



# Programmietechnik 1

## Teil 2: Java Daten

### Literale / Variablen / Typen

# Java Literale: Ganze Zahlen

Schreibweisen für ganze Zahlen (*Integers*):

- **dezimal** — wie gewohnt, erste Ziffer ungleich 0

123      456      7890

- **oktal** — erste Ziffer 0

0173      0710      017322

- **hexadezimal** — am Anfang 0x

0x7b      0x1c8      0x1ed2

- **binär** — am Anfang 0b

0b1111011      0b111001000      0b1111011010010



*Nicht vergessen:  
der Rechner kennt  
zur Laufzeit nur Binärzahlen,  
der Compiler muss  
deshalb alle Schreibweisen  
in Binärzahlen wandeln*

Man kann Zahlen mit Unterstrichen gliedern:      0b0010\_0011

Man kann Zahlen mit Vorzeichen versehen:      -123      +456



Man kann mit Zahlen arithmetische Ausdrücke bilden:      (1 + 2) \* 3 - 4 / 5

# Beispiel-Programm Zahlen-Literale



- Quellcode *IntLiteral.java* :

```
public final class IntLiteral {  
    private IntLiteral() { }  
  
    public static void main(String[] args) {  
        System.out.println(12);  
        System.out.println(012);  
        System.out.println(0x12);  
  
        System.out.printf("%x%n", 12);  
        System.out.printf("%d%n", 012);  
        System.out.printf("%o%n", 0x12);  
    }  
}
```

*println macht unformatierte Ausgabe mit plattformspezifischem Zeilentrenner*

*printf macht formatierte Ausgabe*

*System.out ist der Standardausgabekanal*

*%x ist hexadezimaless Format*  
*%d ist dezimales Format*  
*%o ist oktales Format*  
*%n ist plattformspezifischer Zeilentrenner*

Konsolenausgabe  
des Programms:

```
12  
10  
18  
c  
10  
22
```

*standardmäßig  
dezimale  
Ausgabe*

# Java Literale: Gleitkomma-Zahlen

---

Schreibweisen für Gleitkomma-Zahlen (*Floating Point Numbers*):

- **dezimal**

1.      .23      0.456  
78.9    .789e2    789e-1

.789e2 steht für  $0,789_{10} \cdot 10^2$

zur Unterscheidung von ganzzahligen Literalen muss ein Punkt und/oder Exponent vorkommen, eine 0 vor dem Punkt darf fehlen

- **hexadezimal**

0x1p0    0x.1p4    0x10p-4

0x.1p4 steht für  $0,0001_2 \cdot 2^4$

zur Unterscheidung von ganzzahligen Literalen muss ein Exponent p vorkommen, der Punkt und eine 0 vor dem Punkt dürfen fehlen

Achtung: nur die Mantisse ist hexadezimal, der Exponent hinter dem p ist dezimal

Nicht vergessen: Gleitkomma-Zahlen sind ungenau!

$(1e-30 + 1e30) - 1e30$	ist	0
$1e-30 + (1e30 - 1e30)$	ist	$10^{-30}$

*Auch bei Gleitkomma-Literalen wandelt der Compiler alle Schreibweisen in ein Binärformat, in diesem Fall IEEE 754.*

# Beispiel-Programm Gleitkomma-Literale



- Quellcode *DoubleLiteral.java* :

```
public final class DoubleLiteral {  
    private DoubleLiteral() { }  
  
    public static void main(String[] args) {  
        System.out.println( (1e-30 + 1e30) - 1e30 );  
        System.out.println( 1e-30 + (1e30 - 1e30) );  
        System.out.println( 12.3456789 );  
        System.out.println( 1234567.89 );  
        System.out.printf( "%f%n", 12.3456789 );  
        System.out.printf( "%f%n", 1234567.89 );  
        System.out.printf( "%e%n", 12.3456789 );  
        System.out.printf( "%e%n", 1234567.89 );  
    }  
}
```

*%f ist Festkommaformat  
%e ist Gleitkommaformat  
%g entspricht je nach Zahl %f oder %e*

Konsolenausgabe  
des Programms:

```
0.0  
1.0E-30  
12.3456789  
1234567.89  
12,345679  
1234567,890000  
1,234568e+01  
1,234568e+06
```

*formatierte Ausgabe  
standardmäßig mit  
6 Nachkommastellen*



Schreibweisen für Einzelzeichen (*Characters*):

- in **Einfachhochkommas**

`'a'` `'A'` `'1'` `'.'` `' '`      Buchstaben, Ziffern, Satzzeichen, Leerstelle, ...

Zeichen können auch numerisch als UTF-16 Code Units geschrieben werden:

`'\ooo'`      Codenummer oktal (1 bis 3 Oktalziffern `o`, maximal `377`)

`'\uxxxx'`      Codenummer hexadezimal (4 Hex-Ziffern `x`)      *Vorsicht, Fehlerquelle (siehe 2-8)!*

- Steuerzeichen mit Standardnamen

`'\b'`      Rückschritt (*Backspace*)

`'\f'`      Seitenvorschub (*Formfeed*)

`'\n'`      Zeilenende (*Newline*)

`'\r'`      Wagenrücklauf (*Carriage-Return*)

`'\t'`      Horizontal-Tabulator

- Metazeichen mit Backslash

 `'\''`      das Einfach-Hochkomma

`'\"'`      das Doppel-Hochkomma

`'\\'`      der Backslash

*Der Compiler wandelt alle Schreibweisen in eine binäre UTF-16 Code Unit (hexadezimale Schreibweise bereits beim Einlesen des Quellcodes).*

# Java Literale: Zeichenketten

---

Schreibweise für Zeichenketten (Strings):

- in **Doppelhochkommas**

**"Hallo"**

**" "** *leere Zeichenkette*

- zwischen den Doppelhochkommas sind alle Schreibweisen für Einzelzeichen erlaubt, wobei die Einfachhochkommas entfallen

**"Hallo\n"** *Hallo mit Linux-Zeilenende (Windows verwendet \r\n)*

**"Hallo\12"** *Hallo mit Zeilenende als oktale Codenummer*

**"\u0048\u0061\u006c\u006c\u006f\n"** *Hallo hexadezimal mit Zeilenende*

- mit + verknüpfte Zeichenketten fasst der Compiler zusammen

**"Hal" + "lo"** *das gleiche wie "Hallo"*

# Beispiel-Programm Zeichen-Literale



- Quellcode *CharLiteral.java* :

```
public final class CharLiteral {  
    private CharLiteral() { }  
    public static void main(String[] args) {  
        System.out.print('H');  
        System.out.print('a');  
        System.out.print('l');  
        System.out.print('l');  
        System.out.print('o');  
        System.out.print('\n');  
        System.out.print("Hallo\12");  
        System.out.println("Hal" + "lo");  
        System.out.printf("%s\n", "Hallo");  
        System.out.printf("%c%c%c%c%c\n", 'H', 'a', 'l', 'l', 'o');  
    }  
}
```

*print macht unformatierte Ausgabe ohne Zeilen-trenner*

*\12 ist die oktale Codenummer von \n*

*%c ist Einzelzeichenformat  
%s ist Zeichenkettenformat*

Konsolenausgabe  
des Programms:

```
Hallo  
Hallo  
Hallo  
Hallo  
Hallo
```



# Java Literale: Empfehlungen

---

## Zahlen-Literale:

- echte Zahlen immer dezimal schreiben
- Bitmuster in der Regel hexadezimal schreiben, bei Bedarf auch binär oder oktal

*Zahlen-Literale, die keine Trivial-Werte sind, nur zum Initialisieren von Konstanten verwenden ( trivial sind z.B. 0 oder 1 , Konstanten und Initialisierung werden später noch erklärt ).*

## Zeichen-Literale:

- oktale und hexadezimale Codenummern nur, wenn Zeichen auf der Tastatur fehlen und auch kein Zeichenname ' \...' definiert ist

### *Achtung:*

*Der Compiler ersetzt hexadezimale Codenummern bereits beim Einlesen des Quellcodes durch Codenummern. Dadurch kann z.B. das Zeilenende in einem Zeichen-Literal nicht hexadezimal geschrieben werden.*

# Java Variablen: Eigenschaften

---

Variablen dienen dazu, Werte im Hauptspeicher abzulegen und anzusprechen.

- Eine Variable hat einen **Namen**:

Besteht aus Buchstaben und Ziffern (und weiteren Symbolen, sofern sie in Java nicht schon eine Sonderbedeutung haben).

Darf zur Unterscheidung von Zahlen-Literalen nicht mit einer Ziffer beginnen und darf kein von Java reserviertes Schlüsselwort sein.

*Zusätzlich sollten übliche Namenskonventionen beachtet werden, die festlegen, wie "man" Variablennamen wählt (z.B. immer mit Kleinbuchstabe beginnend).*

- Eine Variable hat einen **Typ**:

Legt fest, wie viel Platz die Variable im Hauptspeicher belegt.

Legt fest, welche Art von Werten die Variable aufnehmen kann (z.B. nur ganze Zahlen).

Legt fest, welche Operationen erlaubt sind (z.B. Addition usw.).

*Java gibt einige Grundtypen fest vor und erlaubt zusätzliche benutzerdefinierte Typen.*

- Eine Variable hat einen **Wert**:

Steht in binärer Zahlendarstellung im Hauptspeicher.

# Java Variablen: Syntax

---

- Variablen-Definition:

Erst nach ihrer Definition ist eine Variable benutzbar.

**Typ Name;**                      *Semikolon nicht vergessen!*

- Wert:

Eine Initialisierung gibt einer Variablen gleich bei der Definition auch einen Wert.

**Typ Name = Wert;**

Eine Zuweisung gibt einer bereits definierten Variablen einen (neuen) Wert.

**Name = Wert;**

Bei einer Konstanten ist nur genau eine Initialisierung oder Zuweisung erlaubt.

**final Typ Name = Wert;**

*final kennzeichnet eine Variable als Konstante (sie hat einen "endgültigen" Wert)*

# Java Datentypen: Übersicht

## Werttypen (*primitive Datentypen*)

- Logischer Typ:  
**boolean**
- Zahltypen
  - > Zeichentyp:  
**char**
  - > Ganzzahlige Typen:  
**byte**, **short**, **int**, **long**
  - > Gleitkommatypen:  
**float**, **double**

## Referenztypen

*die ausgegrauten  
Referenztypen werden  
in Teil 4 und 5 behandelt*

- Felder: **Typ[ ]**
- Klassen
  - > Fundamentalklassen:  
Zeichenketten **String**, ...  
Wrapperklassen **Boolean**, ...  
Wurzelklassen **Object**, **Throwable**  
Typinformation **Class<T>**, ...  
...
  - > Anwendungsklassen:  
**enum** *Name*  
**class** *Name*  
**interface** *Name*

# Java Werttypen: boolean

- **Variablen-Definition**: `boolean aBool = true;`
- **Wert**: Entweder `true` oder `false` (Standardwert ist `false`)
- **Platzbedarf**: mindestens 1 Byte  
*obwohl 1 Bit eigentlich reichen würde*

Ist Ergebnistyp der Vergleichsoperatoren `<`, `<=`, `>`, `>=`, `==`, `!=`

`1 < 2` liefert als Ergebnis `true`

Ist Operandentyp der logischen Operatoren `!`, `&&`, `||`, `&`, `|`, `^`

`aBool || !aBool` liefert als Ergebnis `true`

Ist Bedingungstyp bei Verzweigungen und Schleifen

```
if (aBool) {  
    System.out.println("aBool ist true");  
} else {  
    System.out.println("aBool ist false");  
}
```

# Beispiel-Programm boolean-Variable



- Quellcode:

```
public final class BooleanVar {  
    private BooleanVar() { }  
    public static void main(String[] args) {  
        boolean aBool = true; // oder false  
        System.out.printf("%b%n", 1 < 2);  
        System.out.printf("%b%n", aBool);  
        System.out.printf("%b%n", !aBool);  
        System.out.println(aBool && !aBool);  
        System.out.println(aBool || !aBool);  
        if (aBool) {  
            System.out.println("aBool ist true");  
        } else {  
            System.out.println("aBool ist false");  
        }  
    }  
}
```

Konsolenausgabe  
des Programms:

```
true  
true  
false  
false  
true  
aBool ist true
```

# Java Werttypen: char

---

- Variablen-Definition: `char aChar = 'c';`
- Wert: ein Einzelzeichen  
(als Codenummer der Unicode Basic Multilingual Plane, Standardwert ist 0)
- Platzbedarf: 2 Byte

Wird als Variablentyp eher selten verwendet, weil man oft statt Einzelzeichen lieber Zeichenfolgen speichern möchte (*dafür gibt es `char[]`, `String`, ...*)

## Achtung:


Der Unicode ist seit Version 2.0 ein 21-Bit-Code.

Eine Variable vom Typ `char` kann deshalb nicht alle gültigen Codenummern (*bei Unicode "Code Points" genannt*) aufnehmen!

Für Codenummern größer als `\uffff` müssen Variablen vom Typ `int` mit 4 Byte Platzbedarf verwendet werden.

# Java Werttypen: byte, short, int, long



- Variablen-Definition:  
`byte aByte = 123;`  
`short aShort = 123;`  
`int anInt = 123;`  
`long aLong = 123L; // 123 mit L hat Typ long`
- Wert: Ganze Zahlen mit Vorzeichen  
(negative Zahlen als Zweierkomplement, Standardwert ist 0)
- Platzbedarf: 1 (byte), 2 (short), 4 (int) oder 8 (long) Byte 

`int` und `long` sind mögliche Operanden- und Ergebnistypen  
der arithmetischen Operatoren `+`, `-`, `*`, `/`, `%`, ...

*Rest der ganzzahligen Division*

`int` verwenden, wenn kein zwingender Grund für `byte`, `short` oder `long` spricht.  
*`byte` und `short` werden in arithmetischen Ausdrücken automatisch nach `int` konvertiert.*  
*Bei Mischung von `int` und `long`, wird `int` automatisch nach `long` konvertiert.*



# Java Werttypen: float und double



- **Variablen-Definition:** `float aFloat = 3.14f; // 3.14 mit f hat Typ float`  
`double aDouble = 3.14; // oder 3.14d`
- **Wert:** Gleitkommazahlen  
(einfach bzw. doppelt genaues Format IEEE 754, Standardwert ist 0.0).
- **Platzbedarf:** 4 (float) oder 8 (double) Byte

float und double sind mögliche Operanden- und Ergebnistypen der arithmetischen Operatoren `+`, `-`, `*`, `/`, `%`, ...

double verwenden, wenn kein zwingender Grund für float spricht.

*Bei Mischung von float und double, wird float automatisch nach double konvertiert.*

*Bei Mischung von ganzzahligen und Gleitkomma-Operanden werden die ganzzahligen Operanden in den betreffenden Gleitkommatyp konvertiert.*

# Java Werttypen: Empfehlungen

---

- Vorzugsweise die Werttypen

**boolean**, **char**, **int**, **double**

verwenden.

*Die anderen Grundtypen nur verwenden, wenn es einen zwingenden Grund gibt.*

- Den Zusatz **final** verwenden, wenn eine Variable ihren Wert nach der Initialisierung nicht mehr ändern soll.

**final** double pi = 3.141592653589793;

- Achtung:

Die Mischung unterschiedlicher Zahltypen kann überraschende Ergebnisse liefern.

double x = 8.5 + 1/2; // setzt x auf 8.5 statt 9 ! Warum?

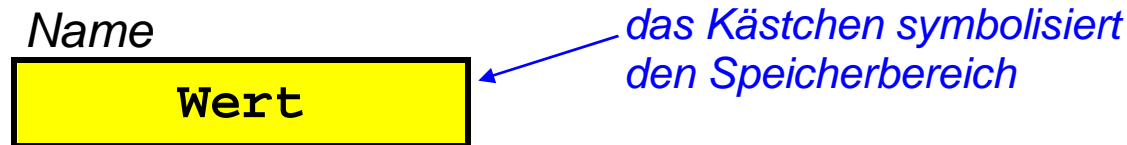
*double*      *int*      *int*

# Java Werttypen versus Referenztypen

---

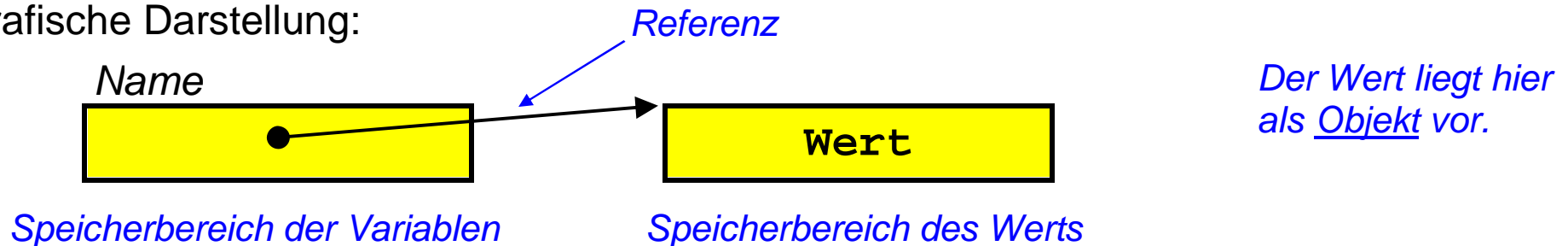
Variablen mit **Werttyp** speichern unmittelbar einen Wert des angegebenen Typs.

- beim Variablenvergleich werden die Werte verglichen
- bei einer Zuweisung wird der Wert kopiert
- Grafische Darstellung:



Variablen mit **Referenztyp** speichern, wo ein Wert des angegebenen Typs steht (*sie enthalten im Prinzip die Hauptspeicheradresse des Werts*).

- beim Variablenvergleich werden die Referenzen verglichen (*d.h. die Adressen der Werte*)
- bei einer Zuweisung wird die Referenz kopiert
- Grafische Darstellung:



# Java Referenztypen: String (1)

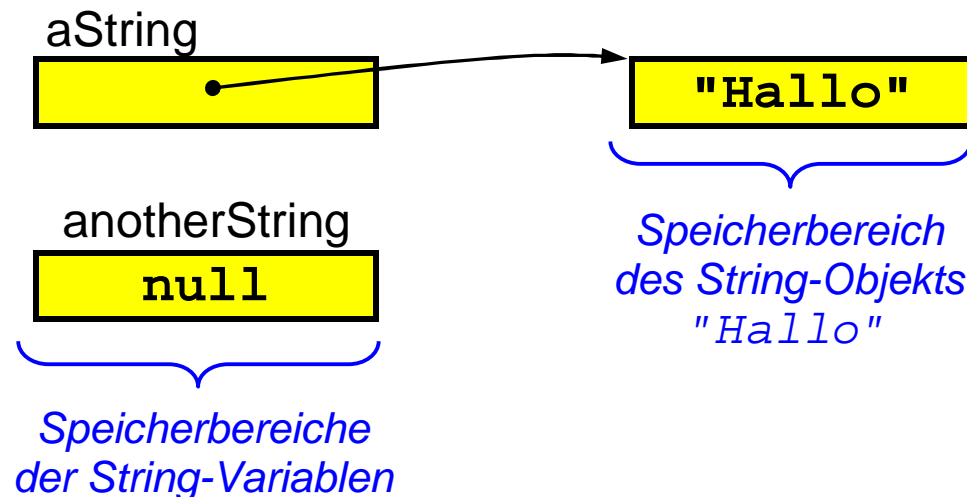


Ein String ist eine Folge von Einzelzeichen in Unicode-Codierungsform UTF-16.

- Variablen-Definition: `String aString = "Hallo";`  
`String anotherString = null;`
- Wert: Referenz auf ein String-Objekt (Standardwert ist `null` )

*Der Wert `null` bedeutet, dass kein String-Objekt referenziert wird (ungültige Referenz)*


Grafische Darstellung:



*String-Objekte sind konstant:  
einmal angelegt, kann die  
gespeicherte Zeichenkette  
nicht mehr geändert werden*


## Java Referenztypen: String (2)

---

- ein String hat eine Länge: `aString.length ( )`  
*liefert die Anzahl der UTF-16 Code-Units als **int**-Wert*
- Einzelzeichenzugriff mit `.charAt`: `aString.charAt ( index )`  
*liefert **char**-Wert (Index muss zwischen 0 und `aString.length ( ) - 1` liegen)*
- Test auf Wertgleichheit mit `.equals`: `aString.equals ( anotherString )`  
*liefert bei Gleichheit **true**, sonst **false***  
*(nicht verwechseln mit Test auf Objektidentität: `aString == anotherString`)* 
- Allgemeiner Wertvergleich mit `.compareTo`: `aString.compareTo ( anotherString )`  
*liefert **int**-Wert kleiner / gleich / größer 0 bei `aString` kleiner / gleich / größer `anotherString`*
- String-Konkatenation mit `+`: `aString + anotherString`  
*liefert ein neues String-Objekt, das als Wert die aneinander gehängten Ursprungsstrings hat*

# Java Referenztypen: String (3)

---

- String-Kopie mit `new`: `new String(aString)`  
*liefert ein neues String-Objekt, das den gleichen Wert wie der Ursprungsstring hat*
- gleiche Referenz für gleiche Werte mit `.intern()`: `aString.intern()`  
*liefert Referenz auf das erste wertgleiche String-Objekt, für das `.intern` aufgerufen wurde  
(`s1.intern()` == `s2.intern()` ist **true** genau dann, wenn `s1.equals(s2)` **true** ist)*
- aus jedem String-Literal macht der Compiler ein String-Objekt: `"Literal"`  
*für wertgleiche Literale ist es immer dasselbe Objekt (mittels `.intern`-Mechanismus)*
- alles lässt sich in Strings wandeln mit `.valueOf()`: `String.valueOf(irgendwas)`  
*liefert ein String-Objekt, das als Wert die String-Darstellung von irgendwas hat  
(z.B. bei einer ganzen Zahl die Ziffernfolge)* 
- formatierte Umwandlung in Strings mit `.format()`: `String.format("Format", Wert ...)`  
*liefert ein String-Objekt, das die formatierte String-Darstellung der genannten Werte enthält  
(Formatangabe wie bei `System.out.printf`)*

# Beispiel-Programm String-Variablen



- Quellcode:

```
public final class StringVar {  
    private StringVar() { }  
    public static void main(String[] args) {  
        String a = "Hallo";  
        String b = new String("Hallo");  
        String c = a + b;  
        System.out.println(a == "Hallo");  
        System.out.println(b == "Hallo");  
        System.out.println(a == b);  
        System.out.println(a == b.intern());  
        System.out.println(a.equals(b));  
        System.out.println(a.compareTo(b) == 0);  
        System.out.println(a.compareTo(c) < 0);  
    }  
}
```

Konsolenausgabe  
des Programms:

```
true  
false  
false  
true  
true  
true  
true
```

# Java Referenztypen: Felder (Arrays) (1)



- **Variablen-Definition:**



*Feldlänge ergibt sich aus Anzahl der angegebenen Werte*

```
Typ[] anArray = new Typ[] { Wert_1, Wert_2, ..., Wert_N };
```

```
Typ[] anotherArray = new Typ[Feldlänge];
```

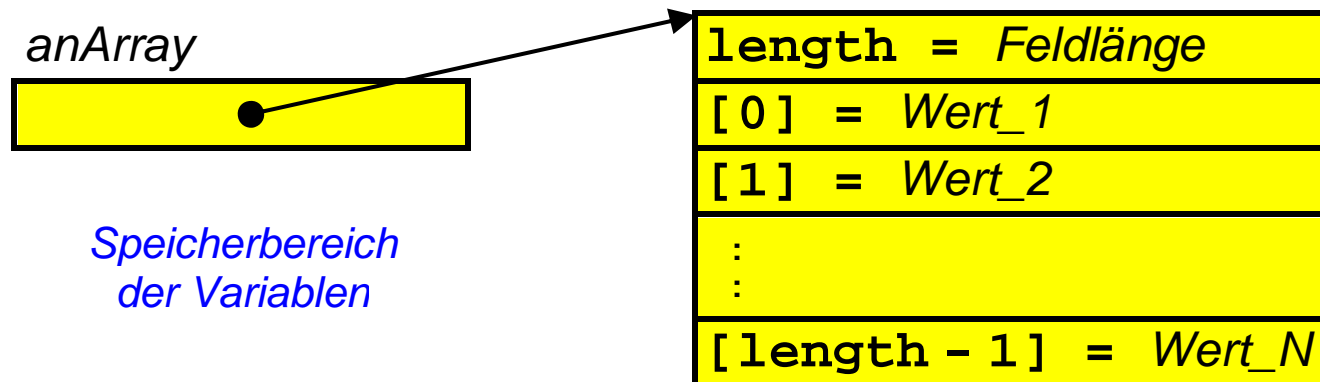
*die Feldelemente werden mit dem Standardwert des Typs initialisiert*

*Für Typ kann jeder gültige Javatyp eingesetzt werden.*

*Die Feldlänge muss vom Typ `int` sein oder automatisch nach `int` konvertierbar sein.*

- **Wert:** Referenz auf eine Folge von Werten gleichen Typs  
(Standardwert ist die ungültige Referenz `null` ).

Grafische Darstellung:



*Die Feldelemente sind von 0 bis `length - 1` durchnummeriert.*

*Speicherbereich des Felds ( mit `new` reserviertes Feldobjekt )*



# Java Referenztypen: Felder (Arrays) (2)

---

- Die im `new`-Ausdruck festgelegte Feldlänge ist als `length` abfragbar:  
    `anArray.length`      *die Feldlänge kann nach dem `new` nicht mehr geändert werden*
- Der Indizierungs-Operator `[ ]` macht die Feldelemente zugreifbar:  
    `anArray[Index]`  
    *Der Index muss vom Typ `int` sein (oder automatisch nach `int` konvertierbar)*  
    Index 0      *erstes Feld-Element*  
    Index `length - 1`      *letztes Feld-Element*  
    *Indices außerhalb dieses Bereichs erzeugen `ArrayIndexOutOfBoundsException`*

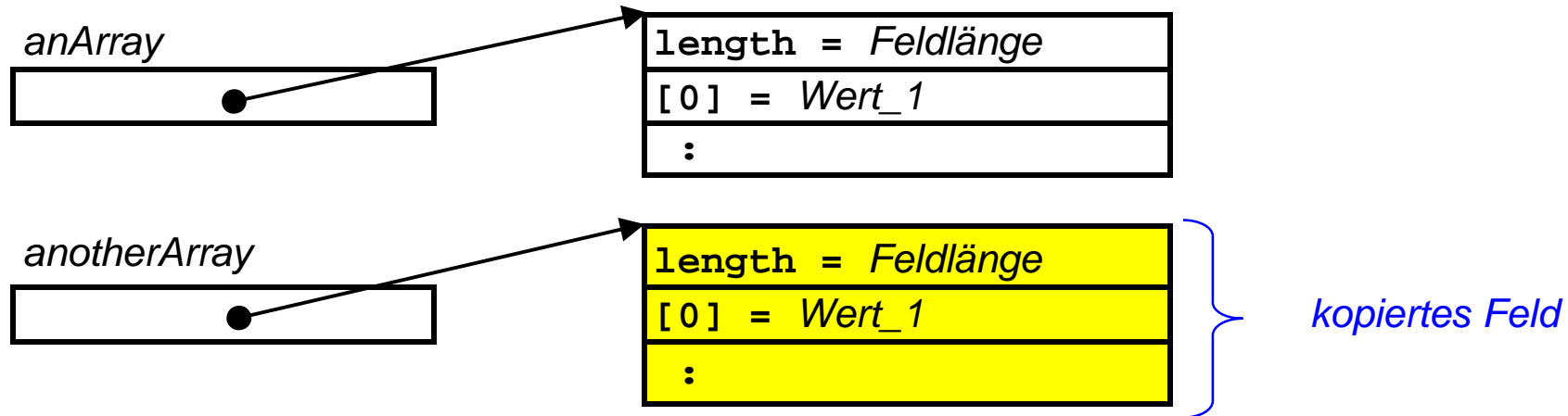
Verwendung von `.length` und `[ ]` meist innerhalb einer Schleife:

*Initialisierung der Laufvariablen i*      *Fortsetzungsbedingung*      *Fortschaltung ( i um 1 erhöhen )*

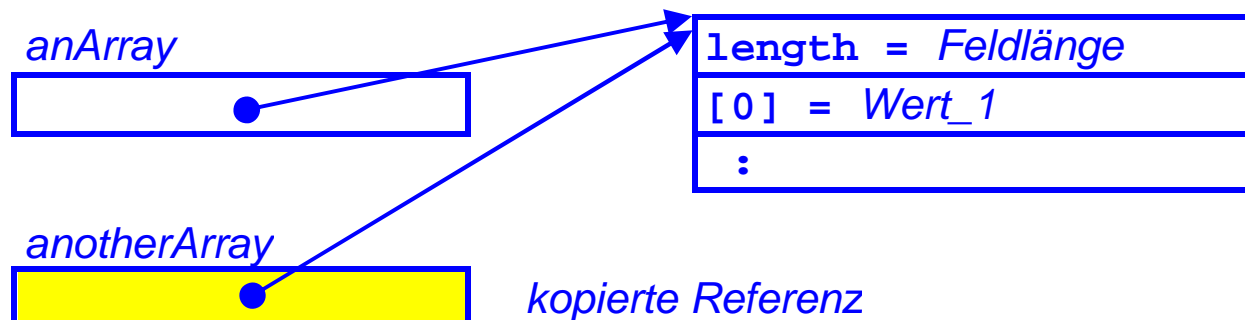
```
for (int i = 0; i < anArray.length; ++i) {  
    System.out.println(anArray[i]);  
}
```

## Java Referenztypen: Felder (Arrays) (3)

- Feldkopien mit `clone()`: `anotherArray = anArray.clone();`



*Achtung! Eine einfache Zuweisung kopiert nur die Referenz:* `anotherArray = anArray;`



# Beispiel-Programm Feld-Variable

- Quellcode:

```
public final class ArrayVar {  
    private ArrayVar() { }  
    public static void main(String[] args) {  
        int[] anIntArray = new int[] {3421, 3442, 3635, 3814};  
        for (int i = 0; i < anIntArray.length; ++i) {  
            System.out.printf("%d: %d%n", i, anIntArray[i]);  
        }  
        int[] anotherIntArray = {0}; // kurz für new int[1] oder new int[]{0}  
        for (int i = 0; i < anotherIntArray.length; ++i) {  
            System.out.printf("%d: %d%n", i, anotherIntArray[i]);  
        }  
        anotherIntArray = anIntArray.clone();  
        for (int i = 0; i < anotherIntArray.length; ++i) {  
            System.out.printf("%d: %d%n", i, anotherIntArray[i]);  
        }  
    }  
}
```

*Was gibt das Programm auf der Konsole aus?*

# Java Referenztypen: mehrdimensionale Felder

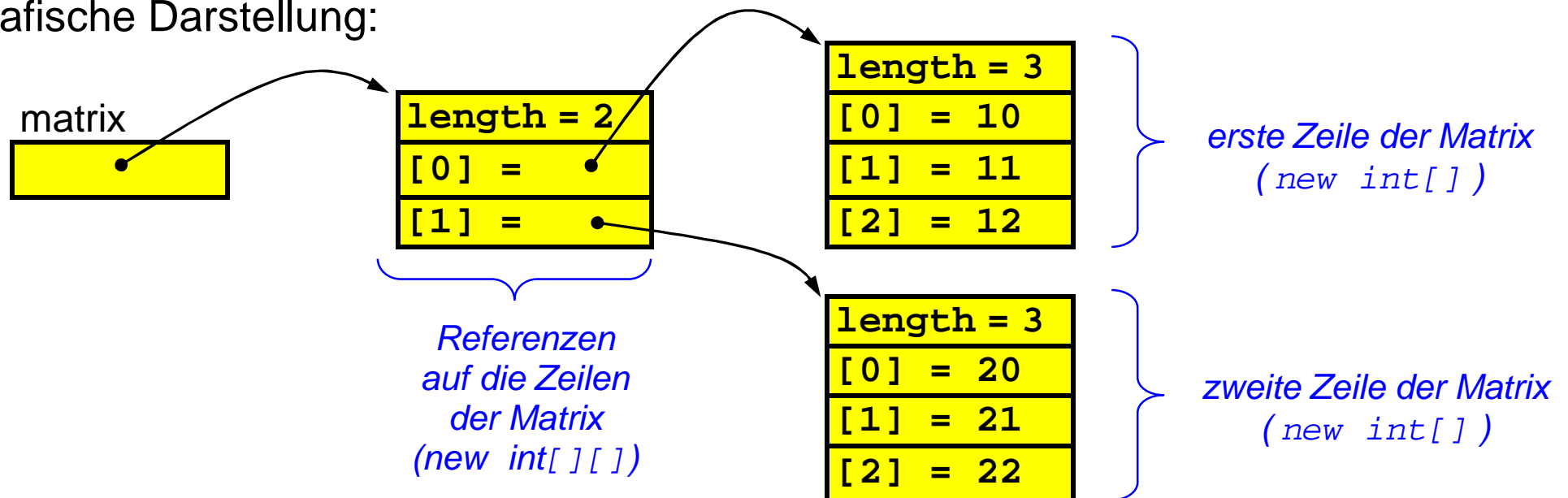
Felder von Feldern (am Beispiel einer 2x3-Matrix ganzer Zahlen)

- **Variablen-Definition:**

```
int[][] matrix = new int[][] {new int[] {10, 11, 12},  
                               new int[] {20, 21, 22}};
```

- **Wert:** Referenz auf eine Folge von Referenzen auf eine Folge von `int`-Werten.

Grafische Darstellung:



# Beispiel-Programm Matrix-Variable

- Quellcode:

```
public final class MatrixVar {  
    private MatrixVar() { }  
    public static void main(String[] args) {  
        int[][] matrix = {{10, 11, 12}, {20, 21, 22}};  
        for (int i = 0; i < matrix.length; ++i) {  
            for (int j = 0; j < matrix[i].length; ++j) {  
                System.out.printf("%3d", matrix[i][j]);  
            }  
            System.out.println();  
        }  
        int[][] anotherMatrix = matrix.clone();  
        for (int i = 0; i < matrix.length; ++i) {  
            anotherMatrix[i] = matrix[i].clone();  
        }  
    }  
}
```

*Initialisierung hier  
in Kurzschreibweise  
ohne new*

*rechtsbündig mit Mindestfeldbreite 3*

*Matrixkopie erfordert  
je ein clone()  
pro Speicherbereich*

*Was gibt das Programm auf der Konsole aus?*

# Java Referenztypen: Aufzählungen (1)



Eine Aufzählung ist ein benutzerdefinierter Referenztyp.

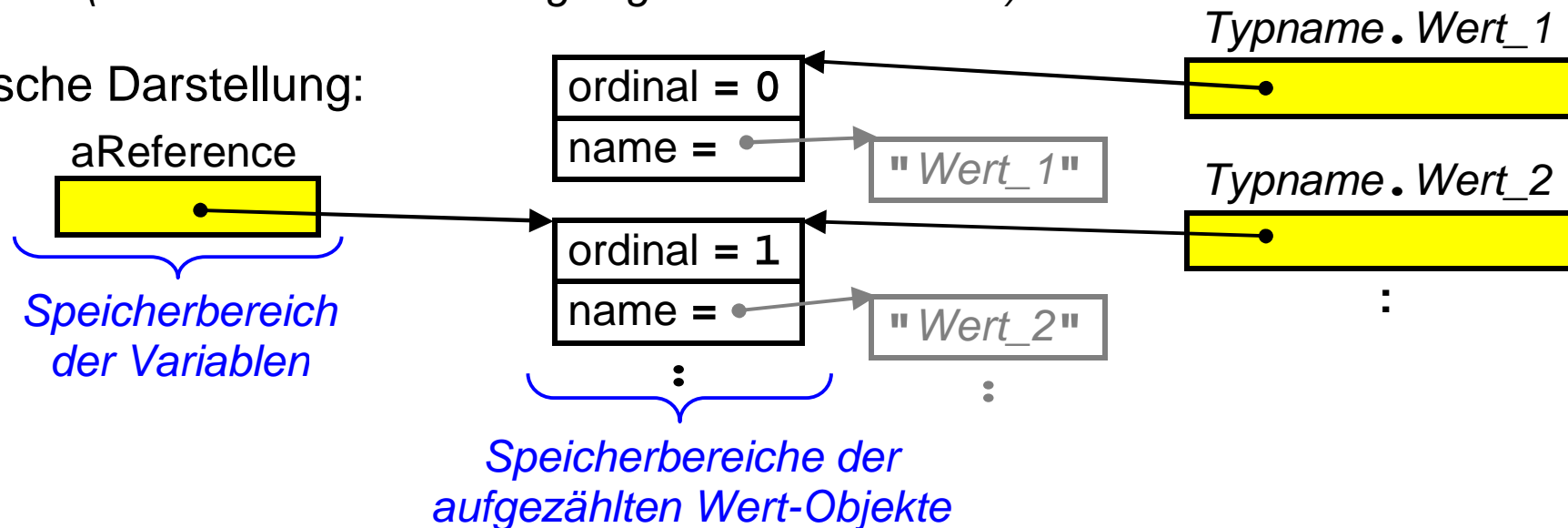
- **Typ-Definition**: in einer Datei `Typname.java` *mögliche "Werte" des Typs*

```
public enum Typname { Wert_1, Wert_2, ..., Wert_N }
```

- **Variablen-Definition**: `Typname aReference = Typname.Wert_2;`

- **Wert**: Referenz auf einen der in der Typ-Definition aufgezählten Werte  
(Standardwert ist die ungültige Referenz `null`)

Grafische Darstellung:



## Java Referenztypen: Aufzählungen (2)

---

- alle möglichen Werte einer Aufzählung können abgefragt werden:

```
Typname[ ] allValues = Typname.values( );
```

*man erhält eine Referenz auf ein Feld, das die Referenzen auf alle Werte enthält*

- die Werte einer Aufzählung können in einen String gewandelt werden:

```
String s = String.valueOf( Typname.Wert_2 ); // ergibt String "Wert_2"
```

```
String t = Typname.Wert_2.toString( ); // ergibt auch String "Wert_2"
```

- ein passender String kann in einen Aufzählungswert gewandelt werden:

```
Typname aReference = Typname.valueOf( "Wert_2" ); // Aufzählungswert Wert_2
```

*ein unpassender String erzeugt einen Fehler `IllegalArgumentException`*

- die Werte einer Aufzählung sind von 0 an durchnummeriert:

```
int n = Typname.Wert_2.ordinal( ); // ergibt 1
```

# Beispiel-Programm enum-Variable



```
// Jahreszeit.java
```

```
public enum Jahreszeit { FRUEHLING, SOMMER, HERBST, WINTER }
```

```
// EnumVar.java
```

```
public final class EnumVar {  
    private EnumVar() { }  
    public static void main(String[] args) {  
        Jahreszeit fruehling = Jahreszeit.FRUEHLING;  
        Jahreszeit sommer = Jahreszeit.valueOf("SOMMER");  
        System.out.printf("%s\n%s\n", String.valueOf(fruehling), sommer.toString());  
        Jahreszeit[] jahreszeiten = Jahreszeit.values();  
        for (int i = 0; i < jahreszeiten.length; ++i) {  
            if (jahreszeiten[i] != fruehling && jahreszeiten[i] != sommer) {  
                System.out.println(jahreszeiten[i]);  
            }  
        }  
    }  
}
```

*Was gibt das Programm auf der Konsole aus?*



# Java Daten: Index

---

<b>!</b> 2-12	<b>&lt;</b> 2-12	<b>if</b> 2-12
<b>!=</b> 2-12	<b>&lt;=</b> 2-12	<b>int</b> 2-15
<b>"x"</b> 2-6	<b>=</b> 2-12	Konstante 2-10
<b>&amp;&amp;  </b> 2-12	<b>==</b> 2-12	Literal 2-1 bis 2-8
<b>'x'</b> 2-5,2-7	<b>&gt;</b> 2-12	<b>long</b> 2-15
<b>+ - * / %</b> 2-15	<b>&gt;=</b> 2-12	<b>new</b> 2-21,2-23
<b>++</b> 2-24	Array 2-11,2-23 bis 2-28	Objekt 2-18
<b>.charAt(...)</b> 2-20	Aufzählung 2-29,2-30	Referenztyp 2-11,2-18
<b>.clone()</b> 2-25,2-26,2-28	<b>boolean</b> 2-11,2-12,2-13	<b>short</b> 2-15
<b>.compareTo(...)</b> 2-20	<b>byte</b> 2-15	String 2-6,2-19 bis 2-22
<b>.equals(...)</b> 2-20	<b>char</b> 2-14	<b>String.format(...)</b> 2-21
<b>.intern()</b> 2-21	Datentyp 2-9,2-11	<b>String.valueOf(...)</b> 2-21
<b>.length</b> 2-24,2-26,2-28	<b>double</b> 2-16	<b>System.out.printf(...)</b> 2-2,2-4,2-7
<b>.length()</b> 2-20	Einzelzeichen 2-5,2-14	<b>System.out.println(...)</b> 2-2
<b>0173</b> 2-1	<b>else</b> 2-12	<b>true</b> 2-12
<b>0b1111011</b> 2-1	<b>enum</b> 2-29,2-31	Variable 2-9,2-10
<b>0x7b</b> 2-1	<b>false</b> 2-12	Werttyp 2-11,2-18
<b>123</b> 2-1	Feld 2-11,2-23 bis 2-28	Zeichenkette 2-6
<b>1e-30</b> 2-3	<b>float</b> 2-16	<b>[]</b> 2-24
<b>78.9</b> 2-3	<b>for(...; ...; ...)</b> 2-24,2-26,2-28	