PM2 Java: Arrays

Fahrplan

- Einführung: Definitionen
- Speichermodell
- Deklaration und Initialisierung; Defaultwerte und Arrayliterale
- Elementzugriff und Elementzuweisung
- Typkompatibilität
- Über Arrays iterieren
- Arrays mit Inhalt füllen
- Mehrdimensionale Arrays
- Arrays kopieren: flache (shallow) versus tiefe Kopie (deep copy)
- Arrays vergleichen
- Die Hilfsklasse Arrays

EINFÜHRUNG: DEFINITIONEN

Einführung-Definitionen

- Arrays sind Anordnungen fester Länge von "***gleichartigen***" Objekten. Die Ordnung ist definiert durch die Anordnung der natürlichen Zahlen.

- Die Länge eines Arrays wird bei der Initialisierung festgelegt und kann nicht verändert werden (im Unterschied zu Ruby).

- Zugriff auf die Elemente eines Arrays erfolgt indiziert über die Positionen der Elemente. Die Zählung der Positionen in einem Array beginnt mit 0 und endet mit (Länge-1).

- Indizierter Zugriff außerhalb des Intervall [0,Länge-1] führt immer zu einem Fehler: ***ArrayIndexOutOfBoundsException***

Einführung-Definitionen

- Arrays haben einen **Arraytyp** und einen **Komponententyp**. Der Komponententyp ist der Typ der Elemente eines Arrays. Für den Arraytyp ***int[]*** ist ***int*** der Komponententyp.

- Alle Objekte in einem Array müssen zum Komponententyp kompatibel sein oder in diesen umwandelbar sein (-> siehe **Coercion**).

- Arrays definieren eine **Familie von Typen**, da jeder Typ in Java (auch Arraytypen) Komponententyp sein kann. **Ausnahme**: generische Typen sind als Komponententyp nicht zulässig.

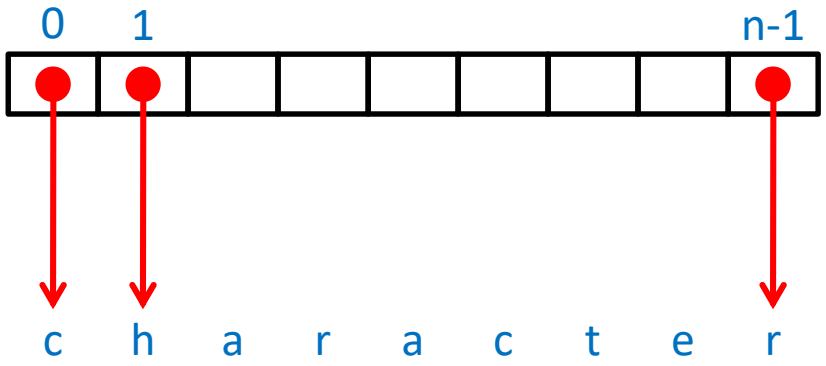
- Arraytypen sind Referenztypen. Basisdatentypen und Referenztypen definieren die Typen in Java.

- Arraytypen verfügen über ein begrenzte Zahl von Methoden und können nicht erweitert werden. (Ruby: hier durfte von Array abgleitet und erweitert werden).

SPEICHERMODELL

Grafik zur Veranschaulichung des Speichermodell

```
char[] cAry = {'c','h','a','r','a','c','t','e','r'};
```



Gegeben ein char Array
char[] cAry = {'c','h','a','r','a','c','t','e','r};
der Länge n=9.
Die Positionen von cAry sind 0 ... 8 (=n-1).
Jede Position ist eine Variable die einen char referenziert.

Default-Werte und Array-Literale

DEKLARATION UND INITIALISIERUNG

Arrays deklarieren

- Arrays werden durch Nennung des Komponententyps und nachgestelltem ***[]*** deklariert.

- Die Komponententypen zu den Arraytypen der rechten Seite sind in der gegebenen Reihenfolge:
  - ***boolean***
  - ***int***
  - ***double***
  - ***float***
  - ***Integer***
  - ***Double***
  - ***Float***
  - ***Person***
  - ***Object***
  - ***int[]***

```
boolean[] bAry;
int[] iAry;
double[] dAry;
float[] fAry;
Integer[] iWrapAry;
Double[] dWrapAry;
Float[] fWrapAry;
Person[] perAry;
Object[] oAry;
int[][] iMatrix;
```

Arrays initialisieren

- Deklarierte Arrays belegen noch keinen Speicher. Sie sind noch nicht initialisiert.
- Zugriff auf ein nicht initialisiertes Array ➔ Compilerfehler.
- Bei der Initialisierung wird Speicher für das Array reserviert.
- Man unterscheidet (1) Initialisierung mit Default-Werten und (2) Initialisierung durch explizite Werte eines Array-Literals

```
p(iAry);      // Compilerfehler iAry
nicht initialisiert
```

Arrays mit Defaults initialisieren

- Bei der Initialisierung von Arrays wird immer der Speicher für das Array festgelegt. Speicher = Speicher für den Komponententyp * Länge des Arrays.
- Initialisierung mit Default-Werten allokiert Speicher und füllt das Array mit den Default-Werten des Komponententyps.
- Initialisierung mit Arrayliteralen allokiert Speicher und belegt die Elemente des Arrays mit den Objekten des Literals.

```
bAry = new boolean[10];
iAry = new int[10];
dAry  = new double[10];
fAry = new float[10];
iWrapAry = new Integer[10];
dWrapAry = new Double[10];
fWrapAry = new Float[10];
perAry = new Person[2];
oAry = new Object[10];
iMatrix = new int[4][6];
```

Arrays mit Defaults initialisieren

- Ausgabe des Inhalts der Array mit der statischen Methode Klasse ***util.Printer.pAry(anAry)*** zeigt z.B. für ***int*** den Default 0, für Referenztypen den Default ***null***.
- **Ausnahme**: Komponententyp Array.

```
pAry("boolean Defaults", bAry);
pAry("int Defaults",iAry);
pAry("double Defaults",dAry);
pAry("float Defaults",fAry);
pAry("Integer Defaults",iWrapAry);
pAry("Double Defaults",dWrapAry);
pAry("Float Defaults", fWrapAry);
pAry("Person Defaults",perAry);
pAry("Object Defaults",oAry);
pAry("int Matrix Defaults", iMatrix);
```

```
boolean Defaults [false,false,false,false,false,false,false,false,false,false]
int Defaults [0,0,0,0,0,0,0,0,0,0]
double Defaults [0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
float Defaults [0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
Integer Defaults [null,null,null,null,null,null,null,null,null,null]
Double Defaults [null,null,null,null,null,null,null,null,null,null]
Float Defaults [null,null,null,null,null,null,null,null,null,null]
Person Defaults [null,null]
Object Defaults [null,null,null,null,null,null,null,null,null,null]
int Matrix Defaults [[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]]
```

Default-Werte nach Komponententyp

Typ	Default-Wert
Referenztyp (außer Arrays)	*null*
Array	Wert des Komponententyps (rekursiv)
boolean	*false*
char	*0* (int-Wert des *char*)
short	*0*
int	*0*
long	*0*
float	*0.0*
double	*0.0*

Arrays mit Array-Literalen initialisieren

- **Array-Literal**: eine Komma-separierte Aufzählung von Objekten oder Werten in geschweiften Klammer {}.
- Initialisierung mit Array-Literalen allokiert Speicher und belegt die Elemente des Arrays mit den Objekten des Literals.
- Initialisierung mit Literalen, die separat von der Deklaration eines Arrays erfolgt, muss immer explizit Speicher für das Liter-al reservieren: siehe ***iAry = new int[ ]....*** im Beispiel rechts.

```
//Initialisierung nach der
   Deklaration falsch
iAry =
   {11,12,13,14,15,16,17,18,19,110};
   // Compiler Fehler


//Initialisierung nach der
   Deklaration korrekt
iAry = new int[]
   {11,12,13,14,15,16,17,18,19,110};
pAry(iAry);
// Initialisierung bei der
   Deklaration korrekt
int[] iAry2 =
   {1,2,3,4,5,6,7,8,9,10};
pAry(iAry2);
```

```
[11,12,13,14,15,16,17,18,19,110]
[1,2,3,4,5,6,7,8,9,10]
```

ELEMENT-ZUGRIFF/-ZUWEISUNG

Elementzugriff

- Zugriff auf Elemente eines Arrays erfolgt über den Index eines Elementes.
- Der Index entspricht einer "Instanz"-Variablen eines Arrays. Indizierter Zugriff ist daher das Auslesen von Werten aus Instanz-Variablen in Arrays.
- Liegt der Index außerhalb des gültigen Bereichs, dann wird ein Laufzeitfehler generiert: ***ArrayIndexOutOfBoundsException***
- Gültiger Bereich: Index >=0 || Index < Länge des Arrays

```
p("Elementzugriff");
char[] cAry =
    {'c','h','a','r','a','c','t','e',
    'r'};
p(cAry[cAry.length-1]);
p(cAry[0]);
p(cAry[cAry.length]);
```

```
Elementzugriff
r
c
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsEx
    ception: 9
at
    arrays.ArrayBasicsDemo.main(Array
    BasicsDemo.java:102)
```

Elementzuweisung

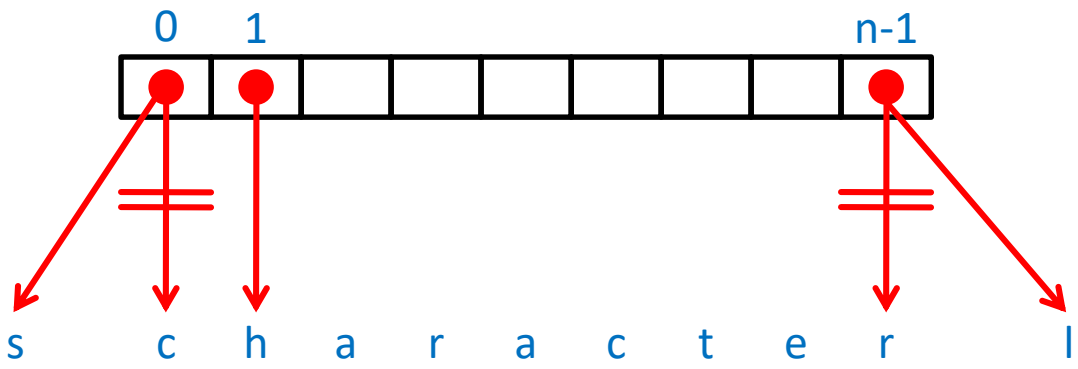
- Elementzuweisung weist einer "Instanz"-Variablen eines Arrays über indizierten Zugriff einen neuen Wert zu.
- Die Variable zeigt nach der Zuweisung auf diesen neuen Wert.
- Auch hier gilt: liegt der Index außerhalb des gültigen Bereichs, dann wird ein Laufzeitfehler generiert: ***ArrayIndexOutOfBoundsException***

```
p("Elementzuweisung");
cAry[0] = 's';
cAry[cAry.length-1] = 'l';
pAry(cAry);
cAry[-1] = 'x';
```

```
Elementzuweisung
[s,h,a,r,a,c,t,e,l]
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsEx
    ception: -1
at
    arrays.ArrayBasicsDemo.main(Array
    BasicsDemo.java:109)
```

Elementzuweisung graphisch erläutert

```
cAry[0] = 's';
cAry[cAry.length-1] = 'l';
```



TYPKOMPATIBILITÄT

Typkompatibilität von Arraytypen

1. Ein Array, dessen Komponententyp ein Basisdatentyp ist, ist **nur** zu Arrays desselben Komponententyps kompatibel.

2. Ein Array, dessen Komponententyp ein Basisdatentyp ist, ist **nicht** zu Arrays mit den korrespondierenden Wrappertypen kompatibel.

3. Für Arrays, deren Komponententyp ein Referenztyp ist, gilt: wenn ***A < B*** dann ist ***A[]*** kompatibel zu ***B[]***.

4. Für Arrays, deren Komponententyp ein Array ist, gelten die obigen Regeln.

5. Alle Arrays sind zu ***Object*** kompatibel.

1.)

```
// iAry = iWrapAry;  // Compilerfehler
// oAry = iAry;      // Compilerfehler
// dAry = iAry;      // Compilerfehler
// dAry = fAry;      // Compilerfehler
```

2.)

```
// iWrapAry = iAry;  // Compilerfehler
// iAry = iWrapAry;  // Compilerfehler
```

3.)

```
oAry = perAry; //Typ von oAry: Object[]
oAry = iWrapAry;
oAry = dWrapAry;
```

4.)

```
Person[][] perMatrix = new Student[3][3];
    // ok
// Integer[][] intMatrix = new int[3][3];
    // Compilerfehler
// int[][] iMatrix = new short[3][3];
    // Compilerfehler
```

Alle Arraytypen sind zu *Object* kompatibel

- Alle Arraytypen sind zu ***Object*** kompatibel, da sie von ***Object*** abgeleitet sind.
- Alle Arrays kennen daher die Methoden, die in ***Object*** definiert sind (z.B. ***equals, hashCode, toString, clone***) (***dazu später mehr***)

```
Object o;
o = iAry;
pAry(o);
o = iWrapAry;
pAry(o);
o = oAry;
pAry(o);
o = perAry;
pAry(o);
o = new int[][]{{1,2},{3,4}};
pAry(o);
```

```
Alle Arrays sind  zu Object kompatibel
[11,12,13,14,15,16,17,18,19,110]
[null,null,null,null,null,null,null,null,null,null]
[null,null,null,null,null,null,null,null,null,null]
[null,null,null,null,null,null,null,null,null,null]
[[1,2],[3,4]]
```

Typkompatibilität bei Elementzuweisung

1. In einem Array, dessen Komponententyp ein Basisdatentyp ***B*** ist, dürfen Werte vom Typ ***A*** zugewiesen werden, wenn Werte vom Typ ***A*** in Typ ***B*** „passen“. (→ Coercion)
2. In einem Array, dessen Komponententyp ein Basisdatentyp ist, dürfen Werte des korrespondierenden Wrappertypen zugewiesen werden, und umgekehrt.
3. In einem Array, dessen Komponententyp ein Referenztyp ***B*** ist, dürfen Objekte vom Typ ***A*** zugewiesen werden, wenn gilt, dass ***A < B***. Für Werte von Basisdatentypen greift das ***Autoboxing***.
4. In einem Array, dessen Komponententyp ein Array ist, gelten die Kompatibilitätsregeln für Arrays, wenn ein Array zugewiesen wird, sonst die Kompatibilitätsregeln für die Elementzuweisung.

```
1.  iAry[0] = (short)89;
    dAry[0] = 12.34f;

2.  iAry[0] = new Integer(4);
    iWrapAry[0] = 4;

3.  oAry = new Object[3];
    oAry[0] = new Person("Donald",
    "Knuth");
    perAry[1] = new Student("Donald",
    "Knuth",1111111);
    oAry[1] = new Integer(1);
    oAry[2] = 4;

4.  perMatrix[0] = new Student[] {new
    Student("Donald",
    "Knuth",1111111),null,null};
    Integer[][] intMatrix = new
    Integer[3][3];
    intMatrix[0][0] = 67;
    //intMatrix[0] = new int[]{1,7,9};
                        // Compilerfehler
```

Ungeschützte Kovarianz von Arrays

- **Kovarianz** beschreibt die Typkompatibilität zwischen Arrays, die sich aus der Kompatibilität der Komponententypen ableitet.
- ***A < B => A[] < B[],*** sprich aus ***A*** kompatibel zu ***B*** folgt ***A[]*** kompatibel zu ***B[]***.
- Der Quelltext auf der rechten Seite ist ein Beispiel für die ungeschützte Kovarianz von Arrays.
- Die Typverletzung in der letzten Zeile, das Einfügen eines ***int*** in ein ***Object*** Array, das real aber ein ***String-Array*** ist, ist nach den Compiler-Regeln ok, erzeugt jedoch zur Laufzeit den Fehler: ***ArrayStoreException.***

```
p("Ungeschützte Kovarianz");
Object[] oa = new String[3];
oa[0] = 4;
```

```
Exception in thread "main"
    java.lang.ArrayStoreException:
    java.lang.Integer
at
    arrays.ArrayBasicsDemo.main(ArrayBa
    sicsDemo.java:140)
```

ITERIEREN

Formen der Iteration

1. Mit dem ***for each*** Konstrukt:
   - Geeignet um Inhalte eines Arrays zu untersuchen z.B. um diese auszugeben.
   - **keine** Änderung des Inhalts des Arrays über eine lokale Variable im ***for each*** möglich.

2. Indiziert mit ***for***: Geeignet, um
   - Inhalte zwischen Arrays zu kopieren oder
   - Inhalte von zwei Arrays zu vergleichen.
   - etc...

Iterieren mit for each ändert den Inhalt eines Arrays nicht

- Die neuen Werte, die der Variable ***obj*** im ***for each*** zugewiesen werden, sind **keine** Elementzuweisung auf das Array ***oAry,*** da ***obj*** eine lokale Variable des ***for each*** Konstruktes ist.

- ***oAry*** ist daher nach der Iteration ***unverändert***.

```
pAry(oAry);
for (Object  obj : oAry) {
     p(obj);
     obj = new Person("Never In", "oAry");
}
pAry(oAry);
```

```
[P(Donald,Knuth),1,4]
P(Donald,Knuth)
1
4
[P(Donald,Knuth),1,4]
```

Iterieren mit *for* und Index zum Kopieren von Arrays

- Sollen Werte in einem Array während des Iterierens geändert werden, dann müssen die Werte durch Elementzuweisung geändert werden.

- Dies geht nur durch Verwendung eines Index in einer Fortschaltanweisung mit ***for.***

```
p("Inhalt eines Arrays in ein anderes kopieren");
pAry("iAry",iAry);
for (int i = 0; i < iAry2.length; i++) {
    iAry2[i] = iAry[i];
}
pAry("iAry2",iAry2);
```

```
iAry [4,12,13,14,15,16,17,18,19,110]
iAry2 [4,12,13,14,15,16,17,18,19,110]
```

Iterieren mit *for* und Index für den Vergleich von Arrays

- Sollen die Inhalte zweier Arrays verglichen werden, dann müssen immer die Elemente auf den gleichen Positionen verglichen werden.

- Dies geht nur durch Verwendung eines Index in einer Fortschaltanweisung mit ***for.***

```
private static void compareCharArys(char[] ary1, char[] ary2) {
   int min = ary1.length <= ary2.length ? ary1.length : ary2.length;
   for(int i = 0; i <min ; i++ ) {
      if (ary1[i] < ary2[i]) { p("ary1 <=> ary2: " + -1); return;}
      if (ary1[i] > ary2[i]) { p("ary1 <=> ary2: " + 1); return;}
      if (i == min && ary1.length == min) {p("ary1 <=> ary2: " + -1); return;}
      if (i == min && ary2.length == min) {p("ary1 <=> ary2: " + 1); return;}
   }
   p("ary1 <=> ary2: " + 0);
}
```

Beispiel: Vergleich von Arrays: Ordnung auf char[] Arrays

```
char[] cAry1 = {'N','e','u','m','a','n','n'};
char[] cAry2 = {'N','e','u','m','a','r','k','t'};
compareCharArys(cAry1,cAry2);
cAry1 = new char[] {'N','e','u','m','a','r','k','t'};
compareCharArys(cAry1,cAry2);
cAry1 = new char[] {'N','e','u','w','a','r','k','t'};
compareCharArys(cAry1,cAry2);
```

```
Vergleich von Arrays: Beispiel Ordnung auf char[] Arrays
ary1 <=> ary2: -1
ary1 <=> ary2: 0
ary1 <=> ary2: 1
```

ARRAYS BEFÜLLEN

Arrays mit Inhalt füllen

1. über Arrayliterale (✓)
2. durch Iterieren mit ***for*** und Index (✓)
3. für ***char[]*** mit der ***toArray***, ***getChars*** Methode von ***String*** und ***StringBuffer***
4. mit der ***toArray*** Methode der ***Collection*** Klassen. (***später dazu mehr***)

Arrays mit Inhalt füllen

```
p("3'tens mittels der toArray(), getChars() Methode von String/StringBuffer");
String s = "In the computer business, soon means the same thing as manana in
Spanish, but without the same kind of urgency.";

cAry = s.toCharArray();
pAry(cAry);


char[] cAry3 = new char[s.length()];
s.getChars(10, 15, cAry3, 0);
pAry(cAry3);
```

```
3'tens mittels der toArray(),  getChars() Methode von String StringBuffer
[I,n, ,t,h,e, ,c,o,m,p,u,t,e,r, ,b,u,s,i,n,e,s,s,,, ,s,o,o,n, ,m,e,a,n,s, ...
[p,u,t,e,r,e, ,c,o,m,p,u,t,e,r, ,b,u,s,i,n,e,s,s,,, ,s,o,o,n, ,m,e,a,n,s, ...
```

Arrays mit Inhalt füllen

```
p("4'tens: toArray Methode der Collection Klassen SPÄTER GENAUER!!!!");
List<String> ls = new ArrayList<String>();
String[] sAry;
ls.add("one");
ls.add("two");
ls.add("three");
sAry = new String[ls.size()];
sAry = ls.toArray(new String[0]);
pAry(sAry);
```

```
4'tens: toArray Methode der Collection Klassen SPÄTER GENAUER!!!!
[one,two,three]
```

MEHRDIMENSIONALE

Mehrdimensionale Arrays

- Arrays sind Typen in Java und können selber wieder Komponententyp sein.
- Dadurch entstehen mehrdimensionale Arrays.
  - mit 2-dimensionalen Arrays lassen sich Matrizen darstellen.
  - mit 3-dimensionalen Arrays lassen sich z.B. Würfel darstellen.
- Der Elementzugriff über den ersten Index eines n-dimensionalen Arrays liefert ein Array der Dimension n-1.
- Elementzugriff über die ersten beiden Indizes eines n-dimensionalen Arrays liefert ein Array der Dimension n-2. etc...

1.) Matrizen = 2-dim Arrays

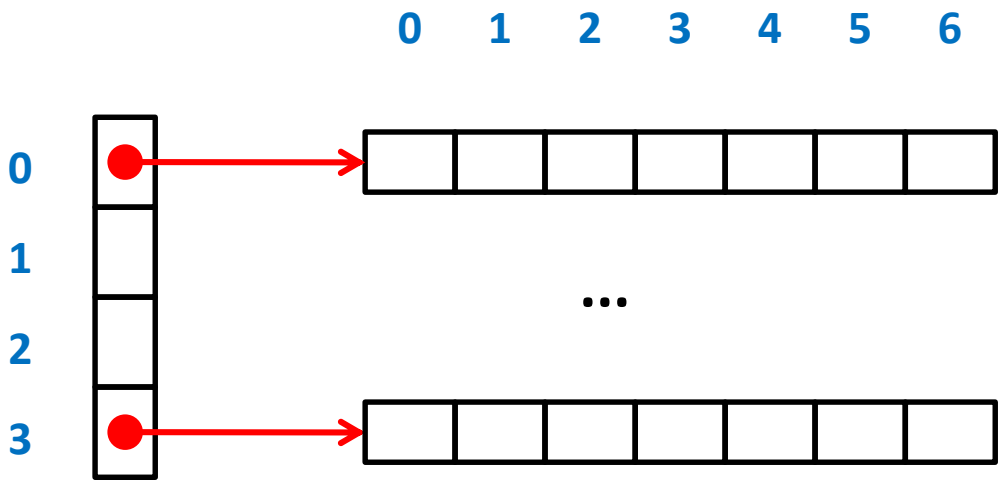
```
int[][] matrix1 = new int[4][7];
int[][] matrix2 = new int[7][5];
int[][] matrix3 = new int[5][4];
```

2.) Würfel = 3-dim Arrays

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

Speichermodell einer Matrix (vereinfacht)

```
int[][] matrix1 = new int[4][7];
```



Der erste Index entspricht den Zeilen einer Matrix. Jede Zeilenvariable zeigt auf ein Array der Länge 7, die Anzahl der Elemente, die im zweiten Index, dem Spaltenindex, festgelegt ist.

Elementzugriff in mehrdimensionalen Arrays

```
int[][] matrix1 = new int[4][7];
int[][] matrix2 = new int[7][5];
int[][] matrix3 = new int[4][5];
matrix1 = new int[][]
    {{11,12,13,14,15,16,17},{21,22,23,24,25,26,27},{31,32,33,34,35,36,37},{4
    1,42,43,44,45,46,47}};
pAry("matrix1:", matrix1);
pAry("matrix1[0]:",matrix1[0]);
pAry("matrix1[0][1]:",matrix1[0][1]);
```

```
matrix1:
    [[11,12,13,14,15,16,17],[21,22,23,24,25,26,27],[31,32,33,34,35,36,37],[4
    1,42,43,44,45,46,47]]
matrix1[0]: [11,12,13,14,15,16,17]
matrix1[0][1]: 12
```

Elementzugriff in mehrdimensionalen Arrays

```
int[][][] cube = new int[3][3][3];
for(int i=0; i< cube.length; i++){
    for(int j=0; j< cube[0].length; j++){
          for(int z=0; z<cube[0][0].length; z++){
                    cube[i][j][z] = Integer.parseInt(""+ (i+1) +(j+1)+(z+1));
          }
    }
}
pAry("cube:", cube);
pAry("cube[0]:", cube[0]);
pAry("cube[0][2]:",cube[0][2]);
pAry("cube[1][1][1]:",cube[1][1][1]);
```

```
cube:
    [[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,2
    33]],[[311,312,313],[321,322,323],[331,332,333]]]
cube[0]: [[111,112,113],[121,122,123],[131,132,133]]
cube[0][2]: [131,132,133]
cube[1][1][1]: 222
```

Matrizen zeilen- und spaltenweise füllen

- Üblicherweise wird der erste Index einer 2-dimensionalen Arrays als Zeile und der zweite als Spalte einer Matrix interpretiert.

- Sollen Matrizen zeilenweise gefüllt werden, so kann jeder Zeile ein Array mit den Spaltenwerten zu gewiesen werden.

- Sollen Matrizen spaltenweise gefüllt werden, muss man über die Spalten und dann die Zeilen iterieren und Elemente in den Spalten einzeln setzen.

Matrix zeilenweise durchlaufen und füllen

```
p("Matrizen mit Werten füllen");
p("Wir weisen den Zeilen einer Matrix Werte zu, indem wir jeder Zeile ein
    eindimensionales Array zuweisen");
p("Dazu iterieren wir über die Zeilen einer Matrix");
int[] sevenInts = {1,2,3,4,5,6,7};
for (int i = 0; i < matrix1.length; i++) {
    matrix1[i] = sevenInts.clone();   // Kopien, um Seiteneffekte zu unterbinden
}
pAry("matrix1: ", matrix1);

int[] fiveInts = {1,2,3,4,5};
for (int i = 0; i < matrix2.length; i++) {
    matrix2[i] = fiveInts.clone();
}
pAry("matrix2: ", matrix2);
```

```
matrix1:  [[1,2,3,4,5,6,7],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7]]
matrix2:
[[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]]
```

Matrix spaltenweise durchlaufen und füllen

```
p("Soll eine Matrix spaltenweise gefüllt werden, muss man die Elemente einzeln in den
    Spalten setzen");
p("Dazu iterieren wir über die Spalten und füllen einzeln die Positionen in den
    Zeilen");
p("si ist der Spaltenindex, zi ist der Zeilenindex");
int[] fourInts = {8,9,10,11};
for (int si = 0; si < matrix1[0].length; si++) {
     for(int zi=0; zi < matrix1.length; zi++) {
          matrix1[zi][si] = fourInts[zi];
          //p("matrix1[" + zi + "][" + si + "]= " + matrix1[zi][si]);
     }
}
pAry("matrix1:",matrix1);
```

```
matrix1: [[8,8,8,8,8,8,8],[9,9,9,9,9,9,9],[10,10,10,10,10,10,10],[11,11,11,11,11,11,11]]
```

Matrixmultiplikation

- Zwei Matrizen, eine (n x m) Matrix m_1 und eine (m x k) Matrix m_2

$$m_1 = (a_{i,j})_{i=1..n;\, j=1..m} = \begin{pmatrix} a_{11} \ldots a_{1m} \\ \ldots \\ a_{n1} \ldots a_{nm} \end{pmatrix} \qquad m_2 = (b_{i,j})_{i=1..m;\, j=1..k} = \begin{pmatrix} b_{11} \ldots b_{1k} \\ \ldots \\ b_{m1} \ldots b_{mk} \end{pmatrix}$$

- werden multipliziert, indem die Summe aus den Produkten der Elemente in den Zeilen von m_1 mit den Elementen der Spalten in m_2 gebildet werden.

- Das Ergebnis ist die (n x k) Matrix m_3

$$m_3 = (c_{ij})_{i=1..n;\, j=1..k} = \begin{pmatrix} \sum_{i=1}^{m} a_{1i} * b_{i1} \ldots \sum_{i=1}^{m} a_{ni} * b_{ik}. \\ \\ \sum_{i=1}^{m} a_{ni} * b_{i1} \ldots \sum_{i=1}^{m} a_{ni} * b_{ik}. \end{pmatrix}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} * b_{kj}$$

Matrixmultiplikation von *matrix1* und *matrix2*

$$m_1 = \begin{pmatrix} 1\,2\,3\,4\,5\,6\,7 \\ 1\,2\,3\,4\,5\,6\,7 \\ 1\,2\,3\,4\,5\,6\,7 \\ 1\,2\,3\,4\,5\,6\,7 \end{pmatrix} \qquad m_2 = \begin{pmatrix} 1\,2\,3\,4\,5 \\ 1\,2\,3\,4\,5 \\ 1\,2\,3\,4\,5 \\ 1\,2\,3\,4\,5 \\ 1\,2\,3\,4\,5 \\ 1\,2\,3\,4\,5 \\ 1\,2\,3\,4\,5 \end{pmatrix}$$

$$m_3 = m_1 \times m_2 = \begin{pmatrix} 28 & 56 & 84 & 112 & 140 \\ 28 & 56 & 84 & 112 & 140 \\ 28 & 56 & 84 & 112 & 140 \\ 28 & 56 & 84 & 112 & 140 \end{pmatrix}$$

$$c_{11} = \sum_{k=1}^{m=7} a_{1k} * b_{k1} = a_{11} * b_{11} + a_{12} * b_{21} + ...a17 * b71$$

$$= 1*1 + 2*1 + 3*1 + 4*1 + 5*1 + 6*1 + 7*1$$

$$= 28$$

$$c_{12} = \sum_{k=1}^{m=7} a_{1k} * b_{k2} = a_{11} * b_{12} + a_{12} * b_{22} + ...a_{17} * b_{72}$$

$$= 1*2 + 2*2 + 3*2 + 4*2 + 5*2 + 6*2 + 7*2$$

$$= 2 * c_{11} = 56$$

Übersetzung der Matrizenmultiplikation in ein Programm

- Um die Summe der Einzelprodukte für ein Element $c_{ij} = \sum_{k=0}^{m-1} a_{ik} * b_{kj}$ in ***matrix₃*** zu berechnen iterieren wir über den Zeilenindex von ***m2.***

```
int cij = 0;
for (int k = 0; k < m2.length; k++) {
        cij += m1[i][k]*m2[k][j]
}
```

*

- Um alle Element einer Spalte von ***m3*** zu berechnen iterieren wir über den Spaltenindex von ***m2***. ***m3*** und ***m2*** haben die gleiche Anzahl an Spalten.

```
for (int j=0; j < m2[i].length; j++) {
  *

  m3[i][j] = cij;
}
```

**

- Um alle Zeilen in ***m3*** zu berechnen iterieren wir über den Zeilenindex von ***m1*** . ***m3*** und ***m1*** haben die gleiche Anzahl an Zeilen.

```
for (int i = 0; i< m1.length; i++) {

        **
}
```

Programm zur Matrixmultiplikation

```
private static int[][]  matrixMult(int[][] m1, int[][] m2) {

    int[][] m3 = new int[m1.length][m2[0].length];

    for (int i = 0; i< m1.length; i++) {


        for (int j = 0; j < m2[i].length; j++) {

            int cij = 0;
            for (int k = 0; k < m2.length; k++) {
                cij += m1[i][k] * m2[k][j];
            }

            m3[i][j] = cij;
        }
    }
    return m3;
}
```

m3 Zeilen = m1 Zeilen
m3 Spalten = m2 Spalten

Zeilen in m3

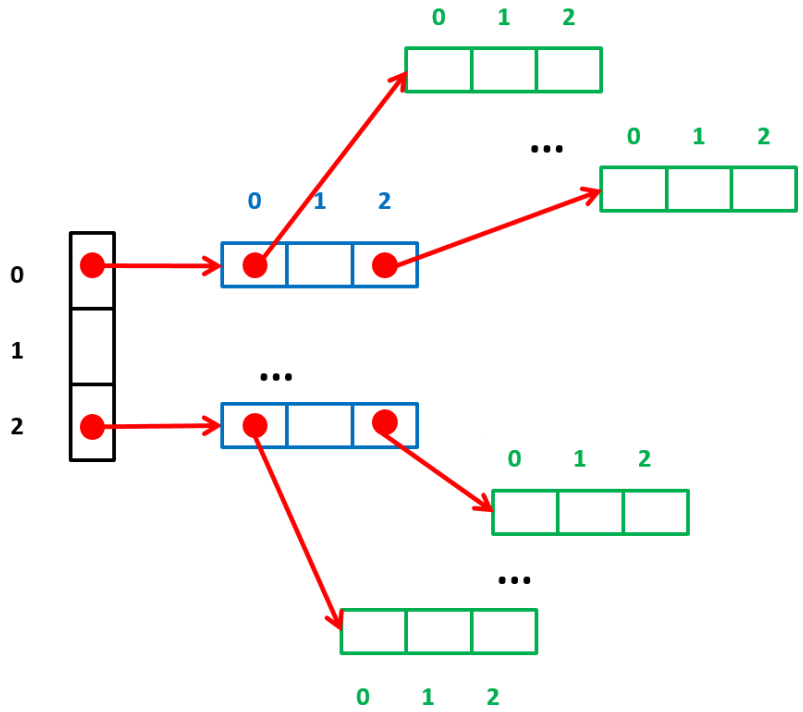
Spalten in m3

Summe der Produkte der Element der Zeile i in m1 und der Spalte j in m2

Summe der Produkte in m3 an Position [i][j] eintragen

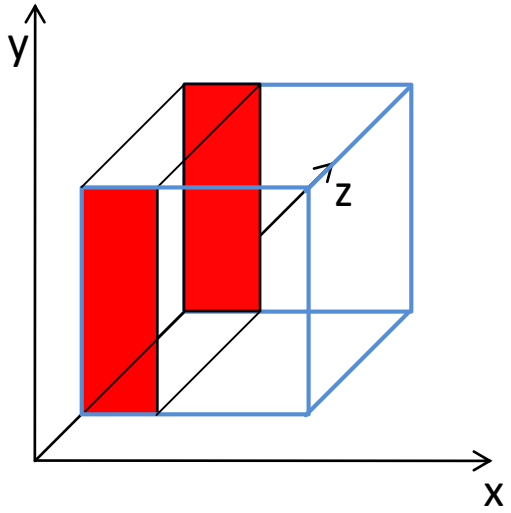
Speichermodell eines 3-dim Arrays (vereinfacht)

```
int[][][] cube = new int[3][3][3];
```

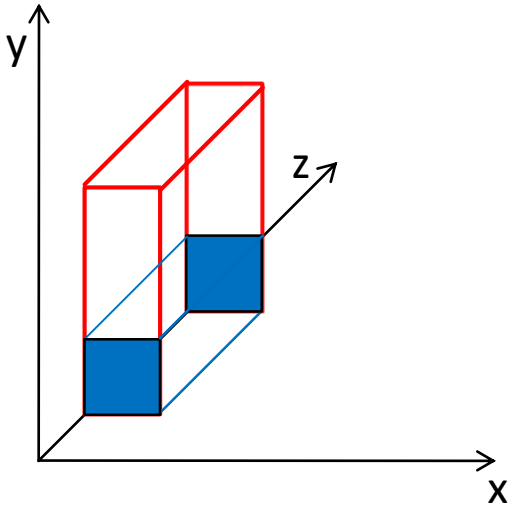


Graphik zur Veranschaulichung der Zugriffe auf einen Würfel

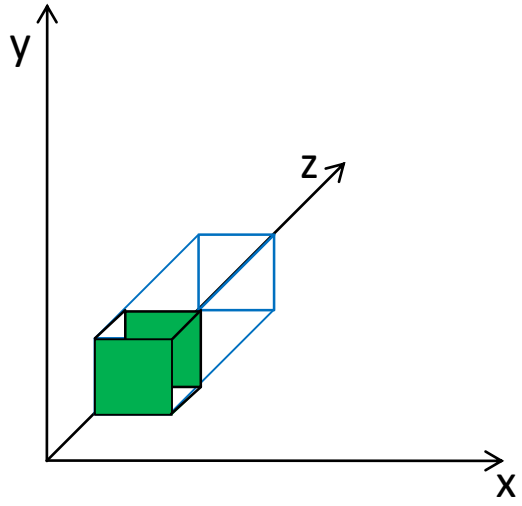
`cube[0]`



`cube[0][0]`



`cube[0][0][0]`



3-dim Arrays „scheibenweise“ über den ersten Index füllen

```
p("Würfel scheibenweise über den ersten Index  befüllen");
int[][] slices1 = {{1,2,3},{1,2,3},{1,2,3}};
int[][] slices2 = {{4,5,6},{4,5,6},{4,5,6}};
int[][] slices3 = {{7,8,9},{7,8,9},{7,8,9}};

int[][][] threeSlices = new int[][][]{slices1,slices2,slices3};
// oder auch direkt cube = new int[][][]{slices1,slices2,slices3};

for (int i = 0; i < cube.length; i++) {
   cube[i] = threeSlices[i].clone();
}
pAry("cube ",cube);
```

```
cube
[[[1,2,3],[1,2,3],[1,2,3]],[[4,5,6],[4,5,6],[4,5,6]],[[7,8,9],[7,8,9],[7,8,9]]]
```

3-dim Arrays „streifenweise“ über den 1‘ten und 2‘ten Index füllen

```
p("Würfel streifenweise über den 1'ten und 2'ten Index  befüllen");
int[] threePerRow = {6,7,8};

for (int i = 0; i < cube.length; i++) {
   int[][] slice = cube[i];
   for (int j = 0; j < slice.length; j++) {
      cube[i][j] = threePerRow.clone();
   }
}
pAry("cube ",cube);
```

```
cube
[[[6,7,8],[6,7,8],[6,7,8]],[[6,7,8],[6,7,8],[6,7,8]],[[6,7,8],[6,7,8],[6,7,8]]]
```

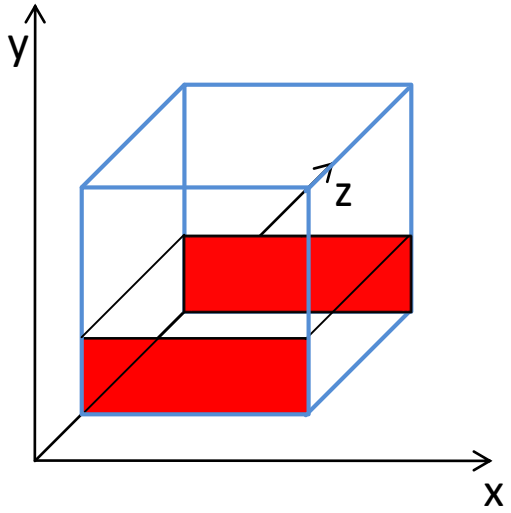
3-dim Arrays „elementweise“ füllen

```
p("Würfel elementweise befüllen");
for(int i=0; i< cube.length; i++){
   for(int j=0; j< cube[0].length; j++){
      for(int z=0; z<cube[0][0].length; z++){
         cube[i][j][z] = Integer.parseInt(""+ (i+1) +(j+1)+(z+1));
      }
   }
}
pAry("cube ",cube);
```

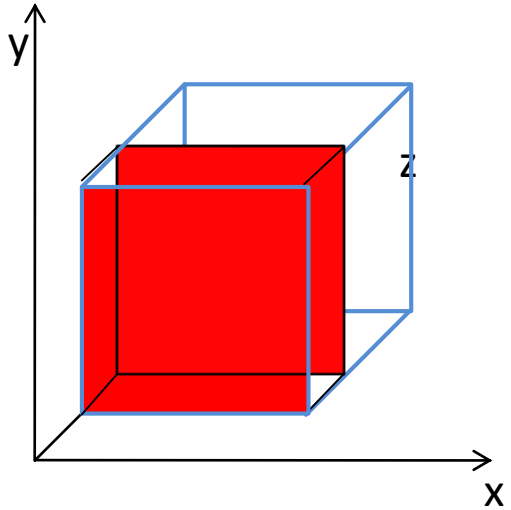
```
cube
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231
,232,233]],[[311,312,313],[321,322,323],[331,332,333]]]
```

Würfelscheiben über den 2‘ten und 3‘ten Index adressieren

Scheiben über den 2‘ten Index
cube[0..2][0][0..2]



Scheiben über den 3‘ten Index
cube[0..2][0..2][0]



3-dim Arrays über den 2‘ten Index „scheibenweise“ ausgeben

```
p("Würfel über den 2'ten Index scheibenweise ausgeben");
int xDim = cube.length;
int zDim = cube[0][0].length;
int yDim = cube[0].length;

int[][] slice = new int[3][3];

for(int i = 0; i < yDim; i++ ) {
   for(int j =0; j < xDim; j++ ){
      for(int k = 0; k < zDim; k++){
         slice[j][k] = cube[j][i][k];
      }
   }
   pAry("slice", slice);
}
```

```
Würfel über den 2'ten Index scheibenweise ausgeben
slice [[111,112,113],[211,212,213],[311,312,313]]
slice [[121,122,123],[221,222,223],[321,322,323]]
slice [[131,132,133],[231,232,233],[331,332,333]]
```

3-dim Arrays über den 3‘ten Index „scheibenweise“ ausgeben

Zur Übung!

Flache und tiefe Kopien

ARRAYS KOPIEREN

Arrays kopieren

Varianten

- explizites Übertragen der Elemente in Schleifen
- Methode ***clone***, die Arrays von der Klasse ***Object*** erben
- Methoden ***copyOf*** und ***copyOfRange*** der Hilfsklasse ***Arrays***.
- Methode ***System.arraycopy***

Eigenschaften der Varianten

- erzeugen flache Kopien eines Arrays A_1 .
- es werden die Instanz-Variablen aber nicht deren Referenzen kopiert.

➔ Eine Zuweisung eines neuen Objektes, Arrays oder Wertes auf eine Position in A_1 ändert den Inhalt der Kopien nicht und umgekehrt.

➔ Enthält A_1 Objekte oder Arrays, dann werden alle Änderungen an referenzierten Objekten bzw. Arrays in allen Kopien von A_1 sichtbar und umgekehrt.

Flache Kopien von Arrays erzeugen

```
p("Variante 1: in Schleifen");
Person[] perAry2 = new Person[perAry.length];
for( int i =0; i <perAry.length; i++) {
   perAry2[i] = perAry[i];
}
pAry("Variante 1:", perAry2);
p("Variante 2: mit der Methode clone, die Arrays von Object erben");
perAry2 = perAry.clone();
pAry("Variante 2:", perAry2);
p("Variante 3: mit Arrays.copyOfRange");
perAry2 = Arrays.copyOf(perAry,0);
perAry2 = Arrays.copyOfRange(perAry,0,perAry.length);
pAry("Variante 3:", perAry2);
p("Variante 4: mit System.arraycopy");
System.arraycopy(perAry, 0, perAry2, 0, perAry.length);
pAry("Variante 4:", perAry2);
```

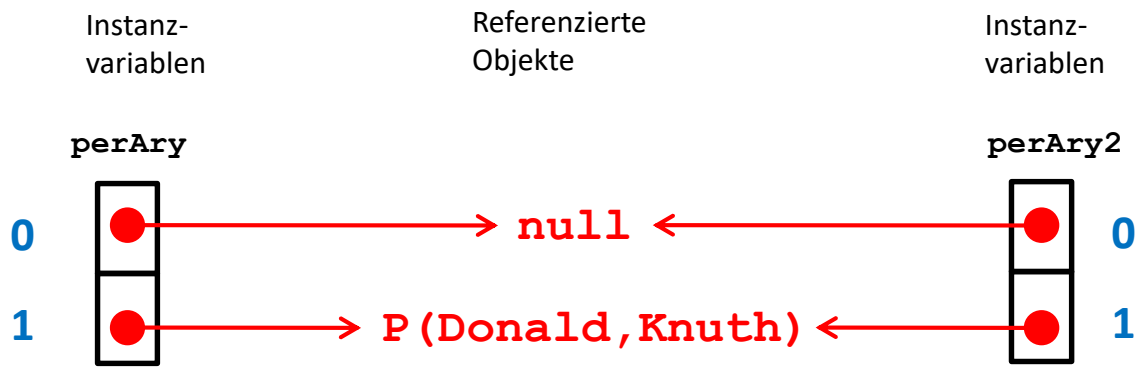
```
Variante 1: [null,P(Donald,Knuth)]
Variante 2: [null,P(Donald,Knuth)]
Variante 3: [null,P(Donald,Knuth)]
Variante 4: [null,P(Donald,Knuth)]
```

Flache Kopien und Elementzuweisung

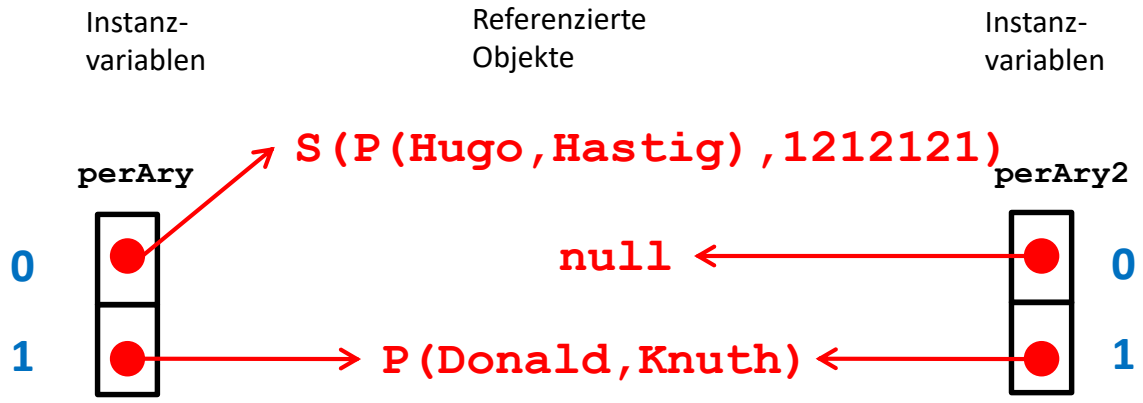
```
p("Flache Kopien und Elementzuweisung");
perAry[0] = new Student("Hugo","Hastig",1212121);
pAry("Original" ,perAry);
pAry("Kopie", perAry2);
```

```
Original [S(P(Hugo,Hastig),1212121),S(P(Donald,Knuth),1111111)]
Kopie [null,S(P(Donald,Knuth),1111111)]
```

Flache Kopien und Elementzuweisung




```
perAry[0] = new
Student("Hugo","Hastig",1212121);
```

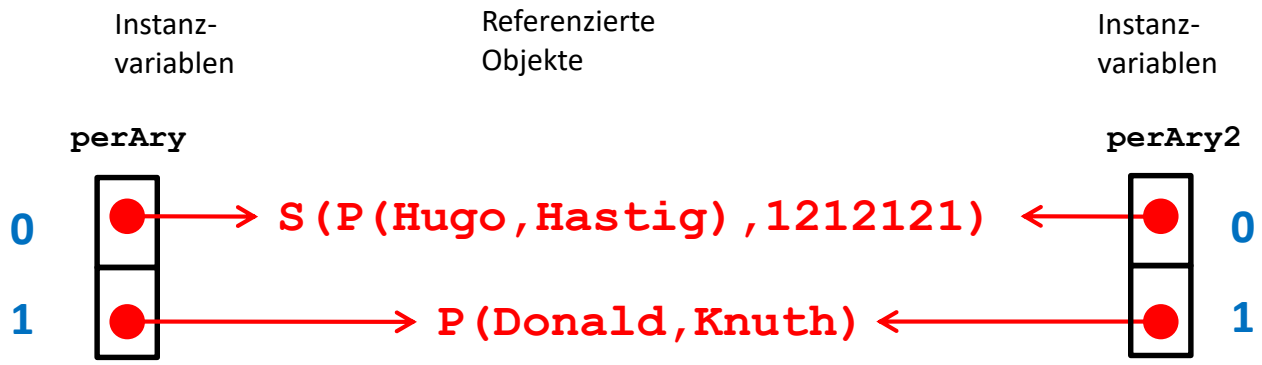


Flache Kopien und Änderung referenzierter Objekte

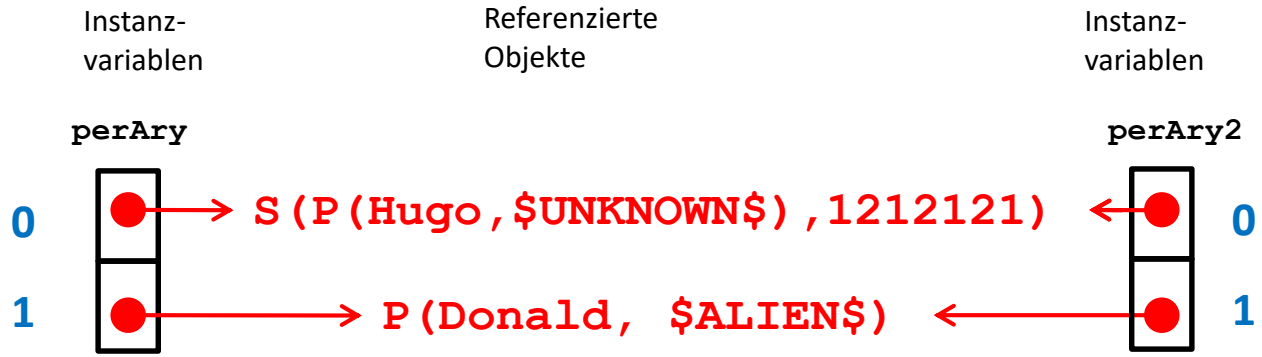
```
p("Flache Kopien und Änderung referenzierter Objekte");
perAry2 = perAry.clone();
perAry[0].setName("$UNKNOWN$");
perAry2[1].setName("$ALIEN$");
pAry("Original" ,perAry);
pAry("Kopie", perAry2);
```

```
Original [S(P(Hugo,$UNKNOWN$),1212121),S(P(Donald,$ALIEN$),1111111)]
Kopie [S(P(Hugo,$UNKNOWN$),1212121),S(P(Donald,$ALIEN$),1111111)]
```

Flache Kopien und Änderung referenzierter Objekte




```
perAry2 = perAry.clone();
perAry[0].setName("$UNKNOWN$");
perAry2[1].setName("$ALIEN$");
```



Tiefe Kopien von Arrays mit Klonen der enthaltenen Objekten

```
p("Tiefe Kopien von Arrays müssen eigenhändig erzeugt werden");
p("Wir erzeugen beim Kopieren der Personen zwischen Arrrays, flache Kopien der Objekte mit clone. "
+ "Dazu haben wir zuvor Personen Cloneable gemacht und die Methode clone von Object " +
    "in der Klasse Person geeignet überschrieben.");

perAry = new Person[]{ new Student("Hugo","Hastig",1212121), new Person("Donald","Knuth")};
perAry2 = new Person[2];

for(int i =0; i <perAry.length; i++){
   perAry2[i] = (Person)perAry[i].clone();
}

perAry[0].setName("$UNKNOWN$");
perAry2[1].setName("$ALIEN$");
p("Jetzt sind die Änderungen auf den enthaltenen Elementen nur in dem Array sichtbar, auf das " +
      "die Änderungen angewendet wurden.");
pAry("Original" ,perAry);
pAry("Kopie", perAry2);
```

```
Original [S(P(Hugo,$UNKNOWN$),1212121),P(Donald,Knuth)]
Kopie [S(P(Hugo,Hastig),1212121),P(Donald,$ALIEN$)]
```

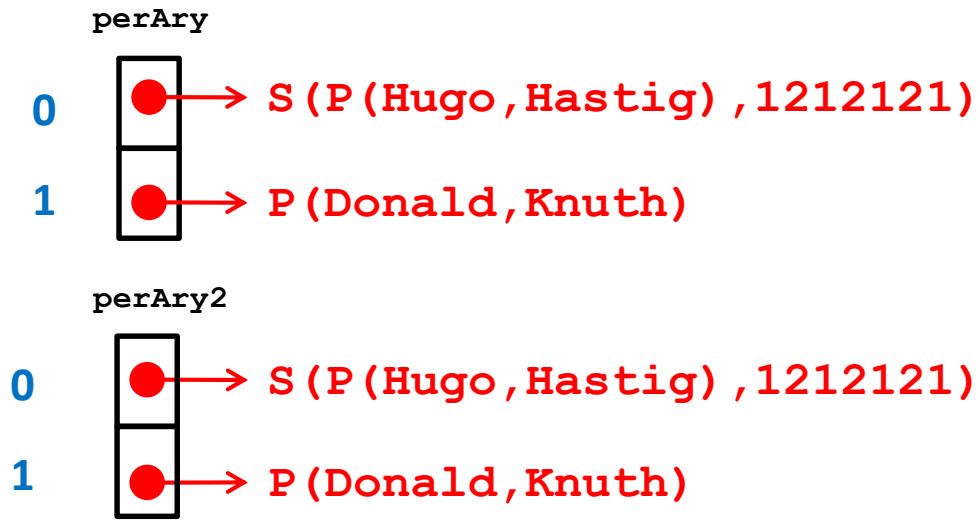
clone in der Klasse Person

```
package misc;

public class Person implements Cloneable {

   private String name;
   private String vorname;
   public Person(String vorname, String name) {
      this.name = name;
      this.vorname = vorname;
   }

   public void setName(String name) {
      this.name = name;
   }

   @Override
   public String toString() {
      return "P(" + vorname + "," + name+ ")";
   }

   @Override
   public Object clone() throws CloneNotSupportedException {
      return super.clone();
   }
}
```

Quelltext im Modul Commons

Tiefe Kopien von Arrays mit clonen der enthaltenen Objekten

```
for(int i =0; i <perAry.length; i++){
   perAry2[i] = (Person)perAry[i].clone();
}
```



Durch Klonen von Personenobjekten beim Kopiervorgang werden den Elementen von *perAry2* Kopien der Personen zugewiesen.

Tiefe Kopien von Arrays mit clonen der enthaltenen Objekten

```
perAry[0].setName("$UNKNOWN$");
perAry2[1].setName("$ALIEN$");
```

Die Änderungen werden auf gleichen, aber nicht identischen Personen vorgenommen. $UNKNOWN$ wird nur für den Student in perAry und $ALIEN$ nur für die Person perAry2 geändert.

Tiefe Kopien von Arrays mit Klonen von enthaltenen Arrays

```
p("Tiefe Kopien von Arrays mit clonen von enthaltenen Arrays");
p("Zunächst die Effekte mit flachen Kopien");
int[][] matCopy = new int[4][7];

for (int i = 0; i < matrix1.length; i++) {
   matCopy[i] = matrix1[i];    // Kopieren durch Elementzuweisung
}
matrix1[0][0]=999999;
matCopy[3][0]=444444;
p("Änderungen in den enthaltenen Arrays werden im Original und der Kopie
sichtbar");
pAry("Original",matrix1);
pAry("Kopie",matCopy);
```

```
Original
[[999999,2,3,4,5,6,7],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7],[444444,2,3,4,5,6,7]]
Kopie
[[999999,2,3,4,5,6,7],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7],[444444,2,3,4,5,6,7]]
```

Tiefe Kopien von Arrays mit clonen von enthaltenen Arrays

```
p("Werden hingegen Kopien der enthaltenen Teilarrays den Elementen von matCopy
zugewiesen, dann werden Änderungen nur jeweils im Original / bzw. der Kopie
sichtbar.");

for (int i = 0; i < matrix1.length; i++) {
   matCopy[i] = matrix1[i].clone();    // flache Kopie der enthaltenen Arrays
}

matrix1[0][0]=-55555;
matCopy[3][0]=111111;

pAry("Original",matrix1);
pAry("Kopie",matCopy);
```

```
Original [[-55555,
2,3,4,5,6,7],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7],[444444,2,3,4,5,6,7]]
Kopie [[999999
,2,3,4,5,6,7],[1,2,3,4,5,6,7],[1,2,3,4,5,6,7],[111111,2,3,4,5,6,7]]
```

ARRAYS VERGLEICHEN

Arrays vergleichen

- Die Methode **`equals`** vergleicht für Arrays auf **Identität** (**==**)
- Für Arrays kann die Methode **equals** **nicht** überschrieben werden.
- Die Methode **equals(a,b)** der Klasse **java.util.Arrays** führt einen inhaltlichen Vergleich auf Arrays durch.
  - primitive Objekte werden mit ==
  - Referenztypen mit **equals** verglichen.
- Bei mehrdimensionalen Arrays scheitert die Methode **Arrays.equals**.
- Die Methode **Arrays.deepEquals** führt einen elementweisen Vergleich für mehrdimensionale Arrays durch.

Arrays vergleichen

```
p("Arrays vergleichen.");
p("equals prüft auf Identität");
perAry2 = perAry.clone();

p("perAry2 == perAry: " + (perAry2 == perAry));
p("perAry2.equals(perAry): " + perAry2.equals(perAry));


p("Arrays.equals prüft Arrays auf Inhaltsgleichheit");
p("Arrays.equals(perAry,perAry2): " + Arrays.equals(perAry2,perAry));
```

```
equals prüft auf Identität
perAry2 == perAry: false
perAry2.equals(perAry): false
Arrays.equals prüft Arrays auf Inhaltsgleichheit
Arrays.equals(perAry,perAry2): true
```

Arrays vergleichen

```
p("Arrays.equals stösst bei mehrdimensionalen Arrays an Grenzen." +
      " Das erste enthaltene Array wird wieder mit equals von Array-Objekten verglichen");

for (int i = 0; i < matrix1.length; i++) {
   matCopy[i] = matrix1[i].clone();
}

p("Arrays.equals(matCopy,matrix1): " + Arrays.equals(matCopy,matrix1));

p("für den Inhaltsvergleich mehrdimensionaler Arrays muss die Methode deepEquals der Klasse
Arrays verwendet werden.");
p("Arrays.deepEquals(matCopy,matrix1): " + Arrays.deepEquals(matCopy,matrix1));
```

```
Arrays.equals(matCopy,matrix1): false
Arrays.deepEquals(matCopy,matrix1): true
```

UTILITY KLASSE *ARRAYS*

Arrays

- Methoden zum Kopieren und Vergleichen von Arrays (siehe vorausgehende Folien).
- Methoden zur Umwandlung in ***Strings*** für ***Object*** Arrays. (***deepToString***)
  - Diese Methode ist nicht auf eindimensionale Arrays anwendbar, deren Komponententyp ein Basisdatentyp ist.
- Methoden zum Sortieren und Suchen in Arrays. ***sort*** und ***binarySearch.***
- ***sort*** modifiziert das zu sortierende Array destruktiv.
- Damit Arrays sortiert werden können, müssen die Komponententypen vom Typ ***Comparable*** sein und die Methode ***compareTo*** überschreiben. (siehe Klasse ***Person*** und ***Student*** im beigefügten Source Code)

Umwandlung in Strings

```
p("Hilfsklasse Arrays für die Ausgabe von Object Arrays.");
p("deepToString" + Arrays.deepToString(matrix1));
// p("deepToString" + Arrays.deepToString(new int[]{1,2,3,4})); Fehler kein
Object-Array
```

```
Hilfsklasse Arrays für die Ausgabe von Object Arrays.
deepToString[[-55555, 2, 3, 4, 5, 6, 7], [1, 2, 3, 4, 5, 6, 7], [1, 2, 3,
4, 5, 6, 7], [444444, 2, 3, 4, 5, 6, 7]]
```

Sortieren und Suchen

```
p("Hilfsklasse Arrays zum Sortieren / Suchen  von / in Arrays");
perAry[0]= new Person("Alex","Wollik");
pAry(perAry);
Arrays.sort(perAry);    // destruktiv, Objekte müssen das Interface Comparable
implementieren
pAry("sort ", perAry);
p("search and found at " + Arrays.binarySearch(new int[]{1,2,3,4,5,6,7,8,9},
5));
p("search and found at " + Arrays.binarySearch(new int[]{1,2,3,4,5,6,7,8,9}, -
1));
```

```
Hilfsklasse Arrays zum Sortieren / Suchen  von / in Arrays
[P(Alex,Wollik),P(Donald,Knuth)]
search and found at 4
search and found at -1
```

Zusammenfassung

- Arrays sind Sammlungen **gleichartiger** Objekte mit einer festen Reihenfolge.
- Arrays haben einem **Komponenten-** und einen **Arraytyp**.
- Arrays definieren eine **Familie** von Typen, da jeder Typ in Java Komponententyp sein kann. Ausnahme: Generische Typen dürfen kein Komponententyp sein.
- Arrays haben in Java eine bei der Initialisierung **festgelegte Größe** (Anzahl von Elementen). Zugriffe außerhalb dieser Länge führt zu einem Laufzeitfehler.
- Arrays sind **Referenztypen**, haben aber nur eine fest definierte Anzahl von Methoden mit spezieller Syntax. Arrays sind **nicht erweiterbar**!
- Mit einem **vereinfachten Speichermodell** kann man sich die Positionen eines Arrays als Instanzvariablen vorstellen, die auf die einzelnen Elemente des Arrays zeigen.
- Die **Deklaration und Initialisierung** von Arrays kann in einer Anweisung oder in zwei getrennten Anweisungen erfolgen. Array-Literale sind nur erlaubt, wenn Deklaration und Initialisierung in einer Anweisung erfolgen.
- Für die **Initialisierung** gibt es abhängig vom Datentyp **Default-Werte**.
- **Element-Zuweisung** und Element-**Zugriff** sind nur im gültigen Indexbereich erlaubt.

Zusammenfassung

- Für das **Iterieren über die Inhalte** eines Arrays wird das **for each-**Konstrukt verwendet.
- **Iterieren**, um **Inhalte zwischen Arrays zu kopieren oder zu vergleichen** geht nur mit einer Fortschaltanweisung und indiziertem Zugriff.
- Für Arrays, deren Komponententyp ein Basisdatentyp ist, gelten strenge Typ-Kompatibilitäts-Regeln.
- Für Arrays, deren Komponententyp ein Referenztyp ist, gilt **Kovarianz**.
- (Ungeschützte) Kovarianz kann Ursache für Typfehler zur Laufzeit sein. (`ArrayStoreException`)
- **Mehrdimensionale Arrays** entstehen, wenn der Komponententyp ein Array ist. Die Dimensionalität ist nicht eingeschränkt.
- Die Standard-Methoden für das Kopieren von Arrays erzeugen **flache Kopien**.
- Für den **Array-Vergleich** existieren **3 Methoden** mit **unterschiedlicher Tiefe** des inhaltlichen Vergleiches `equals`, `Arrays.equals`, `Arrays.deepEquals`.
- Mit `Arrays.deepToString` lassen sich Arrays beliebiger Dimension in eine lesbare Darstellung umwandeln. Voraussetzung ist, dass der Komponententyp eine Referenztyp ist.
- Die Hilfsklasse `Arrays` bietet darüber hinaus eine Reihe von nützlichen Methoden für die Verarbeitung von Array-Objekten.