

Arrays

Softwareentwicklung II (IB)

Prof. Dr. Oliver Braun

Letzte Änderung: 25.06.2018 07:23

Allokieren und Initialisieren

- **Arrays** (auch „Feld“, „Reihung“) vordefiniert, ohne weitere Maßnahmen verfügbar
- Werden von praktisch allen Programmiersprachen angeboten
- Tief in Java verankert, von der JVM intern genutzt
- Arrays sind **Containertypen** (ähnlich wie Lists, Maps, Sets): Speichern Elemente anderer Typen
- **Elementtyp** beliebig, aber gleich für alle Elemente
- Beispiel: Array mit fünf **int**-Elementen
- Werte einzelner Elemente austauschbar (im Gegensatz zu Strings)
- Anzahl Elemente eines Arrays („Arraylänge“) unveränderlich

- Arrays = Familie von ähnlichen Typen, kein einzelner Typ (vergleichbar mit Enums)
- Typangabe: Elementtyp + leere eckige Klammern `type[]`
- Beispiel: Array mit `int`-Elementen (kurz „`int`-Array“) `int[]`
- Zu jedem Elementtyp ein korrespondierender **Arraytyp**

- Beispiele:

Elementtyp	Arraytyp
int	int[]
double	double[]
boolean	boolean[]
char	char[]
String	String[]
Rational	Rational[]
Color	Color[]

- Arraytypen sind Referenztypen
- Ein Arraytyp legt keine Länge fest
- Ein konkretes Exemplar eines Arrays hat eine feste, unveränderliche Länge

- Erzeugen eines neuen Arrays mit einer bestimmten Anzahl Elemente vom Typ `type`:

```
new type[expression]
```

- `expression` = Anzahl Elemente, beliebiger `int`-Ausdruck
- Beispiele: Arrays mit 4, 69, 10 und 97 Elementen:

```
new int[4];  
new double[1 + 17*4];  
new String["new String".length()];  
new Rational['a'];
```

- Anzahl Elemente eines neuen Arrays ...
 - ... wird im **new**-Aufruf festgelegt
 - ... wird zur Laufzeit berechnet, nicht vom Compiler
 - ... kann später nicht mehr verändert werden
- Bildhafte Vorstellung: Array = Liste namenloser Variablen, werden gemeinsam definiert, bleiben für die Lebensdauer des Arrays beisammen

- Definition von Variablen von Arraytypen („Array-Variablen“)
- Beispiel:

```
int[] a;
```

- Zuweisung eines Arrays an eine Arrayvariable:

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

- Verschiedene Arrays mit unterschiedlichen Längen als Werte einer Arrayvariablen:

```
int[] a;  
a = new int[10];  
a = new int[1];    // etwas eigenartig, aber zulässig  
a = new int[10_000];
```

- Elemente eines Arrays beim Allokieren automatisch mit Defaultwerten vorbesetzt (ebenso wie Objekt- und Klassenvariablen)
- Beispiel:

```
Rational[] r = new Rational[97];
```

`r` referenziert ein Array mit 97 `Rational`-Elementen, alle mit dem Wert `null` initialisiert

- Erzeugt nur das Array, keine Objekte, ruft keinen Element-Konstruktor auf

- **Arrayliteral** = Konstante eines Arraytyps
- Allokiert neues Array aus einer Liste vorgegebener Werte
- Schema:

```
new type[] {expression, expression, ..., expression};
```

Länge der Arrays = Anzahl Elemente

- Beispiel:

```
new int[] {71, -4, 7220, 0, 238};
```

- Listenelemente = beliebige Ausdrücke, kompatibel zum Elementtyp des Arrays

Arrayliterale (2/2)

- Beispiel:

```
int[] a = new int[]  
    {123, -4, 7220, 703%19, (int)(715.0/3)};
```

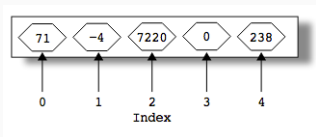
Kurzfassung für:

```
int[] a = new int[5];  
a[0] = 123;  
a[1] = -4;  
a[2] = 7220;  
a[3] = 703%19;  
a[4] = (int)(715.0/3);
```

- Sinnvoll, wenn Anzahl und Werte von Elementen im Quelltext bekannt sind

Elementzugriff

- Elemente eines Arrays folgen linear aufeinander
- Jedes Element hat ganzzahligen **Index**
- Index des ersten Elementes = 0, dann fortlaufend weiter
- Index des letzten Elementes = (Arraylänge - 1)
- Beispiel: 5 Elemente mit Index 0 bis 4



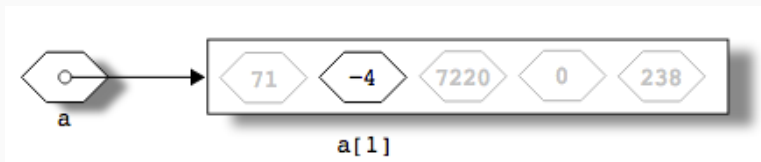
- Zugriff auf alle Element ungefähr gleich schnell = random access

Elementzugriff (1/2)

- Ansprechen eines einzelnen Arrayelements mit Index
- Schema für „Array-Elementzugriff“:

```
array[expression]
```

- Index zur Laufzeit berechnet aus `int`-Ausdruck `expression`
- Beispiel: Zugriff auf das zweite Element von Array `a` `a[1]`



- Zugriff auf ein Element berührt die anderen Elemente des Arrays nicht

- Arrayelement benutzbar wie gewöhnliche Variable des Elementtyps
- Beispiele:

```
int[] a = new int[5];  
a[1] = -3;  
a[3] = 0;  
a[1]-;  
a[152%3] = -a[1]*1805;  
a[a[3]] = 71;  
a[a[1]*-1] = 167 + a[a[2]%7220];
```

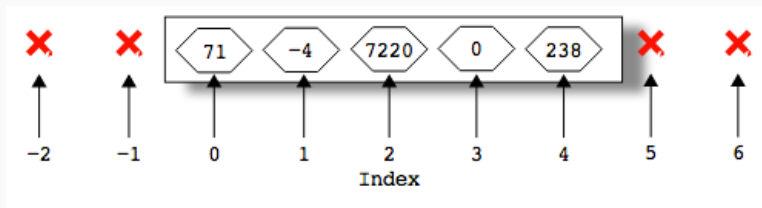
- Unzulässige Indexwerte werfen

`ArrayIndexOutOfBoundsException`

```
int[] a = new int[5];
```

```
a[5] = 23; // Indexfehler
```

- Negativer Index immer unzulässig

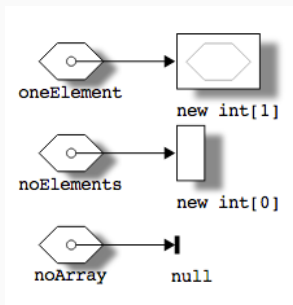


- JVM prüft zur Laufzeit jeden Array-Elementzugriff

Kurze Arrays

- Anzahl Elemente eines Arrays (konzeptionell) beliebig
- Schließt Arrays mit einem und keinem Element ein:

```
int[] oneElement = new int[1];  
int[] noElements = new int[0];  
int[] noArray = null;
```



- Ein Array ohne Elemente ist ein Objekt, `null` ist keines
- Vermeidet Sonderbehandlungen, wenn die Arraylänge automatisch bestimmt wird und 0 bzw. 1 nicht ausgeschlossen werden können

- Eckige Klammern syntaktisch in verschiedenen Kontexten

Arraytyp	<code>int[]</code>	Arraytyp
Array erzeugen	<code>new int[5]</code>	Ausdruck, Arraytyp
Array-Literal	<code>new int[] {1, 2, 3}</code>	Ausdruck, Arraytyp
Elementzugriff	<code>a[1]</code>	Ausdruck, Elementtyp

- Beispiel

```
int[] a;           // Arraytyp
a = new int[5];    // Array erzeugen
a = new int[] {1, 2, 3}; // Array-Literal
a[1] = 1;         // Elementzugriff
```

- Anzahl Elemente als öffentlich lesbare `final`-Objektvariable `length`
- Zugriff wie Objektvariablen:

```
array.length
```

- Beispiel:

```
int[] a = new int[] {71, -4, 7220, 0, 238};  
System.out.println(a.length); // gibt „5“ aus
```

- Array-Ausdruck = Ausdruck mit Arraytyp
- Elementzugriff über beliebigen Arrayausdruck, nicht nur isolierte Arrayvariable
- Beispiele:
 - Rückgabewert

```
int[] getArray() {...}  
...  
getArray()[2] = 23;
```

- Bedingter Operator

```
int[] fst = ...;  
int[] snd = ...;  
(x > 0 ? fst : snd)[2] = 23;
```

- Öffentliche Objektvariable mit Arraytyp

```
class Foo {  
    int[] elements;  
}  
  
...  
Foo foo = ...;  
foo.elements[2] = 23;
```

char-Arrays vs. Strings

- char-Arrays \neq Strings

- Umwandlung

- String \Rightarrow char-Array:

```
String s = ...;  
char[] a = s.toCharArray();
```

- char-Array \Rightarrow String:

```
char[] a = ...;  
String s = new String(a);
```

- Klasse **String** verwendet intern char-Array
- String = Luxusversion eines char-Array: einfacher, effizienter, bequemer, aber immutable

foreach-Schleifen

Motivation

- Sequentieller Array-Durchlauf = Elemente von vorne nach hinten der Reihe nach verarbeiten
- Beispiel mit **for**-Schleife:

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

- Indexvariable **i** nur zur Elementauswahl, nichts sonst
- Allgemein

```
for (int i = 0; i < a.length; i++) {
    T e = a[i];
    // e verwenden, aber nicht i
}
```

- Einfacher: **foreach**-Schleife
- Schema:

```
for (type variable: array)  
    statement
```

- Beispiel:

```
for (int e: a)  
    System.out.println(e);
```

- Kurzform einer **for**-Schleife für bestimmten Zweck
- Anwendbar auf weitere Datenstrukturen
 - bereits kennengelernt mit **List**, **Set**, **Map**

Äquivalenz zur **for**-Schleife

- foreach-Schleife ersetzbar durch **for**-Schleife (T = Elementtyp des Arrays):

```
for (T e: a) {  
    ...  
}
```

äquivalent zu

```
for (int i = 0; i < a.length; i++) {  
    T e = a[i];  
    ...  
}
```

- Neue Schleifenvariable **e** in jedem Durchgang

- Nur Lesen, kein Schreiben des Arrays
- Start immer mit erstem Element
- Sequentieller Durchlauf, keine Sprünge
- Nur ein Array, nicht mehrere parallel
- Durchlauf vorzeitig abbrechen nur mit **break**

- foreach geeignet für beispielsweise ...
 - Ausgabe von Elementen
 - Suche nach Element
 - Änderungen in Elementen
- Nicht brauchbar für
 - Initialisierung
 - Kopieren von Arrays
 - Vergleich zweier Arrays
 - Austausch von Elementen

Varargs

Definition

- Bisher feste Anzahl Argumente bei Methodenaufrufen
- Mit **Varargs** (variable length argument lists) veränderliche Anzahl Argumente
- Methodendefinition mit **Vararg-Parameter**
- Vararg-Parameter syntaktisch markiert mit Typ und drei Punkten:

```
type ... name
```

- Beispiel:

```
int sum(int ... args) {  
    ...  
}
```


Vararg-Parameter im Rumpf

- Varargs ausschließlich in Parameterlisten erlaubt, nirgends sonst
- Varargs-Parameter im Methodenrumpf verwendbar wie ein Array
- Beispiel:

```
int sum(int ... args) ist im Rumpf der Methode gleichwertig mit  
int sum(int[] args)
```

- Beispiel: Addition aller Argumente

```
int sum(int ... args) { // verwendbar wie: int[] args  
    int s = 0;  
    for(int i: args)    // alle Argumente addieren  
        s += i;  
    return s;          // Summe zurückliefern  
}
```

- Aufrufer liefert beliebig viele Argumente für einen Vararg-Parameter

```
sum(1, 2, 3)
```

- Jedes einzelne Argument muß kompatibel zum Vararg-Parameter sein
- Bei der Parameterübergabe:
 1. Allokieren eines neuen Arrays mit Länge = Anzahl Argumente
 2. Initialisieren des Arrays mit Argumentwerten
 3. Zuweisen des Arrays an den Vararg-Parameter

- Beispiel:

```
System.out.println(sum(1, 2, 3));           // Ausgabe „6“  
System.out.println(sum());                 // Ausgabe „0“  
System.out.println(sum(1, sum(2, 3)));     // Ausgabe „6“
```

- Statt einer Argumentliste kann der Aufrufer als Argument ein komplettes Array übergeben

```
int[] a = new int[] {1, 2, 3};  
System.out.println(sum(a));           // Ausgabe „6“
```

- Im Rumpf der Methode nicht unterscheidbar
- An einen Vararg-Parameter kann ...
 - entweder ein Argument-Array
 - oder eine Liste von einzelnen Argumenten

Arrayargumente (2/3)

übergeben werden, aber keine Mischung von beiden

```
int[] a = new int[] {1, 2, 3};  
System.out.println(sum(4, 5, 6)); // Ok, einzelne Argumente  
System.out.println(sum(a)); // Ok, Argument-Array  
System.out.println(sum(a, 4, 5, 6)); // Fehler!
```

- Zweck: Vararg-Parameter kann weitergegeben werden:

```
int printSum(int ... args) {  
    for (int i: args)  
        System.out.println(i);  
    return sum(args); // Vararg an Vararg  
}
```

- `null` als Argument syntaktisch korrekt, aber Laufzeitfehler:

```
System.out.println(sum(null)); // NullPointerException
```

- Einschränkungen:
 - nur ein Vararg-Parameter
 - Vararg-Parameter letzter in der Parameterliste
- Vorausgehende Parameter werden normal behandelt

- Beispiel:

```
boolean inRange(int low, int high, int ... values) {  
    boolean result = 0;  
    for (int x: values) {  
        if (x < low || x > high) {  
            result = false;  
        }  
    }  
    return result;  
}
```

- Mindestens zwei Argumente beim Aufruf:
`inRange(5, 10, 6, 10, 8) → true`
`inRange(5, 10, 6, 12, 8) → false`
`inRange(5, 10) → true`
- Formatierte Ausgabe und `String.format` benutzen Vararg-Parameter

- Per Definition: Alle Methoden ohne Varargs „passen genauer“ als Methoden mit Varargs
- Beispiel: Definitionen

```
void foo(double d)
void foo(int ... i)
```
- Aufruf von ...

```
foo(1)      - foo(double) passt ohne Varargs
foo(1, 2)   - foo(int ... ) passt alleine
```
- Überladen mit Vararg- und Arrayparameter nicht zulässig:

```
int sum(int ... args)
int sum(int[] args)      - Fehler!
```

- Überladen mit Vararg-Parameter an verschiedenen Positionen ok:

```
int sum(int ... args)
```

```
int sum(int arg0, int ... args)
```

Aufrufer muss Array-Argument übergeben, sonst mehrdeutig:

```
sum(1, 2, 3) - Fehler, mehrdeutig
```

```
sum(1) - Fehler, mehrdeutig
```

```
sum(1, new int[] {2, 3}) - ok
```

```
sum(1, null) - ok (aber in diesem Bsp
```

```
NullPointerException)
```

Geschachtelte Arrays

- Zu jedem Javatypt ein korrespondierender Arraytyp
- Auch für Arrays selbst: **geschachtelte Arrays** (auch „zweidimensionale Arrays“)

- Typangabe:

```
type[][]
```

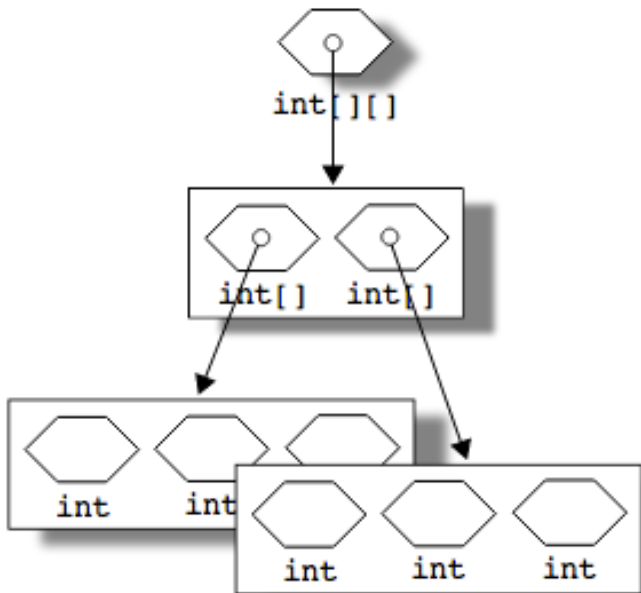
- Beispiel: Matrix m:

```
int[][] m;
```

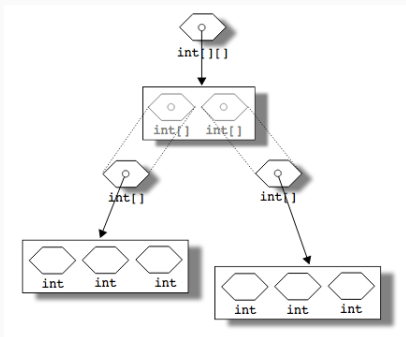
- Allokieren mit **new** + Anzahl Elemente in jeder Dimension

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

erzeugt ein Array mit 2 Elementen, von denen jedes ein Array mit 3 `int`-Elementen ist



- Innere Struktur aus zwei „Ebenen“; Auf jeder Ebene gewöhnliche, eindimensionale Arrays



- Array der ersten Ebene bleibt i.d.R. unsichtbar

- Ein Index für jede Dimension. Beispiel:

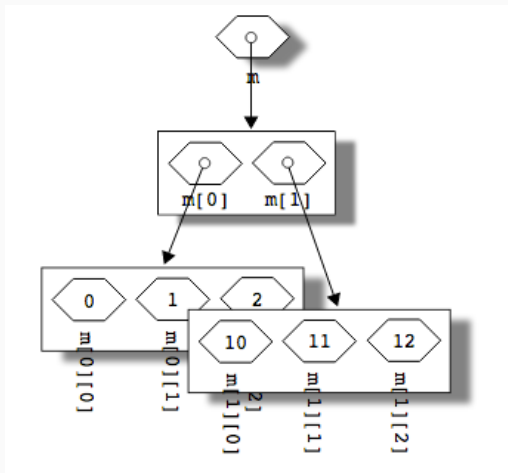
```
m[1][0] = 10;
```

Erster Index [1] für das Array der ersten Ebene,
zweiter Index [0] für das Array der zweiten Ebene

- Beispiel: Array auffüllen


```
int[][] m = new int[2][3];  
m[0][0] = 0;  
m[0][1] = 1;  
m[0][2] = 2;  
m[1][0] = 10;  
m[1][1] = 11;  
m[1][2] = 12;
```

- Speicherstruktur:



Iteration über die Elemente (1/2)

- Geschachtelte Schleifen zum Durchlauf (**while**, **for**, **foreach**)
- Beispiel mit **for**-Schleifen:

```
int[][] m = ...;
for (int x = 0; x < m.length; x++) {
    for (int y = 0; y < m[x].length; y++) {
        System.out.println(m[x][y]);
    }
}
```

- Ausdruck `m.length` = Anzahl der Elemente des Arrays auf erster Ebene

- Äquivalent: foreach-Schleifen ohne Index

```
int[][] m = ...;
for (int[] a: m) {
    for (int e: a) {
        System.out.println(e);
    }
}
```

Initialisierung zweidimensionaler Arrays

- Zweidimensionale Arrayliterale: Liste von Listen
- Schema:

```
new type[][] {  
    {expression00, expression01, ..., expression0n},  
    {expression10, expression11, ..., expression1n},  
    {expression20, expression21, ..., expression2n},  
    ...  
    {expressionm0, expressionm1, ..., expressionmn}  
}
```

- Beispiel:

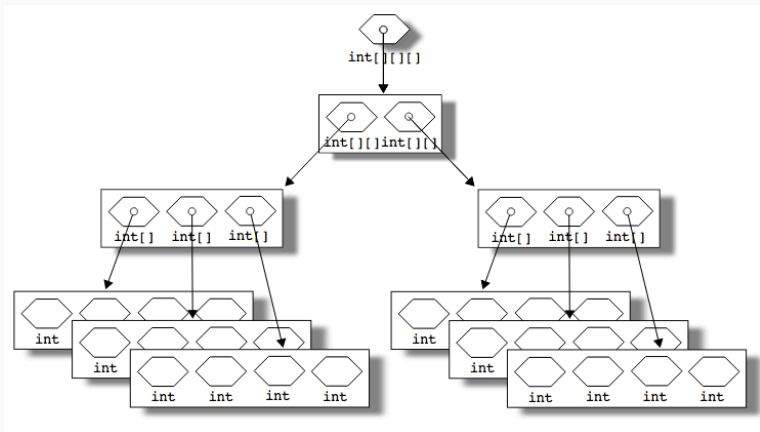
```
int[][] m = new int[][] {  
    { 0, 1, 2},  
    {10, 11, 12}  
};
```

- Arrays mit drei oder mehr Dimensionen entsprechend zweidimensionalen Arrays
- Beispiel: „quaderförmiges“ Array mit 2·3·4 Elementen:

```
int[][][] q = new int[2][3][4];
```

Mehrdimensionale Arrays (2/4)

- Organisation in mehreren Ebenen:



- Elementzugriff: Ein Index pro Dimension

```
for (int x = 0; x < q.length; x++) {  
    for (int y = 0; y < q[x].length; y++) {  
        for (int z = 0; z < q[x][y].length; z++) {  
            q[x][y][z] = 100*x + 10*y + z;  
        }  
    }  
}
```


- Zugriff ohne Index mit foreach-Schleifen:

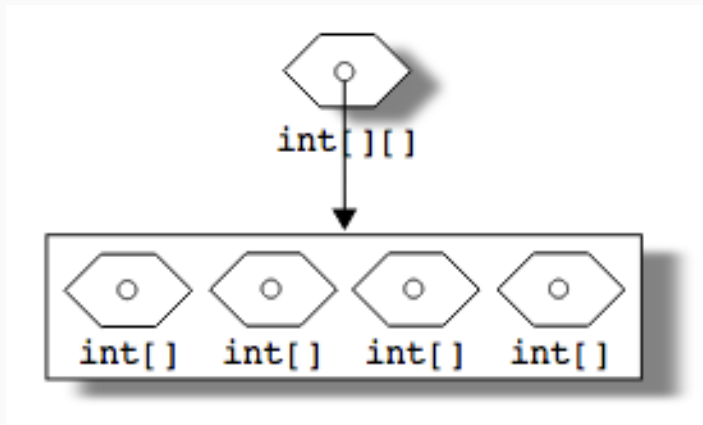
```
for (int[][] m: q) {  
    for (int[] a: m) {  
        for (int e: a) {  
            System.out.println(e);  
        }  
    }  
}
```

Nicht-rechteckige Arrays (1/6)

- Arrays auf verschiedenen Ebenen = unabhängige Elemente übergeordneter Arrays
- **new** mit mehreren Dimensionen erzeugt auf jeder Ebene automatisch gleich lange Arrays (siehe Beispiel)
- Expliziter Aufbau eines nicht-rechteckigen Arrays
 1. Erste Ebene allokiieren:

```
int[][] triangle = new int[4][];
```

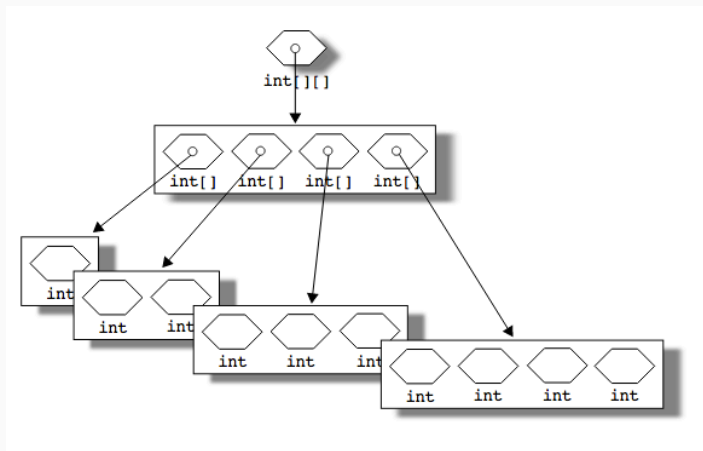
Nicht-rechteckige Arrays (2/6)



2. Arrays auf zweiter Ebene mit Längen 1 bis 4 einzeln erzeugen und zuweisen

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

Nicht-rechteckige Arrays (4/6)



Nicht-rechteckige Arrays (5/6)

- Arrayliteral mit unterschiedlich langen Elementlisten:

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

- Elementzugriff: Länge der inneren Schleife (`triangle[x].length`) bei jedem Durchlauf anders:

```
for (int x = 0; x < triangle.length; x++) {  
    for (int y = 0; y < triangle[x].length; y++) {  
        System.out.println(triangle[x][y]);  
    }  
}
```

- Einfacher mit foreach-Schleifen:

```
for (int[] a: triangle) {  
    for (int e: a) {  
        System.out.println(e);  
    }  
}
```

Kopieren und Vergleichen

- Arrays haben Referenzsemantik
- Wertzuweisung kopiert Referenz, nicht Array, nicht Elemente

```
int[] src = new int[] {71, -4, 7220, 0, 238};  
int[] dst = src;
```

- Änderungen über eine Variable (`src` oder `dst`) in beiden sichtbar (`src` und `dst`, „Aliasing“):

```
src[0] = 23;  
System.out.println(dst[0]);    // gibt 23 aus
```

- Kopie eines Arrays mit primitiven Elementen:
 1. neues Array allokalieren
 2. Original elementweise übertragen

```
int[] src = new int[] {71, -4, 7220, 0, 238};
```

```
// 1.)
```

```
int[] dst = new int[src.length];
```

```
// 2.)
```

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

- Zwei unabhängige Arrays, können einzeln und unabhängig manipuliert werden:

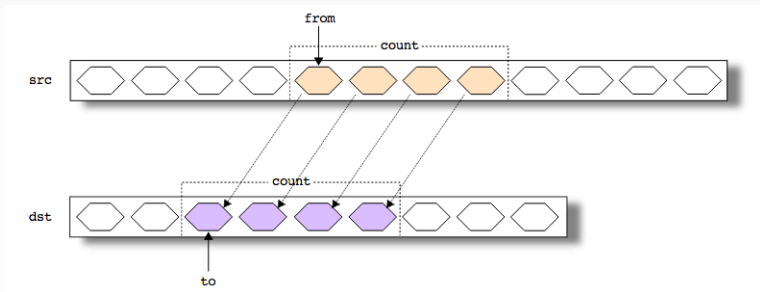
```
src[0] = 23;  
System.out.println(dst[0]);    // gibt 71 aus
```

- Vordefiniert zum Kopieren von Arrays: statische Methode **arraycopy** in Klasse **System**
- Argumente:

src	Original-Array, wird gelesen
from	Index des ersten Elementes in src , das kopiert werden soll
dst	Ziel-Array, wird geschrieben (gleicher Elementtyp wie src)
to	Index in dst , ab dem geschrieben wird
count	Anzahl der Elemente

Methode `System.arraycopy` (2/3)

- Skizze



- Beispiel mit `arraycopy`:

```
int[] src = new int[] {71, -4, 7220, 0, 238};  
int[] dst = new int[src.length];  
System.arraycopy(src, 0, dst, 0, src.length);
```

- `arraycopy` arbeitet korrekt mit überlappenden Bereichen im selben Array
- Beispiel: die vorderen drei Elemente an das Ende kopieren:

```
int[] src = new int[] {71, -4, 7220, 0, 238};  
System.arraycopy(src, 0, src, 2, 3);
```

- `arraycopy` benutzt Wertzuweisungen

- Kopierschleife mit Wertzuweisungen erzeugt flache Kopie, ebenso wie `System.arraycopy`
- Ausreichend bei Wertesemantik des Elementtyps (primitive und unveränderliche Typen)
- Unzureichend für Arrays mit (veränderlichen) Objekten als Elementen

```
Rational[] src = ...  
Rational[] dst = new Rational[src.length];  
for (int i = 0; i < src.length; i++) {  
    dst[i] = src[i];  
}
```

- `src` und `dst` referenzieren die selben Objekte

- Arrays definieren `Object.clone`
- `clone` von Arrays erzeugt Kopie des Original-Arrays
- Kopiert Elemente einzeln per Wertzuweisung = flache Kopie
- Beispiel:

```
int[] src = new int[] {71, -4, 7220, 0, 238};  
int[] dst = src.clone();
```


Verfahren für flache Kopie (1/2)

- Gleiches Ergebnis bei allen drei Verfahren
- Kopierschleife

```
int[] src = ...  
int[] dst = new int[src.length];  
for (int i = 0; i < src.length; i++) {  
    dst[i] = src[i];  
}
```

- System.arraycopy

```
int[] src = ...  
int[] dst = new int[src.length];  
System.arraycopy(src, 0, dst, 0, src.length);
```

- clone

```
int[] src = ...  
int[] dst = src.clone();
```

- Tiefe Kopie: Erst ganzes Array, dann Elemente einzeln duplizieren
- Beispiel mit Kopier-Konstruktor

(ignoriert dynamischen Elementtyp):

```
Rational[] src;  
...  
Rational[] dst = new Rational[src.length];  
for (int i = 0; i < src.length; i++) {  
    dst[i] = new Rational(src[i]);  
}
```

- Beispiel mit `clone`:

```
Rational[] src;  
  
...  
Rational[] dst = new Rational[src.length];  
for (int i = 0; i < src.length; i++) {  
    dst[i] = src[i].clone();  
}
```

- Vergleich mit `=` liefert Aussage über Identität, nicht Gleichheit
- Arrays erben `Object.equals`, überprüft lediglich Identität:

```
int[] a = new int[] {1, 2};  
int[] b = new int[] {1, 2};  
System.out.println(a.equals(b)); // false
```

- `equals` wäre zu redefinieren - nicht möglich bei Arrays

- Inhaltlicher Vergleich von Arrays: Erst Längen vergleichen, dann paarweise Elemente
- Elementvergleich mit \equiv ausreichend bei Typen mit Wertesemantik
- Beispiel: primitive Arrays

Paarweiser Elementvergleich (2/3)

```
int[] a = ...;
int[] b = ...;
// Längen vergleichen
boolean eq = (a.length == b.length);
// Elemente vergleichen
for(int i = 0; eq && i < a.length; i++) {
    eq = (a[i] == b[i]);
}

System.out.println(eq ? "equal" : "not equal");
```

- Bei Elementen von Referenztypen: Elemente paarweise mit `equals` vergleichen

- Beispiel: Rational-Arrays

...

```
for (int i = 0; eq && i < a.length; i++) {  
    eq = a[i].equals(b[i]);  
}
```

...

- Statische Methode `Arrays.equals` vergleicht ...
 - die Längen der Arrays
 - falls gleiche Länge, die Elemente paarweise mit ...

<code>=</code>	bei primitiven Elementen
<code>equals</code>	bei Referenztypen

- `Arrays.equals` redefiniert nicht `Object.equals`!
- Ausreichend für eindimensionale Arrays

Tiefer Vergleich mehrdimensionaler Arrays (1/2)

- `Arrays.equals` ruft `equals` der Elemente auf
- Scheitert bei mehrdimensionalen Arrays: Elemente sind selbst Arrays, diese erben wieder `Object.equals`, dieses vergleicht wieder nur Identität
- Andere Methode `Arrays.deepEquals` berücksichtigt untergeordnete Arrays:

Elementtyp	Vergleich mit ...
primitive Typen	<code>=</code>
Arrays	<code>Arrays.deepEquals</code>
andere Referenztypen	<code>equals</code>

- Beispiel: Inhaltlicher Vergleich auf allen Ebenen bis zu den Elementen

```
int[][][] a = ...;  
int[][][] b = ...;  
if (Arrays.deepEquals(a, b))  
    ...
```

- In `java.util.Arrays` nützliche statische Hilfsmethoden

`void fill(a, x)` a mit Kopien von x füllen

`void sort(a)` a nach aufsteigenden Werten sortieren (Elemente primitiv oder `Comparable`)

`boolean equals(a, b)` Elementweiser Vergleich von a und b (eindimensionale Arrays)

`boolean deepEquals(a, b)` Tiefer Vergleich von a und b (mehrdimensionale Arrays)

`int binarySearch(a, x)` x im sortierten Array a suchen (Elemente primitiv oder `Comparable`)

`String toString(a)` lesbare Darstellung von a als Liste von Elementen

- Arraytypen inkompatibel zu allen anderen Javatypen (außer **Object**)
- Kompatibilität der Arraytypen abhängig vom Elementtyp:

Invarianz bei primitive Elementtypen Arraytypen inkompatibel,
unabhängig von der Kompatibilität der Elementtypen:

```
double d = 1; // ok
double[] a = new int[] {1}; // Fehler
```

Covarianz bei Referenztypen Kompatibilität der Arraytypen „folgt“ der
Kompatibilität der Elementtypen

```
Object x = "one"; // ok
Object[] a = new String[] {"one"}; // ok
```

allgemein:

A[] kompatibel zu B[], wenn

A kompatibel zu B

- Covarianz unterläuft die statische Typprüfung des Compilers!
- Beispiel: Wird übersetzt, scheitert mit Laufzeitfehler:

```
Object[] a = new String[5];  
a[0] = new Rational();           // ArrayStoreException
```

- Unbefriedigend: vollständige statische Typprüfung = wichtiges Entwurfsziel von Java, hier verfehlt