Inhaltsverzeichnis

1 Einführung in Java

1.1 Algorithmus

- Algorithmus: Schritt-für-Schritt-Beschreibung zur Lösung eines Problems
- Pseudocode: in strukturierter natürlicher Sprache geschriebener Algorithmus

1.2 Prozedurale Zerlegung

- Zerlegung eines Problems in Methoden
- Dekomposition: Trennung in Teile, wobei jeder Teil einfacher als das Ganze ist
- Redundanz: Die gleiche Folge von Anweisungen taucht mehrfach in einem Programm auf

1.3 Bezeichner und Schlüsselwörter

- Bezeichner: Name für ein Programmelement (Daten, Methoden…) Regeln wie Klassen groß usw.
- Schlüsselwörter: Bezeichner, der in Java reserviert ist und daher nicht selbst vergeben werden kann wie primitive Datentypen oder public static void main

1.4 Kommentare

- /** … */ => Java doc
- /* … */ => Abschnitte
- // => Zeile

1.5 Lesenotizen

1.5.1 Escape sequence

1. \t Tabulator-Zeichen
2. \n Zeichen für neue Zeile
3. \“ Anführungsstriche
4. \\ Backslash

1.5.2 Primitive Datentypen

Java Datentyp	Größe	Wertebereich
boolean	8 bit	true/false
byte	8 bit	-2^7 bis 2^7-1
short	16 bit	-2^{15} bis $2^{15}-1$
char	16 bit	0 bis 65535
int	32 bit	-2^{31} bis $2^{31}-1$
float	32 bit	+/-1,4E-45 bis +/-3,4E+38
long	64 bit	-2^{63} bis $2^{63}-1$
double	64 bit	+/-4,9E-324 bis +/-1,7E+308

1.5.3 Präzedenzregeln für Operator

- () ist höhergestellt als *, / und %,diese haben gleiche Präzedenz und höhergestellt als + und -

2 Primitive Daten & definite Schleifen

2.1 Typumwandlung

- Eine Umwandlung von einem Typ in einen anderen

Beispiele:

```
double result = (double) 19 / 5;      // 3.8
int result2 = (int) result;           // 3
```

- Nimmt nur 1. Zeichen danach, deshalb Klammern bei ganzen Ausdrücken wie:

```
double average = (double) (a + b + c) / 3;
```

2.2 String-Konkatenation

- Mit +, aber:

```
1 + 2 + "abc"  ist "3abc"
"abc" + 9 * 3  ist "abc27"
```
(Präzedenzregel: * vor +)

2.3 Geltungsbereich

- Variablen existieren nur zwischen den geschweiften Klammern => lokale Variablen

2.4 Klassenkonstanten

- Eine Variable, die im gesamten Programm genutzt werden kann

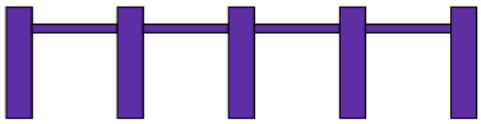
```
public static final <type> <name> = <value> ;
```

- Über main schreiben

2.5 Zaunpfahlproblem

- Lösung: Extra-Statement außerhalb der Schleife für den ersten Zaunpfahl
- Nennt man auch *fencepost loop* oder eine "loop-and-a-half" Lösung
  - Der korrekte Algorithmus:

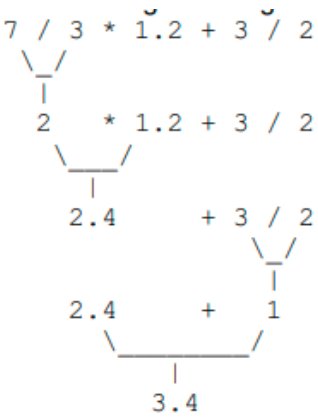
```
setze einen Pfahl.
for (Länge des Zauns – 1) {
    befestige einen Querbalken.
    setze einen Pfahl.
}
```



2.6 Lesenotizen

2.6.1 int und double mischen

- werden immer in double umgewandelt



Beachte: `3 / 2` ist eine Ganzzahldivision mit Zwischenergebnis `1` und nicht `1.5`.

2.6.2 Mehrfache Variablendefinition

- Nur für einen Datentyp möglich `int a = 2, b = 3, c = -4;`

2.6.3 for-Schleife mit einem Statement

```
for (int i = 1; i <= 3; i++)
    System.out.println("Dies wird 3x gedruckt");
    System.out.println("Dies auch ..., oder?");
```

2.6.4 Degenerierte for-Schleifen

Bezeichnet man Schleifen, die die Test-Bedingung in der Initialisierung nicht erfüllen oder Endlosschleifen

2.6.5 Zahlenfolgen erzeugen

count	Gewünschte Ausgabe	5 * count	5 * count - 3
1	2	5	2
2	7	10	7
3	12	15	12
4	17	20	17
5	22	25	22

```
for (int count = 1; count <= 5; count++) {
    System.out.print(5 * count - 3 + " ");
}
```

2.6.6 Lokale Variablen

Eine in einer Methode deklarierte Variable nennt man lokale Variable. Man soll immer so lokal wie möglich die Variablen deklarieren

3 Parameter und Objekte

3.1 Objekte und Klassen

Objekt: Ein Ding, das Daten und Verhalten enthält.

Klasse: Ein Programmteil, der eine Schablone für eine bestimmte Sorte von Objekten definiert.

- Beispiele:
  - Die Klasse `String` repräsentiert Objekte, die Text speichern und verarbeiten können.
  - Die Klasse `Point` repräsentiert Objekte, die Daten in der Form (x, y) speichern und verarbeiten können.
  - Die Klasse `Scanner` repräsentiert Objekte, die Informationen von der Tastatur, aus Dateien oder aus anderen Quellen lesen können.

3.2 Konstruktion

Konstruktion: Erzeugung eines neuen Objekts.

- Objekte werden mit dem Schlüsselwort `new` konstruiert (erzeugt).
- Objekte müssen vor ihrer Benutzung erzeugt werden
- Syntax der Objekt-Konstruktion:
  <type> ***<name>*** `=` `new` ***<type>*** `(` ***<parameters>*** `);`
  - Beispiele:
    ```
    Point p = new Point(7, -4);
    DrawingPanel window = new DrawingPanel(300, 200);
    Color orange = new Color(255, 128, 0);
    ```
  - Klassennamen beginnen normalerweise mit einem Großbuchstaben (`Point`, `Color`).
  - Hier gleich der erste Sonderfall: Objekte der Klasse String können ohne `new` erzeugt werden:
    ```
    String name = "Amanda Ann Camp";
    ```

3.3 Aufruf von Objektmethoden

- Syntax des Methodenaufrufs:
  <variable> `.` ***<method name>*** `(` ***<parameters>*** `)`
  - Beispiele:
    ```
    String gangsta = "G., Ali";
    System.out.println(gangsta.length());     // 7

    Point p1 = new Point(3, 4);
    Point p2 = new Point(0, 0);
    System.out.println(p1.distance(p2));      // 5.0
    ```

3.4 Pointobjekte

- In einem `Point`-Objekt gespeicherte Daten:

Attributname	Beschreibung
`x`	X-Koordinate des Punktes
`y`	Y-Koordinate des Punktes

- Methoden für `Point`-Objekte:

Methodenname	Beschreibung
`distance(`*p*`)`	Berechnet, wie weit der Punkt von einem anderen Punkt *p* entfernt ist
`setLocation(`*x*`, `*y*`)`	Setzt die Koordinaten des Punkts auf gegebene Werte
`translate(`*dx*`, `*dy*`)`	Verändert die Koordinaten des Punkts um die gegebenen Verschiebungen in X- und Y-Richtung.

- `Point`-Objekte können durch `println` Statements ausgegeben werden:

```
Point p = new Point(5, -2);
System.out.println(p);    // java.awt.Point[x=5,y=-2]
```

3.5 Wert-Semantik

Wert-Semantik (call by value): Bei der Parameterübergabe und bei der Zuweisung werden Variablenwerte kopiert.

- Primitive Datentypen
- Primitive Parameter sind lokale Variable ohne Außenwirkung

3.6 Zeiger-Semantik

Zeiger-Semantik (call by pointer value / passing object references by pointer value): Bei der Parameterübergabe und bei der Zuweisung werden keine Werte, sondern Zeiger kopiert.

- Objekte
- Zeigervariablen speichern Adresse und nicht Wert
- Objekte Parameter haben Außenwirkung

3.7 Lesenotizen

3.7.1 Klasse-Math

Methode	Beschreibung
`abs(`*value*`)`	Absolutbetrag
`ceil(`*value*`)`	Aufrunden
`cos(`*value*`)`	Cosinus vom Bogenmaß
`floor(`*value*`)`	Abrunden
`log(`*value*`)`	Logarithmus zur Basis e
`log10(`*value*`)`	Logarithmus zur Basis 10
`max(`*value1*`, `*value2*`)`	der größere zweier Werte
`min(`*value1*`, `*value2*`)`	der kleinere zweier Werte
`pow(`*basis*`, `*exponent*`)`	*basis* potenziert zum *exponent*
`random()`	Zufallswert `double` >=0.0 und <1.0
`round(`*value*`)`	Kaufmännisches Runden auf die nächste ganze Zahl
`sin(`*value*`)`	Sinus vom Bogenmaß
`sqrt(`*value*`)`	Quadratwurzel

`toRadians(`*value*`)`	Umrechnung von Grad in Bogenmaß
`toDegrees(`*value*`)`	Umrechnung von Bogenmaß in Grad

Darüber hinaus besitzt `Math` einige oft genutzte Konstanten:

Konstante	Beschreibung
`E`	2.7182818...
`PI`	3.1415926...

3.7.2 Stringobjekte

Methodenname	Beschreibung
`charAt(`*index*`)`	Zeichen an der gegebenen Indexstelle
`indexOf(`*str*`)`	Index, an dem der als Parameter gegebene String *str* in dem String-Objekt beginnt (-1, wenn er nicht vorkommt)
`length()`	Anzahl der Zeichen im String-Objekt
`substring(`*index1*`, `*index2*`)`	Die Zeichen von einschließlich *index1* bis ausschließlich *index2*

Methodenname	Beschreibung
`toLowerCase()`	Ein neuer String in Kleinbuchstaben
`toUpperCase()`	Ein neuer String in Großbuchstaben

3.7.3 Scannerobjekte

Methode	Beschreibung
`nextInt()`	Liest und gibt die Benutzereingabe als `int` zurück
`nextDouble()`	Liest und gibt die Benutzereingabe als `double` zurück
`next()`	Liest und gibt die Benutzereingabe als `String` zurück
`nextLine()`	Liest und gibt die nächste Eingabezeile als `String` zurück

```
Scanner console = new Scanner(System.in);
System.out.print("Wie alt sind Sie? ");     // prompt
int alter = console.nextInt();
System.out.println("Sie werden 40 in " + (40 - alter) + " Jahren.");
```

- Tokens Blöcke von Zeichen zwischen white space

4 Bedingte Ausführung

4.1 If-/else-statement

```
if (<condition>) {
    <statement> ;
    <statement> ;
    ...
    <statement> ;
}
```

```
if (<condition>) {
    <statement(s)> ;
} else {
    <statement(s)> ;
}
```

```
if (<condition>) {
    <statement(s)> ;
} else if (<condition>) {
    <statement(s)> ;
} else {
    <statement(s)> ;
}
```

4.2 Feinheiten beim Vergleichen

4.2.1 Equals-Methode

- Bei Objekten notwendig, da == nur die Zeigeradressen vergleicht
- Bei String kann == funktionieren da Compiler sehr schlau ist und bereits initialisierte Ausdrücke selben Speicherort referenziert. Bei Eingaben mit der Konsole geht es nicht, da Wert noch unbekannt

```
if (name.equals("Bond")) {
```

4.2.2 Double-Rundungsfehler

- Differenz bestimmen und kleiner Epsilon sein

```
public static final double EPSILON= 0.001;

...

double euro= 0.01 + 0.02 + 0.10 + 0.02 + 0.20 + 0.05;
if (Math.abs(euro - 0.4) < EPSILON) {
    System.out.println("Hier ist Dein Kaugummi");
} else if (euro > 0.4) {
    System.out.println("Das war zuviel");
} else {
    System.out.println("Nachzahlen bitte");
}
```

4.2.3 Min/Max-Schleifen

- Min/Max Wert deklarieren und durch iterieren

```
Scanner console= new Scanner(System.in);
int max= Integer.MIN_VALUE;
for (int i=1; i<=10; i++) {
    System.out.print("Zahl "+i+": ");
    int n= console.nextInt();
    if (n > max) {
        max= n;
    }
}
System.out.println("Maximum: "+max);
```

4.3 Char

- Wird auch als Zahl interpretiert dessen Zahlenwert bei ASCII & Verkettung mit String geht
- Char primitiver Datentyp keine eigenen Methoden deshalb Character

```
if (Character.toLowerCase(s.charAt(i)) == c) {
    count++;
```

Methode	Beschreibung	Beispiel
`getNumericValue(ch)`	Wandelt ein Zeichen, das aussieht wie eine Ziffer, in eine Zahl um	`Character.getNumericValue('6')` liefert `6`
`isDigit(ch)`	Prüft, ob ch eines der Zeichen '0' bis '9' ist	`Character.isDigit('X')` liefert `false`
`isLetter(ch)`	Prüft, ob ch ein Buchstabe ist	`Character.isLetter('f')` liefert `true`
`isLowerCase(ch)`	Prüft, ob ch ein Kleinbuchstabe ist	`Character.isLowerCase('q')` liefert `true`
`isUpperCase(ch)`	Prüft, ob ch ein Großbuchstabe ist	`Character.isUpperCase('q')` liefert `false`
`toLowerCase(ch)`	Liefert den zugehörigen Kleinbuchstaben	`Character.toLowerCase('Q')` liefert `'q'`
`toUpperCase(ch)`	Liefert den zugehörigen Großbuchstaben	`Character.toUpperCase('q')` liefert `'Q'`

4.4 If/else u. return

- Immer alle Pfade return-Statement geben, am Ende mit else oder normal und for-Schleife die nicht durchlaufen werden berücksichtigen

4.5 Exceptions erzeugen

- Exceptions sind Laufzeitfehler.
- Beispiel: `int x=1/0;` ⇒ … `ArithmeticException: / by zero`

Vorbedingung: Eine Bedingung, die vor der Ausführung einer Methode erfüllt sein muss, damit die Methode eine Aufgabe durchführen kann.

Nachbedingung: Eine Bedingung, die von einer Methode als Garantie nach ihrer Ausführung gegeben wird.

- Sinnvoll: Vorbedingung prüfen und Exception erzeugen.
  Beispiel:

```
/** Vorbdg: jahre muss >= 0 sein
    Nachbdg: liefert 12 x jahre */
public static int alterInMonaten(int jahre) {
    if (jahre < 0) {
      throw new IllegalArgumentException("jahre muss >= 0 sein.");
    }
    return jahre*12;
}
```

4.6 Klasse Random

Methode	Beschreibung
`nextInt()`	Liefert eine zufällige ganze Zahl
`nextInt(`*max*`)`	Liefert eine ganzzahlige Zufallszahl aus {0,1,2,…,*max*-1}
`nextDouble()`	Liefert eine reelle Zufallszahl im halboffenen Intervall [0.0, 1.0)

- Beispiel:

```
Random rand = new Random();
int randomNumber = rand.nextInt(10);
// randomNumber has a random value between 0 and 9
```

- Üblicher Weg zur Erzeugung von Zufallszahlen zwischen 1 und N:
  - Beispiel: N=20 (inklusive):

```
int n = rand.nextInt(20) + 1;
```

- Zufallszahl in beliebigem Intervall [*min, max*]:

```
nextInt(<max> - <min> + 1) + <min>
```

  - Beispiel: Zufallszahl zwischen 4 und 9 (inklusive):

```
int n = rand.nextInt(6) + 4;
```

4.7 Zufallsindex für Text

```
public static char zufallsVokal() {
    Random rand = new Random();
    String vokale = "aeiou";
    int laenge = vokale.length();
    char c = vokale.charAt(rand.nextInt(laenge));
    return c;
}
```

- Zufälligen Buchstaben auswählen von A-Z

```
public static char zufallsBuchstabe() {
    Random rand = new Random();
    char c = (char)('A' + rand.nextInt(26));
    return c;
}
```

4.8 Lesenotizen

4.8.1 Switch

Die allgemeine Syntax des `switch`-Statements ist:

```
switch (<expression>) {
case <const expression> :
  <statement(s)> ;
  break;
...
case <const expression> :
  <statement(s)> ;
  break;
default:
  <statement(s)> ;
}
```

5 Programmlogik und indefinite Schleifen

5.1 While-Schleife

`while`-Schleife: Führt Anweisungen durch so lange eine Bedingung wahr ist.

- Syntax:

```
while (<condition>) {
    <statement(s)> ;
}
```

- Beispiel:

```
int number = 1;
while (number <= 200) {
    System.out.print(number + " ");
    number *= 2;
}
```

Ausgabe:

```
1 2 4 8 16 32 64 128
```

5.2 Sentinel-Werte

Sentinel-Wert: Ein spezieller (Eingabe-)Wert, der das Ende einer Folge von Daten(-eingaben) signalisiert.

Sentinel-Schleife: Eine Schleife, die Wiederholungen bis zum Eintreffen des Sentinel-Werts durchführt.

- Häufig Zaunpfahlproblem-Struktur

5.3 Kurzschluss-Auswertung

Kurzschluss-Auswertung (short-circuited evaluation): Die Eigenschaft der Operatoren `&&` und `||`, die verhindert, dass der zweite Operand ausgewertet wird, wenn bereits nach Auswertung des ersten das Ergebnis feststeht.

5.4 do/while-Schleife

`do/while`-Schleife: Führt Anweisungen wiederholt aus, bis eine am Ende des Anweisungsblocks getestete Bedingung falsch ist.

- Unterschied zur `while`-Schleife: Der Rumpf wird unabhängig von der Test-Bedingung mindestens einmal ausgeführt.
- Syntax:

```
do {
    <statement(s)> ;
} while (<condition>);
```

- Beispiel:

```
// weiter würfeln, so lange eine 3 kommt
Random rand = new Random();
int dice;
do {
    dice = rand.nextInt(6)+1;
} while (dice == 3);
```

5.5 break-Schlüsselwort

`break`-Anweisung: Beendet eine Schleife unmittelbar.

5.6 Benutzereingaben prüfen

Methode	Beschreibung
`hasNext()`	Prüft, ob der nächste Token als `String` gelesen werden kann *(immer wahr für Konsoleneingabe)*
`hasNextInt()`	Prüft, ob der nächste Token als `int` gelesen werden kann
`hasNextDouble()`	Prüft, ob der nächste Token als `double` gelesen werden kann
`hasNextLine()`	Prüft, ob die nächste Zeile als `String` gelesen werden kann *(immer wahr für Konsoleneingabe)*

- Bei nicht erwartete Werten, diese Verbrauchen notwendig, um die nächste Eingabe zu benutzen
- Bei Unterscheidung, ob int oder double, zuerst int abfragen und dann double, weil int-Eingaben auch als double gewertet werden

5.7 Lesenotizen

5.7.1 Präzedenzregeln

Operator	Rang	Typ	Beschreibung
`++, --`	1	Arithmetisch	Inkrement / Dekrement
`+, -`	1	Arithmetisch	Unäres Plus und Minus
`!`	1	`boolean`	Negation
`(Typ)`	1	Jeder	Typumwandlung
`*, /, %`	2	Arithmetisch	Multiplikative Op.
`+, -`	3	Arithmetisch	Additive Op.
`+`	3	`String`	String-Konkatenation
`<, <=, >, >=`	5	Arithmetisch	Numerische Vergleiche
`==, !=`	6	Primitiv	Gleich-/Ungleichheit von Werten
`==, !=`	6	Objekt	Gleich-/Ungleichheit von Referenzen
`^`	8	`boolean`	Logisches exkl. Oder
`&&`	10	`boolean`	Logisches Und
`||`	11	`boolean`	Logisches Oder
`=`	13	Jeder	Zuweisung
`*=, /=, %=, +=, -=`	14	Jeder	Zuweisung mit Operation

5.7.2 Continue

- Geht zurück zum Schleifenkopf

6 Dateien und Exceptions

6.1 nextLine

- Konsumiert \n aber benutzt es nicht

```
23    3.14 John Smith    "Hello world"
            45.2       19
```

```
input.nextLine()
23\t3.14 John Smith\t"Hello world"\n\t\t45.2   19\n
                                    ^
```

6.2 Token-basierte Verarbeitung eines String

```
Scanner <name> = new Scanner(<String>);
```

6.3 Abwechselndes Token-basiertes und zeilenbasiertes Einlesen

- Am besten vermeiden und eine Typumwandlung durchführen oder ähnliches

- Stromsicht: `12\nMarty Stepp`
- Nach `nextInt()`: `12\nMarty Stepp`
  `^`
- Nach `nextLine()`: `12\nMarty Stepp`
  `^`

6.4 try-catch

- Besser: Auf die Exception reagieren.
- Dazu gibt es das `try/catch`-Statement
- Syntax:

```
try {
  <statement(s) >;
} catch (<exception-type> <name>) {
  <statement(s)>;
}
```

Potentiell fehleranfällige Anweisungen

Fehlerbehebungs-Code

6.5 Wiederkehrende Eingaben ermöglichen

```
Scanner input= null;
Scanner console= new Scanner(System.in);
do {
    System.out.print("Dateiname: ");
    String name= console.nextLine();
    try {
        input = new Scanner(new File(name));
    } catch (FileNotFoundException e) {
        System.out.println("Datei nicht gefunden. Nochmal.");
    }
} while (input == null);
```

6.6 Ausgabe in Dateien

`PrintStream`: Klasse im package java.io für die Ausgabe auf Console und/oder in Dateien.

- Alle Methoden, die wir für `System.out` verwendet haben (`print`, `println`) funktionieren auf jedem `PrintStream`.

- Syntax der Ausgabe in eine Datei:

```
PrintStream <name> =
        new PrintStream(new File("<file name>"));
...
```

  - Wenn die Datei nicht existiert, wird sie angelegt.
  - Wenn die Datei bereits existiert, wird sie überschrieben.
  - Wie beim Anlegen von `Scanner`-Objekten für Dateien kann auch hier eine `FileNotFoundException` erzeugt werden (bspw. weil Sie keine Zugriffsrechte besitzen, weil die Datei schon von einem anderen Prozess geöffnet ist, ...)

- Keine Datei gleichzeitig als Scanner und PrintStream-Objekt benutzten

6.7 Aufräumarbeiten mit finally

- In einer while-Schleife erzeugte Scanner innerhalb der Schleife wieder schließen, weil mehrere Scanner erzeugt werden. Ein close nach der Schleife ist nicht gut
- Man kann auch mehrere catch-Ausdrücke machen

```
public static void main(String[] args) {
    Scanner input= null;
    Scanner erloeseScanner= null;
    try {
        Scanner console= new Scanner(System.in);
        input= getInput(console);
        while (input.hasNextLine()) {
            String produkt= input.nextLine();
            String erloese= input.nextLine();
            System.out.println(produkt+ ": ");
            erloeseScanner= new Scanner(erloese);
            verarbeite(erloeseScanner);
            erloeseScanner.close();
        }
    } catch (Exception e) {
        System.out.println("Sonstiger Fehler");
    } finally {
        if (input != null)          input.close();
        if (erloeseScanner != null) erloeseScanner.close();
    }
}
```

Deklaration und Initialisierung vorab, sonst ist im `finally`-Block kein `input` und kein `erloeseScanner` bekannt.

Der `finally`-Block wird sowohl im Gut-Fall (`try`-Block fehlerfrei) als auch im Fehlerfall „Sonstiger Fehler" ausgeführt.

6.8 Lesenotizen

6.8.1 File-Objekte

- Stellt nur Informationen zu einer Datei zur Verfügung

```
File f = new File("example.txt");
```

Methode	Beschreibung
`canRead()`	Prüft, ob Datei gelesen werden kann
`delete()`	Löscht Datei
`exists()`	Prüft, ob Datei auf dem Datenträger existiert
`getAbsolutePath()`	Gibt den Pfad im Dateisystem zurück (z. B. `"/home/stud/user/datei.txt"`)
`getName()`	Gibt den Dateinamen zurück
`isDirectory()`	Prüft, ob es sich um ein Verzeichnis handelt
`isFile()`	Prüft, ob es sich um eine Datei handelt
`length()`	Liefert die Größe der Datei in Bytes
`mkdirs()`	Erzeugt das repräsentierte Verzeichnis, falls nicht schon vorhanden.
`renameTo(`*file*`)`	Benennt die Datei um in *file*

6.8.2 Scanner und Dateien

Beispiel:

```
File f = new File("numbers.txt");
Scanner input = new Scanner(f);
```

oder:

```
Scanner input = new Scanner(new File("numbers.txt"));
```

6.8.3 Ausnahmebehandlung

- Exceptions/Ausnahmen ein Objekt, das einen Laufzeitfehler anzeigt

Überprüfungsbedürftige Ausnahmen (checked exceptions)

- Eine Ausnahme, deren Prüfung programmiert werden muss entweder catch oder throws
- Wie FileNotFoundException

Nicht überprüfungsbedürftige Ausnahmen (unchecked exceptions)

- Eine Ausnahme, deren Prüfung nicht programmiert werden muss

6.8.4 Throws

- Ignoriert mögliche Fehler

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

6.8.5 Dateien schließen

- Um Speicherverbrauch und den nicht Zugang zur Datei zu vermeiden mit `scanner.close();`

6.8.6 Berücksichtigung der Locale

```
input.useLocale(new Locale("en", "US"));
```

6.8.7 Ausgabe anhängen

- True bedeutet anhängen

```
PrintStream output=
  new PrintStream(new FileOutputStream(new File("output.txt"),
                                       true));
```

6.8.8 Einlesen unerwartete Eingabe

- Es entsteht NoSuchElementException wenn z.B. bei nextDouble kein double kommt

7 Arrays

7.1 Syntax

- Deklarieren `<type> [] <name> = new <type> [ <length> ];`
- Schreiben `<array name> [ <index> ] = <value> ;`
- Lesen `<array name> [ <index> ]`

- Die Elemente werden automatisch initialisiert bei int = 0, String = null usw.
- Index [-1] gibt es nicht

7.2 length

- Keine Klammern: Attribut `<array name> .length`

7.3 Initialisierung

- Nur beim Deklarieren möglich

```
<type> [] <name> = {<value>, <value>, ..., <value>};
```

7.4 Array zu String

```
Arrays.toString(a)
```

7.5 Rückgabewerte & Parameter

- Rückgabewert `public static int[] readAllIQs(Scanner console){`
- Eingabeparameter `public static int maximumIQ(int[] array) {`
- Ausgabeparameter: Objekt als Parameter, dessen Inhalt verändert wird. Returnen nicht notwendig wegen Zeigersemantik

7.6 Klasse Arrays

Methode	Beschreibung
`binarySearch(`*array, wert*`)`	Liefert den Index von "wert" im sortierten "array" (< 0 wenn nicht gefunden)
`equals(`*array1, array2*`)`	liefert `true`, wenn die beiden Arrays die gleichen Elemente in der gleichen Reihenfolge enthalten
`fill(`*array, wert*`)`	Setzt jedes Element im Array auf den gegebenen Wert.
`sort(`*array*`)`	Sortiert die Elemente innerhalb des Array in aufsteigender Reihenfolge.
`toString(`*array*`)`	Liefert eine Zeichenkette für die Ausgabe, z. B. `"[10, 30, 17]"`

7.7 Kommandozeilenargumente

```
public static void main(String[] args) {
    int faktor1= Integer.parseInt(args[0]);
    int faktor2= Integer.parseInt(args[1]);
    System.out.println(faktor1*faktor2);
}
```

```
public static void main(String[] args) {
    Scanner argumente= new Scanner(args[0] + " " + args[1]);
    double faktor1= argumente.nextDouble();
    double faktor2= argumente.nextDouble();
    argumente.close();
    System.out.println(faktor1*faktor2);
}
```

```
public static void main(String[] args) {
    if (args.length != 2) {
        System.out.println("Bitte zwei Argumente angeben!");
        System.exit(1);
    }
```

7.8 Mehrdimensionale Arrays

- Deklaration

<type> `[][]` ***<name>*** `= new` ***<type>*** `[` ***<length>*** `][` ***<length>*** `];`

- Zugriff

- `temperaturen[2]` bezeichnet die dritte Zeile
- `temperaturen[2][0]` bezeichnet die 1. Spalte dieser Zeile

- Iterieren

```
public static void print(double[][] grid) {
    for (int i=0; i<grid.length; i++)  {
        for (int j=0; j<grid[i].length; j++) {
            System.out.print(grid[i][j] + " ");
        }
        System.out.println();
    }
}
```

- Alternative Ausgabe

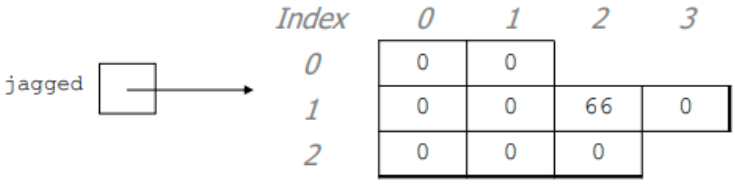
`Arrays.deepToString(temperaturen)` liefert:

```
[[0.0, 0.0, 0.0, 23.5, 0.0], [0.0, 0.0, 0.0, 0.0,
0.0], [19.0, 0.0, 0.0, 0.0, 0.0]]
```

7.9 Jagged Array

Jagged Array: Ein Array, dessen Elemente ***ungleich große*** Arrays sind.

```
int[][] jagged= new int[3][];
jagged[0]= new int[2];
jagged[1]= new int[4];
jagged[2]= new int[3];
jagged[1][2]= 66;
jagged[0][2]= 7; // ArrayIndexOutOfBoundsException
```



7.10 Lesenotizen

7.10.1 NullPointerException

- Nicht aus Methoden bei null-Werten zugreifen
- Deshalb gut zu checken, ob Element initialisiert

```
for (int i = 0; i < words.length; i++) {
    if (words[i] != null) {
        totalLetters += words[i].length();
    }
}
```

7.10.2 Zweiphasen-Initialisierung von Objekt-Arrays

```
Point[] coords = new Point[3];              // phase 1
for (int i = 0; i < coords.length; i++) {
    coords[i] = new Point(0, 0);            // phase 2
}
```

7.10.3 String-Methoden mit Arrays

Methode	**Beschreibung**	**Beispiel** `String s = "long book";`
`toCharArray()`	Separiert den String in ein Array von einzelnen Zeichen	`s.toCharArray()` liefert `{'l', 'o', 'n', 'g', ' ', 'b', 'o', 'o', 'k'}`

`split(`*begrenzer*`)`	Separiert den String anhand des gegebenen Begrenzers in ein Array von Teilstrings	`s.split(" ")` liefert
		`{"long", "book"}`
		`s.split("o")` liefert
		`{"l", "ng b", "", "k"}`

Methode	**Beschreibung**	**Beispiel** `String[] arr = {"a","b","c"};`
`String.join(`*begrenzer*`,` *array*`)`	Setzt die Elemente des Arrays zu einem String zusammen	`String.join("-", arr)` liefert `"a-b-c"`

8 Collections

- Beim printen wird nicht Zeigeradresse angezeigt, nur bei Arrays

8.1 for-each-Schleife

- Dient nur zur Iteration. Nicht geeignet für Veränderungen
- Geht mit Arrays und Collections

```
for (<type> <name> : <collection>) {
    <statement(s)>;
}
```

```
for (String s : list) {
// lesen als: "für jeden String s in list …"
    sum += s.length();
}
```

8.2 Wrapper-Klassen

- Boxing und Unboxing

Primitiver Typ	Wrapper-Klasse
int	Integer
double	Double
char	Character
boolean	Boolean

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

8.3 Klasse Collections

Methode der Klasse `Collections`	Beschreibung
`addAll(`*list, value1, value2, ...*`)`	Fügt mehrere Werte in eine Liste ein
`binarySearch(`*list, value*`)`	Sucht in sortierter Liste nach einem Wert und liefert den Index
`copy(`*dest, source*`)`	Kopiert alle Elemente von einer Liste in eine andere
`fill(`*list, value*`)`	Ersetzt alle Werte durch den gegebenen Wert
`max(`*list*`)`	Liefert den größten Wert in der Liste
`min(`*list*`)`	Liefert den kleinsten Wert in der Liste
`replaceAll(`*list, oldValue, newValue*`)`	Ersetzt alle Vorkommen von *oldValue* durch *newValue*
`reverse(`*list*`)`	Dreht die Reihenfolge der Elemente um
`rotate(`*list, distance*`)`	Verschiebt alle Elemente um die gegebene Anzahl von Indexpositionen
`sort(`*list*`)`	Sortiert die Elemente in der natürlichen Sortierung
`swap(`*list, index1, index2*`)`	Vertauscht die Elemente an den gegebenen Positionen

8.4 Iteratoren

- Collection strukturell nicht verändern, sonst Compilerfehler

Iterator: Objekt zur Repräsentation einer Position in einer Collection

```
Iterator<String> itr = stones.iterator();
while ( itr.hasNext() ) {
  String elem= itr.next();
  System.out.println(elem);
}
```

8.5 Lesenotizen

8.5.1 Arraylist

- Generische (parametrisierte) Klasse: benötigt einen Typ als Parameter

```
ArrayList<String> words = new ArrayList<String>();
words.add("Hallo Welt");
```

Methode	Beschreibung
`add(`*value*`)`	Fügt den gegebenen Wert am Ende der Liste an
`add(`*index*`, `*value*`)`	Fügt den gegebenen Wert in der Liste vor dem gegebenen Index ein
`clear()`	Entfernt alle Elemente
`contains(`*value*`)`	liefert `true`, wenn das Element in der Liste ist
`get(`*index*`)`	Liefert den Wert an der gegebenen Indexposition
`indexOf(`*value*`)`	Liefert den kleinsten Index, an dem der gegebene Wert in der Liste vorkommt (oder -1, wenn nicht gefunden)
`lastIndexOf(`*value*`)`	Liefert den größten Index, an dem der gegebene Wert in der Liste vorkommt (oder -1, wenn nicht gefunden)
`remove(`*index*`)`	Entfernt das Element an der gegebenen Indexposition und liefert es zurück. Nachfolgende Elemente rücken auf.
`set(`*index*`, `*value*`)`	Ersetzt das Element an der gegebenen Indexposition
`size()`	Liefert die aktuelle Anzahl der Elemente in der Liste

8.5.2 Collections

	Listen	
X	`ArrayList`	Größenveränderbares Array
	`LinkedList`	Verkettete Liste
	Mengen (keine Duplikate)	
X	`HashSet`	Schnelle Implementierung einer Menge von Objekten
X	`TreeSet`	Sortierte Menge von Objekten (i. d. R. nur unwesentlich langsamer)
	Maps / Abbildungen (Schlüssel/Wert-Paare)	
X	`HashMap`	Schnelle Implementierung einer Abbildung
X	`TreeMap`	Sortierte Abbildung (i. d. R. nur unwesentlich langsamer)

Methode	Beschreibung
`add(`*value*`)`	Wert hinzufügen
`addAll(`*collection*`)`	Alle Elemente einer als Parameter gegebenen Collection zu dieser Collection hinzufügen
`remove(`*value*`)`	Entfernt den Wert (nur das erste Vorkommen) aus der Collection
`clear()`	Alle Elemente entfernen
`contains(`*value*`)`	liefert `true`, wenn der gegebene Wert enthalten ist
`containsAll(`*collection*`)`	`true`, wenn diese Collection alle Elemente der als Parameter gegebenen Collection enthält
`isEmpty()`	`true`, wenn diese Collection keine Elemente enthält
`removeAll(`*collection*`)`	Entfernt aus dieser Collection alle Werte der als Parameter gegebenen Collection.
`retainAll(`*collection*`)`	Entfernt aus dieser Collection alle Werte, die nicht in der als Parameter gegebenen Collection enthalten sind.
`size()`	Liefert die Anzahl der Elemente
`toArray()`	Liefert ein Array der Elemente
`iterator()`	Liefert ein besonderes Objekt für den Durchlauf durch alle Elemente der Collection.

8.5.3 Maps

```
HashMap<String, String> phoneMap = new HashMap<String, String>()
```

Methode	Beschreibung
`clear()`	Entfernt alle Schlüssel und Werte
`containsKey(`*key*`)`	liefert `true`, wenn der gegebene Schlüssel in der Map existiert
`containsValue(`*value*`)`	liefert `true`, wenn der gegebene Wert in der Map existiert
`get(`*key*`)`	Liefert den Wert, der zum gegebenen Schlüssel gehört (`null`, falls nicht gefunden)
`isEmpty()`	liefert `true`, wenn die Map keine Schlüssel oder Werte enthält
`keySet()`	Liefert eine Menge aller Schlüssel
`put(`*key*`, `*value*`)`	Ordnet dem gegebenen Schlüssel den gegebenen Wert zu
`putAll(`*map*`)`	Fügt alle Schlüssel-Wert-Paare aus der gegebenen Map in diese Map ein
`remove(`*key*`)`	Löscht den gegebenen Schlüssel und den zugehörigen Wert
`size()`	Liefert die Anzahl der Schlüssel-Wert-Paare in der Map
`values()`	Liefert eine Collection aller Werte

- Iterieren

```
for (String key : areaMap.keySet()) {
  System.out.println(key + " => " + areaMap.get(key));
}
```

9 Rekursion

10 Vermischtes

10.1 enum

- Aufzählungstyp

  - **Beispiel:**

```
public enum Spielkarte {
    KARO,
    HERZ,
    PIK,
    KREUZ  //kann optional mit einem ; enden
}
```

  - Die Bezeichner `KARO`, `HERZ`, `PIK` und `KREUZ` können nun wie Konstante verwendet werden (z. B. auch in `switch`-Statements):

```
public static int wert(Spielkarte karte) {
    switch(karte) {
    case KARO:           return 9;
    case HERZ:           return 10;
    case PIK:            return 11;
    default /* KREUZ */: return 12;
    }
}
…
System.out.println("Herz zählt " + wert(Spielkarte.HERZ));
```

10.1.1 Hilfsmethoden

- `name()`: liefert den deklarierten Attributnamen als String
  `String name = Spielkarte.HERZ.name(); // "HERZ"`
- `ordinal()`: liefert die Position einer Enum innerhalb der Deklaration.
  `int idx = Spielkarte.HERZ.ordinal(); // 1`
- `valueOf(String)`: Liefert Enum-Objekt zum Attributnamen.
  `Spielkarte k= Spielkarte.valueOf("HERZ"); // Spielkarte.HERZ`
- `values()`: Liefert ein Array aller Enum-Objekte.
  `for (Spielkarte k : Spielkarte.values()) …`

Anwendung:

```
public static int wert(Spielkarte karte) {
    return karte.ordinal()+9;
}
…
for (Spielkarte karte : Spielkarte.values()) {
    System.out.println(karte.name() + " zählt " + wert(karte));
}
```

Ausgabe:

```
KARO zählt 9
HERZ zählt 10
PIK zählt 11
KREUZ zählt 12
```

10.2 Zeichenketten

- Umwandeln: `int` ↔ Zeichenkette
  - `String.valueOf` und `Integer.toString` wandeln `int` in `String` um
  - `Integer.parseInt` wandelt `String` in `int` um.
    Unterschied zu Scanner: nicht Locale-sensibel
- Zeichenketten manipulieren
  - `StringBuilder` ist eine manipulierbare Zeichenkette
  - Beispiel:

```
StringBuilder sb = new StringBuilder(str);
for (int i=0; i < sb.length(); i++) {
    if (sb.charAt(i) == 'o') {
        sb.setCharAt(i,'e');
    }
}
```

- Zeichenketten auftrennen: `Scanner` oder `String.split`
  - Beispiel: Dateiname Aufgabe2.java isolieren:

```
String pfad= "/home/meier/pr1/ueb1/02/Aufgabe2.java";
String[] arr= pfad.split("/");
System.out.println(arr[arr.length-1]);
```

10.3 Ausgabeformatierung

`println` bietet kaum Möglichkeiten, die Formatierung der Ausgabe gezielt zu beeinflussen.
Es gibt eine bequeme und flexible Möglichkeit, elementare Datentypen formatiert auszugeben: `java.util.Formatter`.

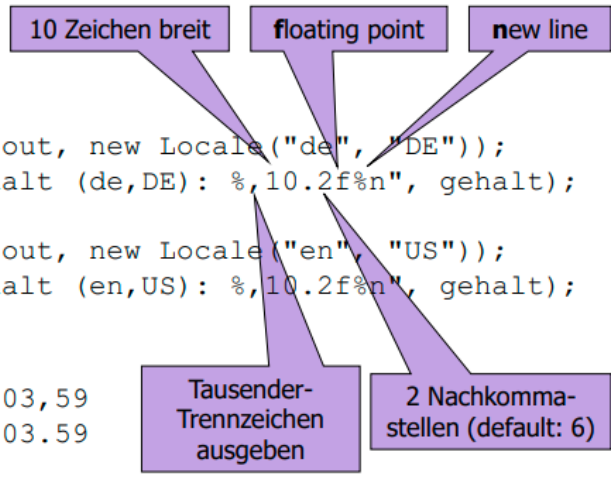
- Kann alle primitiven Datentypen, aber auch Datums-/Zeitwerte in vielfältiger Weise formatiert ausgeben.

Ein `Formatter`-Objekt arbeitet (wie ein `Scanner`) Locale-spezifisch.
Beispiel:

```
double gehalt= 1203.59;
Formatter formatter;

formatter = new Formatter(System.out, new Locale("de", "DE"));
formatter.format("Monatliches Gehalt (de,DE): %,10.2f%n", gehalt);

formatter = new Formatter(System.out, new Locale("en", "US"));
formatter.format("Monatliches Gehalt (en,US): %,10.2f%n", gehalt);
```



Ausgabe:

```
Monatliches Gehalt (de,DE):   1.203,59
Monatliches Gehalt (en,US):   1,203.59
```

Ausgaben können in verschiedene Ziele formatiert geschrieben werden:

- `System.out`
- In jeden `PrintStream` (und damit auch Dateien)
- In einen `StringBuilder`

Beispiel:

```
StringBuilder sb = new StringBuilder();
Formatter formatter= new Formatter(sb, new Locale("de", "DE"));
formatter.format("%,10.2f%n", 1203.59);
String s= sb.toString();
// s enthält nun die Zeichenkette "1.203,59\n"
```

Der erste Parameter der `format`-Methode ist ein Formatstring mit *Formatspezifizierern*.

Allgemeine Syntax:

```
%[argument_idx$][flags][width][.precision]conversion
```

- [] bedeutet: optional
  - Beispiel:
    ```
    formatter.format("%f", 5.7);
    ```
  - Erstellt (z. B. für deutsche Locale) die formatierte Zeichenfolge: `"5,700000"`
- Argument-Index:
  - "`argument idx$`" gibt an, auf welchen Parameter sich die Formatangabe beziehen soll.
  - Insbesondere nützlich, wenn ein Parameter zweimal ausgegeben werden soll
  - "`1$`" steht dabei für das erste Argument nach dem Formatstring, "`2$`" für das zweite usw.
  - Fehlt diese Angabe, werden die Argumente der Reihe nach zugeordnet.
  - Beispiel:
    ```
    int zahl= 5;
    formatter.format("%1$d * %1$d = %2$d%n", zahl, zahl*zahl);
    ```
  - Erstellt die formatierte Zeichenfolge: `"5 * 5 = 25"`

Angabe weiterer Ausgabeoptionen

-	Linksbündige Ausgabe
+	Vorzeichen immer ausgegeben
0	Zahlen werden mit Nullen aufgefüllt
,	Zahlen werden mit Tausenderpunkten ausgegeben
(	Negative Zahlen werden in Klammern eingeschlossen

Beispiel:

```
formatter.format("%08.2f%n", 5.7);
```

Erstellt (z. B. für deutsche Locale) die formatierte Zeichenfolge:

```
"00005,70"
```

10.b

```
Locale enUS= new Locale("en", "US");
Locale deDE= new Locale("de", "DE");
// Ausgabe:
PrintStream output= new PrintStream(new File("Kurse.en.csv"));
Formatter formatter= new Formatter(output, enUS);
// Eingabe:
Scanner input= new Scanner(new File("Kurse.csv"));

// Erste Zeile:
String line= input.nextLine();
formatter.format("%s%n",line.replaceAll(";",","));

while (input.hasNextLine()) {
    line= input.nextLine();
    String[] arr= line.split(";");
    formatter.format("%s",arr[0]); // Datum
    for (int i=1; i<=2; i++) {
        Scanner kursScan= new Scanner(arr[i]);
        kursScan.useLocale(deDE);
        formatter.format("%s%.2f", ",", kursScan.nextDouble());
        kursScan.close();
    }
    formatter.format("%n");
}
input.close();
formatter.close();
```

Erste Lösung

Ergebnisdatei:

```
Datum,Ankaufskurs,Verkaufskurs
31.03.2007,55.34,58.11
01.04.2007,54.66,57.39
02.04.2007,54.16,56.87
03.04.2007,53.44,56.11
04.04.2007,55.66,58.44
05.04.2007,56.90,59.75
06.04.2007,60.07,63.07
07.04.2007,59.99,62.99
```

10.3.1 Hilfsfunktionen für die Formatierung

Da es etwas umständlich ist, immer ein `Formatter`-Objekt anzulegen, gibt es folgende Hilfsfunktionen:

- In der Klasse `String`:
  - Beispiel:

```
String ergebnis=
   String.format(new Locale("de", "DE"), "%08.2f", 5.7);
System.out.println(ergebnis);
```

  - Ausgabe:

```
00005,70
```

- In der Klasse `PrintStream`:
  - Beispiel:

```
System.out.format(new Locale("de", "DE"), "%08.2f%n",5.7);
```

  - Ausgabe:

```
00005,70
```

- Große Zahlen kann man so schreiben: 1_000_000_000

Komplettes Programm

10.b

```
public class Kurse2 {
  public static void main(String[] args) throws FileNotFoundException{
    Locale enUS= new Locale("en", "US");
    Locale deDE= new Locale("de", "DE");
    PrintStream output= new PrintStream(new File("Kurse.en.csv"));
    Scanner input= new Scanner(new File("Kurse.csv"));
    String line= input.nextLine();  // Erste Zeile überlesen
    output.println("date,call price,offer price");

    while (input.hasNextLine()) {
      line= input.nextLine();
      Scanner lineScan= new Scanner(line); // Zeile m. Tokenscanner abarbeiten
      lineScan.useLocale(deDE).useDelimiter(";");
      output.print(convertGermanDateToUS(lineScan.next()));
      for (int i=1; i<=2; i++) {
        output.format(enUS, "%s%.2f", ",", lineScan.nextDouble());
      }
      output.println();
      lineScan.close();
    }
    input.close();
    output.close();
  }

  public static String convertGermanDateToUS(String date){
    String[] arr= date.split("\\.");
    String us= arr[1] + "/" + arr[0] + "/" + arr[2];
    return us;
  }
}
```

Ergebnisdatei:

```
date,call price,offer price
03/31/2007,55.34,58.11
04/01/2007,54.66,57.39
04/02/2007,54.16,56.87
04/03/2007,53.44,56.11
04/04/2007,55.66,58.44
04/05/2007,56.90,59.75
04/06/2007,60.07,63.07
04/07/2007,59.99,62.99
```


10.4 Variable Parameterlisten

Bei Verwendung des Formatters können wir beliebig viele Parameter angeben.

Beispiel:

```
Formatter formatter= new Formatter(System.out);
int a= 7, b= 6, c= 9;
formatter.format("%d %d %d%n", a, b, c);
formatter.format("%d%n", a);
```

10.4.1 Variable Parameterlisten in eigenen Methoden

Einsatz in einer eigenen Methode:

```
public static void printBegruessung(String ... name){
    for (int i=0; i<name.length; i++) {
        System.out.println("Hallo " + name[i]);
    }
}
```

Aufruf:

```
printBegruessung("Dirk", "Jenny", "Lutz");
```

Ausgabe:

```
Hallo Dirk
Hallo Jenny
Hallo Lutz
```

11 Geheimnisse

11.1 Geheimnisprinzip

Geheimnisprinzip: Verbergen von Daten oder Informationen vor dem Zugriff von außen.

- Lösung: Sie geben den "Obertyp" `Iterable` als Rückgabetyp zurück.

```
public static Iterable<String> primaten() {
    HashSet<String> set= new HashSet<String>();
    // ...
    return set;
}
```

- Einen Obertyp kann man gut vergleichen mit einem Oberbegriff:
  - "Iterable" ist ein Oberbegriff von "HashSet" und "ArrayList"

Abstrakter Datentyp (ADT): eine allgemeine Spezifikation eines Typs oder einer Datenstruktur

- Spezifiziert die enthaltenen Daten
- Spezifiziert die Operationen
- Spezifiziert *nicht*, wie die Daten intern strukturiert sind.
- Spezifiziert *nicht*, wie die Operationen implementiert werden.

- Beispiel ADT: **Iterable**
  - Der Iterable ADT spezifiziert, dass Elemente mit einem Iterator durchlaufen werden können.
  - Der Iterable ADT spezifiziert die unterstützte Operation `iterator`
  - `ArrayList`, `HashSet` und `TreeSet` implementieren alle die Daten und Operationen des Iterable ADT.