252-0027
Einführung in die Programmierung

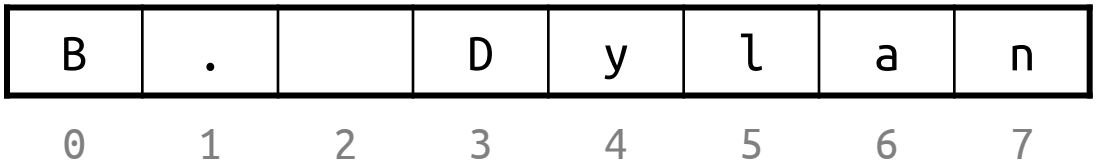
4. Sequenzen

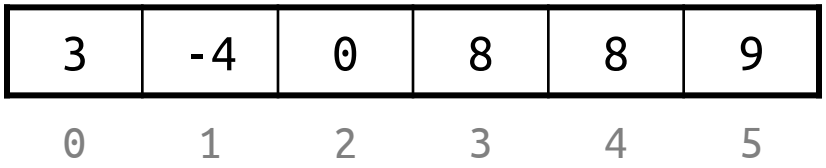
Manuela Fischer, Malte Schwerhoff

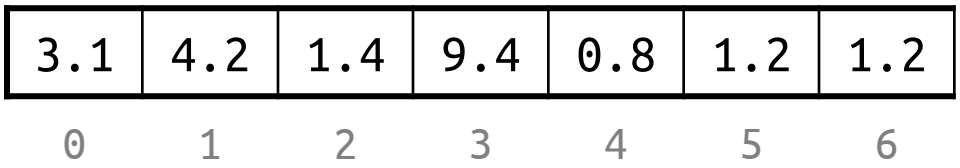
Departement Informatik
ETH Zürich

Sequenzen

geordnete (indexierbare) Reihe von Werten *des gleichen Typs*







4. Sequenzen

4.1 Strings

4.2 Arrays

4. Sequenzen

4.1 Strings

4.2 Arrays

Beispiele

```
String firstName = "James";
String lastName = "Bond";
String fullName = firstName + lastName;
System.out.println("My name is " + lastName + ", " + fullName + "." );
```

```
My name is Bond, James Bond.
```

```
int number = 6;
String numberToString = "" + number + 7; // "67"
```

Konversion nach String

```
String longText = """
                  First line.
                  Second line.""";
System.out.println(longText);
```

Leerzeichen werden dort ignoriert.

```
First line.
Second line.
```

Strings: Funktionalität

- Strings sind Objekte: Objekt `s` vom Typ `String`
  - Strings sind keine Arrays! Arrays folgen bald!
- Standardbibliothek enthält viel Funktionalität für Strings
  - Immer vorhanden, ohne `import java.util.*;`
  - Methoden werden mit dot-Notation aufgerufen: `s.method(parameters)`
    (führe Methode `method` für `s` aus, wende Methode `method` auf `s` an)
  - Ergebnis (Rückgabewert der Methode) kann sein: `String`, `int`, `boolean`, `char`
- Strings sind **unveränderbar** («**immutable**»)
  - Zuweisung verändert nicht den String: Variable verweist auf neuen String
  - Alle Methoden lassen String unverändert

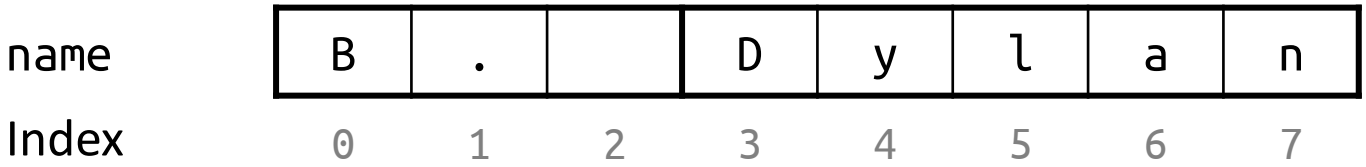
Länge eines Strings

- Ein String ist eine Sequenz von Zeichen
  - Einzelne Zeichen sind Werte des Basistyps `char`
- Länge des Strings ist definiert als Anzahl Zeichen
  - Länge kann mit `length()`-Methode abgefragt werden

```
String name = "B. Dylan";
int numberOfCharacters = name.length(); // 8
```

Zeichen und Indizes

- einzelne Zeichen haben einen Index (0-basiert)
  - Index des ersten Buchstabens ist 0
  - Index des letzten Buchstabens ist 1 weniger als die Länge



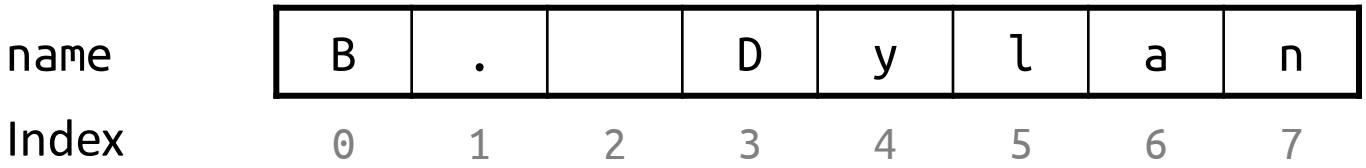
- Strings erlauben Zugriff auf die einzelnen Zeichen mit `charAt`-Methode
  - Liefert einen Wert vom Basistyp `char`
- `indexOf`-Methode gibt Index eines Zeichens zurück
  - Erster Index mit diesem Zeichen, falls es mehrere gibt

Beispiel: Einzelne Zeichen

char-Literale zwischen ' und '

```
String name = "B. Dylan";
char startsWith = name.charAt(0); // 'B'
char endsWith = name.charAt(name.length() - 1); // 'n'
System.out.println(startsWith);
System.out.println(endsWith);
System.out.println("letter y at index " + name.indexOf('y'));
```

```
B
n
letter y at index 4
```



Beispiel: Zeichen zählen

- Methode `count(char c, String s)`, die zählt wie oft ein Zeichen `c` in einem String s vorkommt

```
public static int count(char c, String s) {int count = 0;
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == c) {
      count++;
    }
  }
  return count;
}
```

Strikt kleiner!

`count('c', "computer science")` gibt 3 zurück

String-Methoden, die String liefern

Name der Methode	Beschreibung der Methode

`substring(i1, i2)`
or
`substring(i1)`
	Ein neuer String:
	Der Substring von `i1` (inklusive) bis `i2` (exklusive);
	falls kein `i2` übergeben wird: bis Ende des Strings

`toLowerCase()`	Ein neuer String mit nur Kleinbuchstaben
`toUpperCase()`	Ein neuer String mit nur Grossbuchstaben
`stripLeading()`	Ein neuer String ohne Leerzeichen am Anfang
`stripTrailing()`	Ein neuer String ohne Leerzeichen am Ende

Beispiel: Schreien!

- Text in Variable `String text = "Hello everyone!"` zu Grossbuchstaben

```
text.toUpperCase();
System.out.println(text);
```

```
Hello everyone!
```

String wird zwar erzeugt, aber nicht benutzt!

```
System.out.println(text.toUpperCase());
System.out.println(text);
```

```
HELLO EVERYONE!
Hello everyone!
```

String wird erzeugt und benutzt, aber nicht abgespeichert!

```
text = text.toUpperCase();
System.out.println(text);
```

```
HELLO EVERYONE!
```

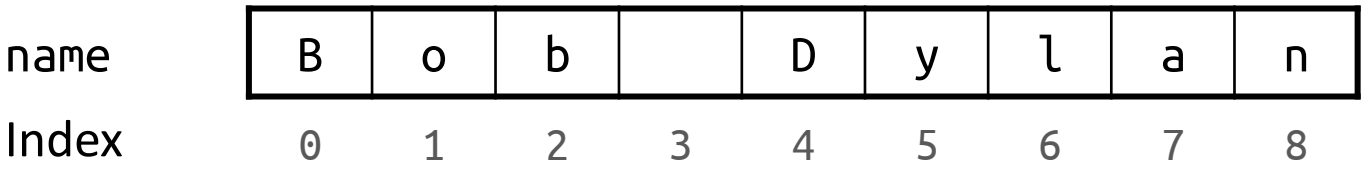
String wird erzeugt und abgespeichert: Alter String wird überschrieben.

Substrings

Indizes 0, 1, 2

```
String name = "Bob Dylan";
String firstName = name.substring(0, 3); // "Bob"
String lastName = name.substring(4); // "Dylan"
String firstCharacter = name.substring(0, 1); // "B"
name = name.toLowerCase(); // "bob dylan"
```

Von Index 4 bis Ende



"B" is same as "" + 'B'

Achtung: `name.substring(0, 1)` ist nicht das gleiche wie `name.charAt(0)`

- `name.substring(0, 1)` hat Wert `"B"` und ist vom Typ `String`
- `name.charAt(0)` hat Wert `'B'` und ist vom Typ `char`

String-Methoden, die `int` liefern

Name der Methode	Beschreibung der Methode
`length()`	Länge des Strings (Anzahl Zeichen)

`indexOf(s, fromIndex)`
`indexOf(c, fromIndex)`
`indexOf(s)`
`indexOf(c)`
	Index wo String `s` oder Zeichen `c` zum ersten Mal nach Index `fromIndex` im String auftaucht (-1 falls nicht gefunden);
	falls `fromIndex` nicht übergeben wird, von Anfang des Strings

```
String s = "S. Beckett";
int indexBeck = s.indexOf("Beck");          // 3
int eFirstOccurrence = s.indexOf('e');      // 4
int eSecondOccurrence = s.indexOf('e', 5); // 7
```

Beispiel 1

```
String vorlesung = "Einfuehrung in die Programmierung";
// index             012345678901234567890123456789012
                                    1         2         3
```

- Wie können wir das Wort "die" extrahieren?

```
// variant 1: less elegant
vorlesung.substring(15, 18);
```

```
// variant 2: more elegant
int index = vorlesung.indexOf("die");
vorlesung.substring(index, index + "die".length());
```

Beispiel 2

```
String vorlesung = "Einfuehrung in die Programmierung";
// index             012345678901234567890123456789012
                                    1         2         3
```

- Wie können wir das Wort "die" gross schreiben und alles andere klein?

```
String word = "die";
int indexStart = vorlesung.indexOf(word);  // 15
int indexEnd = indexStart + word.length(); // 18
String beginning = vorlesung.substring(0, indexStart).toLowerCase();
String end = vorlesung.substring(indexEnd).toLowerCase();
vorlesung = beginning + word.toUpperCase() + end;
System.out.println(vorlesung);
```

```
einfuehrung in DIE programmierung
```

o.m1().m2() ist (o.m1()).m2()

vorlesung zeigt auf neuen String

String-Methoden, die boolean liefern

Name der Methode	**Beschreibung der Methode** bei Aufruf `s1.method(s2)`
`equals(s2)`	ob `s1` und `s2` die gleichen Buchstaben enthalten
`equalsIgnoreCase(s2)`	ob `s1` und `s2` die gleichen Buchstaben enthalten, ohne Berücksichtigung von Gross- und Kleinschreibung
`startsWith(s2)`	ob `s1` mit den Buchstaben von `s2` anfängt
`endsWith(s2)`	ob `s1` mit den Buchstaben von `s2` endet
`contains(s2)`	ob `s1` irgendwo `s2` enthält

Hinweis zu `==` und `equals`

- `==` nur für Basistypen (z.B. `int` oder `char`), nicht für Strings
- `==` für Strings (oder Objekte im Allgemeinen) wird später erklärt!

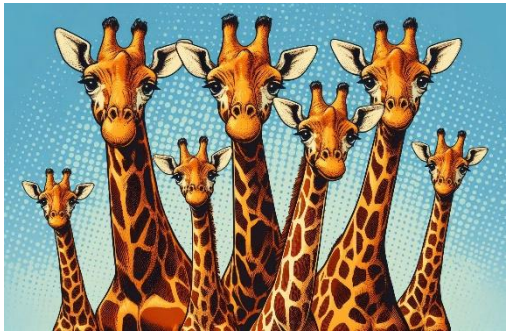
4. Sequenzen

4.1 Strings

4.2 Arrays

Motivation

- Sie wollen Sportdaten analysieren, z.B.
  - Durchschnittsgrösse
  - Anzahl Spieler/innen über Durchschnittsgrösse



Giraffengrafik erzeugt mit *Image Creator in Bing*

```
Anzahl Mitglieder? 6
Groesse in cm: 165
Groesse in cm: 164
Groesse in cm: 158
Groesse in cm: 163
Groesse in cm: 169
Groesse in cm: 181
Durchschnitt in cm = 166.7
Anzahl ≥ Durchschnitt: 2 (33 %)
```

Umzusetzende Schritte und ein Problem

1. Jeder Wert (Grösse) muss eingegeben werden
2. Durchschnitt = Werte aufsummieren, durch Anzahl Personen teilen
3. Zählen, wie viele Personen grösser-gleich der Durchschnitt sind
   - Aber den Durchschnitt kennen wir erst zum Schluss
   - Müssen die einzelnen Messwerte bis zum Ende speichern

Möglich, jeden Wert in einer Variablen zu speichern?

- Nein: Anzahl Personen ist eine *Programmeingabe*, d.h. *statisch* nicht bekannt

→ Wir brauchen einen Weg, beliebig viele Werte zu speichern

4. Sequenzen

4.2 Arrays

4.2.1 Basics: Deklaration, Initialisierung, Zugriff

4.2.2 Referenzen

4.2.3 Anwendungsbeispiele

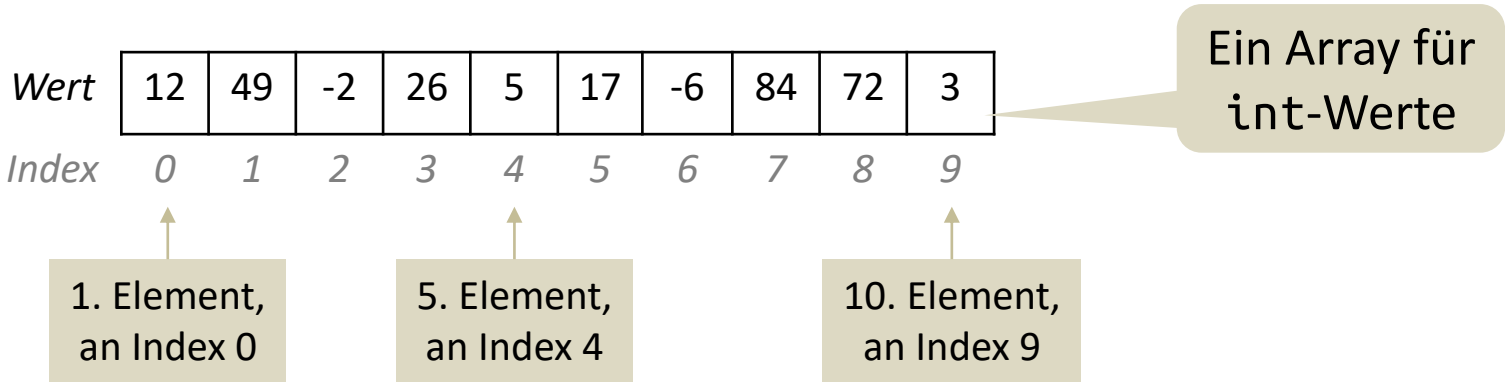
4.2.4 Funktionalität und Arrays-Klasse

4.2.5 Mehrdimensionale Arrays

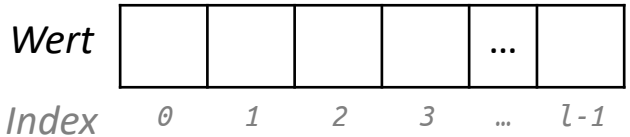
Arrays

Ein **Array** («Reihe») speichert mehrere Werte *des gleichen Typs*

- **Element**: Ein Wert in einem Array
- **Index**: Zahl (≥ `0`), um ein Element eines Arrays auszuwählen
  - **Wichtig**: Das erste Element hat den Index `0`



Ausblick: Arrays nutzen



Wir müssen bzw. wollen

- Arrays deklarieren und initialisieren
- Array-Elemente lesen und schreiben
- Grösse/Länge eines Arrays abfragen
- Arrays an Methoden übergeben bzw. von Methoden zurückbekommen

Arrays deklarieren …

- **Syntax:** `type[] name = new type[length];`
  - Array-getypte Variable *name*
  - Für *einen** gültigen Java-Typ *type*
  - Mit Platz für *length* (`int`, ≥ `0`) Elemente vom Typ *type* — neu

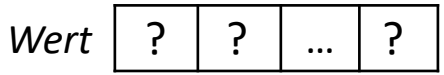
  } analog zu bisherigen Deklarationen
- Operator `new` für Arrays
  - Erzeugt ein neues Array(-Objekt) entsprechender Grösse
  - Initialisiert das Array mit Standardwerten (gleich mehr)
  - Operator `new` ist allgemeiner
    - Tauchte schon vorher auf, z.B. im Kontext von `Scanner`
    - Wird später im Kapitel zu Klassen und Objekten detaillierter behandelt

* Später verallgemeinert zu `type1[] name = new type2[length];`

… und initialisieren

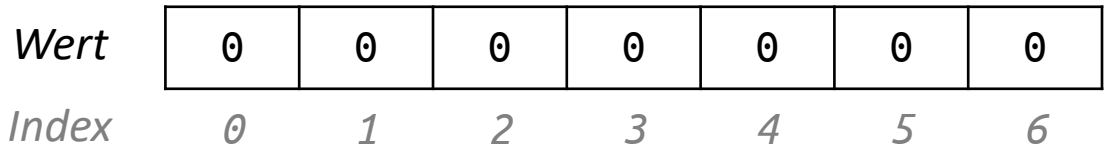
- **Syntax:** `type[] name = new type[length];`
- Array wird mit «Null-äquivalenten» Standardwerten vorbelegt

Typ	Standardwert
`int`, `long`	`0`
`double`	`0.0`
`boolean`	`false`
`String`, Objekte	`null`

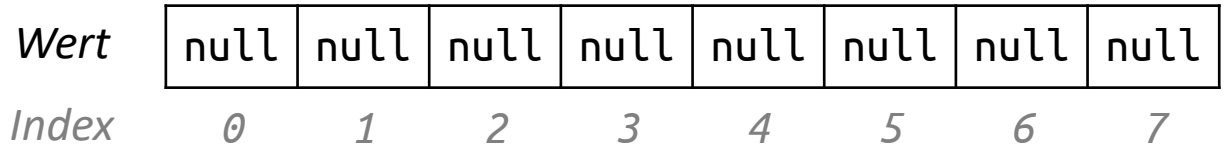


Beispiele

```
int[] numbers = new int[7];
```




```
int x = 4; // a non-negative value
String[] names = new String[2*x];
```



Arrays *nur* deklarieren

- **Syntax:** `type[] name;`
- **Beispiel:** `double[] prices;`
- Deklariert eine (Array-getypte) Variable, aber erstellt (noch) kein neues Array
  - Daher (noch) keine Länge nötig
  - Initialisierung dann später nötig, z.B. `prices = new double[7];`
  - Variable kann erst nach der Initialisierung genutzt werden (wie bisher auch schon)

Arrays lesen und schreiben

Der **Index-Operator** [ ] ermöglicht **Lese-** *bzw.* **Schreibzugriff** *(«***read/write access***»)* auf die Elemente eines Arrays

Lesezugriff

- **Syntax:** *name*[*index*]
- **Beispiele**
  - `System.out.println(names[2]);`
  - ```
    int i = …;
    double total = 2 * prices[i+1];
    ```

Schreibzugriff

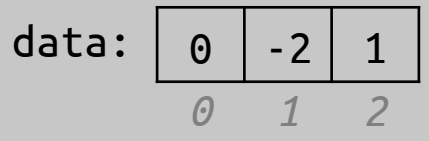
- **Syntax:** *name*[*index*] = expr;
- **Beispiele**
  - `names[i] = "Pip";`
  - `prices[2] = i / 3.0;`
    ```

Beispielausführung

Java

```
int[] data;
data = new int[3];
System.out.println(data[2]);
data[1] = -2;
int i = 11 / 5;
data[i] = i - 1;
System.out.println(data[data[2]]);
```

Zustand

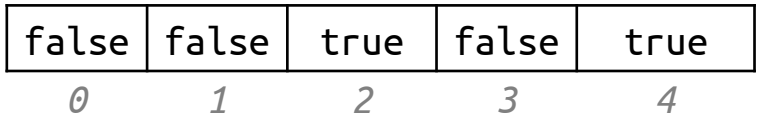


Ausgabe

```
0
-2
```

Beispiele für nicht-numerische Typen

```
boolean[] results = new boolean[5];
results[2] = true;
results[4] = true;
```

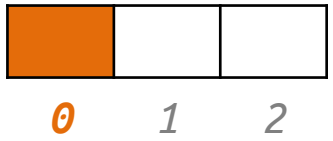



```
String[] addresses = new String[99];
addresses[3] = "Idastrasse 313";
```

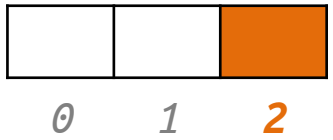
Dies ist eine vereinfachte Visualisierung. Mehr Details folgen später!

Typische Index-Werte

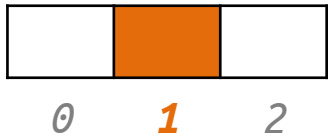
- Erstes Element: `data[0]`

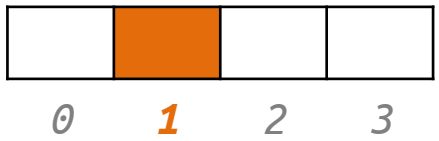


- Letztes Element: `data[data.length - 1]`



- Element in der Mitte: `data[(data.length - 1)/2]`





(Un)Gültige Index-Werte

- Legale Indizes: Zwischen `0` und Array-Länge `-1` (einschliesslich)
- Lese- oder Schreibzugriff mit einem ungültigen Index, d.h. ausserhalb dieses Bereichs, ist ein *Laufzeitfehler* (Out-of-Bounds-Fehler)

  In Java: *Programmabbruch* durch `ArrayIndexOutOfBoundsException`.

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

System.out.println(data[0]); // OK
data[4] = 8; // OK

System.out.println(data[-1]); // ERROR
data[5] = 13; // ERROR
```

Typischer Off-by-One-Fehler

Kleine Arrays direkt initialisieren

- Manchmal nötig: Array mit bekannten Werten belegen

- **Syntax:** *type*`[]` *name* `= {`$expr_1$`, …, `$expr_n$`};`
  - Deklariert und initialisiert ein Array der Grösse *n*
  - Ausdrücke müssen zum Typ *type* passen
  - Nur bei Deklaration (nicht als Zuweisung) möglich
- **Beispiele**
  - `int[] first_five_primes = {2, 3, 2+3, 7, 11};`
  - `String[] my_pets = {"Miezi", "Maunzi", "Brutus"};`

Kleine Arrays direkt initialisieren

- Alle drei Codeschnipsel erstellen das gleiche Array
- Direkte Initialisierung (zuoberst)
  - Kürzer und klarer als die Alternativen
  - Nutzen, insbesondere wenn die Werte keinem Muster folgen und daher nicht in einer Schleife berechnet werden können

```
int[] triple = {5, 8, 9};
```

```
int[] triple = new int[3];
triple[0] = 5;
triple[1] = 8;
triple[2] = 9;
```

```
int[] triple;
triple = new int[3];
triple[0] = 5;
triple[1] = 8;
triple[2] = 9;
```

Arrays und Schleifen

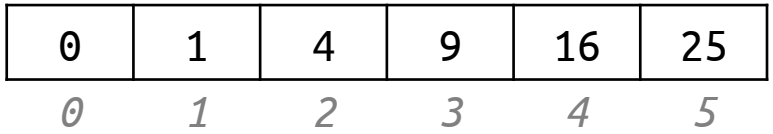
Zugriff auf beliebig viele Array-Elemente? ➜ Schleifen

```
int[] squares = new int[6];

for (int i = 0; i < 6; i++) {
  squares[i] = i * i;
}

for (int i = 0; i < 6; i += 2) {
  System.out.println(squares[i]);
}
```

Ausgabe: 0 4 16



Das `length`-Attribut

Das `length`-Attribut eines Arrays liefert die Anzahl der Elemente.

```
int[] squares = new int[6];

for (int i = 0; i < 6; i++) {
  ...
}
```

```
int[] squares = new int[6];

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

Achtung: ohne Klammern!

- Weniger fehleranfällig, z.B. falls Arraylänge nachträglich geändert wird
- Bei Übergabe an Methoden muss Länge nicht separat mitgegeben werden

Beispiel: Maximum eines Arrays

```
int[] numbers = {4, 2, 6, 5, 7, 1};
int currentMax = Integer.MIN_VALUE;
int currentMaxIndex = -1;
for (int i = 0; i < numbers.length; i++) {
  if (numbers[i] > currentMax) {
    currentMax = numbers[i];
    currentMaxIndex = i;
  }
}
System.out.print("max value " + currentMax);
System.out.println(" at index " + currentMaxIndex);
```

Dummy-Wert, der kleiner ist als alle…

Dummy-Index

```
max value 7 at index 4
```

Zurück zur Analyse von Messdaten

Ein Array erlaubt es uns, diese Datenanalyse zu implementieren

```
Anzahl Mitglieder? 6
Groesse in cm: 165
Groesse in cm: 164
Groesse in cm: 158
Groesse in cm: 163
Groesse in cm: 169
Groesse in cm: 181
Durchschnitt in cm = 166.7
Anzahl ≥ Durchschnitt: 2 (33 %)
```

Messdatenanalyse: Implementation

```java
// Liest Grössen, berechnet Durchschnitt, gibt
// Anzahl und Prozentsatz >= Durchschnitt aus

import java.util.Scanner;

public class Analyse {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Anzahl Mitglieder? ");
    int members = input.nextInt();

    // Grössen einlesen und direkt aufsummieren
    int[] data = new int[members];
    double sum = 0.0;
    for (int i = 0; i < members; i++) {
      System.out.print("Grösse in cm: ");
      data[i] = input.nextInt();
      sum += data[i];
    }
```

```java
    // Durchschnitt berechnen
    double avg = (double) sum / members;
    System.out.println("Durchschnitt in cm = " + avg);

    // Überdurchschnittliche Grössen zählen
    int count = 0;
    for (int i = 0; i < members; i++) {
      if (data[i] >= avg) {
        count++;
      }
    }

    // Ergebnisse ausgeben
    System.out.println(
        "Anzahl >= Durchschnitt: " + count +
        " (" + (count * 100.0 / members) + "%)");
  }
}
```

4. Sequenzen

4.2 Arrays

4.2.1 Basics: Deklaration, Initialisierung, Zugriff

4.2.2 Referenzen

4.2.3 Anwendungsbeispiele

4.2.4 Funktionalität und Arrays-Klasse

4.2.5 Mehrdimensionale Arrays

Methodenaufrufe: Was passiert hier?

```
public static void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

```
int x = 1;
int y = 2;
swap1(x, y);
// Inhalt von x und y?
```

```
public static void swap2(int[] data) {
  int temp = data[0];
  data[0] = data[1];
  data[1] = temp;
}
```

```
int[] pair = {1, 2};
swap2(pair);
// Inhalt von pair?
```

Zuweisungen: Was passiert hier?

```
int x = 1;
int y = x;
y++;
// x == 1, y == 2
```

- Beobachtung
  - Änderung an y hat keinen Einfluss auf x
  - x und y sind «unabhängig»

```
int[] arr1 = {1, 3};
int[] arr2 = arr1;
arr2[0]++;
// arr1 == {2, 3}
```

- Beobachtung
  - Änderung an `arr2` hat einen Einfluss auf `arr1`
  - `arr1` und `arr2` sind «abhängig»

`int y; y = x;` — bzw. `int[] arr2; arr2 = arr1;` — ergeben dasselbe Verhalten

Werte vs. Referenzen

- Primitive Typen haben **Wertesemantik** («**value semantics**»)
  - Z.B. `int`, `double`, `boolean`
  - Bei Zuweisungen (und Methodenaufrufen) wird der Wert *kopiert*

```
int x = 1;
int y = x;
y++;
// x == 1, y == 2
```

- Alle anderen Typen haben **Referenzsemantik** («**reference semantics**»)
  - Inklusive Arrays und Strings
  - Bei Zuweisungen (und Methodenaufrufen) wird ein **Alias** («**alias**», «**aliasing**») erzeugt

```
int[] arr1 = {1, 3};
int[] arr2 = arr1;
arr2[0]++;
// arr1 == {2, 3}
```

Diese Einteilung gilt für Java. Die beiden unterschiedlichen Semantiken finden sich aber in vielen anderen Sprachen wieder.

Warum Referenzsemantik?

1. **Effizienz:** Kopien vermeiden

   Grosses Array → viel Speicherplatz
   → Kopieren kostet Zeit & Platz

Kopie nicht effizient

```
int[] data = …;
boolean result = isSorted(data);
```

2. **Funktionalität:** Gemeinsam Daten verwalten

   Intuitives Beispiel: Mehrere Personen nutzen ein Auto → jede Fahrt ändert die Position *dieses einen* Autos

Kopie nicht sinnvoll

```
Car sharedCar = …;
driveCarTo(sharedCar, "Bern");
```

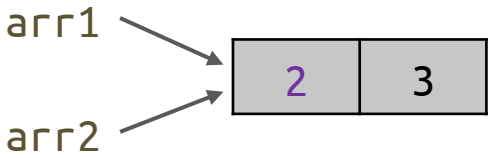
Zuweisungen: Werte vs. Referenzen

```
int x = 1;
int y = x;
y++;
```

- x und y benennen *separate* Speicherplätze
- y bekommt eine (eigene) *Kopie* des Werts von x


```
int[] arr1 = {1, 3};
int[] arr2 = arr1;
arr2[0]++;
```

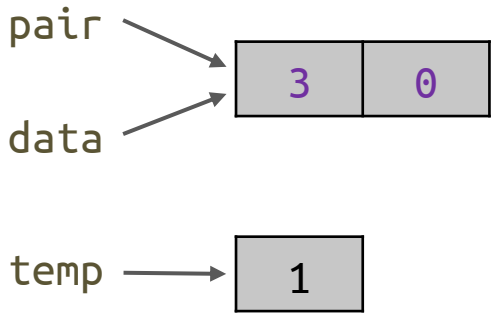
- Für `arr1` wird ein neues Array erstellt
- Anschliessend wird `arr2` ein *Alias* von `arr1`



Referenzen als Methodenparameter

```
int[] pair = {1, 3};
swap2(pair);
pair[1]--;
```

```
// PRE: data.length > 0
public static void swap2(int[] data) {
  int temp = data[0];
  data[0] = data[1];
  data[1] = temp;
}
```

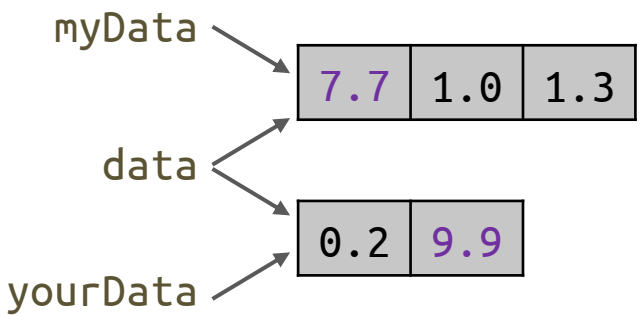


Der Umstand, dass `swap2(pair)` eine Änderung an `pair` bewirkt, wird **Seiteneffekt** («**side effect**») genannt.

Referenzen können «umgehängt» werden

Worauf eine Referenz zeigt, kann geändert werden

```
double[] myData = {1.1, 1.0, 1.3};
double[] yourData = {0.2, 0.5};
double[] data = myData;
data[0] = 7.7;
data = yourData;
data[1] = 9.9;
```



- `data = myData` verändert, worauf `data` zeigt: es wird (neu) ein Alias von `myData`
- `data[0] = 7.7` verändert den Inhalt des Arrays hinter `data`
- Analog für `data = yourData` bzw. `data[1] = 9.9`

Null-Referenzen

Statt auf ein Objekt (z.B. ein Array) können Variablen (von einem Referenztyp) auch auf nichts zeigen. Ihr Wert ist dann `null`.

- **Nullreferenz** («**null reference**»), **Nullzeiger** («**null pointer**»)

```
int[] data = null;
System.out.println(data);
data = new int[3];
```

Ausgabe «`null`»
- Es wurde noch kein Array erzeugt, d.h. kein Speicherplatz belegt (alloziert)
- data zeigt daher auf nichts

Erst jetzt wird ein Array der Grösse 3 erstellt, auf das `data` dann zeigt

NullPointerException

- Auf `null` können keine Operationen ausgeführt werden
  - Wird dies versucht, bricht das Programm mit einer **`NullPointerException`** ab

```
int[] data = new int[2];
...
data = null;
data[0] = 123; // ERROR
```

```
int[] data = null;
int n = data.length; // ERROR
```

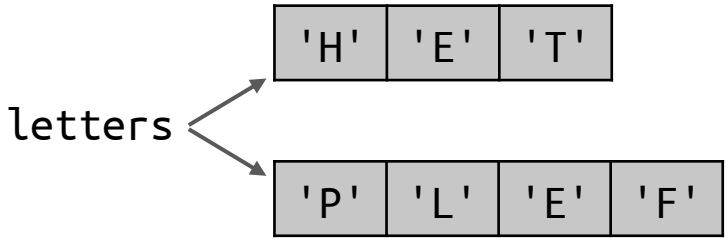
- Mit `if (data != null) { … }` Laufzeitfehler vorbeugen

```
if (data != null && data.length > 0) {
  System.out.println(data[0]);
}
```

Kurzschlussauswertung

Unerreichbare Objekte — Kann das weg?

```
char[] letters = {'H', 'E', 'T'};
char[] l = {'P', 'L', 'E', 'F'};
letters = l;
...
```



Wenn keine Referenzvariable mehr auf ein Objekt zeigt — hier auf das zuerst erzeugte Array — dann ist das Objekt *unerreichbar.*

- Keine direkte Auswirkung auf die Programmausführung
- Wird irgendwann vom Java-Laufzeitsystem entfernt werden («garbage collection»)
  - Details abhängig von vielen Faktoren (Java-Version, Arbeitsspeicher, …)
  - Nicht unser Thema — können uns auf das Erstellen korrekter Programme konzentrieren

4. Sequenzen

4.2 Arrays

Methode `replace`

- **Gewünscht:** Methode `replace(what, array, with)`, die alle Vorkommen von Element `what` in `array` durch `with` ersetzt
- Beispielanwendungen:

```
int[] data = {1, 4, 7, 4, 2};
replace(4, data, 0);
// data now {1, 0, 7, 0, 2}
```

```
int[] data = {3, 2, 1};
replace(7, data, 1);
// data still {3, 2, 1}
```

- Das Programm muss für `int`-Arrays jeder Grösse und Belegung funktionieren

Methode `replace`

```
void replace(int what, int[] array, int with) {
  for (int i = 0; i < array.length; i++) {
    if (array[i] == what) {
      array[i] = with;
    }
  }
}
```

```
int[] data = {1, 4, 7, 4, 2};
replace(4, data, 0);
// data now {1, 0, 7, 0, 2}
```

Beobachtungen

- `int[]` ist bloss ein weiterer Parametertyp (analog zu `int`, `String`, …)
- Typinformation enthält keine Längenangabe (`int[] array`, nicht z.B. `int[5] array`)
- Parameterübergabe auf Aufrufseite (`data`) erfolgt ohne eckige Klammern
- Keine Rückgabe notwendig (`void`), da Arrays per Referenz übergeben werden

Methode `allTrue`

- **Gewünscht:** Methode `allTrue(array)` gibt `true` zurück, genau dann wenn `array` nur `true`s (aber kein `false`) enthält
- Beispielanwendungen:

```
boolean[] data = {true};
allTrue(data); // returns true
```

```
boolean[] data = {true, false};
allTrue(data); // returns false
```

- Das Programm muss für `boolean`-Arrays jeder Grösse und Belegung funktionieren

Methode `allTrue` (v1)

```
boolean allTrue(boolean[] values) {
  boolean result = true;

  // Konjunktion über alle Elemente bilden
  for (int i = 0; i < values.length; i++) {
    result = result && values[i];
  }

  return result;
}
```

```
boolean[] data = {true, false};
boolean res = allTrue(data);
// res == false
```

Implementation kann noch effizienter werden — wie?

Beobachtungen

- Parameterdeklaration und Methodenaufruf wie erwartet (analog zu vorher)
- Passender Rückgabetyp (`boolean`) und entsprechendes `return` notwendig

Methode `allTrue` (v2)

```
boolean allTrue(boolean[] values) {
  for (int i = 0; i < values.length; i++) {
    if (!values[i]) return false;
  }

  return true;
}
```

```
boolean[] data = {true, false};
boolean res = allTrue(data);
// res == false
```

Vergleich v1 vs. v2:

- Methodensignaturen sind identisch
- v1 iteriert immer über alle Elemente. v2 hingegen bricht ab, sobald das erste `false` gefunden wurde.

Methode concat

- **Gewünscht:** Methode `concat(array1, array2)` gibt ein *neues* Array zurück, dass aus den Elementen von `array1` gefolgt von denen aus `array2` besteht
- Beispielanwendungen:

```
int[] arr1 = {1, 3};
int[] arr2 = {0, 2, 4};
int[] arr3 = concat(arr1, arr2);
// arr3 == {1, 3, 0, 2, 4}
```

```
int[] arr1 = {};
int[] arr2 = {1, 2};
int[] arr3 = concat(arr1, arr2);
// arr3 == {1, 2}
```

- Für `int`-Arrays aller Grössen und Belegungen

Methode concat

```
int[] concat(int[] first, int[] second) {
  int[] result
    = new int[first.length + second.length];
  // Elemente aus first rüberkopieren
  for (int i = 0; i < first.length; i++) {
    result[i] = first[i];
  }
  // Elemente aus second dahinter kopieren
  for (int i = 0; i < second.length; i++) {
    result[first.length + i] = second[i];
  }
  return result;
}
```

```
int[] arr1 = {};
int[] arr2 = {1, 2};
int[] arr3 = concat(arr1, arr2);
// arr3 == {1, 2}
```

Beobachtungen

- Rückgabetyp nun ebenfalls ein `int`-Array (keine Überraschung)
- Neues Array
  1. Innerhalb der Methode erstellt
  2. Via Referenz-Semantik zurückgegeben
  3. Auf Aufruferseite als `arr3` bekannt

4. Sequenzen

4.2 Arrays

Was Arrays nicht können

- Arrays haben ihren Ursprung in der hardwarenahen Programmierung
  → effizient, aber (in Java) nicht sonderlich benutzerfreundlich
- Beispieloperationen, die *nicht* unterstützt werden (→ Compilerfehler)
  - `array1 + array2` — Keine arithmetischen Operatoren (+, *, etc.)
  - `array1 < array2` — Keine relationalen Operatoren (<, <=, etc.)

```
jshell> int[] array = {1, 2, 3}
array ==> int[3] { 1, 2, 3 }

jshell> array >= array
|  Error:
|  bad operand types for binary operator '>='
|    first type:  int[]
|    second type: int[]
```

```
jshell> int[] array = {1, 2, 3}
array ==> int[3] { 1, 2, 3 }

jshell> array + array
|  Error:
|  bad operand types for binary operator '+'
|    first type:  int[]
|    second type: int[]
```

- Auch können Arrays nicht zur Laufzeit vergrössert/-kleinert werden

Was Arrays nicht wie erwartet können

- Zudem verhalten sich wichtige Operationen *nicht*, wie vielleicht erwartet:

```
int[] arr1 = {3, 1, 3};
System.out.println(arr1);
```

Gibt *nicht* die Elemente des Arrays aus

```
int[] arr1 = {3, 1, 3};
int[] arr2 = {3, 1, 3};
if (arr1 == arr2) …
if (arr1.equals(arr2)) …
```

Beide Tests evaluieren zu *false*

- Die Standardbibliothek stellt passende Methoden bereit.

Klasse `Arrays` im Paket `java.util`

Die Klasse `java.util.Arrays` stellt zahlreiche praktische Methoden zum Arbeiten mit Arrays zur Verfügung. Auszug:

Methode	Beschreibung
`copyOf(array, length)`	Returns a new copy (of specific length) of an array
`equals(array1, array2)`	Returns true if the two arrays contain the same elements in the same order
`fill(array, value)`	Sets every element to the given value
`sort(array)`	Sorts the array's elements
`toString(array)`	Returns a string representing the array, e.g., `"[2, 7, 1]"`

Bei Interesse: https://docs.oracle.com

Methode `Arrays.toString`

```java
import java.util.Arrays;

double[] percentages = {25, 33.3, 91.7};

System.out.println(percentages);
  // Ausgabe: "[D@3327bd23" (oder ähnlich)

System.out.println(Arrays.toString(percentages));
 // Ausgabe: "[25.0, 33.3, 91.7]"
```

Geht-so hilfreich … 🤷‍♀️

Geht doch!

`Arrays.toString()`

- Funktioniert für alle Array-Typen: `double[]`, `int[]`, `String[]`, …
- Gibt einen `String` zurück (d.h. erzeugt nicht selbst eine Ausgabe)

Methode `Arrays.equals`

```java
import java.util.Arrays;

double[] parts1 = {16, 9, 2024};
double[] parts2 = {16, 9, 2024};

if (parts1 == parts2) { … }
if (parts1.equals(parts2)) { … }

if (Arrays.equals(parts1, parts2)) { … }
```

Beide Tests `false`

Test nun (wie erwartet) `true`

`Arrays.equals()`

- Funktioniert für alle Array-Typen
- Gibt einen `boolean` zurück

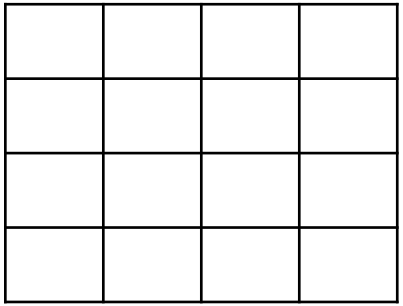
4. Sequenzen

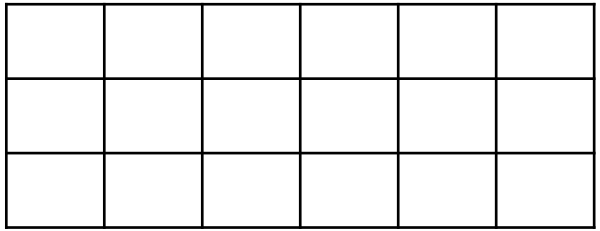
4.2 Arrays

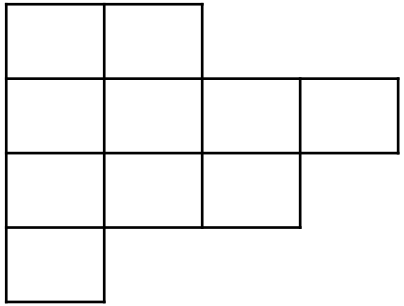
Mehrdimensionale Arrays

Mehrdimensionales («**multidimensional**») Array:

- ein Array von Arrays des gleichen Typs
- nützlich für Tabellen, Matrizen, …







Deklaration und Initialisierung

- Deklaration (ohne Erstellung des Arrays)

```
type[][] name;
```

- Deklaration mit Initialisierung (Null-ähnliche Default-Werte)

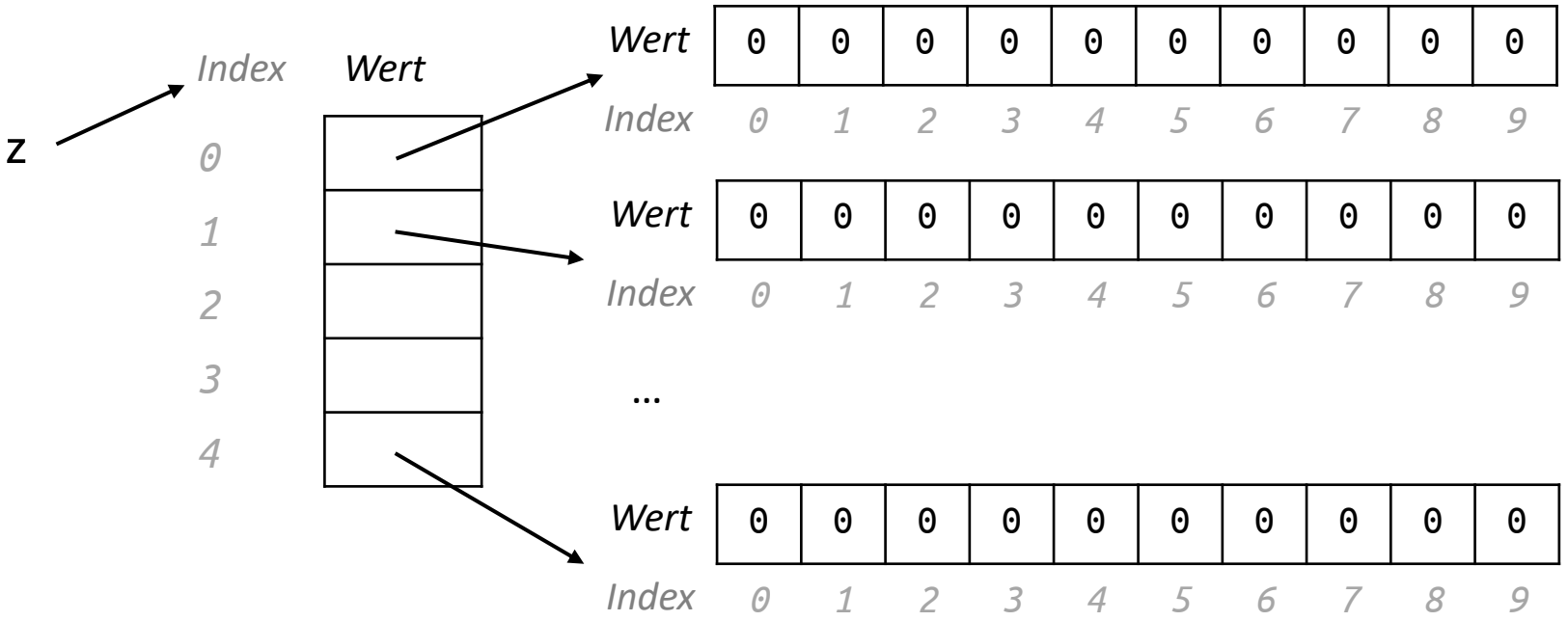
```
type[][] name = new type[nRows][nColumns];
```

- Direkte Initialisierung

```
int[][] sudoku = { { 1, 2, 3 }, // first row
                   { 4, 5, 6 }, // second row
                   { 7, 8, 9 }  // third row
                 };
```

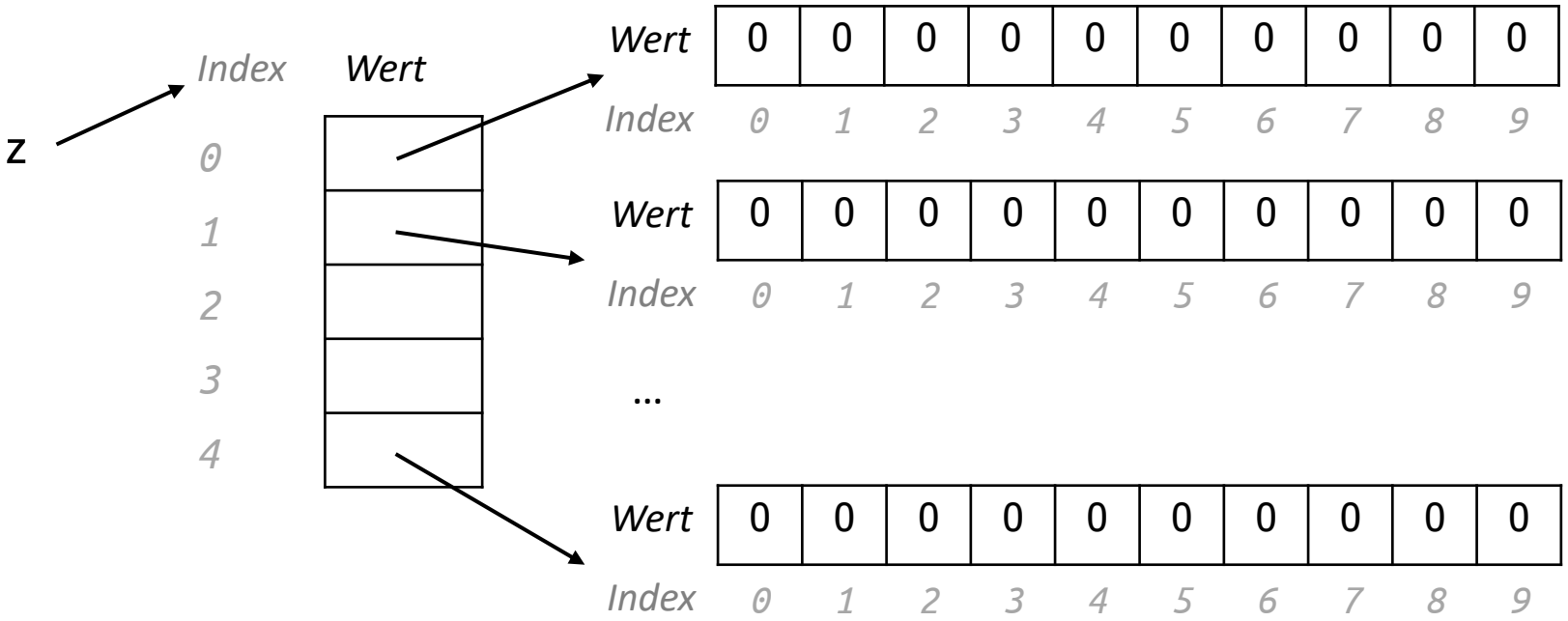
Beispiel

```
int[][] z = new int[5][10];
```



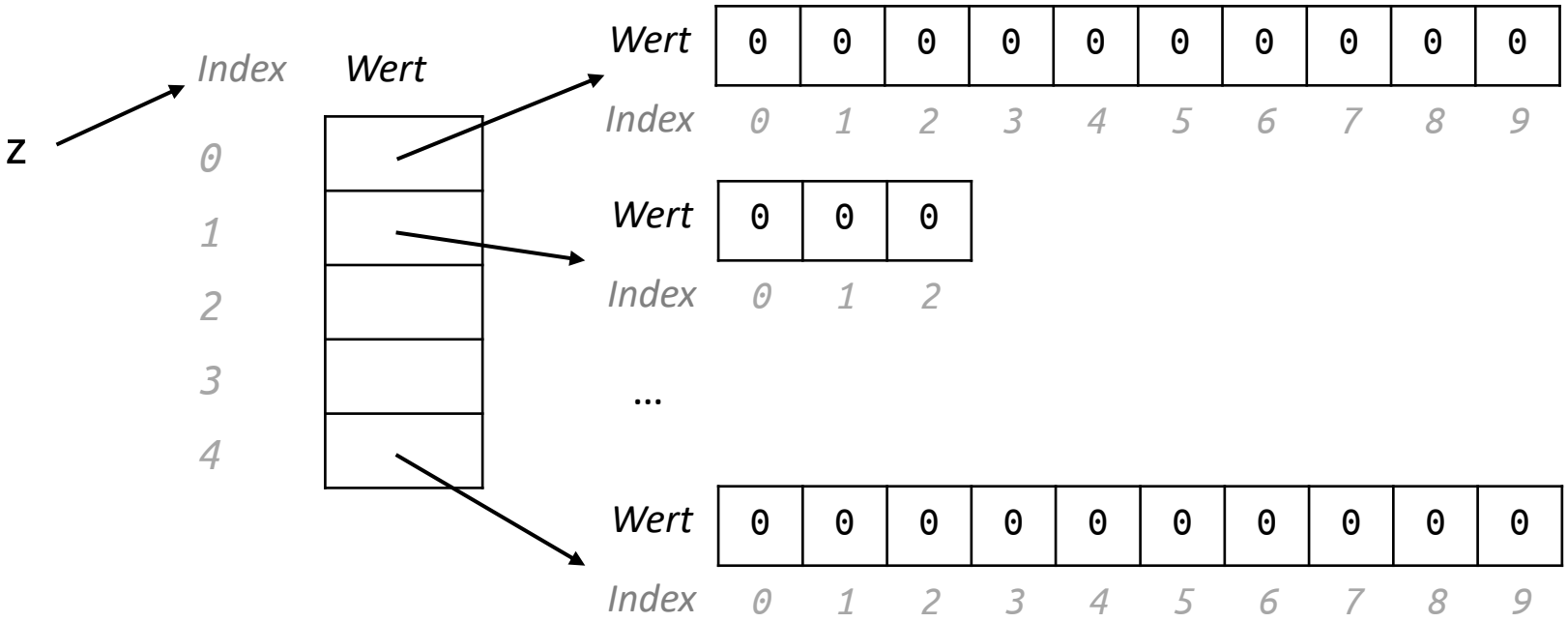
Lese- und Schreibezugriff

`z[row][column]` für ein Element oder `z[row]` für eine Zeile



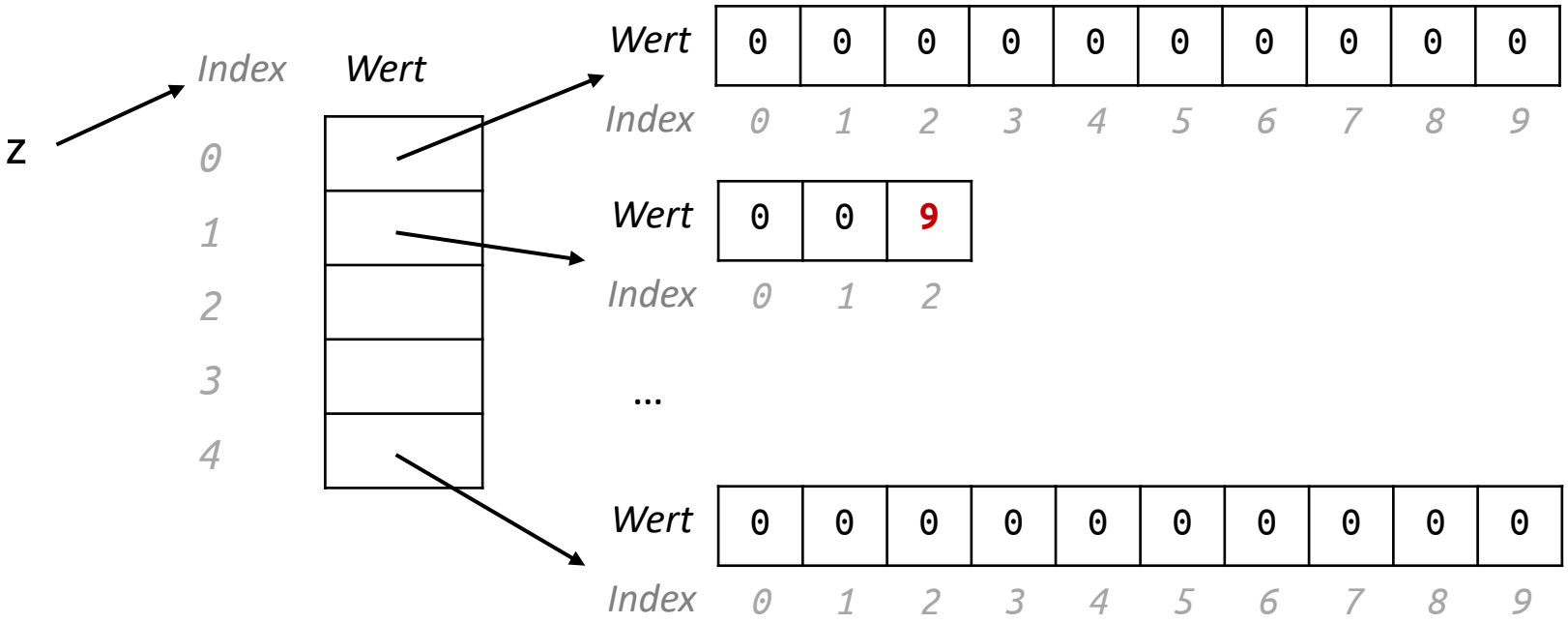
Beispiel (Fortsetzung)

```
z[1] = new int[3];
```



Beispiel (Fortsetzung 2)

```
z[1][2] = 9;
```



Multidimensionale Matrizen ausgeben

```
int[][] table = { {1, 2}, {3, 4}, {5, 6} };
```

- direkt ausgeben

```
System.out.println(table);
// outputs [[I@156643d4
```

- mit `toString` ausgeben

```
System.out.println(Arrays.toString(table));
// outputs [[I@156643d4, [I@123a439b, [I@7de26db8]
```

- mit `deepToString` ausgeben

```
System.out.println(Arrays.deepToString(table));
// outputs [[1, 2], [3, 4], [5, 6]]
```

Multidimensionale Arrays und verschachtelte Schleifen

Java

```
int[][] table = new int[3][6];
for (int row = 0; row < table.length; row++) {
  for (int column = 0; column < table[row].length; column++) {
    table[row][column] = row + column;
    System.out.print(table[row][column]);
  }
  System.out.println();
}
```

Wichtig, insbesondere bei nicht-rechteckigen 2D-Arrays!

Zustand

table:

0	1	2	3	4	5
1	2	3	4	5	6
2	3	4	5	6	7

row: 3

column: 6

Output

```
012345
123456
234567
```

Mehr als 2 Dimensionen…

- Beliebig verallgemeinerbar

```
int[][][] table = { { {1, 2}, {2} },
                    { {1, 2}, {1}, {2}, {3} },
                    { {5}, {6, 7} }
                  };
```