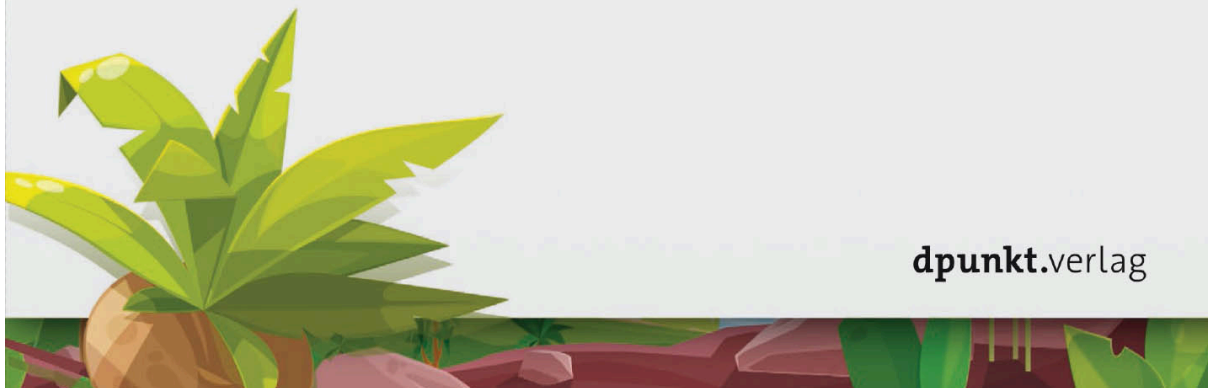




# Einfach Michael Inden Java

Gleich richtig  
programmieren  
lernen



**dpunkt.verlag**

# Inhalt

**Cover**

**Über der Autor**

**Titel**

**Impressum**

**Dedication**

**Inhaltsverzeichnis**

**Vorwort**

**I Einstieg**

1 Einführung

1.1 Java im Überblick

1.2 Los geht's – Installation

1.2.1 Java-Download

1.2.2 Installation des JDKs

1.2.3 Nacharbeiten nach der Java-Installation

1.2.4 Java-Installation prüfen

1.3 Entwicklungsumgebungen

1.3.1 Installation von Eclipse

1.3.2 Eclipse starten

1.3.3 Erstes Projekt in Eclipse

1.3.4 Erste Klasse in Eclipse

2 Schnelleinstieg

2.1 Hallo Welt (Hello World)

2.2 Variablen und Datentypen

2.2.1 Definition von Variablen

2.2.2 Bezeichner (Variablennamen)

## 2.3 Operatoren im Überblick

### 2.3.1 Arithmetische Operatoren

### 2.3.2 Zuweisungsoperatoren

### 2.3.3 Vergleichsoperatoren

### 2.3.4 Logische Operatoren

## 2.4 Fallunterscheidungen

## 2.5 Methoden

### 2.5.1 Methoden aus dem JDK nutzen

### 2.5.2 Eigene Methoden definieren

### 2.5.3 Nützliche Beispiele aus dem JDK

### 2.5.4 Signatur einer Methode

### 2.5.5 Fehlerbehandlung und Exceptions

## 2.6 Kommentare

## 2.7 Schleifen

### 2.7.1 Die for-Schleife

### 2.7.2 Die for-each-Schleife

### 2.7.3 Die while-Schleife

### 2.7.4 Die do-while-Schleife

## 2.8 Rekapitulation

## 2.9 Weiterführende Dokumentation für nächste Schritte

## 2.10 Aufgaben und Lösungen

### 2.10.1 Aufgabe 1: Mathematische Berechnungen

### 2.10.2 Aufgabe 2: Methode und if

### 2.10.3 Aufgabe 3: Selbstabholerrabatt

### 2.10.4 Aufgabe 4: Schleifen mit Berechnungen

### 2.10.5 Aufgabe 5: Schleifen und fixe Schrittweite

### 2.10.6 Aufgabe 6: Schleifen mit variabler Schrittweite

### 2.10.7 Aufgabe 7: Verschachtelte Schleifen – Variante 1

2.10.8 Aufgabe 8: Verschachtelte Schleifen – Variante 2

2.10.9 Aufgabe 9: Verschachtelte Schleifen – Variante 3

3 Strings

3.1 Schnelleinstieg

3.1.1 Gebräuchliche Stringaktionen

3.1.2 Suchen und Ersetzen

3.1.3 Informationen extrahieren und formatieren

3.2 Nächste Schritte

3.2.1 Die Klasse Scanner

3.2.2 Mehrzeilige Strings (Text Blocks)

3.2.3 Strings und char[]s

3.3 Praxisbeispiel: Text in Title Case wandeln

3.4 Aufgaben und Lösungen

3.4.1 Aufgabe 1: Länge, Zeichen und Enthaltensein

3.4.2 Aufgabe 2: Title Case mit Scanner

3.4.3 Aufgabe 3: Zeichen wiederholen

3.4.4 Aufgabe 4: Vokale raten

3.4.5 Aufgabe 5: String Merge

4 Arrays

4.1 Schnelleinstieg

4.1.1 Gebräuchliche Aktionen

4.1.2 Mehrdimensionale Arrays

4.2 Nächste Schritte

4.2.1 Eindimensionale Arrays

4.2.2 Mehrdimensionale Arrays

4.3 Praxisbeispiel: Flächen füllen

4.4 Aufgaben und Lösungen

4.4.1 Aufgabe 1: Durcheinanderwürfeln eines Arrays

4.4.2 Aufgabe 2: Arrays kombinieren

4.4.3 Aufgabe 3: Rotation um eine oder mehrere Positionen

4.4.4 Aufgabe 4: Zweidimensionales String-Array ausgeben

4.4.5 Aufgabe 5: Dreieckiges Array: Upside Down

5 Klassen und Objektorientierung

5.1 Schnelleinstieg

5.1.1 Grundlagen zu Klassen und Objekten

5.1.2 Eigenschaften (Attribute)

5.1.3 Verhalten (Methoden)

5.1.4 Objekte vergleichen – die Rolle von equals()

5.2 Nächste Schritte

5.2.1 Klassen ausführbar machen

5.2.2 Imports und Packages

5.2.3 Übergang zum Einsatz einer IDE

5.2.4 Imports und Packages: Auswirkungen auf unsere Applikation

5.2.5 Verstecken von Informationen

5.3 Vererbung

5.3.1 Basisklassen und abstrakte Basisklassen

5.3.2 Overloading und Overriding

5.4 Die Klasse Object

5.4.1 Beispielklasse Person

5.4.2 Die Methode toString()

5.4.3 Ergänzungen zur Methode equals(Object)

5.4.4 Typprüfung mit instanceof

5.4.5 Pattern Matching bei instanceof

5.5 Schnittstelle (Interface) und Implementierung

5.6 Records

5.7 Aufgaben und Lösungen

5.7.1 Aufgabe 1: Obstkorb

5.7.2 Aufgabe 2: Superheld

5.7.3 Aufgabe 3: Zähler

5.7.4 Aufgabe 4: Zähler mit Überlauf

## 6 Collections

### 6.1 Schnelleinstieg

6.1.1 Die Klasse ArrayList

6.1.2 Die Klasse HashSet

6.1.3 Iteratoren

6.1.4 Die Klasse HashMap

### 6.2 Nächste Schritte

6.2.1 Generische Typen (Generics)

6.2.2 Basisinterfaces für die Containerklassen

### 6.3 Praxisbeispiel: Einen Stack selbst realisieren

### 6.4 Aufgaben und Lösungen

6.4.1 Aufgabe 1: Tennisverein-Mitgliederliste

6.4.2 Aufgabe 2: Liste mit Farbnamen füllen und filtern

6.4.3 Aufgabe 3: Duplikate entfernen – Variante 1

6.4.4 Aufgabe 4: Duplikate entfernen – Variante 2

6.4.5 Aufgabe 5: Hauptstädte

6.4.6 Aufgabe 6: Häufigkeiten von Namen

6.4.7 Aufgabe 7: Objekte mit Maps selbst gebaut

6.4.8 Aufgabe 8: Listenreihenfolge umdrehen (mit Stack)

## 7 Ergänzendes Wissen

### 7.1 Sichtbarkeits- und Gültigkeitsbereiche

### 7.2 Primitive Typen und Wrapper-Klassen

7.2.1 Grundlagen

7.2.2 Casting: Typenerweiterungen sowie -verkleinerungen

### 7.2.3 Konvertierung von Werten

#### 7.3 Ternary-Operator (?-Operator)

#### 7.4 Aufzählungen mit enum

#### 7.5 Switch

#### 7.6 Moderne Switch Expressions

##### 7.6.1 Einführendes Beispiel

##### 7.6.2 Weitere Gründe für die Neuerung

#### 7.7 Pattern Matching bei Switch Expressions (Java 17 Preview)

##### 7.7.1 Einführendes Beispiel

##### 7.7.2 Spezialitäten

#### 7.8 Break und Continue in Schleifen

##### 7.8.1 Funktionsweise von break und continue in Schleifen

##### 7.8.2 Wie macht man es besser?

#### 7.9 Rekursion

#### 7.10 Aufgaben und Lösungen

##### 7.10.1 Aufgabe 1: Würfelspiel

##### 7.10.2 Aufgabe 2: Prüfung auf Vokale mit switch

##### 7.10.3 Aufgabe 3: Temperaturumrechnung

##### 7.10.4 Aufgabe 4: Palindrom-Prüfung mit Rekursion

## **II Aufstieg**

### 8 Mehr zu Klassen und Objektorientierung

#### 8.1 Wissenswertes zu Vererbung

##### 8.1.1 Generalisierung und Spezialisierung

##### 8.1.2 Polymorphie

##### 8.1.3 Sub-Classing und Sub-Typing

#### 8.2 Varianten innerer Klassen

##### 8.2.1 »Normale« innere Klassen

##### 8.2.2 Statische innere Klassen

- 8.2.3 Methodenlokale innere Klassen
- 8.2.4 Anonyme innere Klassen
- 9 Lambdas und Streams
  - 9.1 Einstieg in Lambdas
    - 9.1.1 Syntax von Lambdas
    - 9.1.2 Functional Interfaces und SAM-Typen
  - 9.2 Methodenreferenzen
  - 9.3 Externe vs. interne Iteration
    - 9.3.1 Externe Iteration
    - 9.3.2 Interne Iteration
    - 9.3.3 Das Interface Predicate
  - 9.4 Streams im Überblick
    - 9.4.1 Streams erzeugen – Create Operations
    - 9.4.2 Intermediate und Terminal Operations im Überblick
    - 9.4.3 Zustandslose Intermediate Operations
    - 9.4.4 Zustandsbehaftete Intermediate Operations
    - 9.4.5 Terminal Operations
  - 9.5 Aufgaben und Lösungen
    - 9.5.1 Aufgabe 1: Erwachsene aus Personenliste extrahieren
    - 9.5.2 Aufgabe 2: Stream-API
    - 9.5.3 Aufgabe 3: Informationen mit Stream-API extrahieren
    - 9.5.4 Aufgabe 4: Häufigkeiten von Namen
    - 9.5.5 Aufgabe 5: Kollektoren
- 10 Verarbeitung von Dateien
  - 10.1 Schnelleinstieg
    - 10.1.1 Das Interface Path und die Utility-Klasse Files
    - 10.1.2 Anlegen von Dateien und Verzeichnissen
    - 10.1.3 Inhalt eines Verzeichnisses auflisten

10.1.4 Pfad ist Datei oder Verzeichnis?

10.1.5 Dateiaktionen und die Utility-Klasse Files

10.1.6 Informationen zu Path-Objekten ermitteln

10.1.7 Kopieren

10.1.8 Umbenennen

10.1.9 Löschen

10.2 Dateibehandlung und die Klasse File

10.2.1 Konvertierung von Path in File und zurück

10.2.2 Die Klasse File im Kurzüberblick

10.2.3 Dateiinhalte verarbeiten und Systemressourcen

10.3 Praxisbeispiel: Directory-Baum darstellen

10.3.1 Basisvariante

10.3.2 Variante mit schönerer Darstellung

10.3.3 Finale Variante mit ausgeklügelter Darstellung

10.4 Aufgaben und Lösungen

10.4.1 Aufgabe 1: Texte in Datei schreiben und wieder lesen

10.4.2 Aufgabe 2: Dateigrößen

10.4.3 Aufgabe 3: Existenzprüfung

10.4.4 Aufgabe 4: Rechteprüfung

10.4.5 Aufgabe 5: Verzeichnisinhalt auflisten

11 Fehlerbehandlung mit Exceptions

11.1 Schnelleinstieg

11.1.1 Fehlerbehandlung

11.1.2 Exceptions selbst auslösen – throw

11.1.3 Eigene Exception-Typen definieren

11.1.4 Exceptions propagieren – throws

11.2 Fehlerbehandlung in der Praxis

11.3 Automatic Resource Management (ARM)

## 11.4 Hintergrundwissen: Checked und Unchecked Exceptions

## 12 Datumsverarbeitung

### 12.1 Schnelleinstieg

#### 12.1.1 Die Aufzählungen DayOfWeek und Month

#### 12.1.2 Die Klasse LocalDate

#### 12.1.3 Die Klassen LocalTime und LocalDateTime

### 12.2 Nächste Schritte

#### 12.2.1 Datumsarithmetik

#### 12.2.2 Formatierung und Parsing

### 12.3 Praxisbeispiel: Kalenderausgabe

### 12.4 Aufgaben und Lösungen

#### 12.4.1 Aufgabe 1: Wochentage

#### 12.4.2 Aufgabe 2: Freitag, der 13.

#### 12.4.3 Aufgabe 3: Mehrmals Freitag, der 13.

#### 12.4.4 Aufgabe 4: Schaltjahre

## **III Praxisbeispiele**

### 13 Praxisbeispiel: Tic Tac Toe

#### 13.1 Spielfeld initialisieren und darstellen

#### 13.2 Setzen der Steine

#### 13.3 Prüfen auf Sieg

#### 13.4 Bausteine im Einsatz

### 14 Praxisbeispiel: CSV-Highscore-Liste einlesen

#### 14.1 Verarbeitung von Spielständen (Highscores)

#### 14.2 Extraktion der Daten

#### 14.3 Besonderheiten der Implementierung

### 15 Praxisbeispiel: Worträtsel

#### 15.1 Applikationsdesign – Vorüberlegungen zur Strukturierung

#### 15.2 Einlesen der verfügbaren Wörter

15.3 Hilfsdatenstrukturen

15.4 Datenmodell

15.4.1 Datenspeicherung und Initialisierung

15.4.2 Zufällige Wahl von Richtung, Position, Wort und Buchstabe

15.4.3 Algorithmus zum Verstecken von Wörtern

15.4.4 Wort prüfen und platzieren

15.5 HTML-Erzeugung

15.6 Hauptapplikation

15.7 Ausgabe als HTML und Darstellung im Browser

15.8 Fazit

## **IV Schlussgedanken**

16 Gute Angewohnheiten

16.1 Grundregeln eines guten Programmierstils

16.2 Coding Conventions

16.2.1 Grundlegende Namens- und Formatierungsregeln

16.2.2 Namensgebung

16.2.3 Dokumentation

16.2.4 Programmdesign

16.2.5 Parameterlisten

16.2.6 Logik und Kontrollfluss

16.3 Sourcecode-Prüfung

16.4 JUnit 5: Auch ans Testen denken

16.4.1 Das JUnit-Framework

16.4.2 Schreiben und Ausführen von Tests

17 Schlusswort

## **V Anhang**

A Schlüsselwörter im Überblick

B Schnelleinstieg JShell

C Grundlagen zur JVM und Infos zum Java-Ökosystem

C.1 Wissenswertes zur Java Virtual Machine (JVM)

C.1.1 Einführendes Beispiel

C.1.2 Ausführung eines Java-Programms

C.2 Das Java-Ökosystem im Kurzüberblick

**Literaturverzeichnis**

**Index**

## 4 Arrays

In diesem Kapitel lernen wir Arrays im Detail kennen. Sie bilden die Grundlage für viele andere Datenstrukturen. Arrays werden dazu verwendet, mehrere Dinge gleichen Typs zu speichern, etwa eine Menge von Namen oder Personen. Etwas unpraktisch wäre es, müsste man jeweils eine eigene Variable für jedes Element definieren.

### 4.1 Schnelleinstieg

#### 4.1.1 Gebräuchliche Aktionen

##### Definition eines Arrays

Genau wie andere Variablen besitzen Arrays zunächst einmal einen Typ, allerdings gefolgt von eckigen Klammern und dem gewünschten Namen (Bezeichner). Eine Deklaration eines Arrays von Strings mit Vornamen sieht folgendermaßen aus:

```
String[] firstnames
```

Nun müssen wir das Ganze aber noch mit Leben bzw. Werten füllen. Dazu können wir eine Initialisierung nutzen. Dabei erfolgt die Angabe der Werte in geschweiften Klammern, um eine Vielzahl an Werten, hier Namen, bereitzustellen:

```
jshell> String[] firstnames = { "Tim", "Tom", "Peter", "Mike"
}
```

```
firstnames ==> String[4] { "Tim", "Tom", "Peter", "Mike" }
```

Das geht natürlich auch für Zahlen, hier für einige Primzahlen und Fibonacci-Zahlen gezeigt:

```
jshell> int[] firstPrimes = {2, 3, 5, 7, 11, 13, 17}

firstPrimes ==> int[7] { 2, 3, 5, 7, 11, 13, 17 }

jshell> int[] firstFibonacci = {1, 1, 2, 3, 5, 8, 13}

firstFibonacci ==> int[7] { 1, 1, 2, 3, 5, 8, 13 }
```

### Tipp: Abweichende Notation

Obwohl die bislang gezeigte Definition wohl am weitesten verbreitet ist, kann man die eckigen Klammern auch hinter den Namen des Arrays (Bezeichner) schreiben:

```
jshell> String firstnames[] = { "Tim", "Tom", "Peter",
"Mike" }

firstnames ==> String[4] { "Tim", "Tom", "Peter", "Mike"
}
```

Nach meinem Geschmack ist die zuvor gezeigte Variante mit den Klammern nach dem Typ intuitiver. Die Variante nach dem Variablennamen suggeriert, man würde ein Array von Vornamen und eben nicht ein `String[]` erzeugen.

### Zugriff auf Elemente

Sie greifen auf ein Array-Element zu, indem Sie sich auf die Position beziehen. Diese wird in eckigen Klammern und 0-basiert angegeben. Folgende Anweisung greift auf den Wert des ersten (Index 0) und des letzten Elements (Index 3) im Array `firstnames` zu:

```

jshell> String[] firstnames = { "Tim", "Tom", "Peter", "Mike"
}

firstnames ==> String[4] { "Tim", "Tom", "Peter", "Mike" }

jshell> firstnames[0]

$42 ==> "Tim"

jshell> firstnames[3]

$43 ==> "Mike"

```

**Was passiert, wenn ich an falscher Stelle zugreife?** Array-Zugriffe über die Position haben so ihre Tücken. Deswegen sollten wir uns fragen, was eigentlich passiert, wenn wir mit einer Position zugreifen, für die keine Daten existieren, etwa wie folgt mit negativem Index oder einem Wert jenseits der oberen Grenze, hier 7:

```

jshell> firstnames[-1]

| Exception java.lang.ArrayIndexOutOfBoundsException: Index
-1 out of bounds

    for length 4

jshell> firstnames[7]

| Exception java.lang.ArrayIndexOutOfBoundsException: Index
7 out of bounds for

    length 4

```

Auf Fehlerbehandlungen und Exceptions im Speziellen gehe ich später in Kapitel 11 ein. Hier reicht erst einmal das Wissen, dass nach einer solchen Fehlersituation die Ausführung unseres Java-Programms gestoppt wird.

## Array-Länge: Anzahl der Elemente

Um Zugriffe auf ungültige Positionen am Ende möglichst zu vermeiden oder um sich über die Anzahl der enthaltenen Elemente zu informieren, kann man diese Informationen mit `length` wie folgt abfragen:

```
jshell> firstnames.length
```

```
$48 ==> 4
```

```
jshell> firstPrimes.length
```

```
$49 ==> 7
```

Nochmals zur Erinnerung: Im Beispiel sehen wir die sogenannte Punktnotation, die Zugriff auf Eigenschaften und Methoden ermöglicht. In Kapitel 5 gehe ich auf die objektorientierte Programmierung genauer ein.

## Verändern eines Elements

Wir wollen nun den Wert `Mike` mit dem Wert `Michael` ersetzen und schreiben dazu:

```
jshell> firstnames[3] = "Michael"
```

```
$44 ==> "Michael"
```

```
jshell> firstnames
```

```
firstnames ==> String[4] { "Tim", "Tom", "Peter", "Michael" }
```

Wir sehen, dass die Zuweisung im Prinzip genauso funktioniert, wie bei normalen Variablen, hier halt positionsbasiert.

## Array-Elemente durchlaufen – Variante 1

Sie können die Array-Elemente mit der `for`-Schleife durchlaufen und mit der zuvor kennengelernten Eigenschaft `length` bestimmen, wie oft die

Anweisungen im Schleifenrumpf ausgeführt werden sollen:

```
jshell> for (int i = 0; i < firstnames.length; i++) {  
    ...>     System.out.println(firstnames[i]);  
    ...> }
```

Tim

Tom

Peter

Michael

Hier werden indizierte Zugriffe mit einer for-Schleife kombiniert: Auf diese Weise wird für jede gültige Position der korrespondierende Wert aus der String-Array ermittelt und mit `System.out.println()` auf der Konsole ausgegeben.

### **Array-Elemente durchlaufen – Variante 2**

Weil man gerade beim indizierten Zugriff leicht mal einen Flüchtigkeitsfehler macht, bietet sich die for-each-Variante der for-Schleife an. Übertragen auf unser Beispiel ergibt sich dadurch Folgendes:

```
jshell> for (String currentName : firstnames) {  
    ...>     System.out.println(currentName);  
    ...> }
```

Tim

Tom

```
Peter
```

```
Michael
```

Das obige Beispiel kann wie folgt interpretiert werden: Gib für jedes String-Element (genannt `currentName`) aus den Werten von `firstnames` dessen Wert aus.

## Textuelle Ausgaben

Bislang wurden unsere Arrays immer recht lesbar in der JShell ausgegeben, wenn wir nur deren Namen eingegeben haben. Das ist eine besondere und vor allem nützliche Eigenschaft der JShell. In reinen Java-Programmen können wir lediglich eine Ausgabe mit `System.out.println()` vornehmen – und diese ist zudem kryptisch. Schauen wir uns das Ganze etwas detaillierter an:

```
jshell> String[] strangeOutput = { "Tim", "Tom", "Mike" }

strangeOutput ==> String[3] { "Tim", "Tom", "Mike" }

jshell> strangeOutput

strangeOutput ==> String[3] { "Tim", "Tom", "Mike" }

jshell> System.out.println(strangeOutput)

[Ljava.lang.String;@312b1dae
```

Die ersten beiden Ausgaben sind erwartungskonform und wir kennen diese bereits. Wieso ist die letzte so kryptisch?

Das liegt daran, dass bei Ausgaben mit `System.out.println()` für die übergebenen Werte von Java automatisch deren Methode `toString()` zur Aufbereitung einer schönen textuellen Darstellung aufgerufen wird.

Scheint so, als ob da gerade gründlich etwas schiefgelaufen ist. Das, was wir erhalten, ist ja alles andere als lesbar und verständlich. Das liegt daran, dass Arrays in Java leider keine eigene Implementierung der Methode `toString()`

besitzen. Standardmäßig wird nur der Typ gefolgt von einem @ sowie einer hexadezimalen Zahl ausgegeben. Deswegen sieht man mitunter merkwürdige Ausgaben wie die gezeigte.

Als Abhilfe bietet sich die Methode `Arrays.toString(values)` aus dem JDK an:

```
jshell> Arrays.toString(strangeOutput)

$130 ==> "[Tim, Tom, Mike]"

jshell> System.out.println(Arrays.toString(strangeOutput))

[Tim, Tom, Mike]
```

Um ein wenig Übung zu bekommen, implementieren wir etwas Ähnliches selbst. Hierbei kommt eine `for`-Schleife und ein indizierter Zugriff zum Einsatz:

```
void printArray(String[] values)

{

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

    {

        final String str = values[i];

        System.out.println(str);

    }

}
```

Auch diese Variante liefert eine verständliche Darstellung:

```
jshell> printArray(strangeOutput)
```

```
Tim
```

```
Tom
```

```
Mike
```

### Dynamische Definition eines Arrays

Bislang haben wir bei der Definition einige Werte übergeben und damit die Größe von Arrays festgelegt. Weitaus häufiger hat man es in der Praxis aber mit möglicherweise vorab nicht bekannten Daten oder einer variierenden Menge an Daten zu tun.

**Erzeugung fixer Länge** Beginnen wir mit dem erstem Fall, nämlich einer dynamischen Erzeugung und späteren Befüllung des Arrays. Vorab wissen wir zwar, wie viele Daten gespeichert und verarbeitet werden sollen, die konkreten Werte kennen wir jedoch noch nicht.

Nachfolgend nehmen wir an, dass 20 Werte im Array verwaltet werden sollen. Mit `new` und einer entsprechenden Größenangabe erzeugen wir das passende Array:

```
jshell> int[] values = new int[20]
```

```
values ==> int[20] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }  
  
}
```

Nun könnten Berechnungen oder andere Aktionen erfolgen, um die Werte zu ermitteln. Wir schreiben stellvertretend eine Methode `populate()`, die das Array mit dem Quadrat des jeweiligen Index befüllt:

```
jshell> void populate(int[] inputValues)
```

```

...> {

...>     for (int i = 0; i < inputValues.length; i++)

...>     {

...>         inputValues[i] = i * i;

...>     }

...> }

| created method populate(int[])

```

Rufen wir diese Methode einmal auf und schauen dann stellvertretend an zwei Positionen, ob die Berechnung erfolgreich war:

```

jshell> populate(values)

jshell> values[2]

$198 ==> 4

jshell> values[7]

$199 ==> 49

```

**Dynamische Erzeugung verschiedener Längen** Manchmal ergibt sich die zur Verwaltung und Speicherung der Daten benötigte Länge eines Arrays erst während der Programmausführung. Nachfolgend ist derartiges simuliert und ein zufällig bestimmter Wert im Bereich bis 1000 wird als Größe verwendet:

```

jshell> int computeNeededSize()

```

```

...> {

...>     return (int)(Math.random() * 1000);

...> }

| created method computeNeededSize()

```

Wie kann man eine variierende Größe denn zur Erzeugung des Arrays nutzen? Tatsächlich ist das ganz naheliegend und einfach durch die Angabe einer Variablen oder eines direkten Aufrufs möglich:

```

jshell> int size = computeNeededSize()

size ==> 425

jshell> int[] values1 = new int[size]

values1 ==> int[425] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
... , 0, 0, 0, 0,

    0, 0, 0, 0 }

jshell> int[] values2 = new int[computeNeededSize()]

values2 ==> int[785] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
... , 0, 0, 0, 0,

    0, 0, 0, 0 }

```

Danach kann man dann mit dem Array genauso weiterarbeiten, wie wir es bereits zuvor gesehen haben.

## 4.1.2 Mehrdimensionale Arrays

Ein mehrdimensionales Array ist ein Array, das ein oder mehrere Arrays enthält. Die Anzahl an Dimensionen wird durch die Anzahl an eckigen Klammerpaaren bestimmt, etwa `int[][]` für zweidimensional oder `int[][][]` für dreidimensional. Nachfolgend beschränken wir uns vereinfachend und wegen der besseren Veranschaulichung auf die Betrachtung zweidimensionaler Arrays.

Die einzelnen Werte kann man dann wie gewohnt in geschweiften Klammern notieren. Ein zweidimensionales Array erstellen wir demnach wie folgt:

```
jshell> int[][] twodim = {  
  
    ...>     { 1, 1, 1, 1 },  
  
    ...>     { 2, 2, 2, 2 },  
  
    ...>     { 3, 3, 3, 3 },  
  
    ...>     { 4, 4, 4, 4 }  
  
    ...> }  
  
twodim ==> int[4][] { int[4] { 1, 1, 1, 1 }, int[4] { 2, 2, ...  
}, int[4] { 4,  
  
    4, 4, 4 } }
```

Wir sehen zunächst, dass das Array wiederum Arrays (hier gleicher Länge) enthält. Oftmals hat man es in der Praxis mit solchen rechteckigen Ausrichtungen zu tun. Dann kann man sich ein zweidimensionales Array ein wenig wie eine Schrankwand mit nummerierten Schubladen für Elemente vorstellen. Nachfolgend ist das für ein  $5 \times 3$ -Array (5 Positionen in x-Richtung und 3 Reihen in y-Richtung) schematisch dargestellt.

	0	1	2	3	4
0	0,0	0,1	0,2	0,3	0,4
1	1,0	1,1	1,2	1,3	1,4
2	2,0	2,1	2,2	2,3	2,4

**Abbildung 4-1** Schubladendenkweise für zweidimensionales Array

Schauen wir uns an, wie man eine Ausgabe unabhängig von den konkreten Größen, also flexibel, mit einem zweidimensionalen indizierten Zugriff gestalten kann:

```
jshell> void print2dArray(int[][] values)

...> {

...>     for (int y = 0; y < values.length; y++)

...>     {

...>         for (int x = 0; x < values[y].length; x++)

...>         {

...>             System.out.print(values[y][x]);

...>         }

...>         System.out.println();

...>     }

...> }
```

```
| created method print2dArray(int[][])
```

Rufen wir die Methode einmal auf:

```
jshell> print2dArray(twodim)
```

```
1111
```

```
2222
```

```
3333
```

```
4444
```

Aber Moment, hatten wir nicht schon die Methode `Arrays.toString()` kennengelernt? Das müsste sich doch gewinnbringend für die einzelnen Zeilen nutzen lassen:

```
jshell> for (int y = 0; y < twodim.length; y++)
```

```
...> {
```

```
...>     System.out.println(Arrays.toString(twodim[y]));
```

```
...> }
```

```
[1, 1, 1, 1]
```

```
[2, 2, 2, 2]
```

```
[3, 3, 3, 3]
```

```
[4, 4, 4, 4]
```

## Spezialfall: Nicht rechteckige Arrays

Weil mehrdimensionale Arrays in Java als Arrays von Arrays realisiert sind, lassen sich dort unterschiedliche Längen modellieren, wie hier eine dreieckige Ausrichtung:

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

Rufen wir die Methode zur Ausgabe einmal auf, um die Dreieckseigenschaft noch klarer zu sehen:

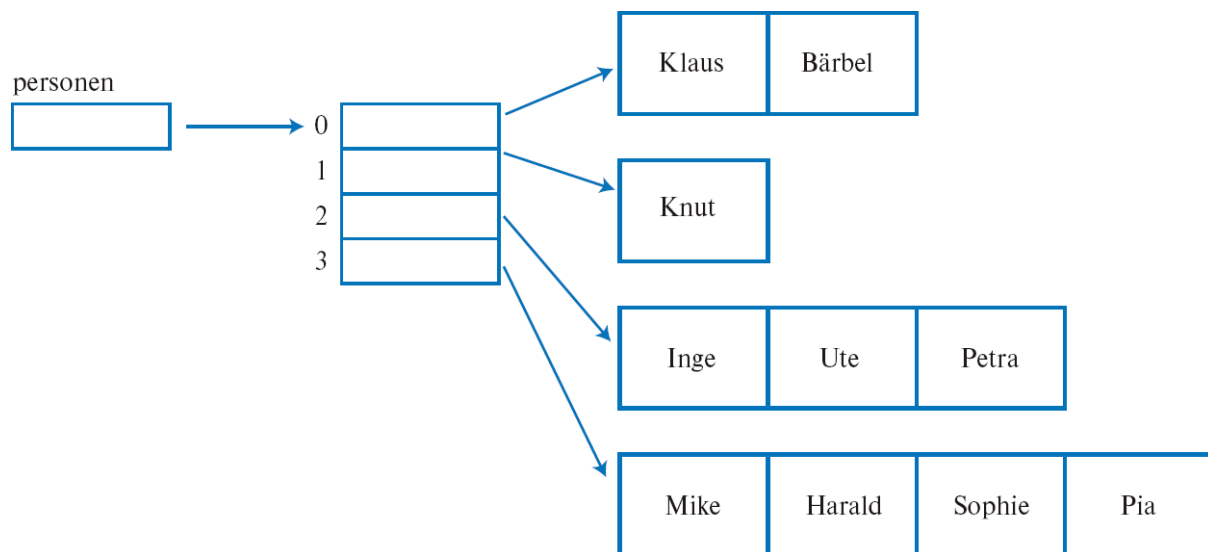
```
jshell> print2dArray(twodimTriangle)  
  
1  
  
12  
  
123
```

Machen wir es noch etwas konkreter für ein String-Array namens `personen`, das dann in beispielsweise so wie in Abbildung 4-2 aussieht.

## 4.2 Nächste Schritte

Nachfolgend lernen wir sowohl eindimensionale als auch mehrdimensionale Arrays noch genauer kennen.

Wir wissen bereits, dass ein Array einen einfachen Datenbehälter bereitstellt, dessen Größe durch die Initialisierung vorgegeben ist und sich über das Attribut `length` ermitteln lässt. Arrays bieten jedoch keinerlei Funktionalitäten in Form von Methoden, etwa zur automatischen Vergrößerung, falls der Platz zur Speicherung von Daten nicht mehr ausreicht. Derartiges muss bei Bedarf in einer nutzenden Applikation selbst programmiert werden. Gerade beim indizierten Zugriff lauern Fehler. Das haben wir in der Einführung schon für Indexwerte außerhalb des Bereichs 0 bis `length - 1` gesehen.



**Abbildung 4-2** Personen-2D-Array

### 4.2.1 Eindimensionale Arrays

Zur Einführung in die Verarbeitung von Daten mit eindimensionalen Arrays und zum Aufbau von Wissen betrachten wir einige Beispiele.

#### Beispiel 1: Tauschen von Elementen

Eine gebräuchliche Funktionalität ist das Vertauschen von Elementen an zwei Positionen. Das ist nachfolgend visualisiert.



```
{  
  
    final int tmp = values[first];  
  
    values[first] = values[second];  
  
    values[second] = tmp;  
  
}
```

Experimentieren wir ein wenig mit der gerade erstellten Methode:

```
jshell> int[] numbers = { 1,2,3,4,5,6,7,8,9 }  
  
numbers ==> int[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }  
  
jshell> swap(numbers, 3, 5)  
  
jshell> swap(numbers, 2, 6)  
  
jshell> numbers  
  
numbers ==> int[9] { 1, 2, 7, 6, 5, 4, 3, 8, 9 }  
  
jshell> swap(numbers, 1, 7)  
  
jshell> swap(numbers, 0, 8)  
  
jshell> numbers  
  
numbers ==> int[9] { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Was haben wir gerade gelernt? Wir können mithilfe von `swap()` eine Funktionalität bauen, die den Inhalt eines Arrays in umgekehrte Reihenfolge

bringt.

**Array Reverse mit swap ()** Weil wir gerade so schön in Fahrt sind, wollen wir eine entsprechende Methode selbst realisieren, die die Werte in einem Array in deren Reihenfolge tauscht. Als Schmankerl wollen wir in der Lage sein, das Umkehren der Werte nicht nur für das gesamte Array, sondern auch Teilbereiche, ausführen zu können. Was benötigen wir dazu? Zunächst einmal je einen Positionszeiger für vorne und hinten. Beide müssen wir zusätzlich an `swap ()` übergeben. Nach jedem Tauschvorgang wandern diese Positionszeiger aufeinander zu. Was uns noch fehlt, ist eine Abbruchbedingung. Wir stoppen, wenn sich die Positionen überlappen. Damit bietet sich eine Realisierung mit einer `while`-Schleife an, die nach dem Tauschen der Elemente die Positionszeiger aufeinander zu bewegt:

```
public static void reverse(final int[] values, int start, int
end)
{
    while (start < end)
    {
        swap(values, start, end);
        start++;
        end--;
    }
}
```

Alternativ können wir auch einfach bis zur Mitte laufen und jeweils die korrespondierenden Elemente tauschen. Damit ergibt sich eine Variante mit der `for`-Schleife:

```

public static void reverse2(final int[] values, int start,
int end)

{

    for (int i = 0; i < ((end-start) / 2); i++)

    {

        swap(values, i + start, end - i);

    }

}

```

Probieren wir das Ganze einmal aus:

```

jshell> int[] numbers = { 1,2,3,4,5,6,7,8,9 }

numbers ==> int[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }

jshell> reverse(numbers, 0, numbers.length - 1)

jshell> numbers

numbers ==> int[9] { 9, 8, 7, 6, 5, 4, 3, 2, 1 }

```

Das Allerbeste kommt zum Schluss: Dadurch, dass wir mit zwei Positionszeigern als Parametern arbeiten, können wir sogar nur Teilbereiche eines Arrays in seiner Reihenfolge umkehren:

```

jshell> int[] numbers = { 1,2,3,4,5,6,7,8,9 }

numbers ==> int[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }

```

```
jshell> reverse(numbers, 2, 6)

jshell> numbers

numbers ==> int[9] { 1, 2, 7, 6, 5, 4, 3, 8, 9 }

jshell> reverse2(numbers, 2, 6)

jshell> numbers

numbers ==> int[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

### **Tipp: Rekursive Variante**

Bei Rekursion geht es darum, die Lösungsfindung durch Selbstaufrufe zu realisieren. In diesem Fall tauschen wir jeweils ein Element und rufen dann die Methode erneut mit veränderten Grenzen auf. Dabei ist wichtig, dass man hier nicht Postinkrement und -dekrement einsetzt, weil dann die Werte nicht verändert werden und die Rekursion nie endet. Details finden Sie später in Abschnitt 7.9.

```
public static void reverseRec(final int[] values, int
start, int end)

{

    if (start < end)

    {

        swap(values, start, end);

        reverseRec(values, start + 1, end - 1);
    }
}
```

```
}  
  
}
```

## Beispiel 2: Suchen in Arrays

Nun wollen wir die Suche nach einem Wert in einem Array einmal selbst implementieren und dazu die Methode `find(int[], int)` schreiben, die in einem `int`-Array nach einem Wert sucht und dessen Position oder `-1` für »nicht gefunden« liefern soll:

```
static int find(final int[] values, final int searchFor)  
{  
  
    for (int i = 0; i < values.length; i++)  
  
        {  
  
            if (values[i] == searchFor)  
  
                return i;  
  
        }  
  
    return -1;  
  
}
```

Das Ganze kann man als typisches Suchproblem mit einer `while`-Schleife lösen – dabei ist die Bedingung am Ende der Schleife als Kommentar angegeben:

```
static int find2(final int[] values, final int searchFor)  
{
```

```

    int pos = 0;

    while (pos < values.length && values[pos] != searchFor)
    {
        pos++;
    }

    // pos >= values.length || values[pos] == searchFor

    return pos >= values.length ? -1 : pos;
}

```

Probieren wir die beiden Varianten einmal aus und suchen in den Werten 1,2,3,4,5,6,7 zunächst nach dem Wert 2 und danach nach dem Wert 42. Ersteres sollte die Position 1 und Letzteres sollte dann -1 für »nicht gefunden« liefern.

```

jshell> int[] numbers = { 1,2,3,4,5,6,7}

numbers ==> int[7] { 1, 2, 3, 4, 5, 6, 7 }

jshell> find(numbers, 2)

$66 ==> 1

jshell> find(numbers, 42)

$67 ==> -1

jshell> find2(numbers, 2)

```

```
$69 ==> 1
```

```
jshell> find2(numbers, 42)
```

```
$70 ==> -1
```

## 4.2.2 Mehrdimensionale Arrays

In diesem Abschnitt wollen wir uns kurz mit mehrdimensionalen Arrays etwas genauer beschäftigen. Wie schon in der Einführung motiviert und weil es in der Praxis häufiger vorkommt und man es sich auch visuell gut vorstellen kann, beschränken wir uns dabei auf zweidimensionale Arrays. Zuvor hatte ich auch darauf hingewiesen, dass in Java mehrdimensionale Arrays als Arrays von Arrays realisiert und somit nicht zwingend rechteckig sein müssen, obwohl dies vermutlich der Normalfall ist.

### Einführendes Beispiel

Ein zweidimensionales rechteckiges Array kann man etwa zur Modellierung eines Spielfelds, eines Sudoku-Rätsels oder einer Landschaft, repräsentiert durch Zeichen, nutzen. Zum besseren Verständnis und Einstieg betrachten wir ein Beispiel: Nehmen wir an, '#' stünde für eine Begrenzungsmauer, '\$' für einen einzusammelnden Gegenstand und 'P' für den Spieler sowie 'X' für den Ausgang aus einem Level. Mit diesen Zeichen kann man ein Spielfeld wie folgt beschreiben:

```
#####  
  
## P           ##  
  
#### $ X ####  
  
##### $ #####  
  
#####
```

In Java lässt sich zur Verarbeitung ein `char [][]` nutzen:

```
public static void main(final String[] args)
{

    final char[][] world = {
        "#####".toCharArray(),

        "## P      ##".toCharArray(),

        "#### $ X ####".toCharArray(),

        "##### $ #####".toCharArray(),

        "#####".toCharArray() };

    printArray(world);
}

public static void printArray(final char[][] values)
{

    for (int y = 0; y < values.length; y++)
    {

        for (int x = 0; x < values[y].length; x++)

            System.out.print(getAt(values, x, y) + " ");
```

```

        System.out.println();

    }

}

```

Es gibt zwei Varianten, wie man beim Speichern die Koordinaten angeben kann: Zum einen mit `[x][y]` und zum anderen, wenn man eher zeilenorientiert denkt, mit `[y][x]`. Das kann zwischen unterschiedlichen Entwicklern zu Missverständnissen und Diskussionen führen. Eine kleine Abhilfe erreicht man, wenn man sich Zugriffsmethoden, etwa `getAt(char [][], int, int)`, schreibt und die jeweilige Präferenz dort berücksichtigt:

```

static char getAt(final char [][] values, final int x, final int y)

{

    return values[y][x];

}

```

Diese Zugriffsmethode werde ich bevorzugt in der Einführung nutzen und später auch immer öfter mal direkte Array-Zugriffe.

Führen wir das obige Programm aus, um die Ausgabefunktionalität in Aktion zu erleben. Im Folgenden werden wir immer mal wieder auf Ähnliches zurückgreifen – neben dem Debugging ist eine Konsolenausgabe gerade bei mehrdimensionalen Arrays ziemlich hilfreich.

```

# # # # # # # # # # # # # # # # # #
# #      P                      # #
# # # #      $ X      # # # #

```

```
# # # # # # $ # # # # # #
```

```
# # # # # # # # # # # # # # # #
```

### Beispiel: Array-Inhalt rotieren

Unsere Aufgabe besteht nun darin, ein Array um 90 Grad nach links bzw. rechts zu drehen. Schauen wir uns das Ganze beispielhaft für zwei Rechtsdrehungen an:

```
1111      4321      4444
2222 => 4321 => 3333
3333      4321      2222
4444      4321      1111
```

Versuchen wir, das Vorgehen ein wenig zu formalisieren. Am einfachsten lässt sich die Rotation realisieren, wenn man ein neues Array erzeugt und dann geeignet befüllt. Zur Ermittlung der Formeln nutzen wir konkrete Beispieldaten, die das Verständnis erleichtern ( $x_n$  und  $y_n$  stehen für die neuen Koordinaten – nachfolgend ist links die Drehung nach links und rechts die Drehung nach rechts zu sehen):

```
      x  0123
      y  --
0     ABCD
1     EFGH

xn  01          xn  01
```

yn	–	yn	–
0	DH	0	EA
1	CG	1	FB
2	BF	2	GC
3	AE	3	HD

Wir erkennen, dass aus einem  $4 \times 2$ -Array ein  $2 \times 4$ -Array wird.

Die Rotation beruht auf folgenden Berechnungsvorschriften, wobei `maxX` und `maxY` die jeweiligen maximalen Koordinaten sind:

	Orig	->	NewX	NewY
-----				
<code>rotateLeft:</code>	<code>(x,y)</code>	->	<code>y</code>	<code>maxX-x</code>
<code>rotateRight:</code>	<code>(x,y)</code>	->	<code>maxY-y</code>	<code>x</code>

Mit diesem Wissen machen wir uns an die Implementierung. Die Rotationsrichtung modellieren wir in Form einer `boolean`-Variablen `rotateLeft` (etwas prägnanter und damit besser geeignet wäre hierfür eine einfache `enum`-Aufzählung. Diese lernen wir jedoch erst in Abschnitt 7.4 kennen).

Kommen wir zur Rotation. Basierend auf den obigen Formeln lässt sich die Rotation wie folgt implementieren, wobei `rotatedArray` das neu erzeugte Array ist und `origValue` der ursprüngliche Wert:

```
if (rotateLeft)
{
```

```

        rotatedArray[maxX - x][y] = origValue;
    }

    else
    {

        rotatedArray[x][maxY - y] = origValue;
    }
}

```

Nun müssen wir diese Logik nur noch geeignet in zwei Schleifen einbetten. Vorher erzeugen wir ein passend großes Array. Dann durchlaufen wir das Originalarray zeilenweise und innerhalb der Zeile positionsweise:

```

public static int[][] rotate(int[][] values, boolean
rotateLeft)
{

    int maxX = values[0].length - 1;

    int maxY = values.length - 1;

    final int[][] rotatedArray = new int[maxX + 1][maxY + 1];

    for (int y = 0; y < maxY + 1; y++)
    {

        for (int x = 0; x < maxX + 1; x++)

```

```

    {

        int origValue = values[y][x];

        if (rotateLeft)

        {

            rotatedArray[maxX - x][y] = origValue;

        }

        else

        {

            rotatedArray[x][maxY - y] = origValue;

        }

    }

}

return rotatedArray;

}

```

Schauen wir uns die Abläufe mal in der JShell an:

```

jshell> int[][] values = {

    ...>     {1,1,1,1},

```

```

...> {2,2,2,2},

...> {3,3,3,3},

...> {4,4,4,4}

...> }

values ==> int[4][] { int[4] { 1, 1, 1, 1 }, int[4] { 2, 2, ...
}, int[4] { 4,

    4, 4, 4 } }

jshell> printArrayJdk(rotate(values, true)

[1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3, 4]

```

Nachfolgend erzeugen wir ein um 90 Grad nach rechts gedrehtes Array, indem wir den booleschen Parameter mit `false` belegen:

```

jshell> printArrayJdk(rotate(values, false)

[4, 3, 2, 1]

[4, 3, 2, 1]

[4, 3, 2, 1]

```

```
[4, 3, 2, 1]
```

Die Hilfsmethode `printArray()` zur Ausgabe wurde wie folgt implementiert:

```
private static void printArray(final int[][] values)
{
    for (int i = 0; i < values.length; i++)
        System.out.println(Arrays.toString(values[i]));
}
```

### Besonderheit bei der Konstruktion

Beim Erzeugen mehrdimensionaler Arrays gibt es noch eine Besonderheit. Nehmen wir an, wir benötigen ein String-Array der Größe  $5 \times 7$ , so können wir das wie folgt erzeugen:

```
jshell> var twodim = new String[5][7]

twodim ==> String[5][] { String[7] { null, null, null, null,
... null, null,

    null, null } }
```

Dabei wird ein mit `null`-Werten initialisiertes Array erzeugt. Zu dessen Ausgabe definieren wir schnell noch (analog zu gerade eben) folgende Methode und rufen diese direkt auf:

```
jshell> private static void printArray(final String[][]
values)

...> {
```

```

...>     for (int i = 0; i < values.length; i++)

...>         System.out.println(Arrays.toString(values[i]));

...> }

| created method printArray(String[][])

jshell> printArray(twodim)

[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, null, null, null, null, null]

```

Weil mehrdimensionale Arrays aber als Arrays von Arrays realisiert sind, kann man auf die Angabe der zweiten Dimension bei der Konstruktion auch verzichten und die einzelnen Zeilen im späteren Verlauf erzeugen oder zuweisen.

Dabei ist es durchaus auch möglich, dass in einigen Positionen noch keine weiteren Daten hinterlegt werden – nachfolgend ist immer nur die jeweils 2. Reihe belegt:

```

jshell> var twodim_special = new String[5][]

twodim_special ==> String[5][] { null, null, null, null, null
}

jshell> twodim_special[0] = new String[] { "A"}

```

```
$34 ==> String[1] { "A" }

jshell> twodim_special[2] = new String[] { "B", "B" }

$35 ==> String[2] { "B", "B" }

jshell> twodim_special[4] = new String[] { "C", "C", "C" }

$36 ==> String[3] { "C", "C", "C" }
```

Schauen wir doch einmal, worin das resultiert:

```
jshell> printArray(twodim_special)

[A]

null

[B, B]

null

[C, C, C]
```

### 4.3 Praxisbeispiel: Flächen füllen

Wir wollen das bislang gesammelte Wissen nun einmal praktisch einsetzen, um eine Methode `void floodFill(char[], int, int)` zu implementieren, die in einem Array alle freien Felder mit einem bestimmten Wert befüllt.

#### Beispiel

Nachfolgend ist der Füllvorgang für das Zeichen '\*' gezeigt. Das Füllen beginnt an einer vorgegebenen Position, etwa in der linken oberen Ecke, und wird dann

so lange in die vier Himmelsrichtungen fortgesetzt, bis die Grenzen des Arrays oder eine Begrenzung in Form eines anderen Zeichens gefunden wird:

```

"  # "      "***# "      "  #  #"      "
#*****#

"  # "      "***# "      "  #  #"      "
#*****#

"#  #" =>  "#***#"      "#  #" =>  "#
#*****#

" # # "      "#*# "      " # #  #"      " # #
*****#

" # "      " # "      " #  #"      " #
*****#

```

## Algorithmus

Wie gehen wir dabei vor? Zunächst prüfen wir das Startfeld. Ist dieses Feld leer, so wird es gefüllt und es werden wiederum dessen vier Nachbarn geprüft. Erreichen wir die Array-Grenzen oder ein gefülltes Feld, so stoppen wir. Das lässt sich wunderbar rekursiv formulieren:

```

static void floodFill(final char[][] values, final int x,
final int y)

{

    // rekursiver Abbruch

    if (x < 0 || y < 0 || y >= values.length || x >=
values[y].length)

        return;

```

```

if (values[y][x] == ' ')
{

    values[y][x] = '*';

    // rekursiver Abstieg: fülle in alle 4 Richtungen

    floodFill(values, x, y-1);

    floodFill(values, x+1, y);

    floodFill(values, x, y+1);

    floodFill(values, x-1, y);

}

}

```

## Bereichsprüfung

Bei dieser Aufgabenstellung lernen wir gleich noch eine nützliche Methode (auch für andere Anwendungsfälle) kennen, nämlich die Prüfung, ob ein übergebener (x,y)-Wert für das Array gültig ist – hier mit der Methode `isOnBoard()` und unter der Annahme, dass das Array rechteckig ist:

```

static boolean isOnBoard(final char[][] values,

                        final int posX, final int posY)

{

    return posX >= 0 && posY >= 0 &&

```

```
        posX < values[0].length && posY < values.length;
    }
}
```

## Prüfung

Um eine Ausgangsbasis zu haben, schreiben wir folgende Methode:

```
private static char[][] firstWorld()
{
    return new char[][] { " # ".toCharArray(),
                          "  # ".toCharArray(),
                          "#  # ".toCharArray(),
                          " # # ".toCharArray(),
                          " #  ".toCharArray()};
}
```

Nun ist es Zeit, die erstellten Bausteine in Aktion zu erleben. Zunächst erzeugen wir durch Aufruf von `firstWorld()` ein `char[][]`, das unsere Spielwelt definiert. Dann rufen wir die Methode `floodFill()` auf. Dadurch wird der innere Bereich mit dem Zeichen '\*' gefüllt:

```
jshell> char[][] world = firstWorld()

world ==> char[5][] { char[6] { ' ', ' ', ' ', '#', ' ', ' ...
', '#', ' ', '
    ', ' ' } } }
```

```
jshell> floodFill(world, 1, 1)
```

Um das zu prüfen, schauen wir noch mal auf die Hilfsmethode zur Ausgabe, die eingangs dieses Abschnitts wie folgt implementiert wurde – hier leicht optimiert mit direktem, indiziertem Zugriff auf das Array:

```
public static void printArray(final char[][] values)
{
    for (int y = 0; y < values.length; y++)
    {
        for (int x = 0; x < values[y].length; x++)
        {
            System.out.print(values[y][x] + " ");
        }
        System.out.println();
    }
}
```

Als Ergebnis erhalten wir die gefüllte Fläche:

```
jshell> printArray(world)
```

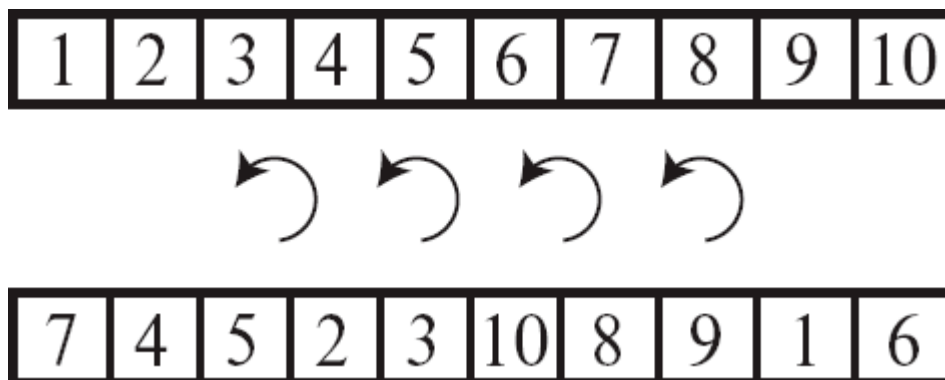
```
* * * #
```

```
* * * * #  
  
# * * * #  
  
# * #  
  
#
```

## 4.4 Aufgaben und Lösungen

### 4.4.1 Aufgabe 1: Durcheinanderwürfeln eines Arrays

Bei dieser Aufgabe geht es darum, die Elemente eines bestehenden Arrays durcheinanderzuwürfeln. Das ist nützlich, wenn man eine zufällige Verteilung benötigt.



**Abbildung 4-4** Durcheinanderwürfeln von Elementen

### Lösung

Überlegen wir kurz, wie wir vorgehen können. Eine Idee besteht darin, zwei zufällige Positionen zu wählen und die Werte zu vertauschen. Wenn man dieses Vorgehen einige Male wiederholt, sollte das Array recht gut durchmixt sein. Sowohl die Berechnung von Zufallszahlen als auch das Tauschen von Elementen sind uns bereits geläufig. Gleiches gilt für die `for`-Schleife. Verbleibt als Letztes die Frage: Wie oft sollen wir tauschen? Als Idee kann man mindestens 10 Mal tauschen. Die obere Grenze, also die maximale Anzahl an Tauschvorgängen, kann man auf die Array-Länge geteilt durch 10 festlegen.

Machen wir uns mit diesen Vorüberlegungen an die folgende Implementierung:

```

jshell> void shuffle(int[] values)

...> {

...>     int maxShuffle = Math.max(10, values.length / 10);

...>

...>     for (int run = 0; run < maxShuffle; run++)

...>     {

...>         int pos1 = (int)(Math.random() *
values.length);

...>         int pos2 = (int)(Math.random() *
values.length);

...>

...>         swap(values, pos1, pos2);

...>     }

...> }

| created method shuffle(int[])

```

Natürlich wollen wir das Ganze in Aktion erleben. Dazu definieren wir ein Array mit Zahlen von 1 bis 15 und lassen unseren Mixer zweimal darauf los:

```

jshell> int[] numbers = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
}

```

```

numbers ==> int[15] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15 }

jshell> shuffle(numbers)

jshell> numbers

numbers ==> int[15] { 11, 2, 3, 6, 1, 4, 7, 5, 9, 15, 12, 10,
13, 8, 14 }

jshell> shuffle(numbers)

jshell> numbers

numbers ==> int[15] { 15, 3, 13, 7, 1, 5, 10, 4, 9, 14, 12,
11, 2, 8, 6 }

```

#### 4.4.2 Aufgabe 2: Arrays kombinieren

Es seien zwei Arrays mit `int`-Werten gegeben. Die Aufgabe besteht nun darin, ein neues Array mit allen Werten zu erstellen. Dabei sollen zunächst alle Elemente aus dem ersten Array und dann alle aus dem zweiten in das Ergebnis übernommen werden. Nachfolgend ist dies visualisiert:



#### Lösung

Um diese Aufgabestellung umzusetzen, schreiben wir eine Methode `int[] arrayConcat(int[] values1, int[] values2)`. Dort müssen wir zunächst die beiden Längen bestimmen und damit ein neues Array passender Größe erstellen. Mithilfe einer `for`-Schleife übertragen wir dann die Werte aus dem ersten Array in das Ergebnis. Das machen wir analog für die Werte aus dem zweiten Array. Damit wir nicht mit dem Versatz der Länge des ersten Arrays im Ergebnis arbeiten müssen, führen wir eine Hilfsvariable `pos` ein. Das Ganze implementieren wir wie folgt:

```

jshell> int[] arrayConcat(int[] values1, int[] values2)

```

```
...> {  
  
...>     int length1 = values1.length;  
  
...>     int length2 = values2.length;  
  
...>  
  
...>     int pos = 0;  
  
...>     int[] result = new int[length1 + length2];  
  
...>     for (int i = 0; i < length1; i++)  
  
...>     {  
  
...>         result[pos] = values1[i];  
  
...>         pos++;  
  
...>     }  
  
...>  
  
...>     for (int i = 0; i < length2; i++)  
  
...>     {  
  
...>         result[pos] = values2[i];  
  
...>         pos++;  
  
...>     }
```

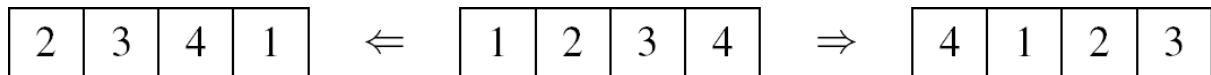
```
...>  
  
...>     return result;  
  
...> }  
  
| created method arrayConcat(int[],int[])
```

Probieren wir das Ganze einmal in der JShell aus:

```
jshell> int[] values1 = { 1, 2, 3 }  
  
values1 ==> int[3] { 1, 2, 3 }  
  
jshell> int[] values2 = { 11, 22, 33, 44 }  
  
values2 ==> int[4] { 11, 22, 33, 44 }  
  
jshell> int[] result = arrayConcat(values1, values2)  
  
result ==> int[7] { 1, 2, 3, 11, 22, 33, 44 }
```

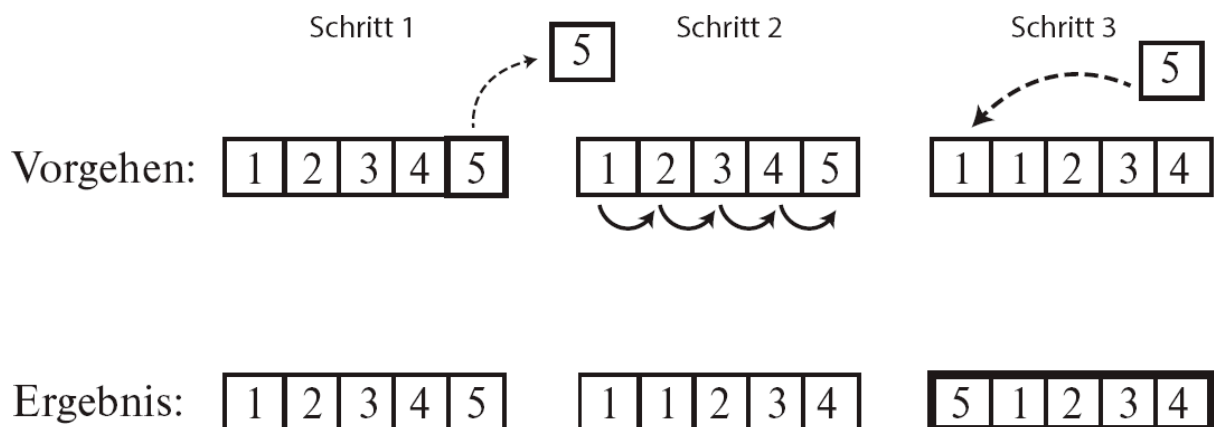
### 4.4.3 Aufgabe 3: Rotation um eine oder mehrere Positionen

In dieser Aufgabe besteht die Problemstellung im Rotieren eines Arrays um  $n$  Positionen nach links bzw. rechts. Dabei sollen die Elemente zyklisch am Anfang bzw. Ende nachgeschoben werden. Das gewünschte Vorgehen ist nachfolgend für eine Rotation um eine Positionen visualisiert, wobei das mittlere Array die Ausgangsbasis bildet:



## Lösung

Der Algorithmus für eine Rotation um ein Element nach rechts ist simpel: Merke dir das letzte Element und dann kopiere wiederholt jeweils das in Rotationsrichtung eins weiter vorne liegende Element in das dahinter.



**Abbildung 4-5** Rotieren von Elementen

Abschließend wird das zwischengespeicherte letzte Element an vorderster Position eingefügt.

```
void rotateRight(final int[] values)
{
    if (values.length < 2)
        return;

    final int endPos = values.length - 1;

    final int temp = values[endPos];

    for (int i = endPos; i > 0; i--)
        values[i] = values[i - 1];
}
```

```
    values[0] = temp;

}
```

Die Rotation nach links arbeitet analog. Hier merken wir uns das vorderste Element. Dann kopieren wir jeweils die dahinterliegenden um eine Position nach vorne und schließlich ersetzen wir das Element an letzter Position mit dem zwischengespeicherten ehemals vordersten Element:

```
void rotateLeft(final int[] values)

{

    if (values.length < 2)

        return;

    final int endPos = values.length - 1;

    final int temp = values[0];

    for (int i = 0; i < endPos; i++)

        values[i] = values[i + 1];

    values[endPos] = temp;

}
```

Probieren wir das Ganze einmal in der JShell aus:

```
jshell> int[] numbers = { 1, 2, 3, 4, 5, 6, 7 }
```

```

numbers ==> int[7] { 1, 2, 3, 4, 5, 6, 7 }

jshell> rotateRight(numbers)

jshell> numbers

numbers ==> int[7] { 7, 1, 2, 3, 4, 5, 6 }

jshell> rotateLeft(numbers)

jshell> numbers

numbers ==> int[7] { 1, 2, 3, 4, 5, 6, 7 }

```

**Rotation um  $n$  Positionen** Eine naheliegende Erweiterung ist das Rotieren um eine bestimmte Anzahl an Positionen. Das kann man dadurch lösen, dass man die gerade entwickelte Funktionalität  $n$  Mal aufruft:

```

static void rotateRightByN_Simple(final int[] values, final
int n)

{

    for (int i = 0; i < n; i++)

        rotateRight(values);

}

```

Diese Lösung ist grundsätzlich akzeptabel, wenn auch durch die häufigen Kopieraktionen nicht performant. Wie es effizienter geht, erkläre ich in meinem Buch »Java Challenge« [5].

## Tipp: Optimierung bei großen Werten für $n$

Zunächst gibt es noch eine Besonderheit zu bedenken: Ist nämlich  $n$  größer als die Länge des Arrays, so muss man nicht ständig rotieren, sondern kann die Anzahl der Rotationen durch die Modulo-Operation  $i < n \% \text{values.length}$  auf das tatsächlich benötigte Maß begrenzen.

### 4.4.4 Aufgabe 4: Zweidimensionales String-Array ausgeben

In diesem Kapitel wurden diverse Ausgaben erstellt, die jedoch oftmals auf `int` gearbeitet haben. Schreiben Sie eine Methode, die ein zweidimensionales String-Array als Eingabe erhält und dann dieses zeilenweise ausgibt und dabei die Zeilennummer protokolliert, etwa wie folgt:

```
Line 0: ONE  
  
Line 1: TWO TWO  
  
Line 2: THREE THREE THREE
```

### Lösung

Wir durchlaufen das Array von oben nach unten. Dabei ergänzen wir in den bereits früher realisierten Varianten ein paar textuelle Angaben sowie Variablen für die aktuelle Zeile und die aktuellen Daten.

```
jshell> void print2dStringArray(String[][] values)  
  
...> {  
  
...>     for (int y = 0; y < values.length; y++)  
  
...>     {  
  
...>         System.out.print("Line " + y + ": ");  
  
...>         for (int x = 0; x < values[y].length; x++)
```

```

...>         {
...>             System.out.print(values[y][x] + " ");
...>         }
...>         System.out.println();
...>     }
...> }

```

| created method print2dStringArray(String[][])

Prüfen wir dies kurz nach:

```

jshell> String[][] sampleValues = {
...>     { "ONE" },
...>     { "TWO", "TWO" },
...>     { "THREE", "THREE", "THREE" }
...> }

sampleValues ==> String[3][] { String[1] { "ONE" }, String[2]
{ "T ... REE", "
    THREE", "THREE" } }

jshell> print2dStringArray(sampleValues)

```

```
Line 0: ONE
```

```
Line 1: TWO TWO
```

```
Line 2: THREE THREE THREE
```

#### 4.4.5 Aufgabe 5: Dreieckiges Array: Upside Down

Bei dieser Aufgabe sei ein dreieckiges Array gegeben. In diesem Array sollen alle Zeilen bis zur Mitte mit den jeweils unteren getauscht werden. Schreiben Sie eine Methode `String[][] upsideDown(String[][])`, um diese Funktionalität bereitzustellen.

Nehmen wir folgendes einfache dreieckige Array:

```
jshell> String[][] original = {  
  
    ...>         { "ONE" },  
  
    ...>         { "TWO", "TWO" },  
  
    ...>         { "THREE", "THREE", "THREE" }  
  
    ...>         }  
  
original ==> String[3][] { String[1] { "ONE" }, String[2] {  
"T ... REE", "THREE"  
  
    , "THREE" } }
```

Dann soll ein Aufruf von `upsideDown(original)` folgendes Resultat liefern:

```
{  
  
    {"THREE", "THREE", "THREE"},
```

```
    {"TWO", "TWO"},  
  
    {"ONE"},  
  
}
```

Wie nahezu immer gibt es verschiedene Lösungswege. Schauen wir uns zwei mögliche Varianten an.

### **Lösung: Brute Force mit neuem Array und viel Kopieren**

Eine Idee ist es, ein neues Array gleicher Länge zu erzeugen und dann einfach das Originalarray von hinten zu durchlaufen und mit den jeweiligen Werten zu befüllen:

```
jshell> String[][] upsideDownV1(String[][] values)  
  
...> {  
  
...>     String[][] result = new String[values.length][];  
  
...>  
  
...>     int resultPos = 0;  
  
...>     for (int i = values.length - 1; i >= 0; i--)  
  
...>     {  
  
...>         result[resultPos] = values[i];  
  
...>         resultPos++;  
  
...>     }
```

```
...>     return result;

...> }
```

Prüfen wir dies kurz nach:

```
jshell> String[][] original = {

...>         { "ONE" },

...>         { "TWO", "TWO" },

...>         { "THREE", "THREE", "THREE" }

...>     }

jshell> print2dStringArray(upsideDownV1(original))

Line 0: THREE THREE THREE

Line 1: TWO TWO

Line 2: ONE
```

**Anmerkungen** Soweit funktioniert das Ganze gut. Die gewünschte Funktionalität ist korrekt umgesetzt. Wie sieht es mit der Effizienz aus? Mittelprächtigt, weil wir nochmals zumindest das äußere Array erstellen, um das Resultat aufzunehmen. Glücklicherweise verweisen die Daten dann auf die Originale, wodurch es speichertechnisch in Ordnung ist.

Aber: Wie könnten wir es effizienter machen? Wie wäre es möglich, die Aufgabenstellung ohne neues Array umzusetzen?

### **Lösung: Tricky Inplace mit Referenztausch**

Tatsächlich ist das machbar. Dabei nutzen wir aus, dass die verschachtelten Arrays wiederum Referenzen auf Arrays enthalten, die man ganz einfach

tauschen kann. Dazu hatten wir in der Einführung schon die Methode `swap()` entwickelt, allerdings für eindimensionale `int[]`. Die dortigen Ideen greifen wir hier wieder auf, eben auf `String[]` ausgerichtet. Nun müssen wir von oben bis zur Hälfte laufen und jeweils die Werte an der ersten und letzten Position tauschen und dies für die nächstinneren Positionen wiederholen, bis man die Mitte erreicht:

```
jshell> void upsideDown(String[][] values)

...> {

...>     for (int i = 0; i < values.length / 2; i++)

...>     {

...>         String[] tmp = values[i];

...>

...>         int endPos = values.length - 1 - i;

...>         values[i] = values[endPos];

...>         values[endPos] = tmp;

...>     }

...> }

| created method upsideDown(String[][])
```

Da wir hier auf den ursprünglichen Daten, also *inplace*, arbeiten, gibt es nun keinen Rückgabewert mehr und wir verwenden `void`.

Auch hier prüfen wir die Funktionalität kurz nach:

```
jshell> upsideDown(original)
```

```
jshell> print2dStringArray(original)
```

```
Line 0: THREE THREE THREE
```

```
Line 1: TWO TWO
```

```
Line 2: ONE
```

```
jshell> upsideDown(original)
```

```
jshell> print2dStringArray(original)
```

```
Line 0: ONE
```

```
Line 1: TWO TWO
```

```
Line 2: THREE THREE THREE
```

# Index

?-Operator, 210

@Override, 146

Abarbeitung

    iterative, 251

    sequenzielle, 251

Ableitungshierarchie

    von Exceptions, 306

Accessor, 135

ActionListener, 248

Angewohnheit

    gute, 357

Annotation, 146, 370

    @Override, 146

Applikation

    als Klasse, 124

    in der IDE, 137

Applikationsdesign, 342

ARM, 68, 285, 305

Array, 83

    Definition, 83

    dynamische Definition, 87

    dynamische Erzeugung, 87, 88

    eindimensionales, 91

- Elemente durchlaufen, 85, 86
- Flächen füllen, 99
- gebräuchliche Aktionen, 83
- Inhalt rotieren, 96
- Länge, 85
- mehrdimensionales, 89, 95
- nicht rechteckig, 90
- Suchen in, 94
- Tauschen von Elementen, 91
- textuell ausgeben, 86
- verändern, 85
- Zugriff, 84

ArrayList, 165

- aus Array erzeugen, 166
- Elemente durchlaufen, 168
- Elemente löschen, 168, 170
- erzeugen, 165
- Größe, 168
- modifizieren, 167
- sortieren, 169
- Teilbereiche ermitteln, 170
- Werte hinzufügen, 166
- Zugriff, 167

ArrayList<E>, 187

Arrays, 254

- stream(), 254

Assert, 369

Assertions, 370

Attribut, 110

- Präfix, 359

- Typpräfix, 360
- Wert ändern, 113
- Zugriff, 112, 134
- Attribute, 112
- Aufzählung
  - mit enum, 211
- Ausführung
  - bedingte, 33
  - wiederholte, 45
- Auto-Boxing, 208
- Auto-Unboxing, 208
- AutoCloseable, 306
- Automatic Resource Management, 68, 285, 305
- Basisklasse
  - abstrakte, 143, 145
  - Definition, 144
  - Einsatzgebiet, 144
  - Vorgehen, 144
- bedingte Ausführung, 33
- Bedingung
  - boolesche, 252
- Berechnung
  - Merkwürdigkeit, 203
- Berechtigung, 277
- Bezeichner, 23
- Bibliothek
  - externe, 388
- Binärdarstellung, 205
- Binden
  - dynamisches, 239

- Block, 32
- Block Scope, 200
- Boilerplate-Code, 247
- Boxing, 208
- break, 214, 222, 224
  - in Schleifen, 222
- Bytecode, 385
- CamelCase-Schreibweise, 358
- can-act-like-Beziehung, 154
- case, 213
- Cast, 41, 206
  - Narrowing, 206
  - Widening, 206
- Casting, 206
  - Restriktionen, 207
- catch, 296
- chars(), 254
- Checked Exception, 306
  - Besonderheit, 303
- Class<T>, 147
- Client, 152
- Coding Conventions, 358
  - Formatierung, 358
  - Namensgebung, 358
- collect(), 262
- Collection, 186
  - parallelStream(), 254
  - stream(), 254
- Collection<E>, 186
  - add(), 186

- addAll(), 186
- isEmpty(), 186
- size(), 186
- Collector, 262
- Collectors, 262
  - counting(), 262
  - groupingBy(), 262
  - joining(), 262
  - partitioningBy(), 262
  - toCollection(), 262
  - toList(), 262
- Comma Separated Values, 337
- Comparator, 247
- compare(), 247
- Conditional-Operator, 366
- Container
  - Basisinterfaces, 186
  - generischer, 183
- continue, 222, 225
  - in Schleifen, 222
- copy(), 277
- counting(), 262
- createDirectory(), 270
- createFile(), 270
- CSV, 337
- Darstellung
  - ausgeklügelte, 289
- Datei
  - anlegen, 270
  - einlesen, 274

- löschen, 280
- schreiben, 274
- Dateibehandlung, 281
- Dateigröße, 277
- Dateiinhalt
  - verarbeiten, 283
- Dateiverarbeitung, 269
- Daten
  - extrahieren, 68, 339
- Datenbank, 388
- Datenmodell, 344
- Datenstruktur
  - Liste, 187
  - Menge, 188
- Datentyp, 20
  - primitiver, 201
- DateTimeFormatter, 319
- Datum
  - aktuelles, 311
  - Informationen, 312
  - repräsentieren, 311
- Datumsarithmetik, 312, 317
- Datumsverarbeitung, 309
  - Formatierung, 319
  - Parsing, 319
- DayOfWeek, 309
- Defaultkonstruktor, 115, 116
- Diamond Operator, 184
- Dictionary, 189
- Directory-Baum

- darstellen, 285
- distinct(), 260
- do-while, 48
- Dokumentation, 50, 363, 389
- Double, 202
  - parseDouble(), 209
- DoubleStream, 254
- Download, 5
- dropWhile(), 258
- Duration, 316
- Dynamic Binding, 239
- Eclipse, 9, 10, 14, 16
  - Basiskonfiguration, 366
  - erste Klasse, 16
  - erstes Projekt, 14
  - Installation, 10
- effectively final, 243
- Eigenschaften, 112
- einzelne Anweisung
  - Besonderheit, 34
- else, 34
- else-if, 35
- entrySet(), 179
- Entwicklungsumgebung, 9
- enum, 211
  - Besonderheit, 212
- equals(), 120, 122, 147, 150
  - Kontrakt, 150
- Exception, 295
  - Checked, 306, 307

- Hintergrundwissen, 306
- IllegalArgumentException, 296
- IllegalStateException, 296
- NullPointerException, 296
- propagieren, 302
- selbst auslösen, 301
- selbst definieren, 302
- Unchecked, 306, 307
- UnsupportedOperationException, 296

Existenz

- prüfen, 276

Experimentiermodus

- interaktiver, 388

Fakultät, 226

Fall Through, 214, 215

Fallstrick, 275

Fallunterscheidung, 32

Fehlerbehandlung, 295

- in der Praxis, 303

Fehlersituation

- behandeln, 363

Fibonacci-Zahlen, 227

File, 68, 281

- Aktionen, 282
- createNewFile(), 282
- delete(), 282
- exists(), 282
- getName(), 282
- isDirectory(), 282
- isFile(), 282

- list(), 283
- listFiles(), 283
- mkdir(), 282
- makedirs(), 282
- renameTo(), 282
- Files, 270, 272
  - lines(), 273
  - list(), 271
  - readAllLines(), 273
  - write(), 272
- filter(), 256
- finally, 300
- Float, 202
  - parseFloat(), 209
- for, 45
- for each, 46
- forEach(), 261
- Formatierung
  - grundlegende Regeln, 358
- Function, 257
  - apply(), 257
- Functional Interface, 246
  - Implementierung von, 247
- FunctionalInterface
  - Annotation, 246
- Gültigkeitsbereich, 199
- Gültigkeitsprüfung, 304
- Garbage Collector, 387
- Generalisierung, 237, 238
- Generics, 395

- generische Container, 183
- Generische Typen, 181
- getClass(), 147
- Groß- und Kleinschreibung, 60
- groupingBy(), 262
- Grundrechenarten, 26
- Hallo Welt, 19
- hashCode(), 147
- HashMap, 177
  - Abbildung löschen, 179
  - containsKey(), 178
  - containsValue(), 178
  - Elemente durchlaufen, 179
  - Elemente löschen, 180, 181
  - entrySet(), 179
  - erzeugen, 177
  - Fallback, 178
  - Größe, 179
  - keySet(), 179
  - modifizieren, 179
  - values(), 179
  - Werte hinzufügen, 177
  - Zugriff, 178
- HashSet, 171
  - alle Elemente löschen, 174
  - durchlaufen, 173
  - Element löschen, 173
  - erzeugen, 171
  - Größe, 173
  - sortieren, 174

- Werte hinzufügen, 172
- Hello World, 19
- Hexadezimaldarstellung, 205
- Highscore-Liste, 337
- Hilfe
  - im Internet, 389
- Hilfsdatenstrukturen, 344
- Hintergrundwissen, 306
- HTML
  - ausgeben, 353
  - erzeugen, 350
  - im Browser, 353
- Identität, 111
- if, 33
- if-else, 34
- if-else-if-else, 35
- Implementierung, 153
- Implementierungsvererbung, 238
- Import, 126, 127
- Information Hiding, 138
- Informationen
  - einlesen, 274
  - schreiben, 274
  - Verstecken von, 138, 139
- Informationen extrahieren, 65
- Informationen formatieren, 66
- innere Klasse, 241
- InputStream, 68, 283
- Installation, 5, 7
- instanceof, 150, 151

- Instanz, 110
- Integer, 202
  - parseInt(), 209, 340
- IntelliJ IDEA, 9
- Interface, 152
  - Abgrenzung zu Vererbung, 154
  - Angebot von Verhalten, 152
- Interfaces
  - mehrere, 153
- Intermediate Operation, 255
  - zustandsbehaftete, 260
  - zustandslose, 255
- IntStream, 254
  - chars(), 254
  - range(), 254
- is-a-Beziehung, 154, 238
- Iterable
  - forEach(), 261
- Iteration, 250
  - externe, 250, 251
  - interne, 250, 251
- Iterator, 175, 251
  - erzeugen, 175
  - zum Durchlaufen nutzen, 175
  - zum Löschen nutzen, 176
- Java
  - Download, 5
  - im Überblick, 3
  - Installation, 7
- Java Development Kit, 388

- Java Virtual Machine, 385
- Java-Ökosystem, 385, 388
- JDK, 388
- JIT, 388
- joining(), 262
- JShell, 19, 381
  - Besonderheiten, 383
  - Kommandos, 382
  - neue Features nutzen, 383
- jshell, 381, 382, 384
  - Historie der Befehle, 382
  - Tab-Completion, 383
  - Tastaturkürzel, 383
- JUnit
  - Annotation, 370
  - assertEquals(), 370
  - assertFalse(), 370
  - Assertions, 370
  - assertNotNull(), 370
  - assertNotSame(), 370
  - assertNull(), 370
  - assertSame(), 370
  - assertThrows(), 371
  - assertTrue(), 370
  - fail(), 371
- JUnit 5, 368
- JUnit-Framework, 368, 370
- Just-in-Time-Compiler, 388
- JVM, 385
- Kalenderausgabe, 321

- Kapselung, 138
- Keep It Human-Readable, 357
- Keep It Natural, 357
- keine Rückgabe, 38
- keySet(), 179
- Keyword, 377
- Klasse, 109, 110
  - abgeleitete, 142
  - anlegen, 129
  - Applikation als, 124
  - Attribute, 112
  - ausführbare, 123
  - ausführen, 131
  - Basis-, 142
  - definieren, 110
  - Eigenschaften, 112
  - Grundlagen, 110
  - innere, 241
    - Varianten, 241
  - Ober-, 142
  - Sub-, 142
  - Super-, 142
  - Theorie, 110
- Klassenhierarchie, 237, 238
- Klassenlänge
  - maximale, 364
- Kommandozeileninterpreter, 19
- Kommentar, 44
  - Überschrift-, 363
- Kompiliertyp, 239

- Konsistenz, 150
- Konstante, 23
- Konstruktor-Chaining, 115
- Kontrakt
  - equals(), 150
- Kontrollfluss, 365
- Konvertierung
  - Path<->File, 282
- Kopieren, 277
- Kurzzeitgedächtnis, 364
- Länge ermitteln, 61
- Löschen, 280
- Lambda, 243, 245, 247
  - als Parameter, 249
  - als Rückgabewert, 249
  - im Java-Typsistem, 246
  - Kurzschreibweisen, 248
- Lambda-Ausdruck, 245
- Laufzeittyp, 239
- Leerzeichen entfernen, 61
- lines(), 273
- LinkedList<E>, 187
- List, 187
- list(), 271
- List<E>, 187
- Liste, 187
- Lizenzpolitik, 6
- Local Variable Type Inference, 22
- LocalDate, 311, 314
- LocalDateTime, 311, 315, 317

- LocalTime, 315
- Logik, 365
- Long, 202
  - parseLong(), 209
- LongStream, 254
- Lookup-Tabelle, 189
- Magic Number, 363
- Magic String, 363
- map(), 257
- Map<K,V>, 189
- Math.random(), 42
- Memory Leak, 387
- Menge, 188
- Method Scope, 200
- Methode, 36, 110, 117
  - abstrakte, 144, 152
  - als Baustein, 39
  - aufrufen, 36
  - aus dem JDK, 36
  - definieren, 36
  - eigene definieren, 37
  - Signatur einer, 43
  - statische, 119
- Methodenlänge
  - maximale, 364
- Methodenreferenz, 249
- Modellierungsfreiheit, 139
- Modulo, 205
- Modulo-Operator, 26
- Monatslänge, 310

- Month, 309
- move(), 279
- Mutator, 135
- Namen
  - aussagekräftige, 360
  - für Containerklassen, 362
  - Lesbarkeit, 362
  - Semantik, 362
  - sinnvoll gewählte, 362
  - sinnvolle, konsistente, 362
- Namensbestandteil, 276
- Namensgebung, 360
  - grundlegende Regeln, 358
- Namenskonsistenz, 361
- Namenskürzel
  - vermeide, 360
- Namensregeln, 358
  - CamelCase-Schreibweise, 358
- Narrowing, 206, 207
- NetBeans, 9
- Noise, 247
- Null-Akzeptanz, 150
- Number, 202
- NumberFormatException, 208, 340
- Object, 147, 246
  - equals(), 147, 150
  - getClass(), 147
  - hashCode(), 147
  - toString(), 147, 149
- Objekt, 110

- erzeugen, 110
- Grundlagen, 110
- Konstruktion, 114
- Theorie, 110
- Vergleich, 120

## Objekte

- vergleichen, 120

Objektorientierung, 109

Objektzustand, 112, 121, 147, 359

Ökosystem, 385

of(), 254, 258

Oktalardarstellung, 205

## OO-Design

- Attribut, 110

- Generalisierung, 238

- Interface, 152

- is-a-Beziehung, 238

- Kapselung, 138

- Methode, 110

- Objektzustand, 112

- Overloading, 145, 146

- Overriding, 145

- Polymorphie, 239

- Realisierung, 153

- Referenz, 111

- Spezialisierung, 238

- Sub-Classing, 240

- Sub-Typing, 240

- Typkonformität, 153

- Vererbung, 142

- OO-Techniken
  - abstrakte Basisklasse, 143, 145
- OpenJDK, 6
- OpenOption, 272
- Operator, 25
- Operatoren
  - arithmetische, 25
  - logische, 31
- Oracle JDK, 6
- Orthogonalität, 364
- OutputStream, 283
- Overloading, 145, 146
- Overriding, 145
- Package, 126
  - anlegen, 129
  - benutzerdefinierte, 126
  - eingebaute, 126
  - Import, 128
  - vordefinierte, 126
- Paging, 261
- parallel(), 254
- Parallelverarbeitung, 254
- Parameterliste, 364
  - kurzhalten, 364
  - Reihenfolge, 364
- partitioningBy(), 262
- Path, 270, 272
- Pattern Matching, 151
- Period, 313
- Polymorphie, 239

- Prädikat, 252
  - boolesche Bedingung, 252
- Predicate, 252
  - boolesche Bedingung, 252
  - test(), 252
- primitiver Datentyp, 202
  - boolean, 202
  - byte, 202
  - char, 202
  - double, 202
  - float, 202
  - int, 202
  - long, 202
  - short, 202
- Programm
  - Ausführung, 387
- Programmdesign, 363
- Programmierstil
  - einheitlicher, 358
  - Grundregeln, 357
  - guter, 357
- Programmierung
  - funktionale, 245
- Projekt
  - anlegen, 129
  - strukturieren, 132
  - Verzeichnislayout, 132
- Prototyping, 381
- provides-Beziehung, 154
- Psychologie, 364

- Rückgabewert
  - prüfen, 363
- range(), 254
- Raw Type, 181
- Read-Eval-Print-Loop, 19, 381, 388
- Readable, 68
- readAllLines(), 273–275
- Reader, 283
- readString(), 274, 275
- Realisierung, 153
- Record, 154
- Refactoring, 9
- Referenz, 111
- Referenzvergleich, 121
- Reflection, 147
- Reflexivität, 150
- Regel
  - Keep It Human-Readable, 357
  - Keep It Natural, 357
- Rekursion, 226
  - Fakultät, 226
  - Fibonacci-Zahlen, 227
- renameTo(), 283
- repeat(), 36
- REPL, 19, 381, 388
- Rolle, 154
- Runnable, 246
- RuntimeException, 307
- SAM-Typ, 243, 246, 247
- Scanner, 68

- Daten extrahieren, 68
  - useDelimiter(), 68
- Schaltjahr, 310
- Schlüsselwort, 377
- Schleife, 45
  - break, 222
  - do-while, 48
  - for, 45
  - for each, 46
  - while, 47
- Schrittweite, 46
- Server, 152
- Set, 188
- Set<E>, 188
- Short, 202
- Short-circuiting Operations, 255
- Sichtbarkeit, 133, 139
  - private, 139
  - protected, 139
  - public, 139
- Sichtbarkeitsbereich, 199
- Sieg
  - prüfen, 333
- Signatur, 43
- Single Abstract Method, 243
- sorted(), 260
- Sourcecode, 4
  - editieren, 130
  - Prüfung, 366
- Spezialisierung, 237, 238

## Spielfeld

- darstellen, 331
- initialisieren, 331

## Spielstand

- einlesen, 337
- modellieren, 337

## Stack

- selbst realisieren, 190

## Standardkonstruktor, 115

## statische Methode, 119

## statische Variable, 119

## Stream, 253

- collect(), 262
- Create Operations, 254
- distinct(), 260
- dropWhile(), 258
- filter(), 256
- Intermediate Operations, 255, 260
- limit(), 261
- map(), 257
- of(), 254, 258
- skip(), 261
- sorted(), 260
- takeWhile(), 258
- Terminal Operations, 255, 261
- toList(), 262

## String, 59, 68

- char[], 72
- format(), 66
- Groß- und Kleinschreibung, 60

- Länge ermitteln, 61
- Leerzeichen entfernen, 61
- mehrzeilig, 71
- repeat(), 36
- replace(), 65
- replaceAll(), 65
- strip(), 340
- Suchen und Ersetzen, 63
- Teilbereiche extrahieren, 63
- Title Case, 74
- toCharArray[], 72
- wiederholen, 63

Stringaktionen

- gebräuchliche, 59

Stringkonkatenation, 59

Strings wiederholen, 63

Strukturierung, 342

Sub-Classing, 240

Sub-Typing, 240

Suchen und Ersetzen, 63

switch, 213, 215, 219

- bisherige Schwachstellen, 215
- Vollständigkeitsprüfung, 216

Switch Expressions, 215, 219

Symmetrie, 150

Systemressource, 283

Tab-Completion, 383

takeWhile(), 258

Teilbereiche extrahieren, 63

TemporalAdjusters, 317, 318

Terminal Operations, 255

Ternary-Operator, 210

Test, 389

test(), 252

Testen, 368

Testfall, 368

Text Block, 71

    HTML, 72

    Platzhalter, 71

throw, 301

throws, 302

Tic Tac Toe, 331

toCollection(), 262

toFile(), 282

Token, 68

toList(), 262

Tool, 388

toPath(), 282

toString(), 147, 149

Transformation

    in Lambda, 247

Transitivität, 150

try, 296

Typ

    primitiver, 201

Type Inference, 248

Typenerweiterung, 206

Typkonformität, 153

Typprüfung

    equals(), 150

- instanceof, 150
- Typumwandlung, 41
- Typverkleinerung, 206
- Umbenennen, 279, 283
- Unboxing, 208
- Unchecked Exception, 306
- Unit Test, 368
  - ausführen, 370
  - mit Eclipse erstellen, 369
  - schreiben, 370
- Unterstrich
  - in Zahlen, 206
- Use-Beziehung, 152
- values(), 179
- var, 22
- Var Args, 43
- Variable, 20
  - Definition von, 20
  - statische, 119
- variable Argumente, 43
- Variablenname, 23
- Verarbeitungsschritt, 255
- Vererbung, 142, 154
  - Wissenswertes, 237
- Vergleich
  - Referenz-, 121
- Vergleichsoperatoren, 30
- Verhalten, 111, 117
  - definieren, 117
- Verhaltensweise, 154

## Verzeichnis

- anlegen, 270

- löschen, 280

## Verzeichnisinhalt

- auflisten, 271

Verzeichnislayout, 132

void, 38

## Wert

- Konvertierung, 207

Wertebereichsprüfung, 138

while, 47

Widening, 206

wiederholte Ausführung, 45

Wiederverwendbarkeit, 364

Wiederverwendung, 142

## Wille

- letzter, 300

## Wort

- reserviertes, 380

Wrapper-Klassen, 201

- Konstruktion, 207

- Konvertierung, 208

write(), 272

Writer, 283

- close(), 284

- flush(), 284

- write(), 283

writeString(), 274, 275

## Zahlenformat

- wissenschaftliches, 204

Zeit

aktuelle, 315

Zeitangabe

unvollständige, 311

Zeitdifferenz, 313, 316

Zeitsprung, 314

Zeitzone, 317

Zufallswert, 41

Zustand, 111

Zuweisungsoperator, 28