf sie über eine Referenz zu.

- Die Zuweisung einer Referenz kopiert lediglich den Verweis auf das Objekt und nicht das Objekt selbst. Soll das Objekt kopiert werden, muss die Methode `clone` verwendet werden.

- Der Gleichheitstest == zweier Referenzen testet, ob die Verweise gleich sind. Soll nur auf inhaltliche Gleichheit getestet werden, ist die Methode `equals` zu benutzen bzw. zu implementieren.

Referenztypen

- Objekte können mithilfe des `new`-Operators explizit erzeugt werden:

```
Vector a = new Vector();
```

- Ein im Hintergrund arbeitender Prozess, der Garbage Collector, sucht periodisch Speicher, der durch Objekte belegt wird, die nicht mehr erreichbar sind. Dieser Speicher wird freigegeben.

Test auf Gleichheit

```
public class Gleichheit {
  public static void main(String[] args) {
    String a = new String("hallo");
    String b = new String("hallo");
    System.out.println(a == b);          // gibt false aus
    System.out.println(a.equals(b));     // gibt true aus
    String c = "hallo";
    String d = "hallo";
    System.out.println(c == d);          // gibt true aus
  }
}
```

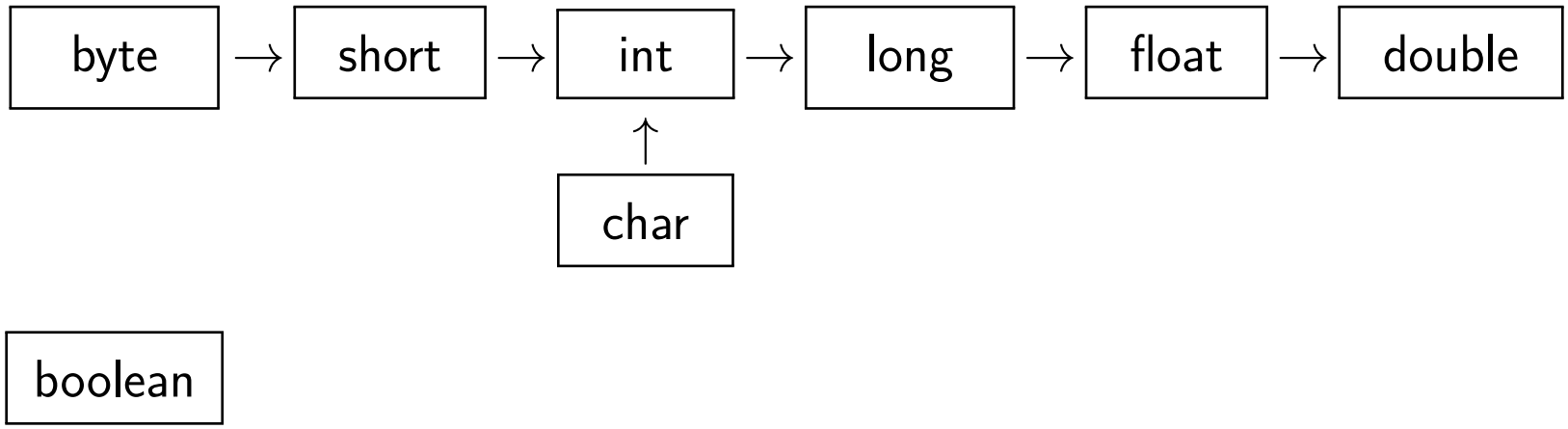
Typkonvertierungen

- Unter einer Typkonvertierung versteht man die Umwandlung eines Wertes eines Typs in einen Wert eines anderen Typs. Beispielsweise kann der int-Wert 5 in den float-Wert 5.0 konvertiert werden.

- Typkonvertierungen werden implizit vom Compiler oder explizit vom Programmierer mithilfe des `cast`-Operators vorgenommen.

- Man unterscheidet zwischen erweiternden und einschränkenden Typkonvertierungen und diese wiederum für primitive Datentypen und für Referenztypen.

Konvertierungen für primitive Datentypen



Die Pfeilrichtung definiert erweiternde Konvertierungen.

Konvertierungen für primitive Datentypen

In den folgenden Fällen nimmt der Compiler implizit erweiternde Konvertierungen vor:

- Bei einer Zuweisung, wenn der Typ des Ausdrucks und der Typ der Variablen nicht identisch sind.

  ```
  double d = 5;
  ```

- Bei der Auswertung eines Ausdrucks, wenn die Typen der Operanden „nicht passen“.

  ```
  ... (5 + 4.0) ...;
  ```

Konvertierungen für primitive Datentypen

- Beim Aufruf einer Methode, falls die Typen der aktuellen Parameter nicht mit denen der formalen übereinstimmen.

```
static double kehrwert(double x) {
  return 1/x ;
}

... kehrwert(3) ...;
```

Konvertierungen für primitive Datentypen

```
class Konvertierung {
  public static void main(String[] args) {
    int i = 5;
    float f = i;            // erweiternde Konvertierung
    System.out.println(f); // gibt 5.0 aus
    float g = 5.9f;
    i = (int) g;            // einschränkende Konvertierung, cast
    System.out.println(i); // gibt 5 aus
    i = 1000000123;
    float h = i;            // Genauigkeitsverlust
    System.out.println(h); // gibt 1.00000013E9 aus
  }
}
```

Java: Grundlagen der Sprache: Speicherung von Werten

Speicherung von Informationen

Warum muss man wissen, wie Daten gespeichert werden?

Wir sehen uns zwei Beispiele an.

Speicherung von Informationen

```
public static void main(String[] args) {
  System.out.println(1*1);
  System.out.println(12*12);
  System.out.println(123*123);
  System.out.println(1234*1234);
  System.out.println(12345*12345);
  System.out.println(123456*123456);
}
```

```
Ausgabe:
  1
  144
  15129
  1522756
  152399025
  -1938485248    Diese Ausgabe ist offensichtlich falsch!
```

Speicherung von Informationen

```
public static void main(String[] args) {
  int z = 256*256*256*128+2147483647;
  System.out.println(z*z);
}
```

Ausgabe:

```
1     Hat er sich schon wieder verrechnet?
```

Speicherung von Informationen

- Die kleinste Informationseinheit ist das Bit. Ein Bit kann eine 0 oder eine 1 speichern. Ein Byte besteht aus 8 Bits.
- Im Rechner werden Informationen in Maschinenwörtern gespeichert. Ein Maschinenwort besteht aus n Bits. Dabei gilt typischerweise $n = 32$ oder $n = 64$.
- In einem Maschinenwort der Länge n können 2^n verschiedene Werte gespeichert werden.
- Beispiel: Eine ganze Zahl vom Typ int wird durch 32 Bits gespeichert. Damit wird der Bereich von $-2^{31} = -2\ 147\ 483\ 648$ bis $2^{31} - 1 = 2\ 147\ 483\ 647$ abgedeckt. Der Bundeshaushalt lässt sich dadurch nicht darstellen.

Der ASCII-Zeichensatz

- Im ASCII-Zeichensatz (American Standard Code for Information Interchange) werden Zeichen durch 7 Bits dargestellt. Es können $2^7 = 128$ verschiedene Zeichen gespeichert werden.

- Der ASCII-Zeichensatz umfasst 32 Steuerzeichen, 84 internationale Schriftzeichen (lateinisches Alphabet, Ziffern, Sonderzeichen) und 12 national festzulegende Zeichen. Er enthält keine Umlaute.

0100000	32	Leerzeichen
0100001	33	!
0100100	36	$
0100101	37	%
0110000	48	0
...	...	...
0110101	57	9
0111111	63	?
1000001	65	A
...	...	...
1011010	90	Z
1100001	97	a
...	...	...
1111010	122	z

Weitere Zeichensätze

- Der ISO-Latin-1-Code erweitert den ASCII-Zeichensatz auf 8 Bits. Er gestattet die Darstellung von Umlauten:

	dezimal	binär	hexadezimal
ä	228	11100100	E4

- Mit dem ISO-Latin-1-Code können alle Zeichen der deutschen Schriftsprache – zum Beispiel aber nicht alle der französischen – dargestellt werden. Beispielsweise sind die Zeichen Œ, œ und Ÿ nicht enthalten.

- Der ISO-Latin-9-Code enthält das Eurozeichen €.

- Der EBCDI-Code ist ein in der IBM-Welt verbreiteter 8-Bit-Code.

Der Unicode-Zeichensatz

- Der Unicode weist den weltweit verwendeten Schriftzeichen und Symbolen Hexadezimalzahlen von 0 bis 10FFFF zu: U+000000 ... U+10FFFF. Damit lassen sich im Prinzip

$$(1 + 10FFFF)_{16} = (110000)_{16} = 16^5 + 16^4 = 1\,048\,576 + 65\,536 = 1\,114\,112$$

  Zeichen darstellen. Bestimmte Zahlen werden nicht für die Codierung von Zeichen verwendet. Der Coderaum ermöglicht 1 111 998 Codepunkte.

- Die Unicode Version 6.2.0 vom September 2012 enthält ca. 110 000 Zeichen. Enthalten sind zum Beispiel auch mathematische Symbole, Noten, Runen, Piktogramme, ... Dieses entspricht ca. 10 % der möglichen Werte.

- Java 7 unterstützt Unicode Version 6.

Der Unicode-Zeichensatz

0000–007F	Basic Latin	entspricht dem ASCII-Zeichensatz
0080–00FF	Latin-1 Supplement	mit Basic Latin zusammen Iso-Latin-1-Code
0100–017F	Latin Extended-A	Erweiterungen für europäische Sprachen
0180–024F	Latin Extended-B	sonstige lateinische Sonderzeichen
...	...	...

- Beispiel: ä $\triangleq$ 228 $\triangleq$ 1110 0100 $\triangleq$ U+00E4

Unicode-Transformations-Formate

- Die Speicherung und Übertragung von Unicode-Zeichen erfolgt auf unterschiedliche Weisen. Man spricht von Unicode-Transformations-Formaten (UTF).
- Bekannte Formate sind UTF-8, UTF-16, UTF-32, UTF-EBCDIC, ...
- UTF-8 ist eine Kodierung mit variabler Länge. Jedem Unicode-Zeichen werden zwischen einem und vier Byte zugeordnet.
- Die Zeichen aus „Basic Latin“ werden durch 1 Byte dargestellt, die aus „Latin-1 Supplement“ und viele weitere durch 2 Byte.
- Bsp: ä $\triangleq$ 228 $\triangleq$ 1110 0100 $\triangleq$ U+00E4 $\triangleq$ 11000011 10100100 $\triangleq$ C3 A4 $\triangleq$ Ã¤

Die ganzzahligen Datentypen

- Ganze Zahlen der Datentypen `byte`, `short`, `int` und `long` werden im Zweierkomplement gespeichert. Das höchste Bit gibt das Vorzeichen an. Ist es 0, so ist die Zahl positiv, ist es 1, so ist die Zahl negativ.

- Zur Erläuterung soll folgendes Beispiel einer Zahl im Zweierkomplement von der Größe eines Bytes dienen:

1	0	1	0	0	1	1	1

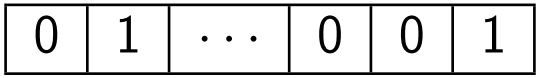
  Dieses Bitmuster stellt die Zahl

$$\text{-}128 + (32 + 4 + 2 + 1) = \text{-}89$$

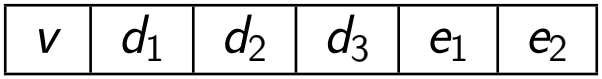
  dar.

Gleitkommadarstellung

Maschinenwort mit n Bits, typisch $n = 32$:



Wir setzen beispielsweise $n = 6$:



Vorzeichen: v
Mantisse: $d_i,\ i = 1, 2, 3$
Exponent: $e_j,\ j = 1, 2$

Gleitkommadarstellung

v	d_1	d_2	d_3	e_1	e_2	Zahl
.	.	.	.	.	.	.
1	1	1	1	1	0	$+1\frac{3}{4}$
1	1	0	0	0	1	$+\frac{1}{2}$
1	1	0	1	0	1	$+\frac{5}{8}$
1	1	1	0	0	1	$+\frac{3}{4}$
1	1	1	1	0	1	$+\frac{7}{8}$
.	.	.	.	.	.	.

$$
\begin{array}{lcr}
d_1 = 0 & \widehat{=} & 0 \\
v = 1 & \widehat{=} & + \\
e = (1,1) & \widehat{=} & 2 \\
e = (1,0) & \widehat{=} & 1 \\
e = (0,1) & \widehat{=} & 0 \\
e = (0,0) & \widehat{=} & -1
\end{array}
$$

Beispiel: 110101

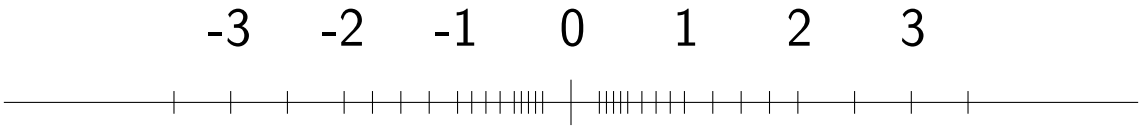
$$\boxed{+\left(\tfrac{1}{2}+\tfrac{0}{4}+\tfrac{1}{8}\right) * 2^0 = +\tfrac{5}{8}}$$

Gleitkommadarstellung

Es lassen sich die folgenden $2^5 + 1 = 33$ Zahlen darstellen:

$$0, \pm\tfrac{1}{4}, \pm\tfrac{5}{16}, \pm\tfrac{3}{8}, \pm\tfrac{7}{16}, \pm\tfrac{1}{2}, \pm\tfrac{5}{8}, \pm\tfrac{3}{4}, \pm\tfrac{7}{8},$$

$$\pm 1, \pm 1\tfrac{1}{4}, \pm 1\tfrac{1}{2}, \pm 1\tfrac{3}{4}, \pm 2, \pm 2\tfrac{1}{2}, \pm 3, \pm 3\tfrac{1}{2}$$



Gleitkommadarstellung

Die üblichen Rechenregeln gelten nicht:

$$\frac{5}{4} + \left(\frac{3}{8} + \frac{3}{8}\right) = \frac{5}{4} + \frac{3}{4} = 2$$

$$\left(\frac{5}{4} + \frac{3}{8}\right) + \frac{3}{8} = ? + \frac{3}{8} \neq 2$$

Die Lage ändert sich nicht grundsätzlich, wenn man $n = 32$ oder $n = 2 * 32$ ("double precision") wählt.

Gleitkommadarstellung

Der Ausdruck

$$\frac{1}{\left(\frac{1}{3}+\frac{1}{3}+\frac{1}{3}\right)-0.999\,999\,998}$$

ist in Gleitkommadarstellung zu schreiben und auszuwerten. Die Rechnung soll mit einer Genauigkeit von 9 Dezimalstellen durchgeführt werden:

Gleitkommadarstellung

	0.333 333 333
+	0.333 333 333
+	0.333 333 333
−	0.999 999 998
=	0.000 000 001
Kehrwert:	1 000 000 000
korrekter Wert:	500 000 000
absoluter Fehler:	500 000 000
relativer Fehler:	100 %

Wir haben uns also um Fünfhundertmillionen verrechnet – und das mit vier Rechenoperationen auf Zahlen, die zwischen 0 und 1 liegen.

Gleitkommadarstellung

Wir betrachten ein weiteres Beispiel:

- Die Zahl 0,2 hat die Binärdarstellung $0{,}001100110011\ldots = 0{,}\overline{0011}$.
- Normalisiert ergibt dies: $0{,}1100110011\ldots \cdot 2^{-2} = 0{,}11\overline{0011} \cdot 2^{-2}$.
- Auf 6 Stellen gerundet: $0{,}110011 \cdot 2^{-2}$.
- Zurückkonvertiert: $0{,}110011 \cdot 2^{-2} = \frac{51}{256} = 0{,}1992187500$.
- Der Test $0{,}2 = 0{,}1992187500$ wird daher bei einer Speicherung von 6 Nachkommastellen `true` liefern.

Statt $a = b$ sollte immer $|a - b| < \varepsilon$ getestet werden.

Gleitkommadarstellung

Ein Java-Beispiel:

```
double a = 1.0/3.0,
       b = 10.0 + a - 10.0, c;
if (a == b)
  c = 0;
else
  c = 1/(a-b);
System.out.printf("%20.5f%n",c);
```

Ausgabe: -1637672591771089.50000, d. h.
- 1 Billiarde 637 Billionen 672 Milliarden 591 Millionen 771 Tausend und 89,5

korrekter Wert: 0

Gleitkommadarstellung

Die Gleitkommadarstellung ist eine Methode zur näherungsweisen Darstellung von reellen Zahlen auf Rechenanlagen. Man spricht von Pseudoarithmetik:

- Es lassen sich nur endlich viele Zahlen darstellen.
- Für sie gelten die bekannten Rechenregeln i. Allg. nicht. Inbesondere gibt es Probleme bei
  - der Summe von Zahlen unterschiedlicher Größenordnung ($x + y = x$, falls $x >> y$),
  - der Differenz von nahezu gleich großen Zahlen (Auslöschung) und
  - dem Test von Zahlen auf Gleichheit.
- Bei jeder Operation „einlesen“, „berechnen“, ... entstehen Rundungsfehler.

Gleitkommadarstellung

- In Java werden `float`-Zahlen durch 32 Bits und `double`-Zahlen durch 64 Bits dargestellt.

- Die Aufteilung erfolgt folgendermaßen:

	`float`	`double`	Beispiel
Vorzeichenbit	1	1	1
Exponent	8	11	2
Mantisse	23	52	3
Anzahl der Bits	32	64	6

Schematisches Speicherbild

```
int b = 107123;
int[] r = {23, 24, 25};
```

symbolische Adresse	Adresse im Speicher	Inhalt der Speicherzelle	Bemerkung
	...		
b	094	107123	ganzzahliger Wert
	...		
r	101	123	Referenz
	...		
–	123	23	ab hier Feld r
	...	24	
	...	25	

Java: Grundlagen der Sprache: Ausdrücke

Ausdrücke

- Ein Ausdruck besteht aus einem Operator und Operanden. Die Anzahl der Operanden heißt Stelligkeit des Operators. Die meisten Operatoren in Java sind ein- oder zweistellig. Der Fragezeichen-Operator ist dreistellig.

- Die Operanden müssen von einem bestimmten Typ sein. Es können aber implizite oder explizite Typumwandlungen vorgenommen werden.

- Jeder Ausdruck hat einen bestimmten Rückgabewert, der durch Operator und Operanden festgelegt wird.

- Prioritäts- und Assoziativitätsregeln regeln die Auswertungsreihenfolge. Durch Klammerung kann jede beliebige Reihenfolge erreicht werden.

- Ausdrücke können Seiteneffekte besitzen, d. h. den Zustand verändern.

Ausdrücke

- Die Reihenfolge der Auswertung der Operanden ist festgelegt: Der linke Operand eines Ausdrucks ist vollständig vor dem rechten Operanden auszuwerten.

- Beispiel: Falls die Variable `i` den Wert 2 besitzt, dann ergibt die Auswertung des Ausdrucks

```
(i=3) * i
```

  den Wert 9 und nicht 6. Falls die Variable `i` den Wert 2 besitzt, dann ergibt die Auswertung des Ausdrucks

```
i * (i=3)
```

  den Wert 6. Diese Ausdrücke besitzen einen Seiteneffekt.

Arithmetische Operatoren

Operator	Stelligkeit	Bedeutung	Bemerkung
+	1	pos. Vorzeichen	
-	1	neg. Vorzeichen	
+	2	Summe	
-	2	Differenz	
*	2	Produkt	
/	2	Quotient	
%	2	Rest, mod	auch für Gleitkommazahlen
++	1	Präinkrement	++a ergibt a+1, dann a = a+1
++	1	Postinkrement	a++ ergibt a, dann a = a+1
--	1	Prädekrement	--a ergibt a-1, dann a = a-1
--	1	Postdekrement	a-- ergibt a, dann a = a-1

Ganzzahlige Division

Theorem. *Für alle Zahlen* $a, b \in \mathbb{Z}, b \neq 0$, *gibt es genau ein* $q \in \mathbb{Z}$ *und ein* $r \in \mathbb{Z}$ *mit*

$$a = q \cdot b + r \text{ und } 0 \leq r < |b|.$$

Beispiel: Für 7 und 3 lauten die verschiedenen Möglichkeiten:

a	b	q	r	
7	3	2	1	7=2·3+1
-7	3	-3	2	-7=(-3)·3+2
7	-3	-2	1	7=(-2)·(-3)+1
-7	-3	3	2	-7=3·(-3)+2

Ganzzahlige Division

Wenn die ganzzahlige Division durch diesen Satz definiert wird, dann erfüllt sie nicht die Gleichung

$$-\frac{a}{b} = \frac{-a}{b}.$$

Ein Gegenbeispiel ist

$$-\frac{7}{3} = -2 \neq -3 = \frac{-7}{3}.$$

Ganzzahlige Division

In Java gilt:

- Ist entweder a oder b negativ, so ist a/b negativ. Der Quotient ist die größte ganze Zahl q, für die $|q\,b| \leq |a|$ gilt.
- a % b = a - (a / b) * b.

a	b	a/b	a%b
7	3	2	1
-7	3	-2	-1
7	-3	-2	1
-7	-3	2	-1

Relationale Operatoren

Operator	Stelligkeit	Bedeutung	Bemerkung
==	2	Gleich	Für Referenztypen `true`, falls beide Werte auf dasselbe Objekt zeigen.
!=	2	Ungleich	Für Referenztypen `true`, falls beide Werte auf verschiedene Objekte zeigen.
<	2	Kleiner	
<=	2	Kleiner gleich	
>	2	Größer	
>=	2	Größer gleich	

Logische Operatoren

Operator	Stelligkeit	Bedeutung
!	1	logisches NICHT
&&	2	UND mit Short-Circuit-Evaluation
||	2	ODER mit Short-Circuit-Evaluation
&	2	UND ohne Short-Circuit-Evaluation
|	2	ODER ohne Short-Circuit-Evaluation
^	2	exklusives ODER

Bei einer Short-Circuit-Evaluation werden nicht
in jedem Falle die beiden Operanden ausgewertet.

Zuweisungsoperatoren

Operator	Stelligkeit	Bedeutung	Bemerkung
=	2	Einfache Zuweisung	`a = b` liefert `b`, weist `b` an `a` zu
+=	2	Additionszuweisung	`a += b` liefert `a+b`, weist `a+b` an `a` zu
−=	2	Subtraktionszuweisung	analog
*=	2	Multiplikationszuweisung	analog
/=	2	Divisionszuweisung	analog
%=	2	Restzuweisung	analog
&=	2	UND-Zuweisung	analog
...	2	...	analog

Eine Zuweisung ist ein Ausdruck mit einem Seiteneffekt. Sie liefert einen Rückgabewert.

Weitere Operatoren

- Bitweise Operatoren: `~ | & ^ >> >>> <<`

  Mithilfe der bitweisen Operationen kann auf die Binärdarstellung von primitiven Datentypen zugegriffen werden. Ein Wert wird dabei als Folge von Bits angesehen, die mit den bitweisen Operatoren einzeln abgefragt und manipuliert werden können.

- Fragezeichen-Operator: `? :`

  Der Fragezeichen-Operator ist der einzige dreistellige Operator in Java. Er erwartet einen logischen Ausdruck und zwei weitere Ausdrücke, die entweder beide numerisch, vom Typ `boolean` oder aber beide von einem Referenztyp sein müssen.

  `a ? b : c` liefert `b`, falls `a` wahr ist, und `c`, falls `a` falsch ist. Der Typ des Rückgabewerts ist der größere der beiden Typen von `b` und `c`.

Weitere Operatoren

- Type-Cast-Operator

  Der Ausdruck `(type) a` wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um. Das Ergebnis ist ein Ausdruck, der nur noch rechts von einem Zuweisungsoperator stehen darf.

  Der Type-Cast-Operator kann immer dann angewendet werden, wenn der Compiler keine oder nicht die gewünschten impliziten Typumwandlungen vornimmt.

- Zugriff auf Array-Elemente

  Wie wir bereits gesehen haben, erfolgt der Zugriff auf Array-Elemente durch eckige Klammern: `a[i]`.

Weitere Operatoren

- String-Verkettung

  Der Operator + wird auch zur Konkatenation von Strings verwendet. Auch dies ist uns bereits bekannt.

- Konkatenation

```
System.out.println(3+4);          // gibt 7 aus
System.out.println(""+3+4);       // gibt 34 aus
System.out.println(3+4+"");       // gibt 7 aus
System.out.println(""+(3+4));     // gibt 7 aus
```

  Erklärung: Die Operanden werden von links nach rechts ausgewertet.

Weitere Operatoren

- Der `instance-of`-Operator

  Der `instance-of`-Operator kann verwendet werden, um herauszufinden, zu welcher Klasse ein bestimmtes Objekt gehört. `a instanceof b` liefert genau dann `true`, wenn `a` und `b` Referenztypen sind und `a` eine Instanz von `b` (oder einer ihrer Unterklassen) ist.

- Der `new`-Operator

  Mit dem `new`-Operator werden Objekte und Arrays erzeugt. Der Rückgabewert ist das jeweils erzeugte Objekt.

Weitere Operatoren

- Member-Zugriff

  Der Zugriff auf Instanz- oder Klassenvariable bzw. -Methoden erfolgt mit dem Punktoperator und hat die Form `a.b`. Dabei ist `a` der Name einer Klasse bzw. einer Instanz der Klasse und `b` der Name auf den zugegriffen werden soll.

Weitere Operatoren

- Methodenaufruf
  - Eine Methode wird durch `f(parameter)` bzw. `f()` aufgerufen.
  - Es wird der durch die `return`-Anweisung angegebene Wert geliefert. Der Rückgabetyp entspricht dem vereinbarten Typ.
  - Ein Methodenaufruf kann Seiteneffekte haben.
  - Wenn überhaupt kein Wert benötigt wird und nur Seiteneffekte beabsichtigt sind, ist `void` als Rückgabetyp zu deklarieren.

Vorrangregeln für Operatoren

- `8-3-2` wird wie `(8-3)-2` ausgewertet und ergibt 3.

  Der Operator - ist linksassoziativ.

- `8-3*2` wird wie `8-(3*2)` behandelt und besitzt das Ergebnis 2.

  * besitzt eine höhere Priorität als -.

Vorrangregeln für Operatoren

- a, b und c seien Variable mit den Werten 1, 2 und 3. Der Ausdruck

  ```
  a-=b+=c
  ```

  wird wie a-=(b+=c) behandelt und bewirkt die Zuweisungen b = 2 + 3 und a = 1 - 5.

  Das Ergebnis des Ausdrucks ist daher -4. Außerdem werden die Zuweisungen b = 5 und a = -4 ausgeführt.

  Die Operatoren += und -= sind rechtsassoziativ.

Vorrangregeln für Operatoren

- Java unterscheidet Prioritätsstufen. Innerhalb einer Stufe sind alle Operatoren entweder links- oder rechtsassoziativ.
- Beispielsweise gehören die Operatoren *, / und % zu einer höheren Prioritätsstufe als die Operatoren + und −, d. h., Punktrechnung geht vor Strichrechnung.
- Die Operatoren <, <=, > und >= stehen auf einer niedrigeren Prioritätsstufe als die arithmetischen Operatoren, == und != besitzen eine noch niedrigere Priorität.
- Die arithmetischen Operatoren und die Vergleichsoperatoren sind linksassoziativ.
- Zuweisungsoperatoren besitzen die niedrigste Priorität und sind rechtsassoziativ.
- Auf der Internet-Seite dieser Veranstaltung finden Sie eine Liste der Operatoren.

Java: Grundlagen der Sprache: Anweisungen

Anweisungen

- Elementare Anweisungen
  - Leere Anweisung
  - Block
  - Deklaration
  - Ausdrucksanweisung
- Anweisungen zur Selektion
  - Die `if`-Anweisung
  - Die `switch`-Anweisung

Anweisungen

- Anweisungen zur Iteration
  - Die `while`-Schleife
  - Die `do`-Schleife
  - Die `for`-Schleife
- Weitere Anweisungen
  - Die `break`-Anweisung
  - Die `continue`-Anweisung
  - Die `return`-Anweisung
  - Später: `throw`-, `try-catch`-, `assert`-Anweisung,...

Die leere Anweisung

Syntax:

```
;
```

Die leere Anweisung besteht nur aus einem Semikolon und hat keine Auswirkung auf die Ausführung des Programms.

Der Block

Syntax:

```
{
  Anweisung_1;
  Anweisung_2;
  ...
  Anweisung_n;
}
```

- Ein Block ist eine Zusammenfassung von Anweisungen, die nacheinander ausgeführt werden. Ein Block gilt als eine einzelne Anweisung.
- Ein Block kann Anweisungen zur Deklaration von Variablen enthalten. Diese sind dann nur in diesem Block sichtbar.

Deklaration von Variablen

Syntax:

```
Typname Variablenname;
```

oder

```
Typname Variablenname = Initialwert;
```

Die Deklaration einer lokalen Variablen ist eine Anweisung. Die Sichtbarkeit der Variablen erstreckt sich von der Deklaration bis zum Ende des umschließenden Blocks. Lokale Variable dürfen einander nicht verdecken, wohl aber Instanz- und Klassenvariable.

Definite Assignment

- Jede lokale Variable muss vor ihrer ersten Benutzung initialisiert werden.
- Der Compiler überprüft dies – mehr oder weniger erfolgreich – mit einer Datenflussanalyse.
- In der Regel ist es ein Fehler, wenn eine Variable nicht initialisiert wird. Die Datenflussanalyse erkennt dann diesen Fehler.
- Solche Analysen haben ihre Grenzen.

Definite Assignment

```
  static void Test(int i) {
    int k;
    if (i > 2) {
      k = 5;
    }
    System.out.println(k);
  }

variable k might not have been initialized
    System.out.println(k);
                       ^
1 error
```

Definite Assignment

Die Meldung tritt auch im folgenden Falle auf:

```
static void Test(int i) {
  int k;
  if (i > 2) {
    k = 5;
  }
  if (i <= 2) {
    k = 6;
  }
  System.out.println(k);
}
```

Definite Assignment

Hingegen beanstandet der Compiler das folgende Programm nicht:

```
static void Test(int i) {
  int k;
  if (i > 2)
    k = 5;
  else
    k = 6;
  System.out.println(k);
}
```

Ausdrucksanweisung

Syntax:

```
ausdruck;
```

Ein Ausdruck ist eine Anweisung. Der Rückgabewert des Ausdrucks wird ignoriert. Wenn ein Ausdruck als Anweisung verwendet wird, kommt es also auf den Seiteneffekt an. Daher sind in diesem Kontext nur die folgenden Ausdrücke zugelassen:

- Zuweisung, Inkrement und Dekrement,
- Methodenaufruf,
- Instanzerzeugung.

Vertauschen zweier Werte

Beispiel: Die Werte der Variablen x und y sollen getauscht werden.

```
int t = x;
x = y;
y = t;
```

Die If-Anweisung

Syntax:

```
if (ausdruck)
  anweisung;
```

```
if (ausdruck)
  anweisung1;
else
  anweisung2;
```

- Es wird zunächst der Ausdruck ausgewertet.
- Ist das Ergebnis `true` wird `anweisung` bzw. `anweisung1` ausgeführt.
- Kommt `false` heraus, so wird keine Anweisung bzw. `anweisung2` ausgeführt.

Die If-Anweisung

Falls If-Anweisungen geschachtelt werden, so gehört ein evtl. auftretender `else`-Zweig stets zum innersten `if`.

```
if (a)
  if (b)
    s;
  else
    t;
```

Dieses Problem wird dangling else genannt.

Maximum zweier Zahlen

Beispiel: Das Maximum von x und y soll an t zugewiesen werden.

```
int t;
if (x <= y)
  t = y;
else
  t = x;
```

Maximum dreier Zahlen

Beispiel: Das Maximum von x, y und z soll an t zugewiesen werden.

```
int t;
if (x <= y) {
  if (y <= z)
    t = z;
  else
    t = y;
}
else if (x >= z)
  t = x;
else
  t = z;
```

Betrag einer Zahl

Beispiel: Der Betrag von x soll an t zugewiesen werden.

```
int t;
if (x >= 0)
  t = x;
else
  t = -x;
```

Die Switch-Anweisung

Syntax:

```
switch (ausdruck) {
  case constant1: anweisung1;
                  break;
  case constant2: anweisung2;
                  break;
  ...
  default:        anweisung;
                  break;
}
```

Die Switch-Anweisung

- Die Switch-Anweisung ist eine Mehrfachverzweigung.
- Zunächst wird der Ausdruck, der von einem der Typen `char`, `byte`, `short` oder `int` bzw. einer einer zugehörigen Wrapper-Klasse oder von einem Enumeration-Typ sein muss, ausgewertet und das Ergebnis mit den Konstanten verglichen. Seit Java 7 sind `String`–Ausdrücke zulässig.
- Kommt es dort vor, so wird die entsprechende Anweisung ausgeführt. Ist kein `break` vorhanden, so wird mit der nächsten Anweisung fortgefahren.
- Tritt das Ergebnis nicht in der Liste der Konstanten auf, so wird die Default-Anweisung ausgeführt. Der Default-Zweig ist optional.

Die Switch-Anweisung

Beispiel: Abhängig vom Wert von i soll ein String an s zugewiesen werden.

```
switch (i) {
  case  1: s = "Januar";
           break;
  ...
  case 12: s = "Dezember";
           break;
  default: s = "Kein gültiger Monat";
           break;
}
```

`switch`-Anweisung

Syntax:

```
switch (ausdruck) {
  case constant1: anweisung1; break;
  case constant2: anweisung2; break;
  ...
  default:        anweisung; break;
}
```

Achtung: Falls der Ausdruck (`ausdruck`) vom Typ `String` ist, erfolgt der Vergleich zwischen dem Ausdruck und den Konstanten mithilfe der `equals`-Methode der `String`-Objekte.

Die While-Schleife

Syntax:

```
while (ausdruck)
  anweisung
```

- Zunächst wird der Ausdruck, der vom Typ `boolean` sein muss, geprüft.
- Ist das Ergebnis `true`, so wird die Anweisung ausgeführt und von vorne begonnen.
- Ist das Ergebnis dagegen `false`, wird die While-Schleife beendet.

Die While-Schleife

Beispiel: Es ist eine Methode zu schreiben, die eine ganze Zahl i als Parameter erhält und deren größten echten Teiler als Ergebnis liefert.

```
static int größterTeiler(int i) {
  int j = i / 2;
  while ( i % j != 0 ) {
    j--;
  }
  return j;
}
```

Die Do-Schleife

Syntax:

```
do
  anweisung
while (ausdruck)
```

- Zuerst wird die Anweisung ausgeführt und anschließend der Ausdruck ausgewertet.
- Ist das Ergebnis `true`, so wird die Do-Schleife erneut ausgeführt, im anderen Fall endet die Schleife.

Die Anweisung wird also immer mindestens einmal bearbeitet. Man sagt, die Do-Schleife sei nicht abweisend, während die While-Schleife abweisend ist.

Die For-Schleife

Syntax:

```
for (init; test; update)
  anweisung;
```

- Der `init`-Teil wird einmal vor dem Start der Schleife aufgerufen. Er dient dazu, Initialisierungen vorzunehmen. Im `init`-Teil dürfen Deklarationen enthalten sein, die dann in der For-Schleife sichtbar sind.

- Der `test`-Teil wird vor jedem Schleifendurchlauf ausgewertet. Ergibt dies `true` wird anschließend die Anweisung ausgeführt, sonst endet die Schleife.

Die For-Schleife

Syntax:

```
for (init; test; update)
  anweisung;
```

- `update` wird nach jedem Durchlauf der Schleife und vor der nächsten Auswertung des Testausdrucks ausgeführt. Hier kann z. B. der Schleifenzähler erhöht werden.

- `init`, `test` und `update` dürfen fehlen bzw. aus mehreren Teilen bestehen. Fehlt `test`, so wird `true` angenommen.

Die For-Schleife

Beispiel: Von allen Zahlen bis 100 soll der größte echte Teiler berechnet und ausgegeben werden:

```
for (int i = 2; i <= 100; i++) {
  System.out.println(größterTeiler(i));
}
```

Maximum eines Feldes

Beispiel: Das Maximum der Elemente des Feldes a soll an t zugewiesen werden.

```
int t = a[0];
for (int i = 1; i < a.length; i++) {
  if (a[i] > t) {
    t = a[i];
  }
}
```

Summe ganzer Zahlen

Beispiel: Es soll die Summe der Zahlen von a bis b, d. h. $\sum_{i=a}^{b} i$, berechnet werden.

```
int sum = 0;
for (int i = a; i <= b; i++) {
  sum += i;
}
```

Fakultät einer Zahl

Beispiel: Es soll a! berechnet und an x zugewiesen werden.

```
int x = 1;
for (int i = 2; i <= a; i++) {
  x = x * i;
}
```

Potenz

Beispiel: Die Potenz a^n soll an p zugewiesen werden.

```
int p = 1;
for (int i = 1; i <= n; i++) {
  p = p * a;
}
```

Primzahlberechnung

Beispiel: Das folgende Programmfragment berechnet die Primzahlen bis 100 und gibt diese aus.

```
static boolean prim(int i) {
  if (i >= 2 && größterTeiler(i) == 1)
    return true;
  else
    return false;
}
 ...
for (int i = 0; i <= 100; i++) {
  if (prim(i))
    System.out.println(i);
}
```

Primzahlberechnung

Zugegeben – der Algorithmus ist nicht sehr geschickt. Er soll ja auch nur die verwendeten Anweisungen und Operatoren illustrieren.

Übungsaufgabe:

Was passiert, wenn statt && der Operator & benutzt wird?

Die erweiterte For-Schleife

Syntax:

```
for (typ variablenname : ausdruck)
  anweisung;
```

- Die erweiterte For-Schleife kann gelesen werden als:
  Führe die Anweisung für alle Werte aus, die die Variable im Ausdruck annehmen kann.

- Initialisierung, Test und Update der Laufvariablen brauchen nicht explizit angegeben zu werden.

Die erweiterte For-Schleife

Beispiel: Es soll die Summe aller Elemente des Felds a berechnet werden.

```
sum = 0;
for (int i = 0; i < a.length; i++)
  sum += a[i];
System.out.println(sum);

sum = 0;
for (int i : a)
  sum += i;
System.out.println(sum);
```

Die erweiterte For-Schleife

Beispiel: Es soll die Summe aller Elemente der Matrix `b` berechnet werden.

```
sum = 0;
for (int i = 0; i < b.length; i++)
  for (int j = 0; j < b[i].length; j++)
    sum += b[i][j];
System.out.println(sum);

sum = 0;
for (int[] zeile : b)
  for (int eintrag : zeile)
    sum += eintrag;
System.out.println(sum);
```

Break- und Continue-Anweisungen

- Tritt innerhalb einer Schleife eine `break`-Anweisung auf, wird die Schleife verlassen und mit der ersten Anweisung nach der Schleife fortgefahren.

- Bei einer `continue`-Anweisung wird der jeweilige Schleifendurchlauf abgebrochen und mit der nächsten Iteration weitergemacht.

- `break` und `continue` können mit Marken versehen werden. Zu jeder Marke muss es eine mit dieser Marke versehene Kontrollstruktur geben.

Die Return-Anweisung

Syntax:

```
return ausdruck;
```

- Hat eine Methode einen Rückgabewert (ist also nicht vom Typ `void`), so wird dieser Wert durch die Return-Anweisung an den Aufrufer übergeben.
- Die Ausführung einer Return-Anweisung führt zum Beenden der Methode.

Fibonacci-Folge

Leonardo von Pisa (ca. 1170–1240), genannt Fibonacci, untersuchte, wie sich eine Kaninchenpopulation unter den folgenden Annahmen verhält:

- Zur Zeit 0 existiert genau ein junges Kaninchenpaar.
- Jedes geschlechtsreife Paar erzeugt in jedem Monat ein weiteres Paar Kaninchen.
- Kaninchen werden nach 2 Monaten geschlechtsreif.
- Kaninchen leben unbegrenzt lange.

Fibonacci-Folge

Die Population wächst also gemäß der folgenden Tabelle:

Zeit	jung	alt	gesamt
0	1	-	1
1	-	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8

Es ergibt sich die Folge: 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci-Folge

Wir erhalten das folgende Programmfragment:

```
int x = 0,
    y = 1;
while (true) {
  System.out.println(y);
  int t = x + y;
  x = y;
  y = t;
}
```

Java: Grundlagen der Sprache: Beispiele aus der Praxis

Vorbemerkungen

- In diesem Abschnitt wollen wir an weiteren Beispielen die bisher vorgestellten Sprachkonzepte von Java üben.
- Wir leiten die Algorithmen her, verzichten jedoch auf einen strengen Nachweis ihrer Korrektheit. Ebenso untersuchen wir nicht im Detail die Komplexität der einzelnen Verfahren.
- Die angegebenen Algorithmen sind nicht in jedem Falle eine optimale Lösung für das jeweilige Problem.
- In einigen Beispielen wurden zur Verdeutlichung konkrete Werte eingesetzt – in der Praxis wird man die Programme jedoch allgemeiner fassen.

Übersicht

- Finanzmathematik
  - Zinseszins
- Zahlentheorie
  - Teilbarkeit
  - Darstellung einer Zahl zu einer beliebigen Basis
  - Schnelle Exponentiation
- Algebra
  - Horner-Schema

Übersicht

- Lineare Algebra
  - Skalarprodukt zweier Vektoren
  - Länge eines Vektors
  - Produkt zweier Matrizen
- Analysis
  - Newton-Verfahren
- Sortierverfahren
  - Bubblesort
- Mathematische Algorithmen
  - Zufallszahlen

Zinseszins

Beispiel: Auf welchen Betrag wächst eine jährliche Einlage nach n Jahren bei gegebenem Zinssatz?

```
double einlage, zins, endbetrag;
int lz;
einlage = ...  // Jährliche Einlage
zins    = ...  // Zinssatz
lz      = ...  // Laufzeit
endbetrag = einlage;
for (int i = 1; i <= lz; i++) {
  endbetrag = endbetrag * (1 + zins) + einlage;
  System.out.println("Wert nach " + i + " Jahren: " + endbetrag);
}
```

Teilbarkeit

Beispiel: Durch welche ihrer Ziffern ist eine dreistellige Zahl teilbar?

```
int zahl      = ... // zu untersuchende Zahl
int einer     = zahl         % 10;
int zehner    = (zahl /  10) % 10;
int hunderter = (zahl / 100);
if (einer != 0 && zahl % einer == 0)
  System.out.println("Zahl ist durch " + einer + " teilbar!");
if (zehner != 0 && zahl % zehner == 0)
  System.out.println("Zahl ist durch " + zehner + " teilbar!");
if (hunderter != 0 && zahl % hunderter == 0)
  System.out.println("Zahl ist durch " + hunderter + " teilbar!");
```

Teilbarkeit

Beispiel: Durch welche ihrer Ziffern ist eine beliebige positive Zahl teilbar?

```
int zahl  = ... // zu untersuchende Zahl;
int dummy = zahl;
while (dummy != 0) {
  int einer = dummy % 10;
  dummy = dummy / 10;
  if (einer != 0 && zahl % einer == 0)
    System.out.println("Zahl ist durch " + einer + " teilbar!");
}
```

Darstellung einer Zahl zu einer beliebigen Basis

Beispiel: Eine in Dezimaldarstellung gegebene Zahl a soll in die Darstellung zur Basis b, $2 \leq b \leq 9$, umgewandelt werden.

```
int x;
String d = "";
x = a;
do {
  d = x % b + d;
  x = x / b;
} while (x > 0);
```

Darstellung einer Zahl zu einer beliebigen Basis

Beispiel: Eine Zahl d in der String-Darstellung zur Basis b, $2 \leq b \leq 9$, soll in die Dezimaldarstellung umgerechnet werden.

```
int s = 0,
    y = 1;
for (int i = d.length() - 1; i >= 0; i--) {
  s += y * ((int)d.charAt(i) - 48);
  y *= b;
}
```

Schnelle Exponentiation

Beispiel: Die Potenz a^n soll effizient berechnet werden.

```
int a1 = a, n1 = n, x = 1;
while (n1 != 0) {
  while (n1 % 2 == 0) {
    n1 = n1 / 2;
    a1 = a1 * a1;
  }
  n1 = n1 - 1;
  x = x * a1;
}
```

Horner-Schema

Beispiel: Ein Polynom a soll an der Stelle t ausgewertet und gleichzeitig durch den Linearfaktor $(x - t)$ dividiert werden.

```
int[] a = {-16, -20, -2, 2};
int t = 5;
int n = a.length - 1, y;
int[] b = new int[n];
b[n-1] = a[n];
for (int i = n - 2; i >= 0; i--) {
  b[i] = t * b[i+1] + a[i+1];
}
y = t * b[0] + a[0];
```

Skalarprodukt zweier Vektoren

Beispiel: Es ist das Skalarprodukt der Vektoren a und b zu berechnen.

```
int[] a = {-2,2,-2,2};
int[] b = {6,-2,4,5};
int t = 0;

for (int i = 0; i < a.length; i++) {
  t = t + b[i] * a[i];
}
```

Länge eines Vektors

Beispiel: Es ist die Länge des Vektors c zu berechnen.

```
int[] c = {1,1,1,0};
int r = 0;
double d;
for (int i = 0; i < c.length; i++) {
  r += c[i] * c[i] ;
}
d = Math.sqrt(r);
```

Produkt zweier Matrizen

Beispiel: Das Produkt der Matrizen aa und bb ist zu berechnen.

```
int[][] aa = {{1,2,3}, {4,5,6}};
int[][] bb = {{1,2,3,0}, {3,4,5,1}, {1,1,2,0}};
int[][] cc = new int[aa.length][bb[0].length];
for (int i = 0; i < aa.length; i++) {
  for (int j = 0; j < bb[0].length; j++) {
    int s = 0;
    for (int k = 0; k < bb.length; k++) {
      s += aa[i][k] * bb[k][j];
    }
    cc[i][j] = s;
  }
}
```

Nullmatrix

Beispiel: Enthält eine Matrix lauter Nullen?

```
boolean nullmatrix = true;
l: for (int i = 0; i < dd.length; i++) {
  for (int j = 0; j < dd[i].length; j++) {
    if (dd[i][j] != 0) {
      nullmatrix = false;
      break l;
    }
  }
}
```

Break- und Continue-Anweisung

Beispiel: Wie viele Zeilen einer Matrix enthalten eine 0?

```
int anzahl = 0;
l1: for (int i = 0; i < dd.length; i++) {
  l2: for (int j = 0; j < dd[i].length; j++) {
    if (dd[i][j] == 0) {
      anzahl += 1;
      break l2;        // oder: continue l1;
    }
  }
}
```

Das Newton-Verfahren

Beispiel: Es soll ein Java-Fragment programmiert werden, das eine „reelle“ Zahl $a \geq 0$ als Eingabe akzeptiert und deren Quadratwurzel $\sqrt{a}$ als Ergebnis liefert:

Lösungsidee:

1. Schritt: Suche eine geeignete Funktion $f : \mathbb{R} \to \mathbb{R}$ mit der Eigenschaft

$$f(x) = 0 \Longleftrightarrow x = \sqrt{a}$$

2. Schritt: Bestimme mithilfe eines numerischen Verfahrens die Nullstelle(n) von f.

Das Newton-Verfahren

1. Schritt: Die Funktion $f : \mathbb{R} \to \mathbb{R}$ mit

$$f(x) = x^2 - a$$

erfüllt wegen

$$\begin{aligned} f(x) = 0 &\iff x^2 - a = 0 \\ &\iff x = \pm\sqrt{a} \end{aligned}$$

die geforderte Bedingung.

Das Newton-Verfahren

2. Schritt: Wir wählen das Newton-Verfahren zur numerischen Bestimmung der positiven Nullstelle von f:

- Wähle einen geeigneten Startwert $x_0 \in \mathbb{R}$.
- Berechne die Folge $x_1, x_2, x_3, \ldots$ mit

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

bis eine Näherung „gut genug“ ist.

Das Newton-Verfahren

1. Wir wählen den Startwert $x_0 = 1.0$.

2. Was heißt „gut genug“? Es gibt mehrere Möglichkeiten, das Verfahren zu beenden:

   - $\left|x_n^2 - a\right| < \varepsilon$
   - $\left|x_{n+1} - x_n\right| < \varepsilon$
   - Es werden genau n_0 Schritte ausgeführt.

   Wir entscheiden uns für die erste Alternative.

Das Newton-Verfahren

Als Beispiel wollen wir die Wurzel aus 3 ziehen. Es ergibt sich somit das folgende Java-Programmfragment:

```
double a = 3.0,
       x = 1.0,
       eps = 0.000001;
do {
  x = (x + a / x) * 0.5;
  System.out.println(x);
} while (Math.abs(x * x - a ) > eps);
```

Das Newton-Verfahren

Die Variable x durchläuft die folgenden Werte:

1.0
2.0
1.75
1.7321428571428572
1.7320508100147274

Danach terminiert der Algorithmus. Der genaue Wert von $\sqrt{3.0}$ auf acht Dezimalen gerundet lautet 1.73205081.

Bubblesort

Beispiel: Das Feld f enthält ganze Zahlen und ist aufsteigend zu sortieren.

```
boolean unsortiert = true;
while (unsortiert) {
  int a = 0, x = 0, y = 1, z;
  while (y < f.length) {
    if (f[x] <= f[y]) {
      a++;
    }
    else {
      z = f[x];
      f[x] = f[y];
      f[y] = z;
    }
```

Bubblesort

```
      y++;
      x++;
    }
    if (a == f.length - 1) {
      unsortiert = false;
    }
  }
```

a zählt die Anzahl der sortierten Paare `f[x]<=f[y]`. Falls ein Paar nicht sortiert ist, werden die Werte getauscht.

Zufallszahlen

Beispiel: Erzeugung von Pseudozufallszahlen mit der linearen Kongruenzmethode.

```
int m = 100000000, m1 = 10000, b = 31415821;
int[] a = new int [1000];
int seed = 7633577;
a[0] = seed;
for (int i = 1; i < a.length; i++) {
  int p1 = a[i-1] / m1,
      p0 = a[i-1] % m1,
      q1 = b / m1,
      q0 = b % m1,
      prod = (((p0*q1+p1*q0)%m1)*m1+p0*q0)%m;
  a[i] = (prod + 1) % m;
}
```

Java: Grundlagen der Sprache: Ein Blick auf imperatives Programmieren

Imperatives Programmieren

- In einem imperativen Programm gibt es Variable, die Werte speichern können. Die Variablen und ihre Werte sowie der nächste auszuführende Befehl legen den Zustand fest. Der Zustand ändert sich mit dem Ablauf des Programms.

- Die Wertzuweisung gestattet es, den Zustand zu verändern.

- ◦ Mit einer Sequenz können Anweisungen nacheinander ausgeführt werden.
  ◦ Die Selektion erlaubt die Auswahl zwischen Anweisungen.
  ◦ Anweisungen können mit der Iteration wiederholt werden.

- ◦ Eingabe-Anweisungen ermöglichen es, den Zustand von außen zu beeinflussen.
  ◦ Ausgabe-Anweisungen erstellen einen Ausdruck (eines Teils) des Zustands.

Imperatives Programmieren

In Java kann man vom

Zustand eines Objektes

und vom

Zustand des Programms

sprechen.

Algorithmus von Euklid

- imperative Formulierung:

```
while b # 0
  do r := a mod b;
     a := b;
     b := r
  od
```

Algorithmus von Euklid (Java)

```
class Euklid {
  static int ggt(int a, int b) {
    int r;
    while (b != 0) {
      r = a % b;
      a = b;
      b = r;
    };
    return a;
  };
  public static void main(String[] args) {
    System.out.printf("Der ggT beträgt %d.%n", ggt(36,52));
  }
}
```

Algorithmus von Euklid (C)

```
int ggt(int a, int b) {
  int r;
  while (b != 0) {
    r = a % b;
    a = b;
    b = r;
  };
  return a;
};
main() {
  printf("Der ggT beträgt %d.\n", ggt(36,52));
}
```

Algorithmus von Euklid (JavaScript)

```
function ggt(a, b) {          // Parameter ohne Typangaben
  var r;                      // keine Typdeklaration
  while (b != 0) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

Algorithmus von Euklid (Pascal)

```
program euklid(output);
function ggt(a, b : integer) : integer;
  var r : integer;
  begin while b <> 0 do
          begin r := a mod b;
                a := b;
                b := r
          end;
        ggt := a
  end;
begin
  writeln('Der ggT beträgt ',ggt(36,52):1,'.')
end.
```

Algorithmus von Euklid (Modula-2)

```
MODULE euklid;
FROM InOut IMPORT
     WriteString, WriteInt, WriteLn;
PROCEDURE ggt(a , b : INTEGER) : INTEGER;
    VAR r : INTEGER;
    BEGIN WHILE b # 0
          DO r := a MOD b;
             a := b;
             b := r
          END;
          RETURN a
    END ggt;
```

Algorithmus von Euklid (Modula-2)

```
BEGIN
    WriteString("Der ggT beträgt ");
    WriteInt(ggt(36,52),1);
    WriteString(".");
    WriteLn;
END euklid.
```

Algorithmus von Euklid (Scheme, imperativ)

```
(define (ggT a b)
   (let ((r 0))
   (do ()
       ((= b 0) a)
        (set! r (remainder a b))
        (set! a b)
        (set! b r))))

(ggT 36 52)
```

Konzepte imperativer Programmierung

- Prozeduren: Abstraktionen von Anweisungen
- Funktionen: Abstraktionen von Ausdrücken
- Datentypen:
  - Primitive Datentypen:
    `boolean`, `char`, `int`, `real`, `enumeration`, `...`
  - Zusammengesetzte Datentypen:
    `array`, `record/struct`, `union`, `pointer`, `...`
  - Typdeklarationen: Abstraktionen von Datentypen

  Typsysteme können unabhängig von Paradigmen und Sprachen definiert und untersucht werden.

Konzepte imperativer Programmierung

- Weitere Kontrollstrukturen
- Module
- Ausnahmebehandlung
- Parallelverarbeitung

Algorithmus von Euklid

- imperative/iterative Formulierung:

```
while b # 0
  do r := a mod b;
     a := b;
     b := r
  od
```

- funktionale/rekursive Formulierung:

$$\texttt{ggT(a,b)} = \begin{cases} \texttt{a} & \texttt{b} = 0 \\ \texttt{ggT(b, a mod b)} & \texttt{b} \neq 0 \end{cases}$$

Algorithmus von Euklid (Scheme, funktional)

```
(define (ggT a b )
   (if (= b 0)
       a
       (ggT b (remainder a b))))

(ggT 36 52)
```

Algorithmus von Euklid (Haskell)

```
ggT :: Integer -> Integer -> Integer
ggT a b | b == 0 = a
        | otherwise = ggT b (a `mod` b)

ggT 36 52
```