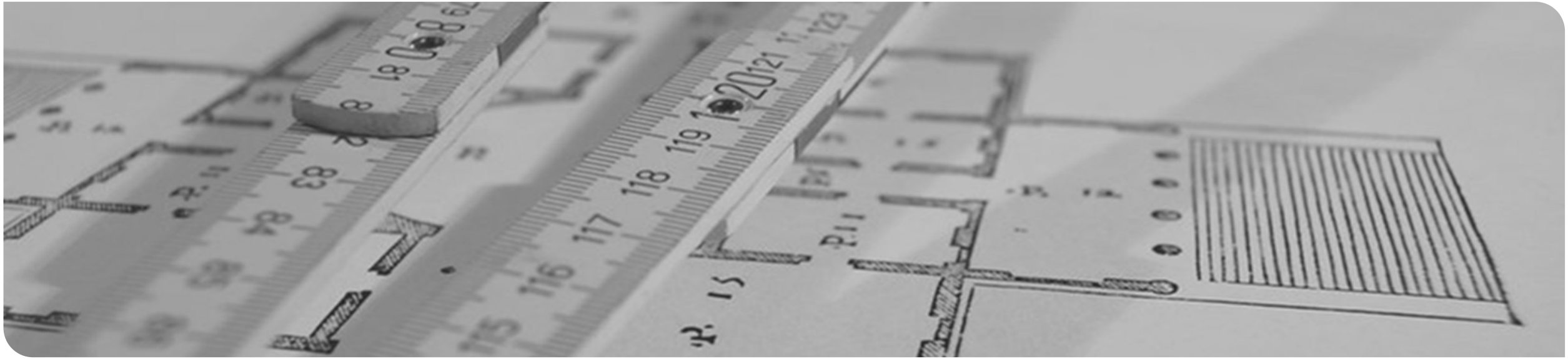
Karlsruher Institut für Technologie

Vorlesung Programmieren

5. Arrays

Prof. Dr.-Ing. Anne Koziolek



EduRef – Education for Refugees

Was machen wir?

- **Programmierkurse** für Geflüchtete und Personen mit Migrationshintergrund **halten**
- Kurse erstellen, Events planen
- Freies commitment (Jeder macht soviel wie er/sie möchte)

Warum mitmachen?

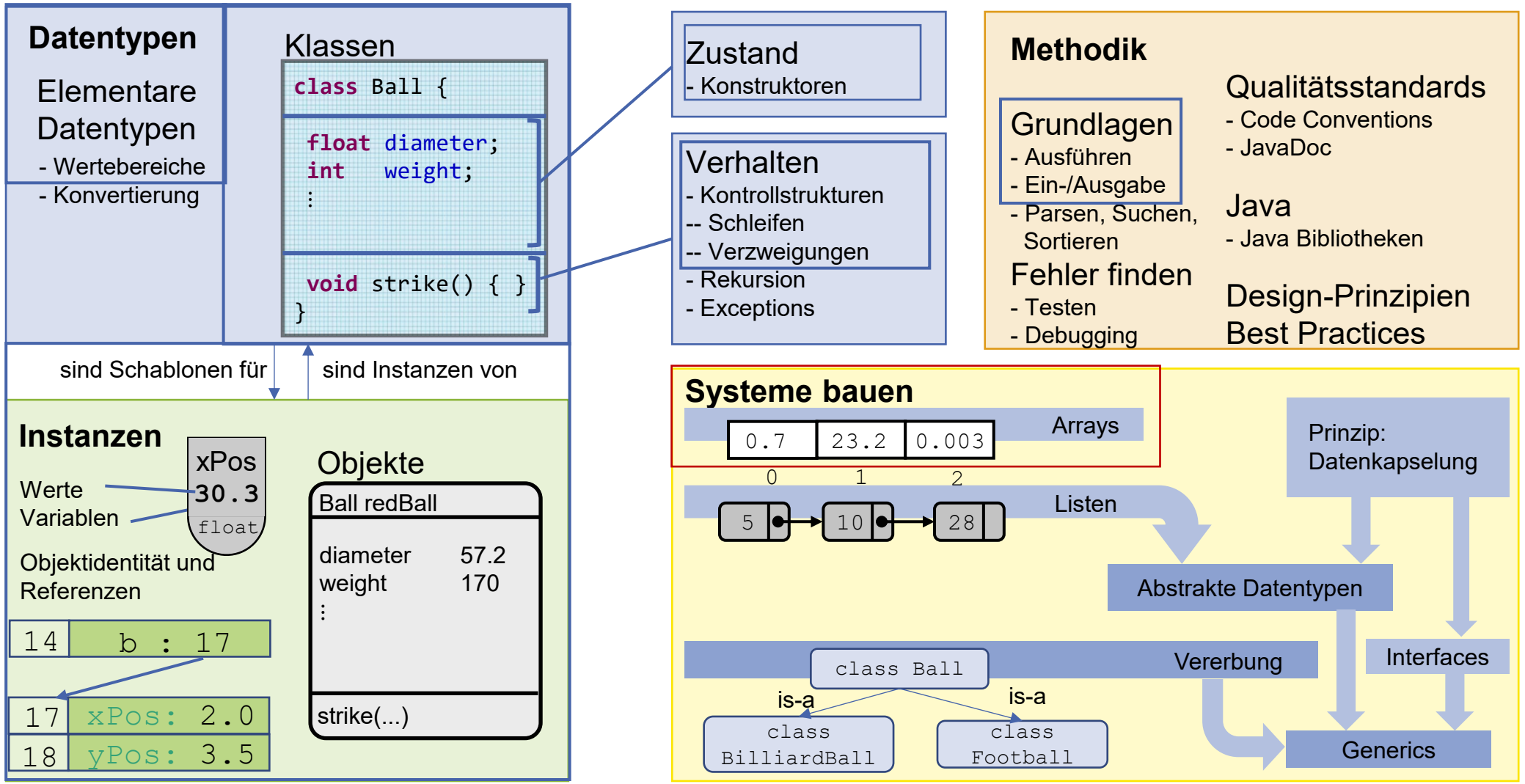
- Präsentationsfähigkeiten Verbessern
- Cooles Ehrenamt, Gleichgesinnte Leute treffen

Lerne uns kennen:

A) Hochschulgruppenmesse **:** 🗓️ 02.12.2024 📍 Audimax

B) 📩 **mitmachen@eduref.eu** oder auf Instagram anschreiben

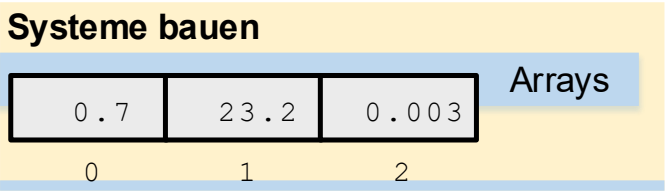
Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Vorlesungsüberblick: Vorläufiger Semesterplan

21.10.2024	Erstsemesterbegrüßung: Einführung
23.10.2024	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
30.10.2024	Typen und Variablen
06.11.2024	Kontrollstrukturen (+Scanner)
13.11.2024	Konstruktoren und Methoden
20.11.2024	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
27.11.2024	Listen und Abstrakte Datentypen
04.12.2024	Vererbung **~~Audimax~~ 50.34 Raum -101 und Raum -102**
11.12.2024	Exceptions; Interfaces
18.12.2024	Generics; Rekursion
08.01.2025	Java-API; Objektorientierte Design-Prinzipien
15.01.2025	Best Practices; Finden und Beheben von Fehlern
22.01.2025	Testen und Assertions
29.01.2025	JUnit; Parsen, Suchen, Sortieren
05.02.2025	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen
12.02.2025	Wrap-Up

Lernziele



Arrays

- Sie können Arrays deklarieren und initialisieren
- Sie können auf Werte in Arrays über den Index zugreifen

- Sie können Arrays in Ihrem Programm sinnvoll einsetzen
  - Sie können die Daten in Ihrem Programm in Arrays ablegen
  - Sie wissen, dass Arrays im Speicher wie Objekte behandelt werden

- Sie können über Arrays mit einer `for-each`-Schleife iterieren
- Sie können mehrdimensionale Arrays deklarieren und verwenden

Quelle: http://phdcomics.com

Definition

- **Array (Feld)**
  - Folge (Kollektion) von Elementen
  - Alle Elemente haben denselben Typ
  - Der Zugriff auf die Elemente des Arrays erfolgt über einen (ganzzahligen) *Index*
  - Einfachste Datenstruktur
- **Länge:** Anzahl *n* der Elemente in der Kollektion
- **Index / Nummerierung:** von 0 bis *n*-1
- **Beispiele:**

Array-Deklaration

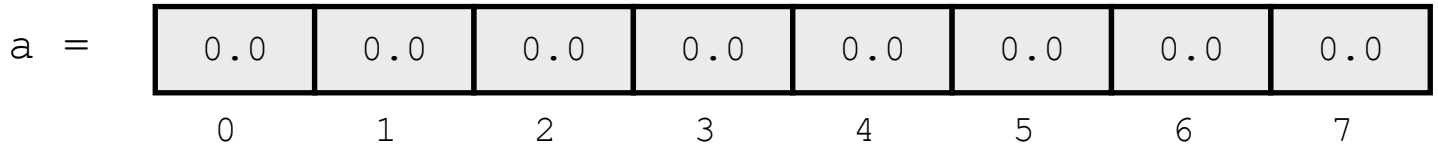
- **Schema:**
  - `Typ[] name;` — deklariert eine Array-Variable mit Namen "*name*" für Elemente des Typs "*Typ*"
  - `Typ[] name = new Typ[N];` — deklariert Array und reserviert Speicherplatz für *N* Elemente (*N* muss vom Typ `int` sein)
  - `Typ[] name = {`v_0`, ..., `v_{n-1}`};` — deklariert Array, reserviert Speicherplatz für *N* Elemente und initialisiert diese mit v_0, ..., v_{n-1}
- **Beispiele:**

```
int[] a; // nur Deklaration, kein Speicherplatz reserviert
byte[] b = new byte[12]; // reserviert Speicherplatz für 12 Elemente
String[] s = { "Maier", "Maier", "Maier", "Maier" }; // reserviert Speicherplatz für 4 Elemente und initialisiert diese
```

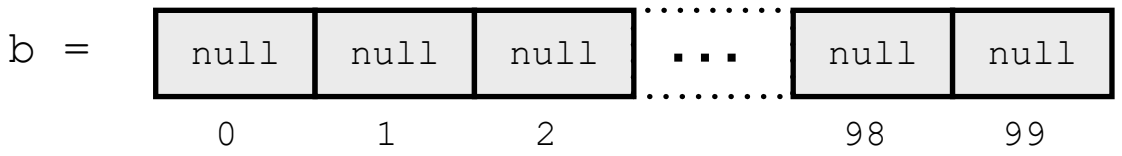
- **Beachte:**
  - Die Größe eines Arrays ist nach Anfordern des Speicherplatzes fest!
  - In der ersten Variante der Speicherplatzreservierung (mit **new** ...) werden die Elemente nach den Regeln der Attributinitialisierung vorbelegt (vgl. Kap. 4 der Vorlesung)

Deklaration und Initialisierung

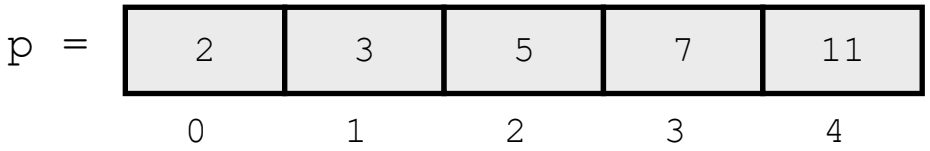
- **Beispiel:**
  - `double[] a = new double[8];`



  - `String[] b = new String[100];`



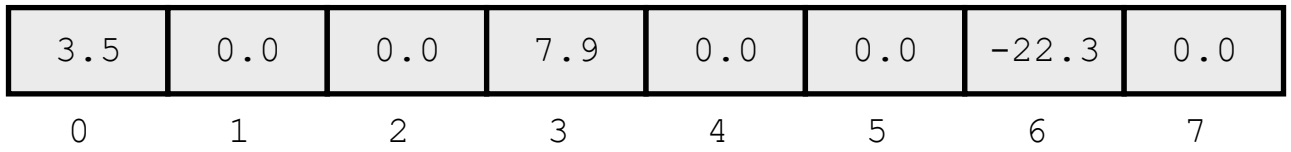
  - `int[] p = { 2, 3, 5, 7, 11 };`



Ansprechen von Elementen / Länge des Arrays

- **Setzen von Elementen:**
  - ```
    double[] a = new double[8];
    a[0] = 3.5;
    a[3] = 7.9;
    a[6] = -22.3;
    ```

a =



- **Selektion / Auslesen von Elementen:**
  - ```
    double x = a[3]; System.out.println(x); // gibt 7.9 aus
    ```
  - Was gibt `System.out.println(a[5]);` aus?

- **Länge des Arrays:**
  - ```
    a.length                    // liefert den Wert 8
    ```
  - ```
    int[] smallPrimes = { 2, 3, 5, 7, 11 };
    ```
  - ```
    int l = smallPrimes.length;  // liefert den Wert 5
    ```
  - `length` ist implizit als `final int` (also Konstante) deklariert
    ```

Arrays und Schleifen: Beispiele

- Erzeuge ein Array mit 100 Zufallswerten:

```
final int N = 100;
double[] a = new double[N];
for (int i = 0; i < N; i++) {
  a[i] = Math.random();
}
```

- Kopiere alles in ein zweites Array:

```
double[] b = new double[a.length];
for (int i = 0; i < a.length; i++) {
  b[i] = a[i];
}
```

- Berechne den Mittelwert der Array-Elemente:

```
double sum = 0.0;
for (int i = 0; i < N; i++) {
  sum += a[i];
}
double average = sum / N;
```

- "Spiegele" das Array (Umkehrung der Reihenfolge):

```
for (int i = 0; i < N / 2; i++) {
  double temp = b[i];
  b[i] = b[N - 1 - i];
  b[N - 1 - i] = temp;
}
```

Quiz

- Ist der folgende Code korrekt?

A

```
double[] a = new double[10];
a.length = 5;
a[6] = 27.5;
```

B

```
int[] b = new int[3000000000L];
```

C

```
int[] c = new int[4];
for (int i = 0; i <= c.length; i++) {
  c[i] = 2*i + 3;
}
```

Beitreten unter
slido.com
#Prog

Arrays und Schleifen: Weitere Beispiele

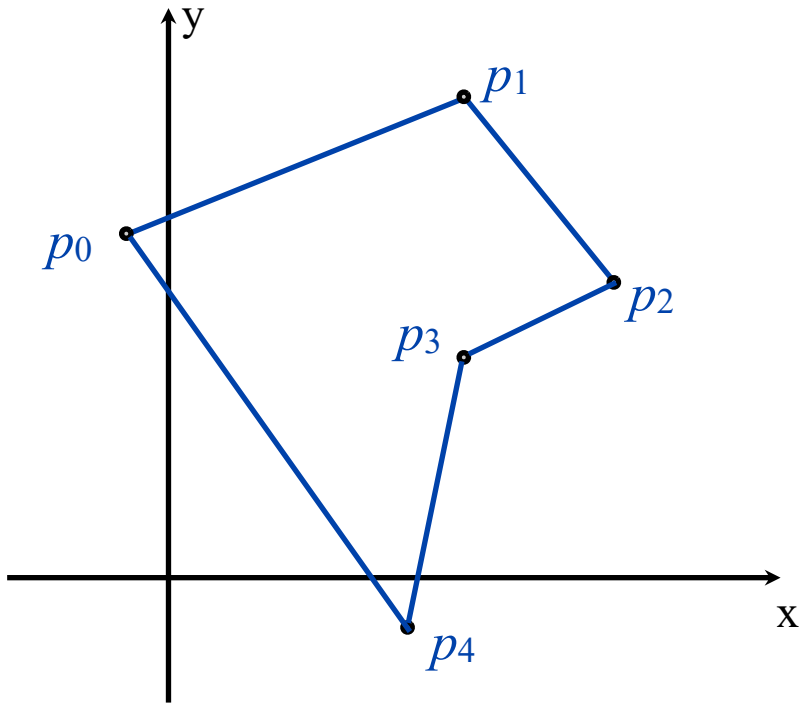
- Bestimme die Häufigkeit einer Zahl in einem Integer-Array:

```
int countOccurrences(int[] a, int number) {
  int count = 0;
  for (int i = 0; i < a.length; i++) {
    if (a[i] == number) {
      count++;
    }
  }
  return count;
}
```

- Suche eine Zahl in einem Integer-Array:

```
boolean search(int[] a, int number) {
  int i;
  for (i = 0; i < a.length; i++) {
    if (a[i] == number) {
      break;
    }
  }
  return (i != a.length);
}
```

Beispiel: Polygone



- Polygone im $\mathbf{R}^2$

```
class Polygon2D {
  // attributes
  Vector2D[] vertices;

  // constructor
  Polygon2D(Vector2D[] vertices) {
    this.vertices = vertices;
  }

  // methods
  int getNrVertices() {
    return vertices.length;
  }
  ...
}
```

Polygone erzeugen

Variante 1:

```
Vector2D[] vertices = new Vector2D[5];
vertices[0] = new Vector2D(-1.0f, 5.0f);
vertices[1] = new Vector2D(5.0f, 7.0f);
vertices[2] = new Vector2D(8.0f, 4.0f);
vertices[3] = new Vector2D(4.5f, 3.0f);
vertices[4] = new Vector2D(3.5f, -1.0f);
Polygon2D poly = new Polygon2D(vertices);
```

Variante 2:

```
Polygon2D poly = new Polygon2D(
  new Vector2D[] {
    new Vector2D(-1.0f, 5.0f),
    new Vector2D(5.0f, 7.0f),
    new Vector2D(8.0f, 4.0f),
    new Vector2D(4.5f, 3.0f),
    new Vector2D(3.5f, -1.0f)
  });
```

```
class Polygon2D {
  // attributes
  Vector2D[] vertices;

  // constructor
  Polygon2D(Vector2D[] vertices) {
    this.vertices = vertices;
  }

  // methods
  int getNrVertices() {
    return vertices.length;
  }
  ...
}
```

Mehr zu Arrays

- **Arrays werden in Java wie Objekte behandelt**
  - `String[] a = { "a0", "a1", "a2", "a3", "a4" };`
    Erstellt ein neues Array (auf dem Heap) und weist der Variablen `a` die Referenz auf das Array zu.
  - `String[] b = a;`
    Beide Variablen (`a` und `b`) referenzieren dasselbe Array
  - `b[1] = "b1";`
    Der Ausdruck `a[1]` wertet zu `"b1"` aus.

Arrays und Schleifen: `for-each`

- Java kennt eine spezielle Variante der **`for`**-Schleife für Arrays (meistens `for-each` genannt)
- **Schema:** `for (Typ element : array) { Anweisungen }`
  - Führe *Anweisungen* für jedes Element (dieses hat in der Schleife den Namen "*element*") des *arrays* (vom Typ `Typ[]`) aus.
- **Beispiel:**

```
final int N = 100;
double[] numbers  = new double[N];

// average
double sum = 0.0;
for (int i = 0; i < N; i++) {
  sum += numbers[i];
}
double average = sum / N;
```

Konventionelle `for`-Schleife

```
final int N = 100;
double[] numbers  = new double[N];

// average
double sum = 0.0;

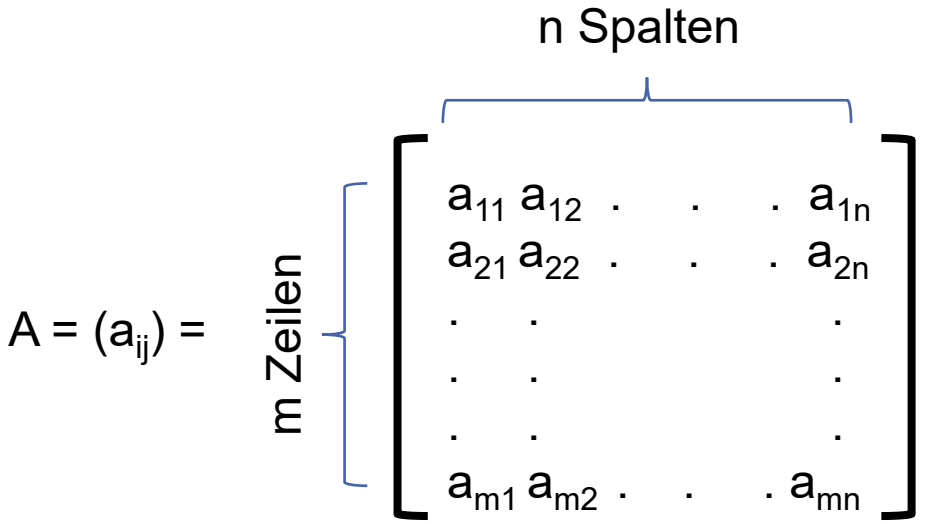
for (double number : numbers) {
  sum += number;
}
double average = sum / N;
```

`for-each`-Schleife

Nutzen wo möglich, da wirkliche Zählschleife!

Matrizen

- Eine Matrix kann als eine Tabelle von Einträgen betrachtet werden
- Einträge der Matrix sind häufig Zahlen
- Eine Matrix mit
  - m Zeilen
  - n Spalten



- a_{ij} ist ein Eintrag in der i-ten Zeile und j-ten Spalte der Matrix

- Wie wird diese Matrix in Java realisiert?

Mehrdimensionale Arrays

- Mehrdimensionale Arrays sind als "Array-von-Arrays" möglich
- **Beispiel:**
  - `int[][] m = new int[10][15];`
    Erzeugt ein 2-dimensionales Array von Ganzzahlen. Genauer ist m dabei ein Array der Größe 10 von Arrays der Größe 15.
  - Die "eingeschachtelten" Arrays können Variablen des passenden Typs zugewiesen werden.
    `int[] v = m[3];  // v ist ein Array der Größe 15 und entspricht 4. „Zeile“ von m`
  - Das Iterieren über mehrdimensionale Arrays erfolgt meist mit geschachtelten **`for`**-Schleifen:

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

Mehr zu mehrdimensionalen Arrays

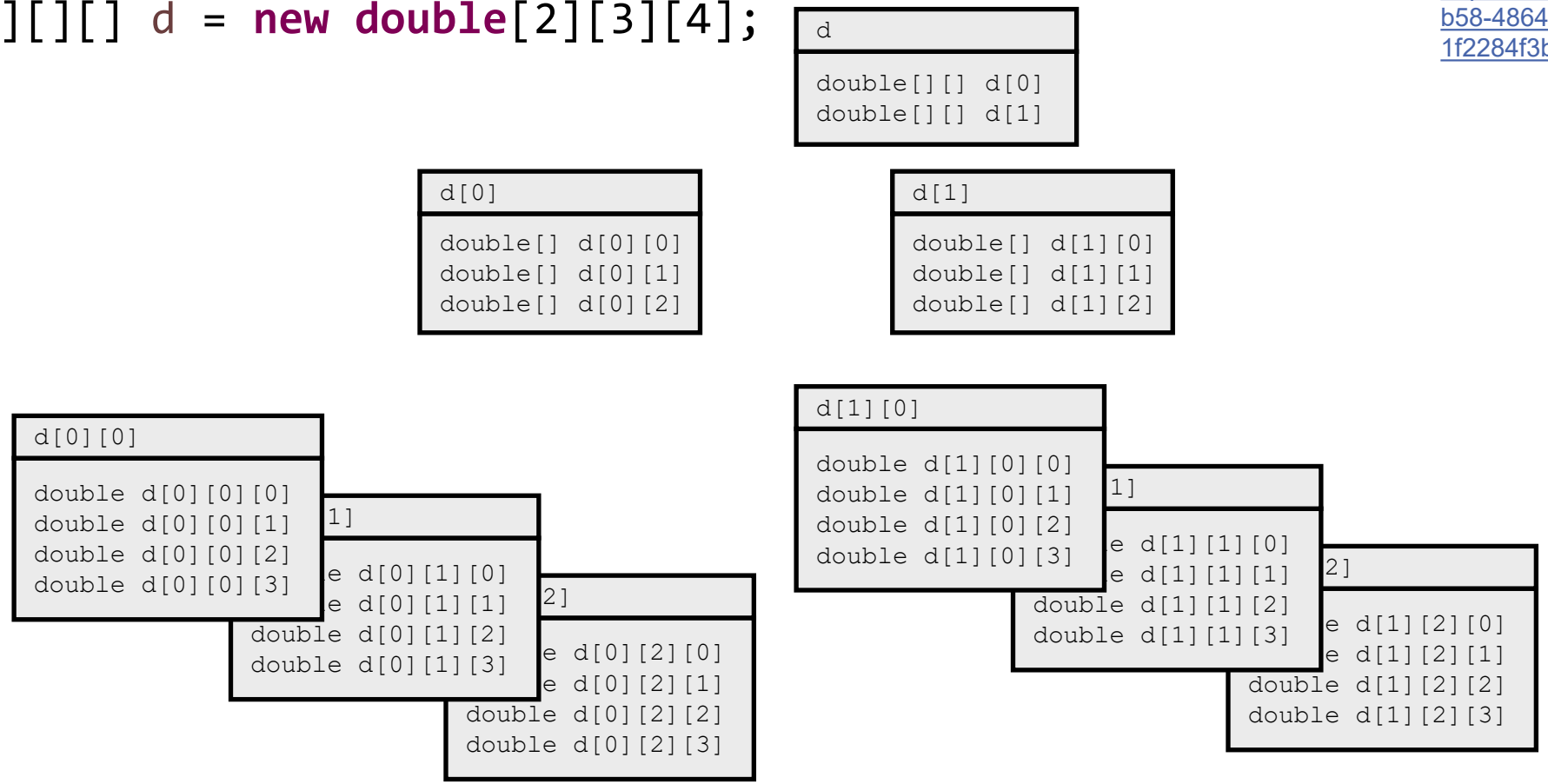
- `int[][] m = new int[10][15];`
  Erzeugt ein Array mit 10 Referenzen auf **`int`**-Arrays der Länge 15.

- `int[][] a = new int[3][];`
  Erzeugt ein Array mit 3 Referenzen des Typs **`int[]`**.

Quiz

- Wieviele Objekte werden in folgender Codezeile erzeugt?
  `double[][][] d = new double[2][3][4];`

(a) 1
(b) 3
(c) 9
(d) 16
(e) 24

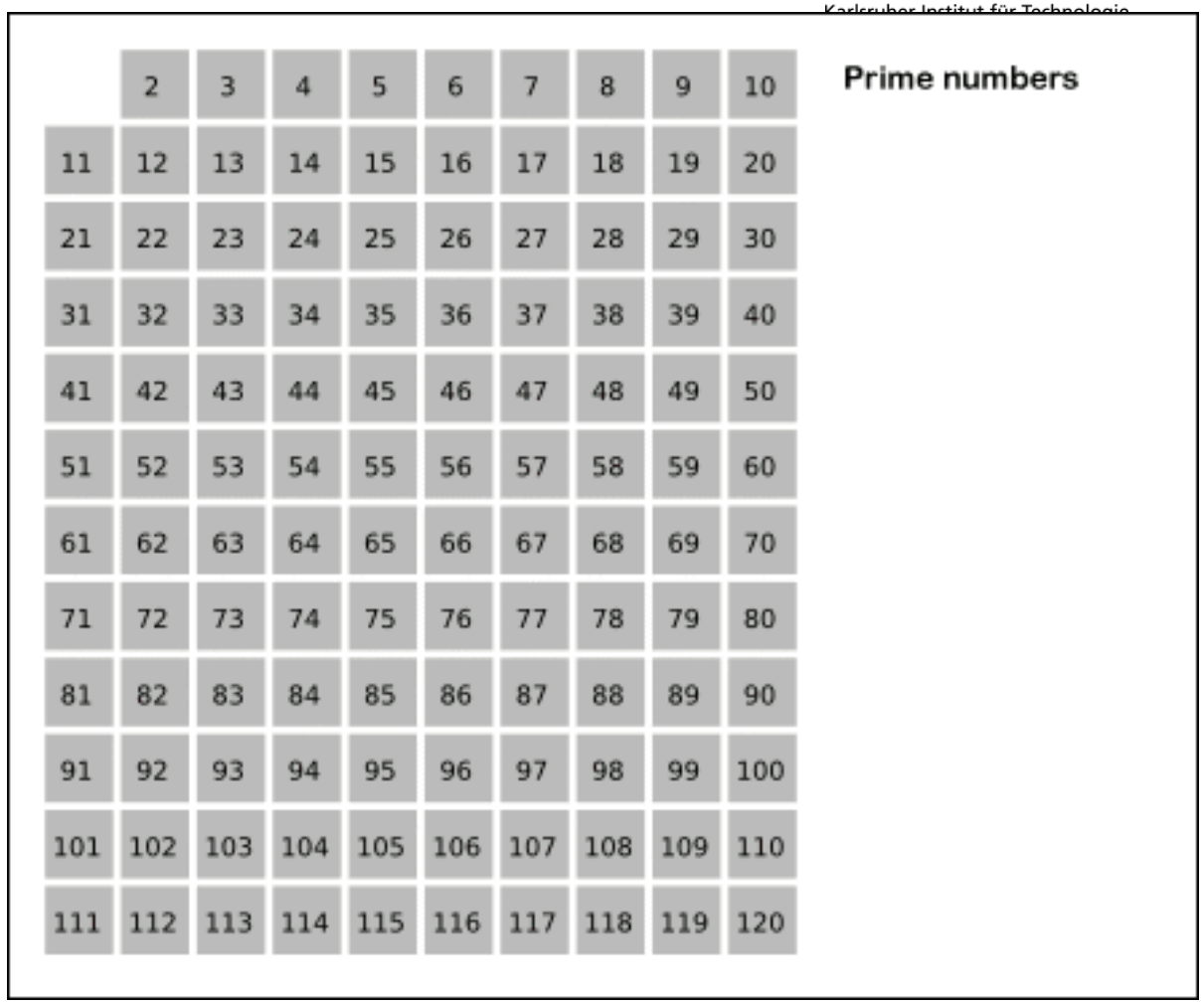


Interestingly, ChatGPT gets this wrong, even in the 2D case: https://chatgpt.com/share/673b4b58-4864-8012-bba7-1f2284f3bc1e

Beispiel: Sieb des Eratosthenes

- **Aufgabe:**
  - **Gegeben:** Eine natürliche Zahl *n*
  - **Gesucht:** Eine Liste aller Primzahlen kleiner *n* (als Array)

- **Lösungsmethode (Sieb des Eratosthenes):**
  - Schreibe alle Zahlen von 2 bis *n* auf.
  - Markiere nun Zahlen nach folgendem Schema:
    1. Wähle die kleinste unmarkierte Zahl und markiere sie als "prim".
    2. Markiere alle Vielfachen dieser Zahl als "nicht-prim".
    3. Solange noch nicht alle Zahlen markiert sind, fahre mit Schritt 1 fort.



"Animation that visualizes the 'Sieve of Eratosthenes' algorithm", including optimization of starting from prime's square by SKopp [CC BY-SA 3.0] from https://commons.wikimedia.org/wiki/File:Animation_Sieve_of_Eratosth.gif

Beispiel: Sieb des Eratosthenes

```
enum Mark { UNMARKED, PRIME, NOT_PRIME };   // Datentyp für Markierungen
public static int[] primeSieve(int n) {
  Mark[] mark = new Mark[n+1];              // Markierung für jede Zahl [0,n]
  mark[0] = mark[1] = Mark.NOT_PRIME;       // 0 und 1 sind keine Primzahlen.
  for (int i = 2; i <= n; i++) {            // Alle anderen Zahlen werden als "unmarkiert"
    mark[i] = Mark.UNMARKED;                // initialisiert.
  }
  // Setze Markierungen
  int p = 2, count = 0;                     // Beginne bei Kandidat p = 2. ('count' zählt
  while (p <= n) {                          // die Anzahl gefundener Primzahlen.)
    mark[p] = Mark.PRIME;                   // Kleinste unmarkierte Zahl p ist prim.
    count++;
    for (int k = p; k * p <= n; k++) {      // Markiere alle Vielfachen von p als nicht-prim (ab p*p, da kleinere schon markiert)
      mark[k * p] = Mark.NOT_PRIME;
    }
    do {// Suche nächste unmarkierte Zahl
      p++;
    } while (p <= n && mark[p] != Mark.UNMARKED);
  }
  // Fülle Ergebnis-Array mit Primzahlen
  int[] primes = new int[count];
  int j = 0;
  for (int i = 0; i < mark.length; i++) {   // Kopiere als 'prim' markierte Zahlen ins
    if (mark[i] == Mark.PRIME) {            // Ergebnis-Array.
      primes[j++] = i;
    }
  }
  return primes;
}
```

Problem:

- Bei der Multiplikation „`k * p`" kann ein Überlauf auftreten!
- Was könnte man dagegen tun?

Beispiel: Sieb des Eratosthenes (optimiert)

- **Beobachtungen**
  - Die äußere Schleife zur Markierung muss nur bis √n durchlaufen werden.
  - `Mark.`***`PRIME`*** und `Mark.`***`UNMARKED`*** können zusammengefasst werden
  - Die Default-Initialisierung von Arrays kann ausgenutzt werden: default "`false`" muss dann für "ist-prim" stehen.

➜ **Optimierungen möglich! (siehe rechts)**

```
class PrimeSieveOpt {
  public static int[] primeSieve(int n) {
    boolean[] notPrime = new boolean[n+1];
    int p = 2, count = 0;
    int sqrn = (int) Math.sqrt(n);
    while (p <= sqrn) {
      count++;
      for (int k = p; k * p <= n; k++) {
        notPrime[k*p] = true;
      }
      while (notPrime[++p]) {}
    }
      while (p <= n) {
        if (!notPrime[p++]) {
          count++;
        }
      }
    int[] primes = new int[count];
    int j = 0;
    for (int i = 2; i <= n; i++) {
      if (!notPrime[i]) {
        primes[j++] = i;
      }
    }
    return primes;
  }
}
```

Generelles zum Optimieren

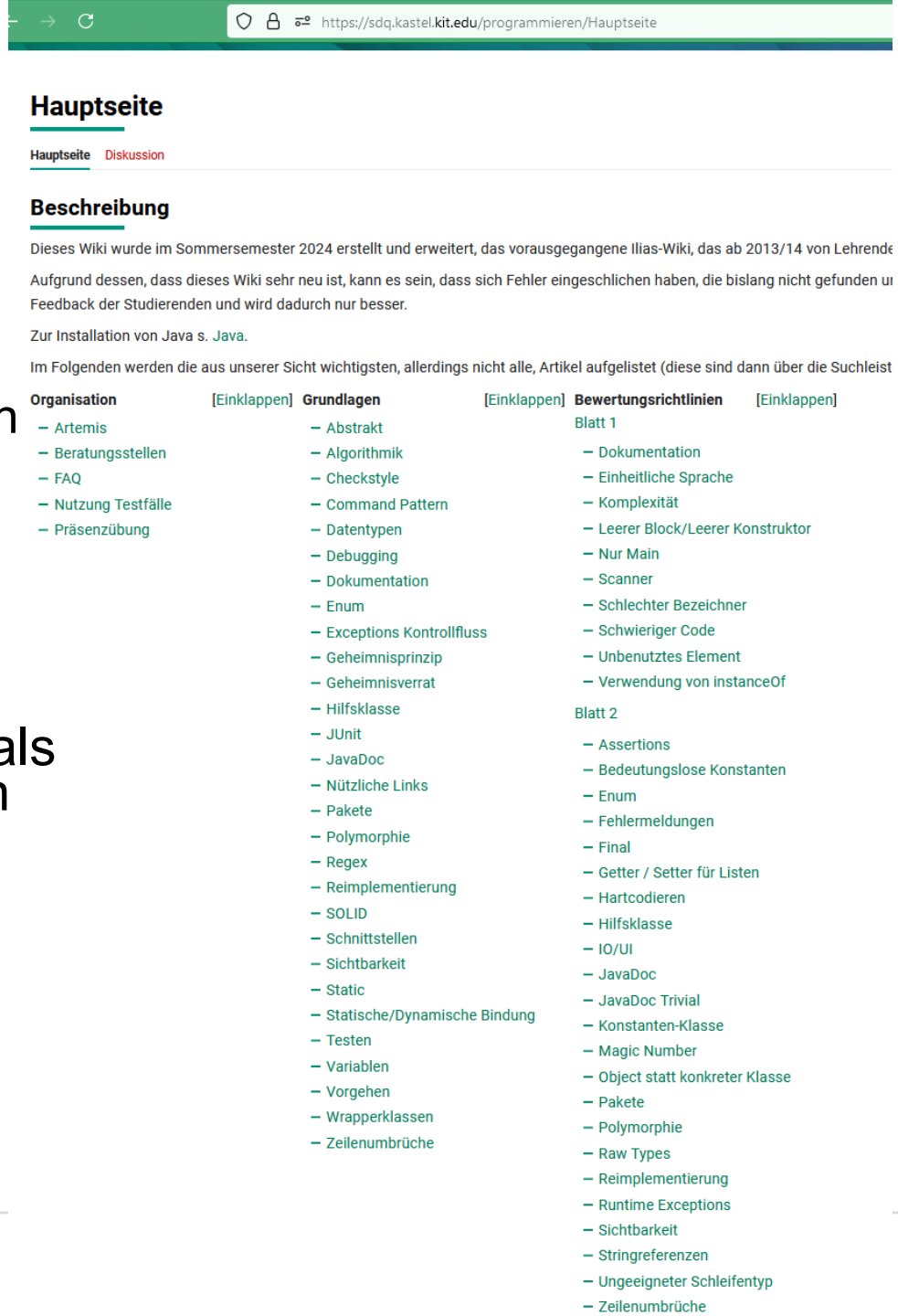
- **Verschiedene Optimierungsmöglichkeiten:**
  - Algorithmus austauschen / verbessern
    - im Bsp.: Äußere Schleife nur bis $\sqrt{n}$
  - Datenstrukturen verbessern
    - `boolean[]` statt `Mark[]`
  - Detailoptimierungen
- **Beste Vorgehensweise:**
  - Optimierungen erst **spät** im Entwurfsprozess
  - Nur "*hot spots*" optimieren
    - Es gibt Werkzeuge (*profiler*) zur Ermittlung von *hot spots*.
  - Optimierung kann Programmcode auch schlechter lesbar machen
    - Dokumentation besonders wichtig

Zweites Übungsblatt (Abgabe 28.11., 6:00 Uhr)

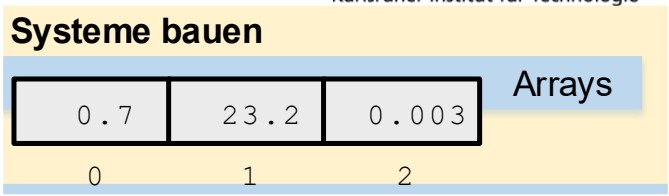
- Aufgabe A: String-Utility
  - Tipp: Sie können die String-Methode `charAt(int index)` verwenden, um auf ein Zeichen eines Strings zuzugreifen
    s. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html#charAt(int)
- Aufgabe B: Minkowski-Distanz
- Aufgabe C: Modellierung einer Lichtsignalanlage

- Tipp: In den Aufgaben B und C eignen sich Arrays als Datenstruktur um mit beliebig Objekten umzugehen
  - Vorgefertigte Listen aus `java.util` dürfen (noch) nicht verwendet werden

- Hinweise zu Bewertungsrichtlinien im Wiki https://sdq.kastel.kit.edu/programmieren/Hauptseite →



Zusammenfassung

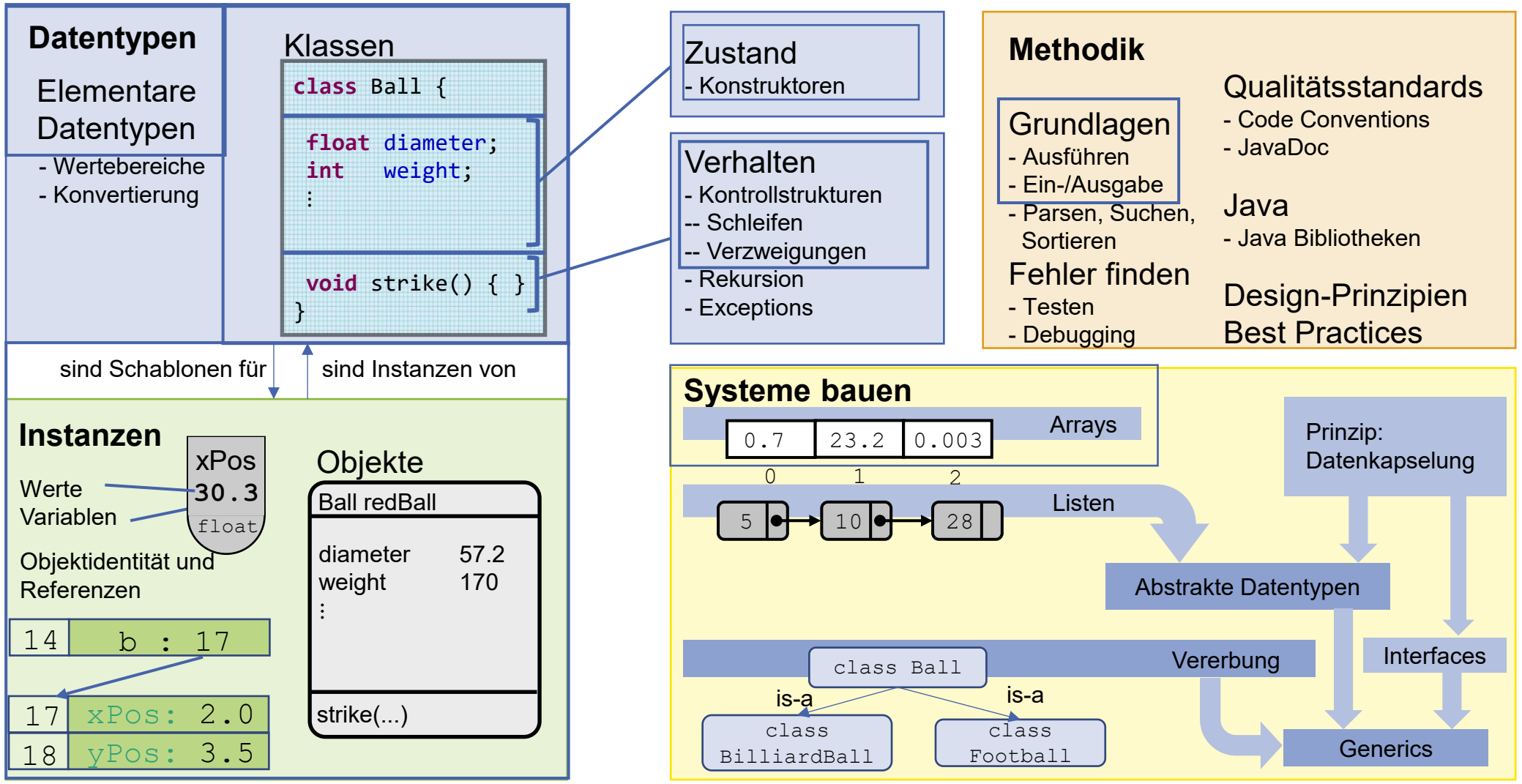


Arrays

- sind Folgen von Elementen desselben Typs
- Elemente werden über einen Index angesprochen

- **Arrays werden in Java wie Objekte behandelt**

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Literaturhinweis – Weiterlesen

- Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger "Grundkurs Programmieren in Java", 7. Auflage, 2014 (mit Java 8), Hansa-Verlag
  - "Referenzdatentypen"