Wo sind wir?

- Java-Umgebung
- Lexikale Konventionen
- **Datentypen**
- Kontrollstrukturen
- Ausdrücke
- Klassen, Pakete, Schnittstellen
- JVM
- Exceptions
- Java Klassenbibliotheken
- Ein-/Ausgabe
- Collections
- Threads
- Applets, Sicherheit
- Grafik
- Beans
- Integrierte Entwicklungsumgebungen

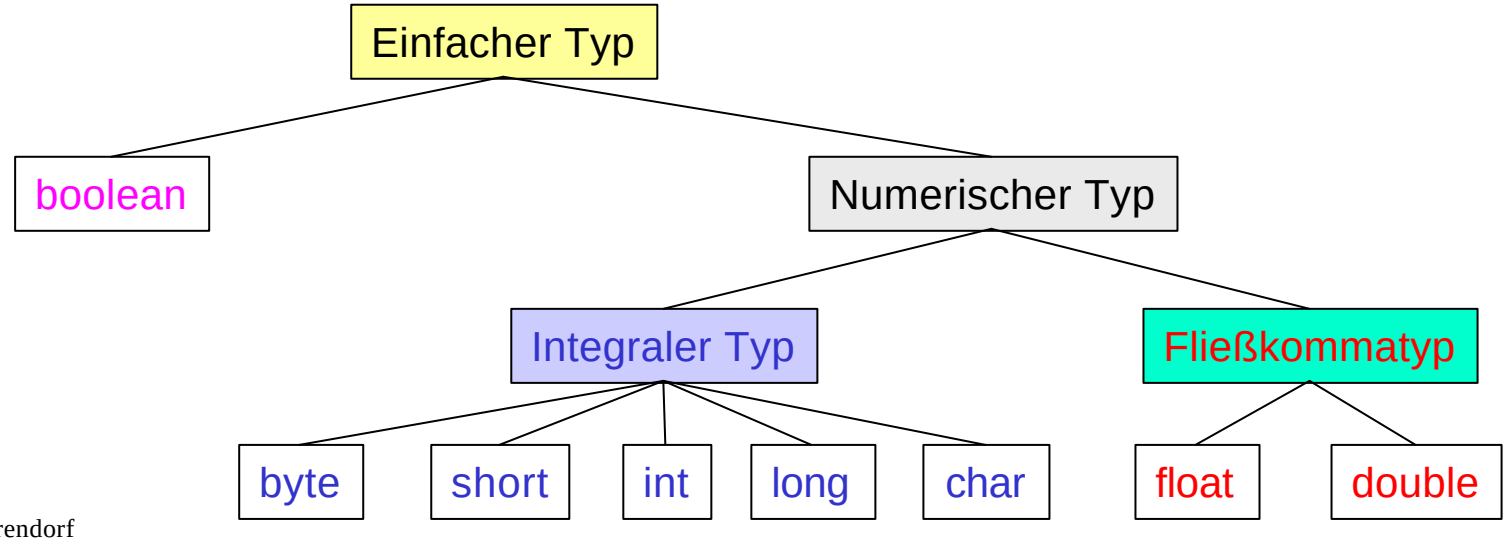
Typen

Java ist eine streng typisierte Sprache: Jede Variable und jeder Ausdruck besitzt einen zur Übersetzungszeit bekannten eindeutigen Typ.

Java teilt alle Typen in zwei Kategorien auf:

- Einfache Typen (`boolean, byte, short, int ,long, char, float, double`)
  Der Wert eines einfachen Typen ist das entsprechende Datum.
- Referenztypen (Klasse, Schnittstelle, Feld)
  Werte eines Referenztyps sind Referenzen (Zeiger) auf Objekte.

Die einfachen Typen unterteilt man weiter in:



Integrale Typen

Typ	Größe (Bit)	Kleinster Wert	Größter Wert
`byte`	8	-128	127
`short`	16	-32.768	32.767
`int`	32	-2.147.483.648	2.147.483.647 ($> 10^9$)
`long`	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807 ($> 10^{18}$)
`char`	16	\u0000	\uffff (65536)

Die Werte von `byte, short, int` und `long` werden in Zweierkomplementdarstellung repräsentiert, Werte vom Typ char als 16 Bit vorzeichenlose ganze Zahlen.

Ganzzahl-Arithmetik generiert bei Operationen niemals einen Fehler wegen Überlauf oder Unterlauf. Weder der Compiler noch das Laufzeitsystem merken dies.

Fließkommatypen

	`float`	`double`
Größe (Bit)	32	64
kleinster negativer Wert	-3.4028235e38	-1.7976931348623157e-308
größter negativer Wert ¹ 0.0	-1.4e-45	-4.9e-324
kleinster positiver Wert ¹ 0.0	1.4e-45	4.9e-324
größter positiver Wert	3.4028235e38	1.7976931348623157e-308
Signifikante dezimale Nachkommastellen	7	14

Die Werte von `float` und `double` werden in IEEE 754 Darstellung repräsentiert, Werte vom Typ `float` im 32-Bit Format (1 Bit Vorzeichen, 23 Bit Mantisse, 8 Bit Exponent), Werte vom Typ `double` im 64-Bit Format (1 Bit Vorzeichen, 52 Bit Mantisse, 11 Bit Exponent).

Dezimalpunkt ist Punkt und nicht Komma.

Spezielle Fließkommawerte

- Unterscheidung +0.0, -0.0
- +∞, -∞
- NaN (Not a Number; z.B. bei Division durch 0)

Alle Fließkommawerte bis auf NaN sind geordnet:

-∞ < negativen Werte < -0.0 = +0.0 < positiven Werte < +∞

Die Unterscheidung zwischen +0.0 und -0.0 ist bei einigen Operationen wichtig:

- 1.0 / +0.0 = +∞
- 1.0 / -0.0 = -∞

Operationen mit NaN als ein Operand liefern NaN. Eine Operation mit Überlauf liefert +/-∞, eine Operation mit Unterlauf liefert +/-0.0.

Der Typ void

Es gibt einen speziellen Typ `void`, der sinngemäß "Nichts" bedeutet. Hauptsächlich wird er bei Methodendeklarationen benutzt, um anzugeben, dass eine Methode keinen Rückgabewert liefert.

Beispiel:

```
class Test {

    static void ausgeben() {

        System.out.println("Hier ist die erwartetet Ausgabe");
        return;           // kein Resultatwert
    }


    public static void main(String[] args) {

        ausgeben();
    }
}
```

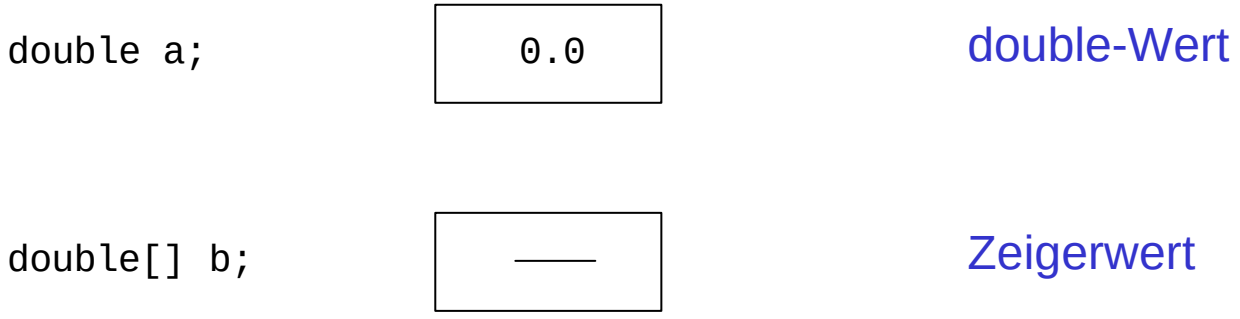
Referenztypen

Es gibt drei Arten von Referenztypen:

- Feldtypen
- Klassentypen (später)
- Schnittstellentypen (verwandt mit den Klassentypen; später)

Unterschiede zu einfachen Typen:

- Ein Referenzwert ist ein Zeiger auf ein Objekt und nicht das Objekt selbst.
- Werte einfacher Typen sind atomare Werte, Klassen- und Feldtypen sind zusammengesetzte Typen.



Felder

Felder (Engl. array), können Elemente eines Grundtyps (beliebiger Typ) bis zu einer bei der Feldinstanziierung vorgegebenen Maximalzahl aufnehmen. Felder sind homogene Datenstrukturen, d.h. enthalten nur Daten eines Grundtyps (im Gegensatz zu einer Klasse mit heterogenen Datenkomponenten).

Auf die Feldelemente kann direkt durch Angabe des Feldnamens und eines Indexausdrucks zugegriffen werden.

Felder spielen eine wichtige Rolle in vielen Anwendungsbereichen, da über sie u.a. mathematische Strukturen wie Vektoren und Matrizen direkt implementiert werden können.

Typ Feld

Will man eine Variable deklarieren (später genauer), so muss man einen Typnamen angeben gefolgt von einem Variablenbezeichner. Einen Feldtyp kann man angeben, indem man den Basistyp notiert gefolgt von [ ].

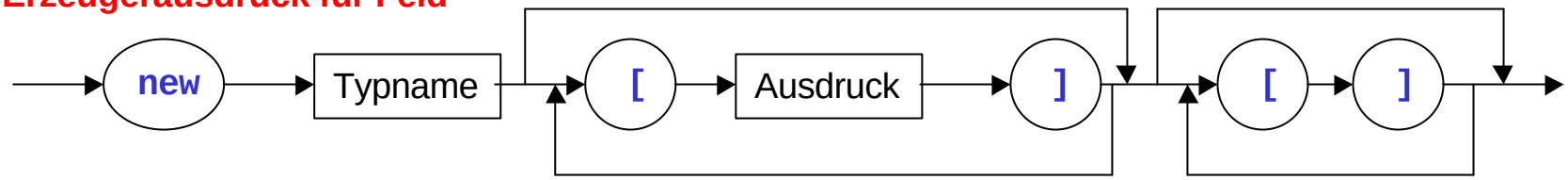
Beispiele:

```
int [ ] feld1;            // Variable feld1 hat Typ Feld von int
double [ ] feld2;         // Variable feld2 hat Typ Feld von double
MeineKlasse [ ] feld3;    // Variable feld3 hat Typ Feld von MeineKlasse
int [ ] [ ] feld4;        // Variable feld4 hat Typ Feld von Feld von int
```

Wichtig:
Feldtypen gehören zu den Referenztypen, d.h. mit der Variablendeklaration wird lediglich eine Referenz auf ein Feldobjekt angelegt, kein Objekt dieses Typs selber. D.h., die Feldelemente existieren damit noch nicht (nächste Folie)!

Erzeugen von Feldern

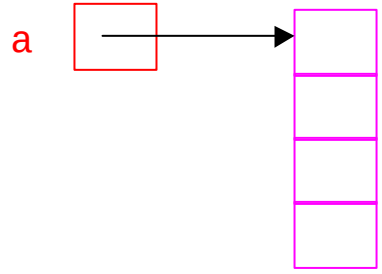
Erzeugerausdruck für Feld



- Typname darf ein Primitivtyp (`int, char` usw.) oder ein Referenztyp sein.
- Die Dimensionierungsausdrücke müssen alle einen ganzzahligen Typen haben (nur `long` ist nicht erlaubt).
- Das Offenlassen von Dimensionsangaben ist nur am Ende der Deklaration möglich. Nach dem ersten [ ]-Paar ohne Dimensionsgröße sind nur noch [ ]-Paare erlaubt.
- Die Größe eines Feldes kann später nicht mehr verändert werden.
- Die Erzeugung kann unabhängig von einer Variablendeklaration erfolgen.

Beispiele:

```
int[] a = new int[4];
int[] b = new int[4][3];
b = new int[12][2];
MeineKlasse[] c = new MeineKlasse[5];
MeineKlasse[][] d = new MeineKlasse[5][];
```



Schritte bei der Erzeugung

Die (interne) Reihenfolge der Schritte bei der Erzeugung eines Feldes durch new ist genau definiert (Grund: reproduzierbarer Ablauf eines Programms auf allen Systemen):

- Bestimmung der Feldgrößenausdrücke von links nach rechts
- Überprüfung der Werte (Exception `NegativeArraySizeException`, wenn ein Wert kleiner 0)
- Speicher für Feld anlegen (Exception `OutOfMemoryException` möglich)
- Anlegen der Feldelemente
  Mehrdimensionale Felder werden als Felder von Feldern realisiert

Beim Erzeugen eines Feldes in der beschriebenen Syntax werden alle Feldelemente automatisch zu ihrem Default-Wert initialisiert:

- `false` für `boolean`
- `'\u0000'` für `char`
- `0` für ganzzahlige Typen
- `0.0` für Fließkommatypen
- `null` für Referenztypen

Mehrdimensionale Felder

Gibt man eine Deklaration der Form

```
int[][] feld = new int [2][3];
```

an, so wird dies intern umgesetzt als:

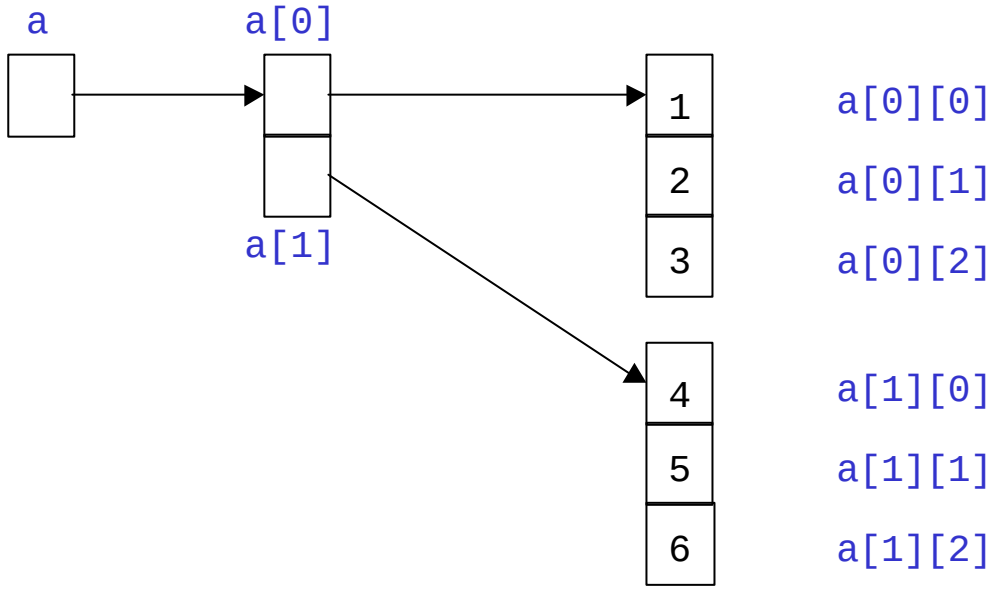
```
feld = new int[2];
for(int i=0; i<2; i=i+1)
    feld[i] = new int[3];
```

Die Werte der Feldelemente der 1. Dimension sind also Referenzen! In anderen Programmiersprachen ist dies anders, dort wird ein zusammenhängender Speicherbereich für alle Feldelemente angelegt.

Mehrdimensionale Felder

Beispiel:

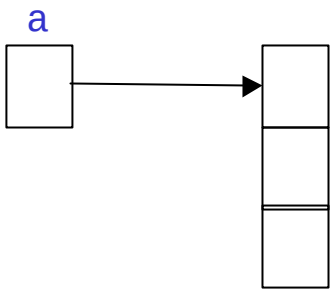
```
int[][] a = new int[2][3];
int i=1, i1, i2;
for(i1=0; i1 < a.length; i1=i1+1)
    for(i2=0; i2 < a[i1].length; i2=i2+1) {
        a[i1][i2] = i;
        i = i + 1;
    }
```

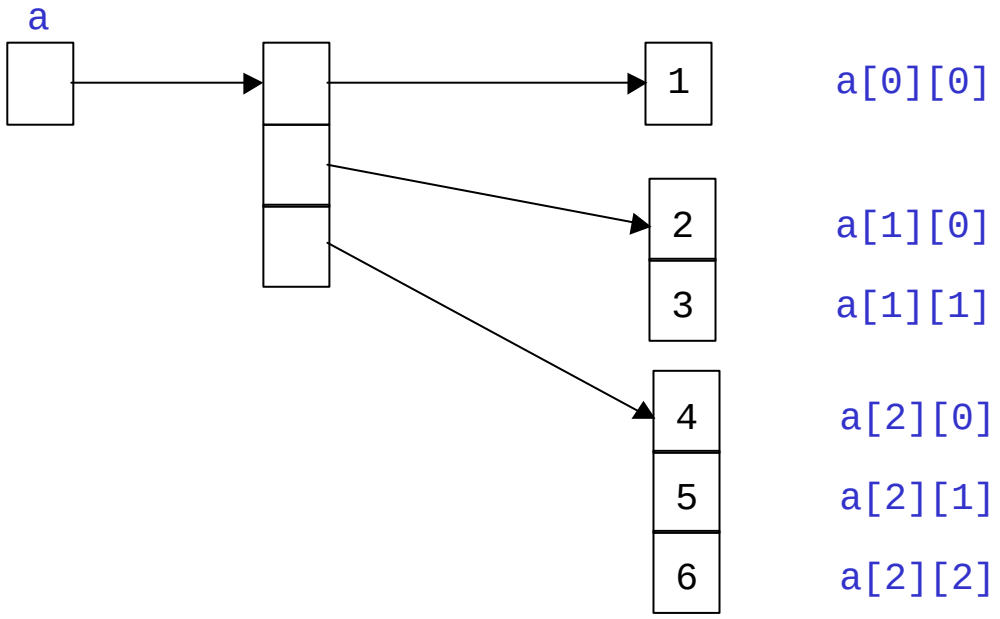


a[1] ist ein Referenzwert, a[1][2] ein int-Wert.

Offenlassen von Dimensionsgrößen

```
int[][] a = new int[3][];
int i=1, i1, i2;

for(i1=0; i1 < a.length; i1=i1+1) {
    a[i1] = new int [i1+1];
    for(i2=0; i2 < a[i1].length; i2=i2+1) {
        a[i1][i2] = i;
        i = i + 1;
    }
```



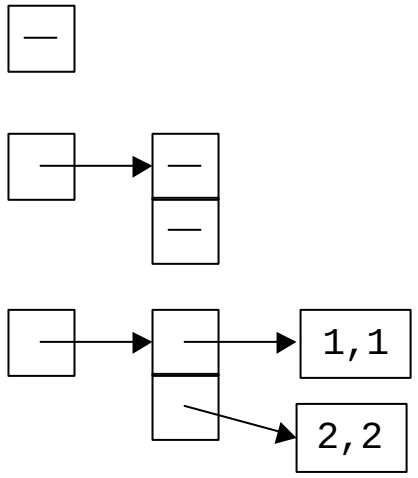


Felder von Referentypen

Bisher wurden in den Beispielen hauptsächlich Felder angegeben, die einen einfachen Typen als Basistyp besitzen. Genauso kann man aber auch ein Feld mit einem Referenztyp als Basistyp angeben.

Beispiel:

```
class Punkt {
    int x, y;
    Punkt() { x = 0; y = 0; }
    Punkt(int xpos, int ypos) { x = xpos; y = ypos; }
}

class Test {
    Punkt[] feld;                    // Feldvariable anlegen
    int i;

    feld = new Punkt[2];             // Feld anlegen


    for(i=0; i<feld.length; i=i+1)
        // Feldelementen Wert zuweisen
        feld[i] = new Punkt(i+1, i+1);
}
```



Zugriff auf Feldelemente

Der Zugriff auf einzelne Feldelemente geschieht über den Variablennamen des Feldes gefolgt von einem Indexausdruck in eckigen Klammern. Bei mehrdimensionalen Feldern werden die Indexausdrücke in eckigen Klammern hintereinander geschrieben.

Oft gemachter Fehler:
Indizes beginnen bei 0 und enden bei Dimensionsgröße-1!
Versucht man auf ein Element außerhalb der Feldgrenzen zuzugreifen, so wird ein `ArrayIndexOutOfBounds` Exception ausgelöst.

Beispiele:

```
int[] a = new int[4];
int[][] b = new int[4][3];
MeineKlasse[] c = new MeineKlasse[5];

a[0] = 5;
b[3][2] = 3;              // letzte Element des Feldes
c[0].methode();           // Methode aufrufen des 0-ten Feldelementes
a[4] = 4;                 /* Fehler: ArrayIndexOutOfBounds Exception wird
                             zur Laufzeit des Programm ausgelöst */
```

Länge eines Feldes

Zu jedem Feld lässt sich die Länge erfragen, die man als normalen Wert verwenden kann, der zur Laufzeit des Programms ermittelt wird.

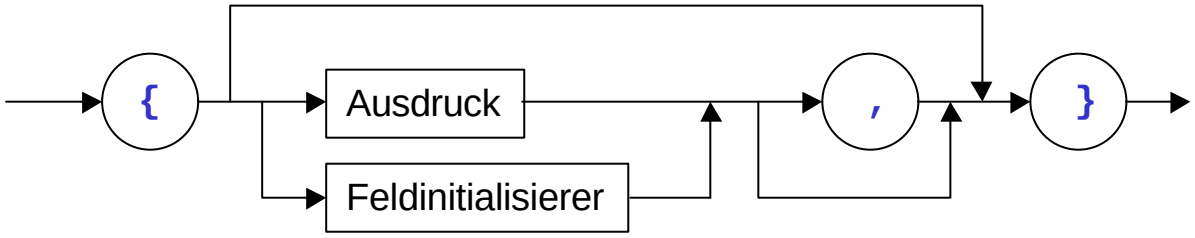
Beispiele:

```
int[] a = new int[4];
int[][] b = new int[4][3];

if(a.length < 10000)
    System.out.println("Mit kleinen Feldern gebe ich mich nicht ab!");

// Dies geht nur, wenn das Feld ein "Rechteck" ist
int groesse_b = b.length * b[0].length * 4;

// häufigste Art der Nutzung (wichtig: nicht Abfrage auf <= in Test)
for(int i = 0; i < a.length; i = i + 1)
    System.out.println(a[i]);
```

Initialisierung eines Feldes

Beim Anlegen eines Feldes kann man gleichzeitig Werte angeben, mit denen die Feldelemente anfangs belegt sein sollen. Ebenfalls kann man durch Angabe von n Initialisierungswerten eine implizite Dimensionierung vornehmen. In diesen Fällen muss die explizite Erzeugung mittels new entfallen.

Feldinitialisierer



Beispiele:

```
int[] a = {0,1,2,3};
int[][] b = { {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11} };
int[] c = {};                          // c hat anschließend die Länge 0
int[] c = {0,1,2,3,4,5};               // d hat anschließend die Länge 6
```

Strings

Wiederholung String-Literale:
String- oder Zeichenketten-Literale sind 0 oder mehr Zeichen in Anführungszeichen eingeschlossen. Längere Strings können in mehrere Teilstrings aufgebrochen werden, die durch + verkettet werden.

Beispiel:

```
"Hallo"
```

Der Typ solch eines Literals ist String, demzufolge ist jedes Zeichenketten-Literal eine Referenz auf eine Instanz der Klasse String!

Es gibt in Java (anders als in anderen Programmiersprachen) keinen Basistyp String (wie `char, int, float`) sondern eine Klasse String (`java.lang.String`; siehe API-Dokumentation).

Deshalb zweite Möglichkeit der Erzeugung eines Strings über den Konstruktor:

```
String s = new String("Hallo");
```

String verknüpfen

Man kann Strings mit + zu einem (neuen!) String verknüpfen.

Beispiel:

```
System.out.println("String 1 " + "String 2 " + "gehören zusammen");
```

Ausgabe:

```
String 1 String 2 gehören zusammen
```

Dies ist nicht nur mit zwei Strings möglich, sondern auch mit einem String und einem beliebigen anderen Objekt (inkl. Wert eines Primitivtyps). Später mehr.

Beispiel:

```
System.out.println("String " + 4711);
System.out.println("String " + 47 + 11);
System.out.println("String " + (47 + 11));
```

Ausgabe:

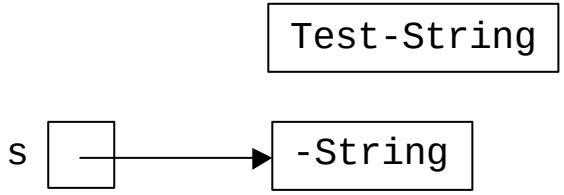
```
String 4711
String 4711
String 58
```

Strings

Ebenfalls anders als in anderen Programmiersprachen sind Strings nach der Erzeugung nicht mehr veränderbar. Methoden, die Strings modifizieren wollen, können ein neues String-Objekt als Resultat der Methode zurückgeben.

Beispiel:

```
String meineMethode(String str) {
    return str.substring(4);    // Zeichen 4 bis Ende
}

public static void main(String[] args) {
    String s = meineMethode("Test-String");
}
```

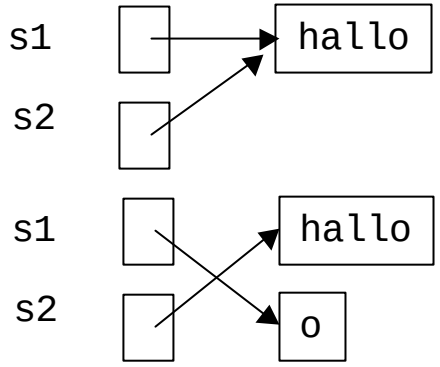


Strings

Nachteil:

Haben zwei String-Variablen den gleichen String als Wert (d.h. zeigen auf das gleiche String-Objekt), so zeigen anschließend beide auf unterschiedliche Strings/Objekte.

```
String s1 = "hallo";
String s2 = s1;
s1 = meineMethode(s1);
```



Alternative zu String:

Java kennt die Klasse `java.lang.StringBuffer`, die die Modifikation des Inhalts erlaubt. Ebenfalls wird ein StringBuffer-Objekt automatisch größer oder kleiner.

Einige Methoden der Klasse String

Klassenmethode (auch ohne String-Objekt aufrufbar):

```
public static String valueOf(int i);
```

Liefert den `int`-Wert als String. Die Methode gibt es auch für andere Typen (`long, float` usw.)

Instanzenmethoden (nur über String-Objekt aufrufbar):

```
public char charAt(int index);
```

Liefert das Zeichen an der Position index (bei 0 beginnend).

```
public int compareTo(String anotherString);
```

Vergleicht zwei Strings und liefert 0, wenn die Strings gleich sind.

```
public int compareToIgnoreCase(String anotherString);
```

Vergleicht zwei Strings (Groß- und Klejnschreibung wird ignoriert) und liefert 0, wenn die Strings gleich sind.

```
public String concat(String str);
```

Hängt an den String des Objektes den String str an.

```
public int length();
```

Liefert die Länge des Strings.

```
public String substring(int beginIndex, int endIndex);
```

Liefert Kopie des Teilstrings, der an der Position beginIndex beginnt und an der Position endIndex endet (Indizes beginnen bei 0).

Komplette Referenz in der API-Dokumentation!

Beispiel

```
public static void main(String[] args) {
    // impliziter Konstruktor
    String s1 = "Hallo";

    // Klassenmethode
    String s2 = String.valueOf(4711);              // liefert "4711"

    // Instanzenmethoden
    int l = s1.length();                           // liefert 5
    String s3 = s1.substring(1,2);                 // liefert "al"
    int vgl1 = s1.compareTo("hallo");              // liefert Wert != 0
    int vgl2 = s1.compareToIgnoreCase("hallo");    // liefert 0
    String s4 = s1.concat(s2);                     // liefert "Hallo4711"
}
```

Einige Instanzenmethoden der Klasse StringBuffer

```
public StringBuffer append(String str);
```

Hängt an den StringBuffer den String str an (gibt es auch für andere Typen als String).

```
public StringBuffer delete(int beginIndex, int endIndex);
```

Entfernt den Substring beginnend bei beginIndex und endend bei endIndex.

```
public StringBuffer insert(int offset, int i);
```

Fügt die String-Darstellung von i ab der Position offset in den StringBuffer ein (gibt es auch für andere Typen als int).

```
public StringBuffer replace(int start, int end, String str);
```

Ersetzt das Stück start-end durch str.

Komplette Referenz in der API-Dokumentation!

Beispiel

```
public static void main(String[] args) {
    // impliziter Konstruktor
    StringBuffer sb1 = new StringBuffer("Hallo");

    sb1.append(" Leute!");                  // ergibt "Hallo Leute!"
    sb1.delete(5,5+7);                      // ergibt "Hallo"
}
```

Kommandozeilenargumente

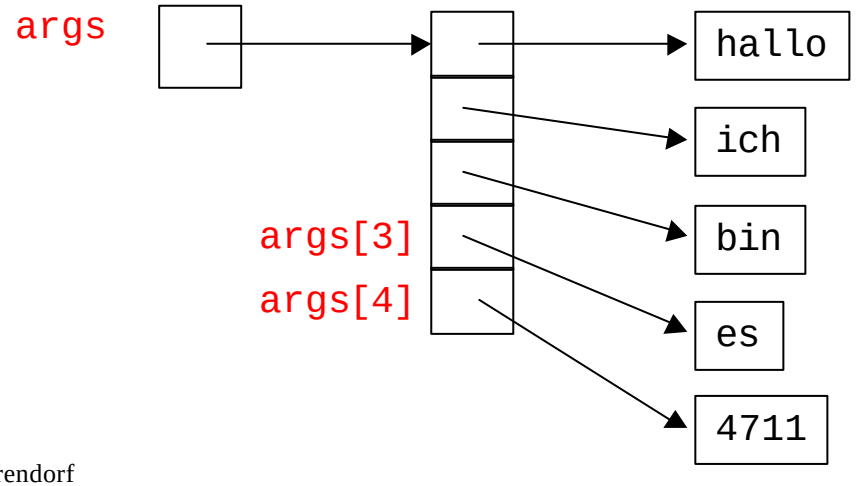
Die Methode main ist wie folgt definiert:

```
public static void main(String[] args);
```

d.h. das Hauptprogramm bekommt als einziges Argument ein Feld von Strings übergeben. In diesem Feld sind die einzelnen Strings der Kommandozeile gespeichert, mit denen der Java-Interpreter aufgerufen wurde.

Beispiel:

```
java MeineKlasse hallo ich     bin es  4711
```



Der String "4711",
nicht die Zahl 4711!

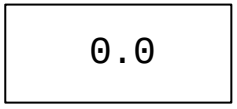
Beispiel

```
// alle Kommandozeilenargumente ausgeben

class Test {

    public static void main(String[] args) {

        int i;

        for(i=0; i<args.length; i=i+1)
            System.out.println(args[i]);
    }

}
```

Variablen

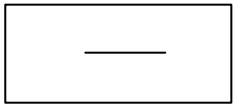
Variablen bezeichnen einen Speicherplatz. Jede Variable hat einen Typ, der die möglichen Werte einschränkt. Der Speicherplatz kann entweder einen Wert dieses Typs aufnehmen (einfache Typen) oder eine Referenz bzw. null (Referenztypen).

```
double a;
```



double-Wert

```
double[] b;
```



Zeigerwert

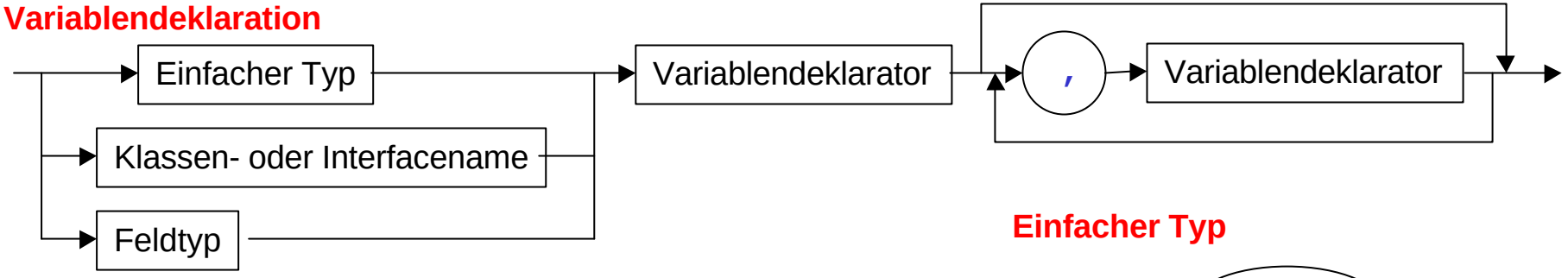
Je nachdem, wo eine Variable verwendet wird, kann sie unterschiedliche Bedeutung haben:

```
a = a + 1;
```

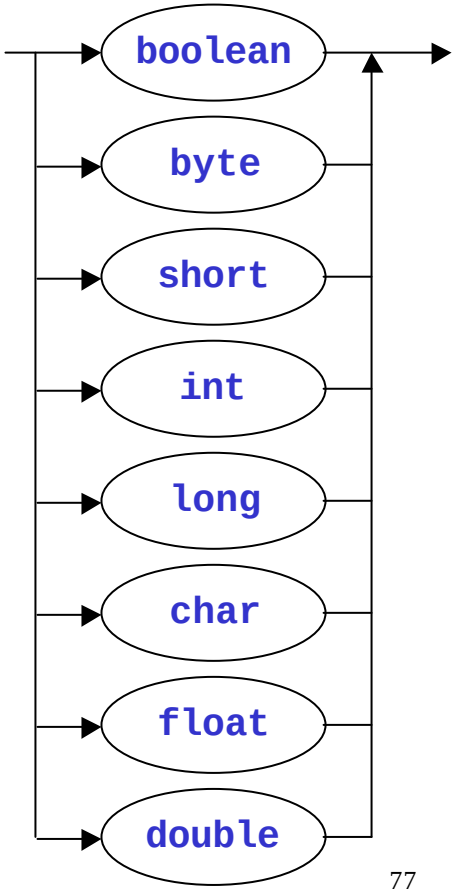
Linke Seite: Der Speicherplatz ist gemeint, d.h. speichere den Wert (a+1) dort ab.
Rechte Seite: Nimm den Wert, der an dem Speicherplatz zu finden ist.

Deklaration einer Variablen

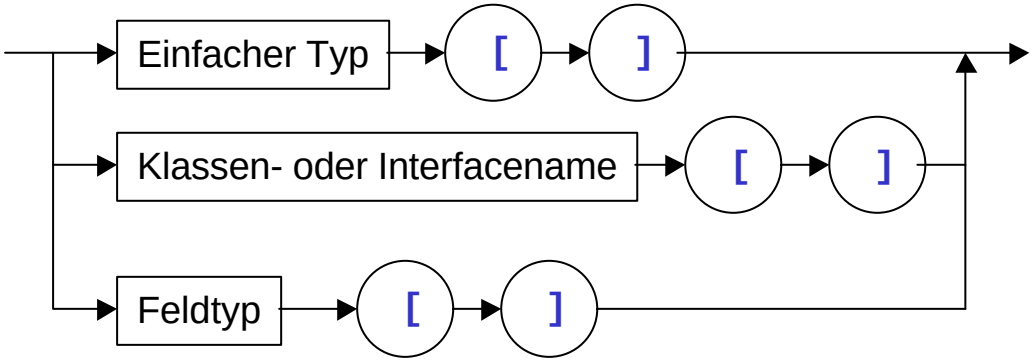
Variablendeklaration



Einfacher Typ

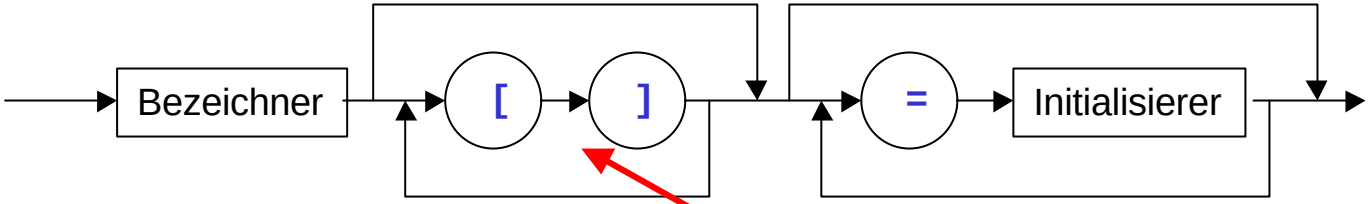


Feldtyp



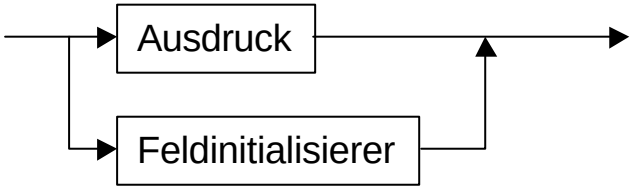
Deklaration einer Variablen

Variablendeklarator

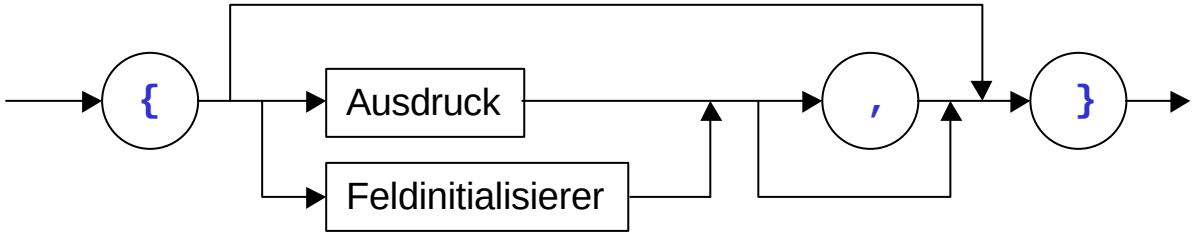


Nur aus Kompatibilität zu C/C++.
Sollte nicht verwendet werden.

Initialisierer



Feldinitialisierer



(Wiederholung, wurde früher schon angegeben)

Beispiele

```
int i1;
int i2,j2;                          // mehrere Variablen einfachen Typs
int i3=5, j3=7, k3;                 // mit Initialisierer (Ausdruck)
MeineKlasse kl4 = new MeineKlasse();// Klassentyp mit Initialisierer (Ausdruck)
float [] f5, g5 = {2,3};            // Feldtyp mit Feldinitialisierer
int[][] i6 = {{1,2,3}, {4,5,6}};    // mehrdimensionaler Feldtyp mit Init.
```

Arten von Variablen

- Eine Klassenvariable ist ein Datenfeld in einer Klassendeklaration, das das Schlüsselwort `static` enthält, oder ein Datenfeld einer Schnittstellendeklaration.
- Eine Instanzvariable ist ein Datenfeld in einer Klassendeklaration ohne das Schlüsselwort `static`.
- Lokale Variable werden in einem Block (später) deklariert.
- Feldelemente sind unbenannte Variable, die erzeugt und mit einem Anfangswert belegt werden, wenn das Feld erzeugt wird.
- Methodenparameter werden bei jedem Aufruf einer Methode als Variable angelegt und mit dem aktuellen Argumentwert initialisiert.
- Konstruktorparameter sind ähnlich Methodenparametern
- Parameter einer Ausnahmeroutine (später)

Beispiel

```
class Point {

    static int NumPoints;          // NumPoints ist eine Klassenvariable

    int x,y;                       // x und y sind Instanzvariablen

    int[] w = new int[10];         // w[0] ist ein Feldelement

    Point(int x, int y) {          // x und y sind Konstruktorparameter
        this.x = x;
        this.y = y;
    }

    int setX(int x) {              // x ist ein Methodenparameter

        int oldx = this.x;         // oldx ist eine lokale Variable
        this.x = x;
        return oldx;
    }
}
```

Die Art einer Variablen hat u.a. Einfluss auf die Lebensdauer einer Variablen.

Anfangswerte von Variablen

- Klassen-, Instanz- und Feldvariablen werden, falls nicht explizit durch einen Initialisierungsausdruck überschrieben, mit einem Defaultwert belegt (`false, '\u0000', 0, 0.0` bzw. `null`)
- Methoden- und Konstruktorparameter werden mit dem aktuellen Argument belegt
- Parameter einer Ausnahmeroutine mit dem ausgelösten Objekt (später)
- Jeder lokalen Variablen muss vor ihrer Nutzung (d.h. der Wert wird in einem Ausdruck genutzt) explizit ein Wert zugewiesen werden! Dies kann entweder mit einem Initialisierungsausdruck geschehen oder durch eine Zuweisung vor der Nutzung.

Beispiel

```
class Test {
    static int i;          // mit 0 initialisiert
    float f;               // bei Instantiziierung mit 0.0f initialisiert
    boolean b;             // bei Instantiziierung mit false initialisiert
    static char c;         // mit '\u0000' initialisiert
    Test o;                // bei Instantiziierung mit null initialisiert

    Test(int i) {          // bei Aufruf mit aktuellem Argument initialisiert
    }

    int methode(int j) {   // bei Aufruf mit aktuellem Argument initialisiert
        int i = 5;
        int j;

        i = j;             // Fehler: Nicht-initialisierte Variable wird genutzt
                           // Hier meckert der Compiler!
    }

}
```