

1. Referenzen

2. Arrays (Felder, Reihungen)

2.1 Der Datentyp der Arrays

2.2 Arrays in Java

2.3 Zeichenketten (Strings)

3. Record-Typen

- Die Umsetzung in Java erfolgt 1-zu-1, allerdings ist zu beachten:
- Wie bei den meisten Programmiersprachen beginnt die Indexmenge in Java bei 0, d.h. für eine n -elementige Reihung gilt die Indexmenge $I_n = \{0, \dots, n - 1\}$.
(Die Operation *LEN* würde für ein Array der Länge n also $n - 1$ zurückgeben!)
- Wie bei uns sind in Java die Arrays *semidynamisch*, d.h. ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden (=statisch), d.h. dynamisches Wachstum von Arrays ist auch in Java nicht möglich!!!

Beispiel

Ein `char`-Array `gruss` der Länge 13:

<code>gruss:</code>	'H'	'e'	'l'	'l'	'o'	','	''	'W'	'o'	'r'	'l'	'd'	'!'
<code>Index:</code>	0	1	2	3	4	5	6	7	8	9	10	11	12

- Der Typ eines Arrays, das den Typ `<type>` enthält, wird in Java als `<type> []` notiert (statt **ARRAY**`<type>`).
- Beispiel: ein `int`-Array ist vom Typ `int []`.
- Variablen und Konstanten vom Typ `<type> []` können wie gewohnt vereinbart werden:

```
<type>[] variablenName;
```

(Konstanten wie immer mit dem Zusatz `final`)

- Und natürlich: Arrays sind *Referenztypen*, d.h. auf dem Zettel einer Variable steht nicht das Array direkt, sondern die Referenz (auch Java-Arrays sind i.d.R. unterschiedlich groß und werden daher auf der Halde (Heap) verwaltet).

Erzeugung (Konstruktions-Operator):

- Es gibt nicht *den* Konstruktions-Operator *INIT*.
- Aber: wie alle Referenztypen werden Arrays bei ihrer Initialisierung *erzeugt*.
- Die Initialisierung (Erzeugung) eines Arrays kann dabei auf verschiedene Arten erfolgen.
- Die einfachste ist, alle Elemente der Reihe nach in geschweiften Klammern {} aufzuzählen:

```
<type>[] variablenName = {elem1, elem2, ...}
```

wobei die einzelnen `elem1`, `elem2`, etc. Literale (Werte) oder Variablen vom Typ `<type>` sind.

Zugriff auf das i -te Element (Projektion):

- Die Operation *PROJ* für den Zugriff auf das i -te Element eines Arrays a notiert man in Java durch den Ausdruck $a[i]$.
- Dabei ist i vom Typ `int` (bzw. vom Compiler implizit in `int` cast-bar)
- Der Wert des Ausdrucks `variablenName[i]` ist der Wert des Arrays `variablenName` an der Stelle i .
- Der Typ des Ausdrucks `variablenName[i]` ist der Typ, über dem das Array `variablenName` gebildet wurde.
- Beispiel:

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

Der Ausdruck `a[1]` hat den Wert 2 und den Typ `int`.

Verändern des i -ten Elements:

- Die Operation *ALT* gibt es nicht explizit.
- Vielmehr ist die Projektion auf das i -te Element eines Arrays a ($a[i]$) auch für den schreibenden Zugriff gedacht.
- Man kann $a[i]$ also nicht nur als Ausdruck verstehen, sondern diesem selbst auch einen Wert zuweisen (es handelt sich nämlich letztlich um eine Variable).
- **Beispiel:** `int [] a = { 1 , 2 , 3 , 4 };`
`a[1] = 6;` verändert das Array a zu `{ 1 , 6 , 3 , 4 }`.

Zugriff auf die Länge:

- Die Operation *LEN* ist in Java etwas anders umgesetzt: es gibt eine Konstante mit Namen `length`, die anzeigt, wie viele Elemente im Array enthalten sind (Vorsicht: das sind nicht $n - 1$ sondern n !!!), d.h. es wird nicht die obere Grenze der Indexmenge sondern die Länge des Arrays gespeichert.
- Der Typ dieser Konstante ist `int`.
- Der Name der Konstanten ist zusammengesetzt aus dem Namen des Arrays (also z.B. `a`) und dem Namen `length` (also `a.length`).
- Beispiel:

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

Der Ausdruck `a.length` hat den Wert 4 und den Typ `int`.

Beispiel

```
char a = 'a';  
char b = 'b';  
char c = 'c';  
char[] abc = {a, b, c};  
System.out.print(abc[0]); // gibt den Character 'a' aus,  
                          // den Wert des Array-Feldes  
                          // mit Index 0. Allgemein: array[i]  
                          // ist Zugriff auf das i-te Element  
  
System.out.print(abc.length); // gibt 3 aus  
int[] zahlen = {1, 3, 5, 7, 9};  
System.out.print(zahlen[3]); // gibt die Zahl 7 aus  
System.out.print(zahlen.length); // gibt 5 aus
```

- Oft vereinbart man eine Array-Variable, bevor man die einzelnen Elemente kennt.
- Die Länge muss man dabei angeben:
`char[] abc = new char[3];`
- Das Schlüsselwort **new** ist hier verlangt (es bedeutet in diesem Fall, dass eine neue (leere) Referenz angelegt wird — es ist also so etwas wie ein Konstruktions-Operator).
- Dann kann man das Array im weiteren Programmverlauf (durch Verändern der einzelnen Elemente) füllen:

```
abc[0] = 'a';
```

```
abc[1] = 'b';
```

```
abc[2] = 'c';
```

- Dass Arrays in Java *semidynamisch* sind, bedeutet: Es ist möglich, die Länge erst zur Laufzeit festzulegen.
- Beispiel

```
// x ist eine Variable vom Typ int  
// deren Wert bei der Ausführung  
// feststeht, aber ggfs. noch nicht beim  
// Uebersetzen des Programmcodes (Kompilieren)  
// (z.B. weil x ein Eingabeparameter ist)  
char[] abc = new char[x];
```

- Was passiert, wenn man ein Array anlegt

```
int[] zahlen = new int[10];
```

aber nicht füllt? Ist das Array dann leer?

- Nein: es gibt in Java keine leeren Arrays.
- Ein Array wird immer mit den Standardwerten des jeweiligen Typs initialisiert.
- Das spätere Belegen einzelner Array-Zellen ist also immer eine Änderung eines Wertes (durch eine Wertzuweisung):

```
int[] zahlen = new int[10];
System.out.print(zahlen[3]); // gibt 0 aus
zahlen[3] = 4;
System.out.print(zahlen[3]); // gibt 4 aus
```

- Das legt den Schluss nahe, dass die einzelnen Elemente des Arrays wiederum Variablen (Zettel) sind, die ich beschreiben und ablesen kann.
- Das ist genau so, wie wir es theoretisch eingeführt haben und so in etwa kann man sich das auch tatsächlich vorstellen.
- In

```
int[] zahlen = { 1 , 2 , 3 }
```

ist die Variable `zahlen` ein (radierbarer) Zettel, auf dem ein Haufen weitere Zettel, nämlich die der einzelnen Elemente `zahlen[i]`, liegen.

- Genauer gesagt enthält `zahlen` die Adresse der Zettel, der einzelnen Elemente `zahlen[i]` und einen zusätzlichen (nicht-radierbaren) Zettel mit der Länge des Arrays (`zahlen.length`).
- Die einzelnen (radierbaren) Zettel `zahlen[1]`, enthalten die konkreten Werte
- `zahlen` ist im Stack gespeichert, die einzelnen `zahlen[i]` stehen im Heap.

- Es gelten die üblichen Eigenschaften für Zettel (Variablen/Konstanten), also z.B. ist auch so etwas erlaubt:

```
char[] abc = { 'a', 'b', 'c' };  
char[] de = { 'e', 'e' };  
abc = de; // d.h. de wird der Wert von abc zugewiesen
```

- Und jetzt eben Achtung: statt einem konkreten Wert wird hier eine Referenz zugewiesen (siehe Implikation “Zuweisung”)!

- Call-by-reference-Effekt:

```
public static void veraendere(int[] a, int i, int wert) {  
    a[i] = wert;  
}
```

```
public static void main(String[] args) {  
    int[] werte = {0, 1, 2};  
    changeValues(werte, 1, 3);  
}
```

Was passiert mit `werte`?

- Gleichheit/Identität:

```
int[] x = {1 , 2};
```

```
int[] y = {1 , 2};
```

```
boolean gleich = (x==y); // Welchen Wert hat gleich???
```

- Zuweisung/Kopie:

```
int [] x = {1 , 2};
```

```
int [] y = {2 , 3};
```

```
x = y;
```

```
y[1] = 5; // Welchen Wert hat x[1]???
```

```
x[1] = 10; // Welchen Wert hat y[1]???
```

- Die Klasse `java.util.Arrays` bietet einige (statische) Methoden rund um Arrays an.
- Die (überladen) Methode `equals`, überprüft die Gleichheit zweier Arrays, z.B.

```
static boolean equals(int[] a, int[] a2)
```

für `int`-Arrays, also:

```
int[] x = {1 , 2};
```

```
int[] y = {1 , 2};
```

```
boolean gleich = java.util.Arrays.equals(x,y);
```

- Die überladenen Methoden `copyOf`, erstellt eine tiefe Kopie zu erstellen, z.B.

```
static int[] copyOf(int[] original, int newLength)
```

für `int`-Arrays (die neue Kopie kann dabei abgeschnitten oder mit zusätzlichen 0.0-Werten aufgefüllt werden — je nach `newLength`), also

```
int[] x = {1 , 2};  
int[] y = {2 , 3};  
x = java.util.Arrays.copyOf(y, y.length);  
// x und y sind nun unabhaengig  
int[]z = java.util.Arrays.copyOf(y, y.length+2);  
// z ist um 2 Elemente laenger, die zusaetzlichen Elemente  
// sind jeweils 0.0
```

- Auch Array-Variablen kann man als Konstanten deklarieren.
- Dann kann man der Variablen keinen neuen Wert zuweisen:

```
final char[] ABC = { 'a', 'b', 'c'};  
final char[] DE = { 'd', 'e'};  
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

- Aber **Achtung**: einzelne Array-Komponenten sind normale Variablen (Zettel), man kann ihnen also einen neuen Wert zuweisen:

```
ABC[0] = 'd'; // erlaubt  
ABC[1] = 'e'; // erlaubt  
System.out.print(ABC.length); // gibt 3 aus  
System.out.print(ABC[0]); // gibt 'd' aus  
System.out.print(ABC[1]); // gibt 'e' aus  
System.out.print(ABC[2]); // gibt 'c' aus
```

- Hä? Wie passt denn das mit unserer Intuition der Zettel(-wirtschaft) zusammen?
- Sehr gut sogar:
- Wie gesagt, eine Variable vom Typ `<type> []` ist ein Zettel, auf dem die Referenz zu weiteren Zetteln (die Elemente) steht.
- Konstanten sind nicht radierbare Zettel, d.h. der Zettel `<type> []` auf dem die Referenz zu den anderen Zettel steht, ist dann nicht radierbar; die Zettel, die referenziert werden, aber natürlich schon.

- Mit Arrays kann man also wunderbar programmieren:
- Beispiel: der Algorithmus *enthalten* von oben

```
public static boolean enthalten(int[] x, int a) {
    boolean gefunden = false;
    int i = 0;
    while(!gefunden && i < x.length) {
        if(x[i] == a) {
            gefunden = true;
        }
        i++;
    }
    return gefunden;
}
```

Versuchen Sie es selbst mal mit einer `for`-Schleife!

- Beispiel: Summe der Elemente in einem `int`-Array

```
public static int summeElemente(int[] x) {  
    int erg = 0;  
    for(int i = 0; i < x.length; i++) {  
        erg = erg + x[i];  
    }  
    return erg;  
}
```


- Beispiel: Wechselgeldalgorithmus

Zur Erinnerung:

- Eingabe eines Rechnungsbetrags $1 \leq r \leq 100$.
- Gesucht ist das Wechselgeld zu einer Bezahlung von r mit einem 100-EUR-Schein als Menge an 1 EUR, 2 EUR Münzen sowie 5 EUR Scheinen (mit dem Ziel möglichst wenige Münzen/Scheine auszugeben).
- Als Ergebnis wollten wir eine Folge an 1er, 2er und 5er ausgeben. Das könnten wir jetzt mit einem Array implementieren.

Dadurch handeln wir uns allerdings ein Problem ein:

- Arrays sind ja semi-dynamisch, d.h. wir müssen in Abhängigkeit von r zunächst bestimmen, wieviel Scheine/Münzen auszugeben sind (d.h. wie lang das Ergebnis-Array wird).

- Beispiel: Wechselgeldalgorithmus (cont.)

Aber die Lösung hatten wir wenigstens schon:

- Der ganzzahlige Quotient $q_1 = DIV(100 - r, 5)$ ist die Anzahl der 5-EUR-Scheine im Wechselgeld.
- Der Rest $r_1 = MOD(100 - r, 5)$ ist der noch zu verarbeitende Wechselbetrag. Offensichtlich gilt $r_1 < 5$.
- r_1 muss nun auf 1 und 2 aufgeteilt werden, d.h. analog bilden wir $q_2 = DIV(r_1, 2)$ und $r_2 = MOD(r_1, 2)$.
- q_2 bestimmt die Anzahl der 2-EUR-Münzen und r_2 die Anzahl der 1-EUR-Münzen.

D.h. die Länge des Ergebnis-Arrays ist $q_1 + q_2 + r_2$.

- Beispiel: Wechselgeldalgorithmus (cont.)

Fragt sich nur noch, wie wir das Ergebnis-Array zu befüllen haben:

- q_1 ist die Anzahl der 5-EUR-Scheine, d.h. die Stellen $0, \dots, (q_1 - 1)$ sind mit der Zahl 5 zu belegen.
- q_2 ist die Anzahl der 2-EUR-Münzen, d.h. die Stellen $q_1, \dots, (q_1 + q_2) - 1$ sind mit der Zahl 2 zu belegen.
- r_2 ist die Anzahl der 1-EUR-Münzen, d.h. die Stellen $(q_1 + q_2), \dots, (q_1 + q_2 + r_2) - 1$ sind mit der Zahl 1 zu belegen.

- Beispiel: Wechselgeldalgorithmus (cont.)

```
public static int[] wechselGeld(int r) {
    int q1 = (100 - r) / 5;
    int q2 = ((100 - r) % 5) / 2;
    int r2 = ((100 - r) % 5) % 2;
    int[] erg = new int[q1 + q2 + r2];

    for(int i=0; i<q1; i++) {
        erg[i] = 5;
    }
    for(int i=q1; i<q1+q2; i++) {
        erg[i] = 2;
    }
    for(int i=q1+q2; i<erg.length; i++) {
        erg[i] = 1;
    }
    return erg;
}
```

- Wir haben übrigens immer noch nicht geklärt, was eigentlich passiert, wenn wir eine Indexposition abrufen wollen, die es gar nicht gibt, also z.B.

```
int[] a = {1, 2, 3}; a[10] = 11;
```

- In diesem Beispiel hat der Compiler die Chance, das theoretisch abzufangen (was er aber nicht tun wird).
- Ganz allgemein ist das sowieso schwierig:

```
int x; // Wert erst zur Laufzeit klar ...  
int[] a = new int[x]; //Laenge erst zur Laufzeit klar!
```

- Der Fehler wird erst zur Laufzeit behandelt durch eine sog. *Ausnahme (Exception)*.
- Was das ist, lernen wir leider erst später kennen, aber schon mal soviel: das Programm wird mit einer Fehlermeldung beendet.

- Da auch Arrays einen bestimmten Typ haben z.B.

`gruss : char []` kann man auch Reihungen von Reihungen bilden.

- Diese Gebilde heißen auch *mehrdimensionale* Arrays.
- Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren.

```
int[] m0 = {1, 2, 3};  
int[] m1 = {4, 5, 6};  
int[][] m = {m0, m1};
```

- Man ist dabei nicht auf „rechteckige“ Arrays beschränkt:

```
int[] m0 = {0};  
int[] m1 = {1, 2};  
int[] m3 = {3, 4, 5};  
int[][] m = {m0, m1, m2};
```

1. Referenzen

2. Arrays (Felder, Reihungen)

2.1 Der Datentyp der Arrays

2.2 Arrays in Java

2.3 Zeichenketten (Strings)

3. Record-Typen

- Wir hatten bereits diskutiert, dass Zeichenketten nicht nur zur Darstellung von Daten benutzt werden können; sie können selbst Gegenstand der Datenverarbeitung sein.
- *Zeichenketten (Strings)* sind in Java Arrays über dem Typ `char` (Folgen über der Menge der druckbaren Zeichen).
- Java stellt einen eigenen Typ `String` für Zeichenketten zur Verfügung, d.h. es gibt eine eigene Sorte (mit Operationen) für Zeichenketten in Java, wir können mit diesem Typ ganz normal „arbeiten“.
- Der Typ `String` ist kein primitiver Typ, sondern wieder ein Referenztyp, genauer eine Klasse von Objekten, ein sog. *Objektyp*.

- Betrachten wir folgendes Beispiel:

```
public class HelloWorld {  
    public static final String GRUSS = "Hello World";  
    public static void main(String[] args) {  
        System.out.println(GRUSS);  
    }  
}
```

- In der Deklaration und Initialisierung

```
public static final String GRUSS = "Hello, World!";
```

entspricht der Ausdruck "Hello, World!" einer speziellen Schreibweise für ein konstantes Array `char[13]`, das in einen Typ `String` *gekapselt* ist.

- **Achtung:** Die Komponenten dieses Arrays können nicht mehr (durch Neuweisung) geändert werden.

- Obwohl `String` (wie `Arrays`) kein primitiver Typ ist, wird dieser Typ in Java sehr ähnlich wie ein primitiver Typ behandelt:
- Z.B. können Werte dieses Typs (wie bei primitiven Typen) durch Literale gebildet werden (in `" "` eingeschlossen).
- Beispiele für Literale der Sorte `String` in Java:
 - `"Hello World!"`
 - `"Kroeger"`
 - `"Guten Morgen"`
 - `"42"`
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (überladenen) Operator `+` konkateniert werden:
 - `"Guten Morgen, "+"Kroeger"` ergibt die Zeichenkette `"Guten Morgen, Kroeger"`

- `String` ist in der Tat ein klassisches Modul, das verschiedene Operationen (statische Methoden) über dieser (und weiterer) Sorte(n) bereitstellt.
- Ein paar davon schauen wir uns im Folgenden an (einige später), ansonsten sei wieder auf die Dokumentation der API verwiesen:
- Eine überladene Typcast-Operation, um aus primitiven Typen Strings zu erzeugen

```
static String valueOf(<type> input)
```

Bei der Konkatination eines Strings mit einem Literal eines primitiven Typs (z.B. `Note: "+1.0`) werden diese Methoden (hier: `static String valueOf(double d)`) implizit verwendet.

- Länge der Zeichenkette durch die Methode `int length()` (Achtung, anders als bei Arrays ist das tatsächlich eine Methode!!!)
- Die Methode `char charAt(int index)` liefert das Zeichen an der gegebenen Stelle des Strings
Dabei hat das erste Element den Index 0 und das letzte Element den Index `length() - 1` (ist ja ein Array).
- Beispiele:
 - der Ausdruck `"Hello, World!".length()` hat den Wert: 13
 - der Ausdruck `"Hello, World!".charAt(10)` hat den Wert: `'l'`

- Nun verstehen Sie auch endlich offiziell den Parameter der `main`-Methode eines Java Programms.
- In Programmbeispielen haben wir bereits die `main`-Methode gesehen, die das selbständige Ausführen eines Programmes ermöglicht.
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus (bzw. gibt eine Fehlermeldung falls, diese Methode dort nicht existiert).
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array, meist als Eingabe-Variablen `args`.
- Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.

- Der Aufruf

```
java KlassenName <Ein1> <Ein2> ... <Ein_n>
```

füllt das `String`-Array (Annahme, der Eingabeparameter heißt `args`) automatisch mit den Eingaben

```
args[0] = <Ein1>
```

```
args[1] = <Ein2>
```

```
...
```

```
args[n-1] = <Ein_n>
```

- Beispiel für einen Zugriff der main-Methode auf das Parameterarray

```
public class Gruss {  
    public static void gruessen(String gruss) {  
        System.out.println("Der Gruss ist: "+gruss);  
    }  
    public static void main(String[] args) {  
        gruessen(args[0]);  
    }  
}
```

Dadurch ist eine vielfältigere Verwendung möglich:

- `java Gruss "Hello, World!"`
- `java Gruss "Hallo, Welt!"`
- `java Gruss "Servus!"`

- Jetzt können wir auch endlich unser Kalenderprojekt etwas cleverer implementieren
- Statt je eine Variable für die einzelnen Termin-Einträge zu verschwenden, verwalten wir diese jetzt in einem Array.
- Statt einem einzigen `char`-Eintrag als Abkürzung können wir jetzt den Eintrag als `String`-Objekt speichern.
- Die Termine an einem Tag (einfach erweiterbar auf eine ganze Woche bzw. auf ganze Jahre):

```
...  
String[] termine = new String[24]  
...
```


- Die Termine sollten zwar noch initialisiert werden (z.B. mit dem leeren String), dies geht aber auch einfach mit einer Schleife:

```
for(int i = 0; i < termine.length; i++) {  
    termine[i] = "";  
}
```

- Die Ausgabe benötigt auch nur solch eine Schleife.
- Einfach geht auch die Eintragung eines neuen Termins:

```
// Termin-Eingabe  
int stunde = ...  
String eingabe = ...  
  
termine[stunde] = eingabe;
```

- Die Erweiterung auf eine ganze Woche kann dann z.B. über ein mehrdimensionales Array erfolgen:

```
String[][] termine = new String[7][];

// die einzelnen Tage mit 24 Stunden anlegen
// und gleich initialisieren:
for(int i = 0; i < termine.length; i++) {
    // anlegen
    termine[i] = new String[24];
    // initialisieren
    for(int j=0; j < termine[i].length; j++) {
        termine[i][j] = "";
    }
}
```

- Frühstück am Dienstag (Tag 2), 8 Uhr tragen wir wie folgt ein:

```
termine [1][7] = "Fruehstueck";
```