Kapitel 4
Arrays

4

Arrays sind wichtige Datenstrukturen, die auch indirekt in Java auftauchen, etwa bei der erweiterten `for`-Schleife oder variablen Argumentlisten. Dieses Kapitel enthält Aufgaben zum Aufbau von Arrays, zum Ablaufen von Arrays und Fragen nach Algorithmen, etwa zur Suche von Elementen in einem Array.

Voraussetzungen

- Arrays anlegen, abfragen, füllen können
- Arrays mit erweiterter `for`-Schleife ablaufen können
- ein- und mehrdimensionale Arrays nutzen können
- variable Argumentlisten aufbauen können
- Utility-Methoden der Klasse `Arrays` kennen

Java ist auch eine Insel, Kapitel 4

In der »Insel« erklärt Kapitel 4, »Arrays und ihre Anwendungen«, die nötigen Voraussetzungen.

Verwendete Datentypen in diesem Kapitel:

- `java.util.Arrays` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html*)
- `java.lang.System` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html*)

4.1 Alles hat einen Typ

Bevor wir uns den Zugriff auf die Elemente anschauen, wollen wir uns mit den Typen näher beschäftigen. Es ist wichtig, den Unterschied zwischen Objekttyp und Referenztyp zu kennen.

4.1.1 Quiz: Array-Typen ★

Arrays sind in Java *kovariant*, das bedeutet zum Beispiel, dass `String[]` ein Untertyp von `Object[]` ist. Das klingt ein bisschen akademisch, ist es vermutlich auch, daher soll die folgende Aufgabe das Verständnis für die Kovarianz von Arrays schärfen.

Überlege, ob alle Anweisungen compilieren bzw. zur Laufzeit funktionieren:

```
/* 1 */ String[] strings1 = new String[ 100 ];
/* 2 */ Object[] a1 = (String[]) strings1;
/* 3 */ Object[] a2 = strings1;
/* 4 */ Object[] strings2 = new String[]{ "1", "2", "3" };
/* 5 */ String[] a3 = (String[]) strings2;
/* 6 */ String[] strings3 = { "1", "2", "3" };
/* 7 */ Object[] a4 = strings3;
/* 8 */ Object[] strings4 = { "1", "2", "3" };
/* 9 */ String[] a5 = (String[]) strings4;

/* A */ int[] ints1 = new int[ 100 ];
/* B */ Object[] a6 = (int[]) ints1;
/* C */ Object[] ints2 = new int[ 100 ];
/* D */ int[] a7 = (int[]) ints2;
```

4.2 Eindimensionale Arrays

Ein Array ist eine Sammlung gleichartiger Elemente. *Eindimensionale Arrays* enthalten direkt die Elemente und keine weiteren Unter-Arrays.

4.2.1 Arrays ablaufen und Windgeschwindigkeit, Windrichtung ausgeben ★

Captain CiaoCiao segelt über das Meer, der Wind bläst von allen Seiten. Er muss die Windgeschwindigkeit und Windrichtung immer im Blick haben.

Aufgabe:

- Deklariere zwei Arrays `int[] windSpeed` und `int[] windDirection`.
- Initialisiere beide Arrays je mit drei ganzzahligen Zufallszahlen (prinzipiell sollte die Anzahl beliebig sein), wobei die Windstärke zwischen 0 und (kleiner) 200 km/h und die Windrichtung zwischen 0 und (kleiner) 360 Grad liegen kann.
- Laufe mit einer Schleife über das Array, und gib alle Pärchen kommasepariert aus.

Beispiel:

- Enthält z. B. das Array `windSpeed` die Werte `{82, 70, 12}` und das Array `windDirection` die Werte `{12, 266, 92}`, soll die Ausgabe auf dem Bildschirm sein:


```
Wind speed 82 km/h and wind direction 12°, Wind speed 70 km/h and wind direction 266°, Wind speed 12 km/h and wind direction 92°
```

Hinweis: Bedenke, dass die Segmente mit einem Komma getrennt werden und am Ende kein Komma steht.

4.2.2 Konstante Umsatzsteigerung feststellen ★

Am Ende eines Monats bekommt Captain CiaoCiao die Umsätze gemeldet, die er und seine Crew – sagen wir mal – erwirtschaftet haben. In der monatlichen Liste ist vermerkt, wie der Gewinn an einem Tag ausfiel. Sie hat dieses Format:

```
//                Tag    1,    2,   3,    4,    5, ... bis maximal 31
int[] dailyGains  = { 1000, 2000, 500, 9000, 9010 };
```

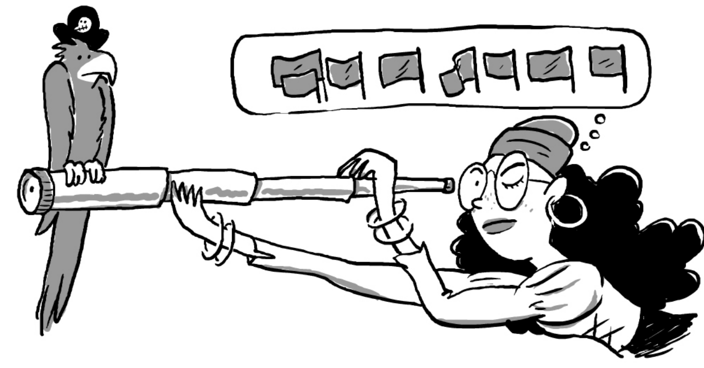
Captain CiaoCiao ist mit den Zahlen zufrieden, und er möchte eine Belohnung zahlen, wenn Gewinne über 5 % gestiegen sind. Von 1.000 auf 2.000 ist ein satter Sprung um 100 %, von 500 auf 9.000 ebenso, doch definitiv nicht von 2.000 auf 500 und auch nicht von 9.000 auf 9.010.

Aufgabe:

- Schreibe eine Methode `int count5PercentJumps(int[])`, die die Anzahl der Umsatzsprünge liefert. Ein Umsatzsprung ist dann gegeben, wenn der Umsatz 5 % über dem des Vortags lag.
- Das übergebene Array darf nicht `null` sein, andernfalls folgt eine Ausnahme.

4.2.3 Aufeinanderfolgende Strings suchen und feststellen, ob Salty Snook kommt ★

Bonny Brain beobachtet die Flaggen der vorbeiziehenden Schiffe, denn sie wartet auf Salty Snook. Sie schaut sich jede Flagge an und weiß, dass Salty Snook nie alleine kommt, sondern sich in einem Konvoi von vier Schiffen bewegt. Die Flaggen selbst kennt sie nicht, nur weiß sie, dass alle die gleiche Aufschrift haben.



Aufgabe:

- Schreibe eine neue Methode `isProbablyApproaching(String[] signs)`, die dann `true` zurückliefert, wenn es im Array vier gleiche Kürzel hintereinander gibt. Bedenke, dass man Strings mit `equals(…)` vergleicht.
- Das übergebene Array darf nicht `null` sein, und kein Element im Array darf `null` sein.

Beispiel:

```
String[] signs1 = { "F", "DO", "MOS", "MOS", "MOS", "MOS", "WES" };
isProbablyApproaching( signs1 );  // true

String[] signs2 = { "F", "DO", "MOS", "MOS", "WES", "MOS", "MOS" };
isProbablyApproaching( signs2 );  // false
```

4.2.4 Array umdrehen ★

Charlie Creevey macht für Captain CiaoCiao die Finanzen. Doch statt die Einnahmen aufsteigend zu sortieren, hat er sie absteigend sortiert. Daher muss die Liste umgedreht werden.

Ein Array *umzudrehen* bedeutet, dass das erste Element mit dem letzten Element vertauscht wird, das zweite mit dem zweitletzten usw.

Aufgabe:

- Schreibe eine neue statische Methode `reverse(…)`, die ein gegebenes Array umdreht:

  ```
  public static void reverse( double[] numbers ) {
    // TODO
  }
  ```

- Die Operation soll *in place* sein, also das übergebene Array ändern. Wir wollen kein neues Array anlegen.
- Die Übergabe `null` führt zu einer Ausnahme.

Beispiel:

- `{ }` → `{ }`
- `{ 1 }` → `{ 1 }`
- `{ 1, 2 }` → `{ 2, 1 }`
- `{ 1, 2, 3 }` → `{ 3, 2, 1 }`

Die Darstellung in den geschweiften Klammern ist nur symbolisch.

4.2.5 Das nächste Kino finden ★★

Die Klasse `java.awt.Point` repräsentiert Punkte mit x/y-Koordinaten. Diese lassen sich gut für Positionen einsetzen.

Im Kino läuft die Neuverfilmung »Unter der Flagge der Freibeuter« an, die Captain CiaoCiao unbedingt sehen muss. Doch wo befindet sich das nächste Kino?

Aufgabe:

- Gegeben ist eine Menge von `Point`-Objekten in einem Array `points` für die Kinopositionen.

  ```
  Point[] points = { new Point(10, 20), new Point(12, 2), new Point(44, 4) };
  ```

- Schreibe eine Methode `double minDistance(Point[] points, int size)`, die den Abstand des Punktes zurückliefert, der die geringste Distanz zum Nullpunkt besitzt. Mit `size` können wir bestimmen, wie viele Elemente des Arrays betrachtet werden sollen, damit das Array auch prinzipiell größer sein kann.
- `null` als Übergabe ist nicht erlaubt, auch dürfen die Punkte nicht `null` sein; es muss eine Ausnahme ausgelöst werden.
- Was müssen wir ändern, wenn der Rückgabetyp `Point` ist, also der Punkt selbst mit dem kleinsten Abstand zurückgegeben werden soll?

> **Tipp**
> Studiere die Javadoc zu `java.awt.Point`, um herauszufinden, ob der Punkt selbst Abstände zu anderen Koordinaten berechnen kann.

4.2.6 Süßigkeitenladen überfallen und fair aufteilen ★★

Captain CiaoCiao überfällt mit seinen Kindern Junior und Jackie einen Süßigkeitenladen. Die Süßigkeiten stehen in einem langen Regal, und jedes Produkt hat ein Gewicht. Die Daten liegen als Array vor:

```
int[] values = { 10, 20, 30, 40, 40, 50 };
```

Junior und Jackie stellen sich links und rechts an entgegengesetzten Enden des Regals auf, und da Captain CiaoCiao beide Kinder gleich lieb hat, sollen sie am Ende auch gleich viel mit nach Hause nehmen. Captain CiaoCiao zeigt im Regal auf eine Süßigkeit, sodass alle Produkte links davon zu Junior gehen und alle Produkte rechts von der Position (inklusive dem gezeigten) für Jackie sind.

Der Captain weiß zwar, was im Regal steht, aber nicht, ab welcher Position links und rechts die gleiche Summe entsteht. Abweichungen von 10 % sind für die Kinder in Ordnung. Für den Unterschied wollen wir auf folgende Formel für die relative Differenz zurückgreifen:

```
private static int relativeDifference( int a, int b ) {
  int absoluteDifference = Math.abs( a - b );
  return (int) (100. * absoluteDifference / Math.max( a, b ));
}
```

Aufgabe:

- Schreibe eine Methode `int findSplitPoint(int[])`, die den Index im Array findet, bei dem links und rechts fair geteilt werden kann. Irgendeine Lösung reicht, es sind nicht alle Lösungen nötig.
- Falls es keine faire Teilung gibt, soll eine Methode `-1` liefern.

Beispiele:

- 10 + 20 + 30 + 40 ≈ 40 + 50, denn 100 ≈ 90, und der Index für die Rückgabe ist 4.
- `10 20 30 40 40 100` führt zu `-1`, denn es gibt keine gültige Partitionierung.

4.3 Erweiterte for-Schleife

Sollen Arrays ab dem ersten Element abgelaufen werden, so lässt sich dafür gut eine erweiterte `for`-Schleife mit unsichtbarem Schleifenzähler nutzen. Das spart Code ein.

4.3.1 Berge zeichnen ★★

Für die nächste Schatzsuche müssen Bonny Brain und die Crew über Berge und Hügel gehen. Sie bekommt vorher die Höhenmeter mitgeteilt und möchte sich vorher einen Eindruck vom Profil machen.

Aufgabe:

- Schreibe ein Programm mit einer Methode `printMountain(int[] altitudes)`, die ein Array mit Höhenmetern in eine ASCII-Darstellung umsetzt.
- Die Höhe soll darstellt werden über ein Multiplikationszeichen `*` in genau dieser Höhe von einer Grundlinie. Die Höhen können beliebig sein, aber nicht negativ.

Beispiel:

Das Array { `0, 1, 1, 2, 2, 3, 3, 3, 4, 5, 4, 3, 2, 2, 1, 0` } soll darstellt werden als:

```
5          *
4         * *
3      ***   *
2    **       **
1  **           *
0 *              *
```

Die erste Spalte dient der Verdeutlichung und muss nicht umgesetzt werden.

Optionale Erweiterung:

Deute statt mit `*` mit den Symbolen `/`, `\`, `-` und `^` an, ob wir aufsteigen, absteigen, auf einem Plateau sind oder an der Spitze stehen.

```
5          ^
4         / \
3      --/   \
2    -/       -\
1  -/           \
0 /              \
```

4.4 Zwei- und mehrdimensionale Arrays

Ein Array kann in Java Verweise auf andere Arrays enthalten, und so definiert man in Java mehrdimensionale Arrays. In Java gibt es keine echten zweidimensionalen Arrays; zweidimensionale Arrays sind nichts anderes als Arrays, die Unter-Arrays referenzieren, und die Unter-Arrays könnten unterschiedlich lang sein.

4.4.1 Mini-Sudoku auf gültige Lösung prüfen ★★

Da Überfälle ziemlich anstrengend sind, braucht Bonny Brain einen Ausgleich und beschäftigt sich mit Sudoku. Ein Sudoku-Spiel besteht aus 81 Feldern in einem 9-×-9-Gitter. Das Gitter lässt sich in neun Blöcke zerlegen, jeder Block ist ein zweidimensionales Array der Größe 3 × 3. In jedem dieser Blöcke muss jede Zahl von 1 bis 9 genau einmal vorkommen – keine darf fehlen.

Aufgabe:

Schreibe ein Programm, das ein zweidimensionales Array mit neun Elementen daraufhin testet, ob alle Zahlen von 1 bis 9 vorkommen.

Beispiel:

- Das folgende Array ist eine gültige Sudoku-Belegung:

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

- Das folgende Array ist keine gültige Sudoku-Belegung:

```
int[][] array = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 8 } };
```

Der Fehler könnte etwa gemeldet werden mit `"missing: 9"`.

4.4.2 Bild vergrößern ★★

Bilder werden im Speicher oft als Tripel von Rot-Grün-Blau-Werten gespeichert, wobei die einzelnen Werte sich zwischen 0 und 255 bewegen können. Da es bei Graustufenbildern keine Farben gibt, ist statt drei Werten nur ein Wert nötig.

Aufgabe:

- Gegeben ist ein zweidimensionales Ganzzahl-Array mit Werten von 0 bis 255; das Array stellt gedanklich ein Graustufenbild dar.
- Schreibe eine Methode `int[][] magnify(int[][] array, int factor)`, die ein neues Array zurückgibt und das Bild um den gegebenen Faktor skaliert. Aus einem Bild der Größe 2 × 3 und dem Faktor 2 wird also ein Bild der Größe 4 × 6. Bildpunkte werden einfach verdoppelt, eine Interpolation der Werte ist nicht gewünscht.

Beispiel:

Nehmen wir folgendes Array an:

```
{ {1, 2, 3},
  {4, 5, 6} }
```

Dann folgt nach einer Verdoppelung:

```
{ {1, 1, 2, 2, 3, 3},
  {1, 1, 2, 2, 3, 3},
  {4, 4, 5, 5, 6, 6},
  {4, 4, 5, 5, 6, 6} }
```

4.5 Variable Argumentlisten

Java erlaubt Methoden, der man eine beliebige Anzahl Argumente übergeben kann. Sie nennen sich *Vararg-Methoden*. Ein Vararg-Parameter darf nur zum Schluss einer Parameterliste stehen und ist ein Array. Beim Aufruf mit variablen Argumenten erzeugt der Compiler automatisch ein neues anonymes Array und übergibt es an die Methode.

4.5.1 SVG-Polygone mit variabler Koordinatenanzahl erzeugen ★

Bonny Brain möchte für ihren nächsten Arbeitsort eine Karte zeichnen, und die soll gedruckt und auf jeder Auflösung immer gut aussehen, weil es auf jedes Detail ankommt. Die beste Technologie dafür ist SVG.

In SVG gibt es verschiedene Primitive, etwa für Linien, Kreise oder Rechtecke. Auch für Linienzüge gibt es ein XML-Element. Ein Beispiel:

```
<polygon points="200,10 250,190 160,210" />
```


Aufgabe:

- Deklariere eine Java-Methode `printSvgPolygon(…)`, der wir beliebig viele Koordinaten-Paare übergeben können. Welche Fehler könnte es bei der Übergabe geben?
- Die Methode soll zu den übergebenen Paaren eine passende SVG-Ausgabe auf dem Bildschirm ausgeben.

Beispiel:

In `printSvgPolygon( 200, 10, 250, 190, 160, 210 )` ist `200, 10` ein Koordinatenpaar, `250, 190` ebenso, genauso wie `160, 210`. Die Bildschirmausgabe soll sein: `<polygon points="200,10 250,190 160,210" />`.

Optional: Studiere das Beispiel unter *https://www.w3schools.com/graphics/tryit.asp?filename=trysvg_polygon*. Kopiere das selbstgenerierte SVG in die Weboberfläche.

4.5.2 Auf Zustimmung prüfen ★

Captain CiaoCiao holt sich von seinen Crewmitgliedern eine Rückmeldung über einen Auftrag. Sämtliche Mitglieder können Ja oder Nein stimmen.

Aufgabe:

- Gesucht ist eine Vararg-Methode `allTrue(…)`, der man eine beliebige Anzahl von `boolean`-Werten übergeben kann, aber mindestens ein Argument übergeben muss.
- Sind alle Argumente `true`, ist auch die Rückgabe `true`; ist einer der `boolean`-Werte `false`, soll die Methode `false` zurückgeben.
- Da ein Vararg intern ein Array darstellt, kann `null` übergeben werden – das muss zu einer Ausnahme führen.

Beispiel:

- `allTrue(true, true, true)` liefert `true`.
- `allTrue(true)` liefert `true`.
- `allTrue(true, false)` liefert `false`.
- `allTrue(true, null)` liefert eine Ausnahme.
- `allTrue()` ergibt einen Compilerfehler.

4.5.3 Hilfe, Tetraphobie! Alle Vieren nach hinten setzen ★★

Bonny Brain trifft befreundete Freibeuter in Hongkong und stellt fest, dass viele an Tetraphobie leiden und abergläubische Angst vor der Zahl 4 haben. Die Buchhalterin muss nun alle Zahlen mit einer 4 nach hinten setzen.

Aufgabe:

- Schreibe eine Methode `fourLast(int... numbers)`, die alle Zahlen, die eine `4` enthalten, hinter die Zahlen stellt, die keine `4` haben. Die Reihenfolge der Zahlen ohne `4` darf sich nicht ändern, die Zahlen mit einer `4` können irgendwo am Ende stehen.
- `fourLast(…)` soll das übergebene Array als Rückgabe haben.
- `null` in der Übergabe muss zu einer Ausnahme führen.

Beispiel:

- `int[] numbers = {1, 44, 2, 4, 43}; fourLast(numbers);` verändert das Array `numbers` so, dass `1` und `2` im Array vor `44`, `4` und `43` stehen. Die `2` darf später nicht vor der `1` stehen.
- `fourLast( 4, 4, 44, 1234 )` liefert das vom Compiler automatisch generierte Array mit den Einträgen zum Beispiel in der Reihenfolge `4`, `4`, `44`, `1234` zurück.

4.6 Die Utility-Klasse Arrays

Arrays »können« in Java nicht viel, interessante Methoden sind in Klassen ausgelagert, etwa `java.util.Arrays`. Eine Methode zum Kopieren von Arrays befindet sich in der Klasse `System`.

4.6.1 Quiz: Arrays kopieren ★

Wofür stehen diese unsinnigen Variablennamen, und was ist der Effekt der folgenden Zeilen?

```
int[] hooey = { 1, 2, 3, 4 };
int[] shuck = new int[ hooey.length - 1 ];
int bushwa = 2;
int kelter = 0;
int piddle = 0;
System.arraycopy( hooey, kelter, shuck, piddle, bushwa );
System.arraycopy( hooey, bushwa + 1, shuck, bushwa,
  hooey.length - bushwa - 1 );
System.out.println( Arrays.toString( shuck ) );
```

4.6.2 Quiz: Arrays vergleichen ★

Wie sind die Ausgaben?

```
Object[] array1 = { "Anne Bonny", "Fortune", "Sir Francis Drake",
  new int[]{ 1, 2, 3 } };
```



```
Object[] array2 = { "Anne Bonny", "Fortune", "Sir Francis Drake",
  new int[]{ 1, 2, 3 } };
System.out.println( array1 == array2 );
System.out.println( array1.equals( array2 ) );
System.out.println( Arrays.equals( array1, array2 ) );
System.out.println( Arrays.deepEquals( array1, array2 ) );
```

4.7 Lösungsvorschläge

Quiz 4.1.1: Array-Typen

Interessant sind weniger die Variablendeklarationen und Zuweisungen von `strings1` bis `strings4` und `int1` und `int2`; die Syntax ist uns bekannt: Ein Array wird entweder mit einer festen Größe oder mit Elementen vorinitialisiert.

Interessant sind die folgenden Typumwandlungen, die explizit oder implizit durchgeführt werden. Wir müssen dabei unterscheiden zwischen Typumwandlungen, die für den Compiler in Ordnung sind, und Typumwandlungen, die erst zur Laufzeit zu Schwierigkeiten führen. Das ist eine wichtige Unterscheidung, die auch in den Begriffen *Objekttyp* und *Referenztyp* deutlich wird. Für den Compiler gibt es Referenzvariablen und einen Referenztyp, der Compiler weiß nicht, was zur Laufzeit vor sich geht. Die Laufzeitumgebung wiederum weiß im Grunde nicht, unter welchem Variablentyp eine Variable deklariert wurde, sie weiß allerdings, was für ein Objekt sie gerade vor sich hat: Wir sprechen daher von dem *Objekttyp*.

Die erste Anweisung ist die ehrlichste. Für den Compiler ist es ein String-Array und für die Laufzeitumgebung ebenfalls.

Die Typumwandlungen in der zweiten Zeile ist irrelevant, weil `String[]` ein Untertyp von `Object[]` ist. Das ist sehr wichtig, denn genau das ist kovariant: **Ein `String`-Array ist ein Untertyp eines `Object`-Arrays, auch ein `Point[]` ist ein besonderes `Object[]`.** Das ist auch in der dritten Zeile ersichtlich, wo die explizite Typumwandlung fehlt, weil sie implizit ist.

In der vierten Anweisung wissen Laufzeitumgebung und Compiler Unterschiedliches. Für die Laufzeitumgebung steht weiterhin ein `String`-Array im Speicher, aber der Compiler kennt das `String`-Array nur als `Object`-Array. Diese Schreibweise ist prinzipiell gültig, und auch hier findet wieder eine implizite Typumwandlung statt.

In der fünften Anweisung werten wir das `Object`-Array zu einem `String`-Array auf. Das funktioniert zur Compilezeit und auch zur Laufzeit.

In der sechsten Anweisung wird wieder direkt ein `String`-Array aufgebaut und als `String`-Array vermerkt. Das ist die übliche Schreibweise. In der siebten Anweisung

finden wir eine implizite Typanpassung vom `String`-Array zum `Object`-Array. Diese Anweisung ist in Ordnung.

Die achte und neunte Anweisung ist heimtückisch: Der Compiler baut bei der Zuweisung kein `String`-Array auf, sondern ein `Object`-Array und legt in dieses `Object`-Array `String`-Referenzen. Es gibt folglich kein `String`-Array, sondern nur ein `Object`-Array, das Strings referenziert. In der neunten Zeile vertraut der Compiler unserer Entscheidung, das `Object`-Array auf ein `String`-Array anzupassen. Der Compiler schluckt das, und es gibt keinen Compilerfehler. Ein Problem tritt aber zur Laufzeit auf. Da die Laufzeitumgebung natürlich weiß, dass in der Variablen `strings4` nur ein `Object`-Array steht und kein besseres `String`-Array, folgt die Ausnahme `java.lang.ClassCastException: class [Ljava.lang.Object; cannot be cast to class [Ljava.lang.String;`.

Die letzten vier Beispiele sind für den Compiler einfacher als falsch zu identifizieren. Während die Deklaration von `ints1` noch korrekt ist, werden B und C und D zu einem Compilerfehler führen. Ein `int`-Array lässt sich nicht in ein `Object`-Array konvertieren, weder explizit wie in Zeile B noch implizit wie in Zeile C. Es gibt hier keine Typanpassung, so wie `Object o = 1` auch falsch ist. Da wir schon Zeile C nicht compilieren können, führt auch Zeile D zu einem Compilerfehler: Es gibt keine Typanpassung von einem `Object`-Array in ein `int`-Array.

Aufgabe 4.2.1: Arrays ablaufen und Windgeschwindigkeit, Windrichtung ausgeben

```
final int MAX_WIND_SPEED = 200;
final int MAX_DEGREE     = 360;

final int LENGTH = 5;
int[] windSpeed     = new int[ LENGTH ];
int[] windDirection = new int[ LENGTH ];

for ( int i = 0; i < LENGTH; i++ ) {
  windSpeed[ i ]     = (int) (Math.random() * MAX_WIND_SPEED);
  windDirection[ i ] = (int) (Math.random() * MAX_DEGREE);
}

for ( int i = 0; i < LENGTH; i++ ) {
  System.out.printf( "Wind speed %d km/h and wind direction %d°",
                     windSpeed[ i ], windDirection[ i ] );
  if ( i != LENGTH - 1 )
    System.out.print( ", " );
}
```

Listing 4.1 com/tutego/exercise/array/Windy.java

Die Lösung besteht aus vier Schritten. Zunächst wollen wir drei Konstanten definieren: für die maximale Windgeschwindigkeit und Windstärke sowie für die Anzahl der Elemente. `LENGTH` initialisieren wir mit `5`, damit wir später beim Aufbau der Arrays `windSpeed` und `windDirection` nicht wieder das Literal `5` als magische Zahl in den Code schreiben müssen; wir können später einfach die Variable ändern, wenn wir größere Arrays wünschen.

Im zweiten Schritt laufen wir mit einer Schleifenvariablen von `0` bis zum letzten Element des Arrays. Das Array ist fünf Elemente groß, also dürfen wir von `0` bis `4` laufen. Im Rumpf der Schleife bilden wir zwei Zufallszahlen und initialisieren die Array-Elemente. Die Berechnung einer ganzzahligen Zufallszahl von `0` bis `200` sieht so aus: Mit `Math.random()` bekommen wir eine Zufallszahl als Fließkommazahl zwischen `0` und echt kleiner als 1. Die Multiplikation mit `200` liefert eine Zufallszahl zwischen `0` und echt kleiner als `200`. Konvertiert `(int)` den Ausdruck in eine Ganzzahl, werden alle Nachkommastellen abgeschnitten, also liegt das Ergebnis als ganze Zahl zwischen `0` und `199` vor. Üblicherweise sind bei Bereichsangaben der Start inklusive und das Ende exklusive.

Die beiden Arrays sind nun initialisiert, und wir können die Paare ausgeben. Mit einer Schleife laufen wir das Array ab; wir greifen an der gleichen Position `i` auf die Arrays `windSpeed` und `windDirection` zurück. Bei der Ausgabe unterstützen uns `printf(…)` und der Formatspezifizierer `%d` für Dezimalzahlen. In den Format-String setzen wir aber kein Komma, denn am Ende der Kette darf kein Komma stehen. Dass nur am Ende ein Komma steht, kann unterschiedlich gelöst werden. Die Herangehensweise hier fragt den Schleifenzähler ab, ob er für das letzte Element steht. Wenn `i` ungleich dem letzten Element ist, dann wird ein Separator gesetzt, andernfalls nicht.

Aufgabe 4.2.2: Konstante Umsatzsteigerung feststellen

```
private static int count5PercentJumps( int[] dailyGains ) {

  if ( dailyGains.length < 2 )
    return 0;

  final double MIN_PERCENT = 5;

  int result = 0;

  // Index variable i starting at 1, second element
  for ( int i = 1; i < dailyGains.length; i++ ) {
    double yesterday = dailyGains[ i - 1 ];
    double today     = dailyGains[ i ];
```

```
    double percent = today / yesterday * 100 - 100;

    if ( percent >= MIN_PERCENT )
      result++;
  }

  return result;
}
```

Listing 4.2 com/tutego/exercise/array/BigProfits.java

Der Methode `count5PercentJumps(…)` wird ein Array übergeben, das im besten Fall eine Reihe von Ganzzahlen enthält. Es kann passieren, dass `null` übergeben wird, was keine gültige Eingabe für das Programm sein soll. Greifen wir auf `length` zurück, gibt es im Fall von `null` eine `NullPointerException` – das ist so gewollt.

Existiert das Array-Objekt, aber enthält es kein oder nur ein Element, betrachten wir das als Fehler und geben `0` zurück.

Kommen wir weiter, wissen wir, dass mindestens zwei Elemente im Array stehen. Wir laufen mit einer `for`-Schleife über das Array, wobei wir mit dem Index `i` bei `1` beginnen und immer zwei Elemente gleichzeitig erfragen: das aktuelle Element an der Position `i` und das Element an der Position vorher, an der Position `i - 1`. Diese Elemente stehen für `today` und `yesterday`. Prinzipiell hätten wir auch bei `0` beginnen und bis `< dailyGains.length - 1` laufen können.

Haben wir den Betrag für den heutigen und gestrigen Tag ausgelesen, müssen wir die relative prozentuale Steigerung berechnen. Das machen wir mit einer einfachen Formel. Wir achten allerdings darauf, dass die Division nicht auf Ganzzahlen durchgeführt wird, sondern auf Fließkommazahlen. Werden nämlich zwei Ganzzahlen dividiert, ist das Ergebnis wieder eine ganze Zahl. Wenn wir vorher die Zahlen aus dem Array herausnehmen und in ein `double` konvertieren, haben wir später durch die Division von zwei Fließkommazahlen ein genaueres Verhältnis. Damit vermeiden wir Probleme bei der Rundung, denn falls wir irgendwann einmal die Konstante verändern wollen, z. B. auf einen viel kleineren Wert, könnte es sein, dass kleine Sprünge nicht korrekt erkannt werden. Sonderfall: Tage, an denen kein Umsatz generiert wurde, funktionieren, weil die Steigerung am Folgetag `Double.Infinity` ist und »Unendlich« größer als `MIN_PERCENT` ist.

Nach der Berechnung der relativen Steigung prüfen wir, ob wir über unsere Konstante von 5 % kommen, und erhöhen die Variable `result`, in der wir uns alle Erhöhungen merken. Am Ende der Schleife geben wir `result` zurück und melden damit, wie viele Erhöhungen wir insgesamt gefunden haben.

Aufgabe 4.2.3: Aufeinanderfolgende Strings suchen und feststellen, ob Salty Snook kommt

```
public static boolean isProbablyApproaching( String[] signs ) {

  final int MIN_OCCURRENCES = 4;

  if ( signs.length < MIN_OCCURRENCES )
    return false;

  for ( int i = 0, count = 1; i < signs.length - 1; i++ ) {
    String currentSign = Objects.requireNonNull( signs[ i ] );
    String nextSign    = Objects.requireNonNull( signs[ i + 1 ] );
    if ( currentSign.equals( nextSign ) ) {
      count++;
      if ( count == MIN_OCCURRENCES )
        return true;
    }
    else // ! currentSign.equals( nextSign )
      count = 1;
  }
  return false;
}
```

Listing 4.3 com/tutego/exercise/array/SaltySnook.java

Die Anzahl der gewünschten Raumschiffe merken wir uns in einer Konstanten `MIN_OCCURRENCES`, sodass wir die Anzahl später leicht ändern können.

Als Erstes prüfen wir, ob das Array mindestens `MIN_OCCURRENCES` viele Elemente hat. Wenn nicht, liefert die Methode dem Aufrufer `false` zurück. Wurde `null` übergeben, folgt eine `NullPointerException` durch den Zugriff auf das Attribut `length`, was den fehlerhaften Parameter deutlich meldet.

Wenn wir aus der Methode nicht aussteigen, gibt es mindestens vier Elemente. Beim Vergleich von nachfolgenden Elementen im Array gibt es in der Regel zwei Ansätze:

- einen Index von `0` bis zum vorletzten Element generieren und dann Zugriff auf zwei Elemente über den Index und Index + `1`
- einen Index von `1` bis zum letzten Element generieren und dann Zugriff auf zwei Elemente über den Index `- 1` und Index

Diese Lösung verwendet die erste Variante.

Die `for`-Schleife deklariert zwei lokale Variablen: `i` für den Index, und in der Variablen `count` merken wir uns die Anzahl der nacheinander gleichwertigen Strings; da ein String mindestens einmal vorkommt, wird die Variable mit `1` initialisiert.

Wir beginnen in der Schleife beim Index `0` und speichern das Element in einer Zwischenvariablen `currentSign`. An der Stelle `1` haben wir zum Start das zweite Element, und auch diese Belegung wird gesichert in einer sprechenden Variablen `nextSign`. `Objects.requireNonNull(…)` wird an dieser Stelle eine Ausnahme auslösen, wenn eines der Array-Elemente `null` ist.

Strings haben eine `equals(…)`-Methode, die zur Bestimmung der Gleichwertigkeit herangezogen wird. Es gibt zwei Ausgänge für den Vergleich:

1. Wenn wir zwei gleiche aufeinanderfolgende Zeichenfolgen gefunden haben, setzen wir den Zähler `counter` hoch und testen, ob er gleich `MIN_OCCURRENCES` ist. In dem Fall liegen vier gleichwertige Zeichenfolgen hintereinander, und wir können mit `return true` aus der Methode aussteigen.
2. Sind `currentSign` und `nextSign` nicht gleichwertig, müssen wir den Zähler auf `1` zurücksetzen.

Wurde am Ende der Schleife keine Zeichenfolge erkannt, die viermal hintereinander vorkommt, wird die Methode mit `return false` verlassen.

Aufgabe 4.2.4: Array umdrehen

```
public static void reverse( double[] numbers ) {
  final int middle = numbers.length / 2;

  for ( int left = 0; left < middle; left++ ) {
    int right = numbers.length - left - 1;
    swap( numbers, left, right );
  }
}

private static void swap( double[] numbers, int i, int j ) {
  double swap  = numbers[ i ];
  numbers[ i ] = numbers[ j ];
  numbers[ j ] = swap;
}
```

Listing 4.4 com/tutego/exercise/array/ArrayReverser.java

Die `reverse(…)`-Methode bekommt als Parameter ein Array. Wir bekommen folglich ein Verweis auf ein Objekt von einer anderen Stelle übergeben. Änderungen finden also auf keiner Kopie statt, sondern wir operieren auf genau dem Array, das der Aufrufer übergeben hat. Da Arrays Objekte sind, die über Referenzen angesprochen werden, ist es möglich, dass der Aufrufer `null` übergeben hat. In dem Fall führt das kommende `numbers.length` zu einer `NullPointerException`, und das ist in Ordnung.

Der Algorithmus selbst ist nicht schwierig. Wir müssen das erste mit dem letzten Element vertauschen, dann das zweite mit dem zweitletzten und so weiter. Das Vertauschen der Elemente lagern wir in eine eigene Methode `swap(…)` aus.

Damit wir uns in `reverse(…)` nicht selbst die Elemente überschreiben, dürfen wir nur bis zur Hälfte laufen. Die Variable `middle` steht für die Hälfte. Zwar wird die Variable nur ein einziges Mal in der Schleife genutzt, doch eine Variable dieser Art hilft, die Bedeutung dieses Ausdrucks genauer zu dokumentieren. Unsere Schleife beginnt mit dem Schleifenzähler `left` bei `0` und läuft bis zur Mitte. Die Variable `right` läuft in die entgegengesetzte Richtung.

Aufgabe 4.2.5: Das nächste Kino finden

```
static double minDistance( Point[] points, int size ) {

  if ( points.length == 0 || size > points.length )
    throw new IllegalArgumentException(
      "Array is either empty or size out of bounds" );

  double minDistance = points[ 0 ].distance( 0, 0 );

  // Index variable i starting at 1, second element
  for ( int i = 1; i < size; i++ ) {
    double distance = points[ i ].distance( 0, 0 );
    if ( distance < minDistance )
      minDistance = distance;
  }

  return minDistance;
}
```

Listing 4.5 com/tutego/exercise/array/MinDistance.java

Als Erstes prüfen wir in der Methode, ob die Parameter `points` und `size` korrekt sind. Wir erwarten mindestens ein Element, und die Anzahl der zu betrachtenden Elemente darf nicht größer sein als die Anzahl der Elemente im Array. War die Übergabe `null`, folgt automatisch durch den Zugriff auf `length` eine `NullPointerException`.

Bei der Frage nach dem größten oder kleinsten Element einer Liste sehen die Algorithmen immer gleich aus. Wir beginnen mit einem Kandidaten und schauen dann, ob dieser Kandidat korrigiert werden muss. Unser Kandidat ist `minDistance`. Wir ini-

tialisieren ihn mit dem Abstand des ersten Punktes zum Nullpunkt. Den Abstand zum Nullpunkt müssen wir nicht selbst ausrechnen, hier hilft uns praktischerweise die `Point`-Methode `distance(x,y)`. Wie übergeben die Koordinaten `0`, `0`, relativ zu denen der Punkt seinen Abstand berechnen soll.

Damit alle Punkte betrachtet werden, laufen wir durch das Array und nutzen `size` als Längenbeschränkung. Vom neuen Punkt berechnen wir ebenfalls den Abstand zum Nullpunkt, und falls wir einen Punkt gefunden haben, der näher am Nullpunkt liegt, müssen wir unsere Wahl korrigieren.

Am Ende der Methode geben wir die minimale Distanz zum Nullpunkt zurück. Falls die Methode jetzt nun nicht die Distanz selbst, sondern einen `Point` zurückgeben soll, schreiben wir die Methode so um, dass wir uns neben `double minDistance` zusätzlich `Point nearest` merken; würden wir auf `minDistance` verzichten, müsste jedes Mal die Distanz neu berechnet werden, was unnötig Performance kostet.

```
static Point minDistance2( Point[] points, int size ) {
  Point  nearest = points[ 0 ];
  double minDistance = nearest.distance( 0, 0 );

  for ( int i = 1; i < size; i++ ) {
    double distance = points[ i ].distance( 0, 0 );
    if ( distance < minDistance ) {
      minDistance = distance;
      nearest = points[ i ];
    }
  }
  return nearest;
}
```

Listing 4.6 com/tutego/exercise/array/MinDistance.java

Aufgabe 4.2.6: Süßigkeitenladen überfallen und fair aufteilen

```
public static int findSplitPoint( int[] values ) {

  if ( values.length < 2 )
    return -1;

  int sumLeft = values[ 0 ];

  int sumRight = 0;
  for ( int i = 1; i < values.length; i++ )
    sumRight += values[ i ];
```



```
  for ( int splitIndex = 1; splitIndex < values.length; splitIndex++ ) {
    int relativeDifference = relativeDifference( sumLeft, sumRight );

    Logger.getLogger( "MuggingFairly" )
          .info( "splitIndex=" + splitIndex
                 + ", sum left/right=" + sumLeft + "/" + sumRight
                 + ", difference=" + relativeDifference );

    if ( relativeDifference <= 10 )
      return splitIndex;

    int element = values[ splitIndex ];
    sumLeft  += element;
    sumRight -= element;
  }
  return -1;
}

// https://en.wikipedia.org/wiki/Relative_change_and_difference
private static int relativeDifference( int a, int b ) {
  if ( a == b ) return 0;
  int absoluteDifference = Math.abs( a - b );
  return (int) (100. * absoluteDifference / Math.max( a, b ));
}
```

Listing 4.7 com/tutego/exercise/array/FairSharing.java

Den Algorithmus für die Lösung können wir gut iterativ oder rekursiv umsetzen. Die Entscheidung ist hier auf die gut verständliche iterative Variante gefallen.

Beginnen wir mit einer einfachen Überlegung, wie wir das Problem lösen können. Wir könnten

- einen Index nehmen, der das Array in zwei Hälften teilt,
- die Summe vom rechten und linken Teil berechnen,
- vergleichen und, wenn die beiden Seiten in etwa gleich sind, das Programm mit einem Ergebnis beenden.

Der Index wandert von vorne nach hinten, und die Summen werden immer neu berechnet. Dieser Algorithmus ist einfach, wir müssen allerdings mehrfach über das Array gehen, sodass letztendlich die Laufzeit quadratisch ist. Das geht besser.

Wenn wir das Array in zwei Teile zerlegen und der Cursor um eine Position nach rechts wandert, verändert sich die Summe nach einem ganz einfachen Muster: Das,

was links zur Summe hinzukommt, wird rechts abgezogen. Das ist die Kernidee der Lösung.

Am Anfang der Methode prüfen wir, ob ein oder kein Element übergeben wurde. Dann kann es auch keine faire Teilung geben, und wir liefern `-1` zurück. Falls die `null`-Referenz übergeben wird, knallt das Programm mit einer `NullPointerException`, was eine gute Reaktion ist.

Im nächsten Schritt deklarieren wir zwei Variablen, die die Summen der linken und rechten Hälfte speichern. Die linke Summe besteht am Anfang nur aus dem ersten Element des Arrays, die rechte Summe bildet sich vom ersten bis zum letzten Element.

Diese beiden Variablen, `sumLeft` und `sumRight`, werden wir im Folgenden anpassen. Die Schleife läuft vom ersten bis zum letzten Element. Da wir schon vor dem Schleifendurchlauf die linke und rechte Summe komplett gebildet haben, können wir jetzt schon die relative Differenz berechnen, und wenn sie `<= 10` ist, dann haben wir tatsächlich schon ein Ergebnis. Falls der Abstand der Werte größer war, wird zum linken Teil das Element addiert und vom rechten Element abgezogen. Zum Schluss geht es weiter in die Schleife, und falls die relative Differenz irgendwann einmal kleiner gleich 10 wird, springen wir mit dem `splitIndex` raus, andernfalls wird die Methode mit `-1` beendet.

Aufgabe 4.3.1: Berge zeichnen

```
private static String mountainChar() { return "*"; }

public static void printMountain( int[] altitudes ) {

  int maxAltitude = altitudes[ 0 ];

  for ( int currentAltitude : altitudes )
    if ( currentAltitude > maxAltitude )
      maxAltitude = currentAltitude;

  // include height 0, so it's >= 0
  for ( int height = maxAltitude; height >= 0; height-- ) {
    System.out.print( height + " " );
    for ( int altitude : altitudes )
      System.out.print( altitude == height ? mountainChar() : ' ' );
    System.out.println();
  }
}
```

Listing 4.8 com/tutego/exercise/array/MountainVisualizer.java

Die Methode `printMountain(int[] altitudes)` nimm ein ganzes Array von Höheninformationen an, und für die grafische Darstellung müssen wir im ersten Schritt den höchsten Wert finden. Das ist die Aufgabe der ersten Schleife. `maxAltitude` speichert das Maximum.

Im nächsten Schritt sind Zeilen zu zeichnen. Jede Zeile steht für eine Höhe. Das Schreiben aller Zeilen übernimmt eine `for`-Schleife mit einem Schleifenzähler `height`. Da es mit der höchsten Höhe beginnt, beginnt die Schleife mit `maxAltitude` und gehen hinunter auf `0`. Die Aufgabenstellung besagt, dass es nicht unter den Nullpunkt geht, das heißt, wir müssen keine zweite Suche für die kleinste Zahl ergänzen.

Im Rumpf der `height`-Schleife geben wir zunächst die Höhe aus gefolgt von einem Leerzeichen. (Wir schreiben `height + " "` und nicht `height + ' '`, warum?) Eine innere `for`-Schleife kümmert sich um die Zeile. Sie läuft wiederholt über die Höheninformationen `altitudes`, die der Methode übergeben wurden. Für jedes Element in `altitudes` fragen wir ab, ob es der Höhe `height` entspricht, und wenn ja, zeichnen wir das Symbol über `mountainChar()`, andernfalls ein Leerzeichen. Am Ende der Zeile wird eine Leerzeile geschrieben.

Das Zeichnen des Bergsymbols übernimmt die Methode `mountainChar()`; sie liefert ein * zurück. Wir hätten das Zeichen direkt zeichnen oder über eine Konstante referenzieren können, doch die Methode ist eine Vorbereitung für die nächste Aufgabe ...

Optionale Erweiterung:

Im ersten Lösungsvorschlag gibt die Methode `mountainChar()` immer * zurück; wenn die Methode andere Symbole zurückgeben soll, braucht sie etwas mehr Kontext, denn sie muss zurück und nach vorne schauen können. Erweitern wir daher die Signatur: `mountainChar(int[] altitudes, int index)`. Die Methode bekommt Zugriff auf das Array und auf die aktuelle Position. Der Aufruf sieht so aus:

```
for ( int height = maxAltitude; height >= 0; height-- ) {
  System.out.print( height + " " );
  for ( int x = 0; x < altitudes.length; x++ )
    System.out.print( altitudes[ x ] == height ?
      mountainChar( altitudes, x ) : ' ' );
  System.out.println();
}
```

Listing 4.9 com/tutego/exercise/array/MoreMountainVisualizer.java

So kann `mountainChar(…)` selbst entscheiden, was das richtige Symbol ist.

```
private static char mountainChar( int[] altitudes, int index ) {
  int previous = index == 0 ? 0 : altitudes[ index - 1 ];
  int current  = altitudes[ index ];
```

```
  int next     = index < altitudes.length - 1 ? altitudes[ index + 1 ] : -1;

  if ( previous < current && current > next )
    return '^';
  if ( current < next )
    return '/';
  if ( current > next )
    return '\\';
  // current == next )
  return '-';
}
```

Listing 4.10 com/tutego/exercise/array/MoreMountainVisualizer.java

Im ersten Schritt werden die Variablen `previous`, `current` und `next` mit den Höhen aus dem Array initialisiert; so lässt sich auf die Höhe des aktuellen Elements schauen, aber auch auf die des Vorgängers und Nachfolgers. Vor dem ersten Element des Arrays soll die Höhe 0 sein, genauso wie nach dem letzten Element.

Abhängig von den Beziehungen kann eine Wahl für das Zeichen auf der Höhe `current` getroffen werden:

- Sind `previous` und `next` kleiner als `current`, so haben wir eine Spitze und zeichnen ein ^.
- Sind wir niedriger als der rechte Nachbar, geht es bergauf, und wir zeichnen /.
- Sind wir höher als der rechte Nachbar, geht es bergab, das Symbol ist \.
- Andernfalls ist der rechte Nachbar auf der gleichen Höhe wie wir selbst, und das wird angedeutet durch -.

Aufgabe 4.4.1: Mini-Sudoku auf gültige Lösung prüfen

```
final int DIMENSION = 3;
for ( int i = 1; i <= DIMENSION * DIMENSION; i++ ) {
  boolean found = false;
  matrixLoop:
  for ( int row = 0; row < DIMENSION; row++ ) {
    for ( int column = 0; column < DIMENSION; column++ ) {
      int element = array[ row ][ column ];
      if ( element == i ) {
        found = true;
        break matrixLoop;
      }
    }
  }
```



```
  if ( found == false )
    System.out.printf( "Missing %d%n", i );
}
```

Listing 4.11 com/tutego/exercise/array/Sudoku3x3Checker.java

Das Array für die Aufgabe deklarieren wir vorweg mit Elementen, ebenso legen wir eine Variable `DIMENSION` an, für die Ausmaße des Arrays. Wir gehen davon aus, dass das 3-×-3-Array genau neun Elemente besitzt.

Zwei verschiedene Lösungen wollen wir uns anschauen. Ist zu testen, ob die Zahlen 1 bis 9 in dem zweidimensionalen Array vorkommen, kann eine Schleife die Werte von 1 bis 9 produzieren, und dann lässt sich prüfen, ob jede dieser Zahlen in dem zweidimensionalen Array vorkommt. Dazu legen wir eine `boolean`-Variable `found` an, die wir am Anfang mit `false` initialisieren und immer dann, wenn im Array das Element vorkommt, auf `true` gesetzt wird; in dem Fall können wir auch die Schleifen abbrechen. Prinzipiell könnten wir natürlich weitersuchen, aber das ist unnötig. Für den Abbruch der Schleife müssen wir auf eine besondere Konstruktion in Java zurückgreifen, die Sprungmarken. Wenn wir in der Fallunterscheidung einfach nur `break` nutzen, beendet das die innerste Schleife, allerdings nicht die äußere Schleife. Mit einer Sprungmarke können wir auch die äußere Schleife mit `break` verlassen. Am Ende der Schleifen fragen wir das Flag `found` ab, und wenn das Flag weiterhin `false` ist, weil es in der Fallunterscheidung nicht auf `true` gesetzt wurde, fehlt die Zahl. Wir geben sie aus.

Der Nachteil der Lösung ist die relativ hohe Laufzeit, außerdem macht ein `break` mit Label den Code unleserlich und schwerer verständlich. Wir müssen neunmal das 3 × 3 große Array ablaufen. Das geht besser. Allerdings müssen wir uns merken, ob wir eine Zahl schon einmal gesehen haben oder nicht.

```
boolean[] numberExisted = new boolean[ DIMENSION * DIMENSION ];

for ( int row = 0; row < DIMENSION; row++ ) {
  for ( int column = 0; column < DIMENSION; column++ ) {
    int element = array[ row ][ column ];
    if ( element >= 1 && element <= 9 )
      numberExisted[ element - 1 ] = true;
  }
}

for ( int i = 0; i < numberExisted.length; i++ ) {
  boolean found = numberExisted[ i ];
```

```
  if ( ! found )
    System.out.printf( "Missing %d%n", i + 1 );
}
```

Listing 4.12 com/tutego/exercise/array/Sudoku3x3Checker.java

Die zweite Lösung deklariert ein `boolean`-Array `numberExisted` als Speicher. Das Praktische bei den Zahlen ist, dass sie zwischen 1 und 9 liegen, also können wir das problemlos auf den Index `0` bis `8` abbilden. Wenn wir aus dem Array eine Zahl holen und daraus einen Index für das Array berechnen, müssen wir uns davor schützen, eine `ArrayIndexOutOfBoundsException` zu bekommen. Daher prüfen wir vorher, ob die Zahl `element` im richtigen Bereich ist. Wenn, dann setzen wir auf der Position `element - 1` den Wert `true`.

Nach dem einmaligen Durchlauf untersuchen wir das Array, und falls wir eine Position finden, die nie beschrieben wurde, wissen wir, dass die Zahl fehlt. Ob eine Stelle im Array mehrfach belegt wurde, spielt dabei keine Rolle.

Aufgabe 4.4.2: Bild vergrößern

```
public static int[][] magnify( int[][] array, int factor ) {
  int width = array[ 0 ].length;
  int height = array.length;
  int[][] result = new int[ height * factor ][ width * factor ];

  for ( int row = 0; row < result.length; row++ ) {
    int[] rows = result[ row ];
    for ( int col = 0; col < rows.length; col++ )
      result[ row ][ col ] = array[ row / factor ][ col / factor ];
  }
  return result;
}

private static void printValues( int[][] array ) {
  for ( int[] rows : array ) {
    for ( int col = 0; col < rows.length; col++ )
      System.out.printf( "%03d%s", rows[ col ],
        col == rows.length - 1 ? "" : ", " );
    System.out.println();
  }
}

private static void fillWithRandomValues( int[][] array ) {
  for ( int row = 0; row < array.length; row++ ) {
```



```
    int[] cols = array[ row ];
    for ( int col = 0; col < cols.length; col++ ) {
      array[ row ][ col ] = ThreadLocalRandom.current().nextInt( 256 );
    }
  }
}

public static void main( String[] args ) {
  int[][] testArray = new int[ 2 ][ 5 ];
  fillWithRandomValues( testArray );
  printValues( testArray );
  int[][] result = magnify( testArray, 2 );
  printValues( result );
}
```

Listing 4.13 com/tutego/exercise/array/ArrayMagnifier.java

Der zentralen `magnify(…)`-Methode haben wir noch ein paar weitere Methoden beiseitegestellt, die ein Array mit Zufallszahlen anlegen und die Informationen in einer Matrix ausgeben.

Um die Übersicht zu erhöhen, werden zum Start der Methode zwei neue Variablen deklariert, die für die Breite und Höhe des zweidimensionalen Arrays stehen. Die nächste Aufgabe besteht darin, ein neues zweidimensionales Array aufzubauen, das in der Höhe und Breite um den `factor` größer ist als das alte Array.

Die Hauptaufgabe übernehmen die zwei ineinandergeschachtelten Schleifen. In der ersten äußeren Schleife laufen wir mit `row` über alle neuen Zeilen. Da zweidimensionale Arrays nichts anderes sind als Arrays in Arrays, hält das äußere Array ganz viele Verweise auf die inneren Arrays, die Zeilen. Die Zwischenvariable `row` steht genau für so eine Zeile. Den inneren Schleifenzähler `col` lassen wir dann von `0` bis zur Breite dieser Zeile laufen.

Der interessante Teil ist in der inneren Schleife. Wir haben die Variablen `row` und `col` in den Wertebereichen des neuen vergrößerten zweidimensionalen Arrays. Es gilt, die Position `result[row][col]` zu initialisieren. Dazu holen wir uns die Werte aus dem alten kleinen `array`. Die Position rechnen wir runter mit `row / factor` für die Zeile und `col / factor` für die Spalte. Zur Erinnerung: `row` geht von `0` bis `height * factor` und `col` bis `width * factor`. Bei den Divisionen `row / factor` und `col / factor` werden Ganzahlen dividiert, und das Ergebnis ist wieder eine Ganzzahl; das hat zur Folge, dass mehrmals dieselbe Zahl aus dem kleinen Ursprungs-Array herausgeholt wird.

Aufgabe 4.5.1: SVG-Polygone mit variabler Koordinatenanzahl erzeugen

```
/**
 * Prints an SVG polygon. Example output:
 * <pre>
 *  <polygon points="200,10 250,190 160,210 " />
 * </pre>
 * @param points of the SVG polygon.
 */
public static void printSvgPolygon( int... points ) {

  if ( points.length % 2 == 1 )
    throw new IllegalArgumentException(
      "Array has an odd number of arguments: " + points.length );

  System.out.print( "<polygon points=\"" );

  for ( int i = 0; i < points.length; i += 2 )
    System.out.printf( "%d,%d ", points[ i ], points[ i + 1 ] );

  System.out.println( "\" />" );
}
```

Listing 4.14 com/tutego/exercise/array/SvgVarargPolygon.java

Zwei Fehler können auftreten:

1. Bei einem Vararg baut der Compiler selbst ein Array aus den übergebenen Argumenten auf, doch auch wir können einen Verweis auf ein Array übergeben. Da Referenzen `null` sein können, könnte es einen Aufruf von `printSvgPolygon(null)` geben. Die Übergabe wird durch `null.length` automatisch zu einer `NullPointerException` führen.
2. Beim Aufruf von `printSvgPolygon()` baut der Compiler ein leeres Array auf; dieses enthält keine Elemente, was prinzipiell für unsere Methode in Ordnung ist. Aber es gibt eine andere Anforderung an die Länge: Die Methode selbst erwartet immer Pärchen von x- und y-Koordinaten. Es wäre ein Fehler, wenn nur eine Koordinate übergeben würde und nicht zwei. Leider kann der Compiler so etwas nicht testen, man kann an die Anzahl der Varargs keine Wünsche stellen wie: »höchstens 102 Elemente«, »die Anzahl der Elemente muss durch 10 teilbar sein« etc. Es bleibt uns nichts anderes übrig, als den Test zur Laufzeit durchzuführen. Wir können den Fehler leicht erkennen, indem wir prüfen, ob die Anzahl der Elemente in dem Array gerade oder ungerade ist. Ist die Anzahl gerade, wurden immer Paare übergeben, für x und y. Ist die Anzahl ungerade, so fehlt eine Koordinate. Wir bestrafen fehlerhafte Aufrufe mit einer `IllegalArgumentException`. Es wäre noch zu überlegen, aus beiden Prüfungen zwei Fallunterscheidungen zu machen, um dann im Fall der ungeraden Übergabe die Anzahl Elemente mit in die Ausnahmemeldung zu setzen.

Das Generieren der Ausgabe besteht aus drei Teilen:

1. Im ersten Teil, dem Prolog, setzen wir das Start-Tag für das Polygon.
2. Im zweiten Teil läuft eine Schleife über alle Elemente des Arrays und greift sich immer zwei Elemente heraus, die mit Komma getrennt auf die Konsole kommen – hinter dem Paar steht ein Leerzeichen. Da wir immer zwei Elemente gleichzeitig aus dem Array nehmen, wird im Fortschaltausdruck der `for`-Schleife der Index um 2 erhöht. Da die Anzahl der Elemente in Array gerade ist, wird es keine `ArrayIndexOutOfBoundsException` geben können.
3. Die Methode endet mit dem Epilog, dem Schließen des Tags.

Aufgabe 4.5.2: Auf Zustimmung prüfen

```
private static boolean allTrue( boolean first, boolean... remaining ) {

  for ( boolean b : remaining )
    if ( b == false )
      return false;

  return first;
}
```

Listing 4.15 com/tutego/exercise/array/AllTrue.java

Bei variablen Argumentlisten ist es nicht möglich, eine Mindestanzahl von Argumenten zu erwarten. Die Lösung für das Problem besteht darin, eine Mindestanzahl von festen Parametern einzuführen und dann zum Schluss ein Vararg für den Rest einzusetzen.

Die Methode hat zwei Pfade, die zu einer Rückgabe `true` oder `false` führen:

1. Als erstes gehen wir das Array ab. Ist einer der Wahrheitswerte im Array `false`, können wir die Methode direkt mit `return false` beenden. Wenn das Array leer ist, passiert nichts.

   Es gibt Autoren, die `boolean`-Werte nicht mit `== false` testen, sondern den Ausdruck negieren, aber ich meine, dass sich `if ( b == false )` besser lesen lässt als `if ( ! b )`; es kommt auch auf den Variablennamen an. Es kann sein, dass als Argument `null` übergeben wurde, dann wird die erweiterte `for`-Schleife eine `NullPointerException` erzeugen, das ist gewollt.

2. Kommt es zu keinem Abbruch der Schleife, müssen alle Elemente in dem Array `true` gewesen sein, und der erste Parameter `first` entscheidet über das Ergebnis.

Aufgabe 4.5.3: Hilfe, Tetraphobie! Alle Vieren nach hinten setzen

```
private static boolean containsFour( int number ) {
  return String.valueOf( number ).contains( "4" );
}

public static int[] fourLast( int... numbers ) {

  if ( numbers.length < 2 )
    return numbers;

  for ( int startIndex = 0; startIndex < numbers.length; startIndex++ ) {
    if ( ! containsFour( numbers[ startIndex ] ) )
      continue;

    // from right to left search the first number without a 4
    for ( int endIndex = numbers.length - 1;
      endIndex > startIndex; endIndex-- ) {
      if ( containsFour( numbers[ endIndex ] ) )
        continue;
      // swap number[i] (with 4) and number[j] no 4
      int swap = numbers[ startIndex ];
      numbers[ startIndex ] = numbers[ endIndex ];
      numbers[ endIndex ]   = swap;
    }
  }
  return numbers;
}
```

Listing 4.16 com/tutego/exercise/array/Tetraphobia.java

Für unsere Lösung schreiben wir neben der gewünschten Methode `fourLast(…)` eine zusätzliche private Methode `boolean containsFour(int)`, die die Frage beantwortet, ob die übergebene Zahl eine 4 enthält. In der Implementierung machen wir es uns einfach und konvertieren die Zahl in einen String, und `contains(String)` prüft, ob der String `"4"` in der String-Repräsentation liegt. Die Prüfung hätten wir natürlich auch numerisch durchführen können, doch das wäre viel komplizierter gewesen. Wir müssten diese Zahl immer durch `10` teilen und uns dann den Rest anschauen und testen, ob er `4` ist. Da ist mehr Code als nur dieser eine Einzeiler nötig.

Die Methode `fourLast(…)` bekommt ein Array übergeben, das wieder `null` sein kann und in so einem Fall durch `numbers.length` zu einer `NullPointerException` führt. Auch könnte das Array nur ein Element beinhalten; in dem Fall geben wir direkt das übergebene Array an den Aufrufer zurück.

Der Algorithmus ist einfach implementiert: Wir laufen mit einer Schleife von links nach rechts, und wenn wir etwas mit einer `4` finden, laufen wir mit einer zweiten Schleife von rechts nach links und suchen den ersten freien Platz ohne `4`. Dann vertauschen wir die Inhalte des Arrays.

Der Lösungsvorschlag ist nicht ganz optimal und sollte von den Lesern verbessert werden.

1. Das Erste, was verbessert werden sollte, ist die Schleife mit `startIndex`, die immer bis `numbers.length` läuft. Das ist unnötig, denn sollte die innere Schleife eine Zahl mit einer `4` finden, dann wird diese Zahl nach hinten gehen, und wir können von der Länge eins abziehen, denn das letzte Element müssen wir ja nicht mehr betrachten.
2. Zweitens ist die innere Schleife nicht optimal, denn sie beginnt immer von rechts nach links, die erste Zahl ohne `4` zu suchen. Allerdings wächst der Block mit den Vieren nur nach links, sodass wir uns in einer zweiten Variablen merken könnten, wo wir unsere letzte `4` untergebracht haben. Läuft die innere Schleife erneut, könnten wir bei dieser Position weitermachen und müssen nicht erst von rechts wieder zu dieser Position finden.
3. Drittens können wir die beiden Verbesserungen kombinieren. Für den Fall, dass in der inneren Schleife keine Vertauschung stattfindet, wären wir fertig.

Quiz 4.6.1: Arrays kopieren

Mit der Methode `arraycopy(…)` lassen sich Bereiche in einem Array verschieben oder Teile eines Arrays in ein anderes Array kopieren. Betrachten wir aus der Javadoc die Parametervariablen von `arraycopy(…)`, die selbsterklärend sind:

```
static void arraycopy(Object src, int srcPos, Object dest,
  int destPos, int length)
```

In unserem Fall wird nicht in einem Array etwas verschoben, sondern es werden zweimal Teile eines Arrays in ein neues Array kopiert. `hooey` ist die Quelle und `shuck` das Ziel. Setzen wir die Konstanten ein und benennen `hooey` und `shuck` um, folgt:

```
int[] src = { 1, 2, 3, 4 };
int[] dest = new int[ 3 ];
System.arraycopy( src, 0, dest, 0, 2 );
System.arraycopy( src, 3, dest, 2, 1 );
```

Es entsteht [1, 2, 4], also ein neues Array, in dem das Element 3 an Index 2 (bushwa) fehlt. Die erste Kopieroperation überträgt von der Quelle ab der ersten Stelle 0 in das neue Array ab der Stelle 0, insgesamt 2 Elemente. Die zweite Kopieroperation kopiert ab der Stelle 3 (überspringt also Stelle 2) alles bis zum Ende auf das Ziel-Array ab der Position 2. Die Anzahl der kopierten Elemente ist eins.

Bei den unsinnigen Variablennamen sollte auch klar geworden sein, dass sauberer Code – hier Variablenbezeichner – menschliche Verarbeitungszeit einspart und Fehler reduziert.

Quiz 4.6.2: Arrays vergleichen

Der Operator == prüft, ob die Objekte, auf die die beiden Referenzvariablen verweisen, identisch sind. Das tun die beiden Variablen array1 und array2 nicht, denn es sind im Speicher zwei völlig getrennt stehende Objekte. Daher wird die Ausgabe auch false sein.

Arrays sind Objekte, und als Objekte haben sie alle Methoden, die auch der Basistyp java.lang.Object bietet. Allerdings können wir mit keiner dieser Methoden irgendetwas anfangen, was wir schnell merken, wenn wir die toString()-Methode auf einem Array-Objekt aufrufen. Die equals(…)-Methode eines Arrays stammt aus Object, und dort steht ein Identitätsvergleich, der, wie im ersten Punkt erklärt, zur Ausgabe false führt.

Um Arrays zu vergleichen, sind wir bei der Klasse java.util.Arrays und der Methode equals(…) ganz richtig. Prinzipiell können wir mit dieser Methode Arrays vergleichen, allerdings haben unsere beiden Arrays eine kleine Besonderheit: Sie enthalten Strings und ein inneres Array mit Ganzzahlen. Wenn dieses unterreferenzierte Array nicht vorhanden wäre, würde tatsächlich bei dieser Ausgabe true erscheinen. Allerdings arbeitet Array.equals(…) flach, das heißt, das referenzierte innere Array müsste identisch sein, da aber auch dieses Ganzzahl-Array jeweils ein neues Array ist, sind die beiden Referenzen auf das Ganzzahl-Array nicht gleich, und somit liefert Array.equals(…) auf unserem Array false.

Nur bei der Methode Array.deepEquals(…) wird true auf dem Bildschirm erscheinen. deepEquals(…) verfolgt auch die referenzierten Unter-Arrays und schaut, ob die Werte gleichwertig sind. Auf die Identität der Unter-Arrays kommt es bei deepEquals(…) nicht an. Die beiden Ganzzahl-Arrays sind gleichwertig, und daher ist auch die Rückgabe von deepEquals(…) true.

Kapitel 16
Java-Stream-API

Die Stream-API ermöglicht eine schrittweise Verarbeitung von Daten. Nachdem eine Quelle Daten emittiert, folgen unterschiedliche Schritte, die Daten filtern und transformieren und auf ein Ergebnis reduzieren.

Streams – auf Deutsch auch »Ströme« genannt, wobei die Bezeichnung mehrdeutig ist und mit Ein-/Ausgabeströmen verwechselt werden könnte – sind eine wichtige Neuerung von Java 8 und greifen auf andere Neuerungen in der Java SE-Bibliothek wie vordefinierte funktionale Schnittstellen oder `Optional` zurück. Zusammen mit Lambda-Ausdrücken und Methodenreferenzen ergeben sich kompakter Code und eine ganz neue Art, deklarativ Verarbeitungsschritte zu konfigurieren.

Die erste Aufgabe in diesem Aufgabenblock nutzt die Helden, die wir im Kapitel über die Klassenbibliothek schon einmal kennengelernt haben. Für diese Heldensammlung kommen alle wichtigen terminalen und intermediären Operationen zum Einsatz. Es folgen unterschiedliche Aufgaben, deren Lösung zeigen, wie elegant die Möglichkeiten der Stream-API sind.

16

Voraussetzungen

- Stream aufbauen können
- terminale und intermediäre Operationen einsetzen können
- primitive Ströme beherrschen
- Lambda-Ausdrücke praktisch einsetzen können

Java SE 9 Standard-Bibliothek, Kapitel 4, und Java ist auch eine Insel, Kapitel 17

In der »2. Insel« führt Abschnitt 4.11, »Stream-API«, ausführlich in die Thematik ein. In der »1. Insel« gibt Abschnitt 17.3, »Java Stream-API«, eine kompakte Einführung.

Verwendete Datentypen in diesem Kapitel:

- `java.util.stream.Stream` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html*)
- `java.util.stream.IntStream` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html*)

- java.util.stream.Collectors (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html*)
- java.util.IntSummaryStatistics (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/IntSummaryStatistics.html*)
- java.util.DoubleSummaryStatistics (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/DoubleSummaryStatistics.html*)
- java.util.regex.Pattern (*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html*)

16.1 Reguläre Ströme mit ihren terminalen und intermediären Operationen

Bei jedem Stream gibt es zwei verpflichtende Schritte und beliebig viele optionale Schritte dazwischen:

1. Aufbau des Streams aus einer Datenquelle
2. optionale Verarbeitungsschritte, genannt *intermediäre Operationen*
3. abschließende Operation, genannt *terminale Operation*

16.1.1 Heldenepos: Stream-API kennenlernen ★

In Kapitel 10, »Besondere Typen aus der Java-Bibliothek«, wurde die Klasse Heroes mit Helden vorgestellt. Darauf greift diese Aufgabe zurück.

Stream-Aufbau:

Baue für die folgenden Aufgabenpunkte immer einen neuen Stream mit den Helden auf, und wende anschließend die terminalen und intermediären Operationen nach folgendem Muster an:

```
Heroes.ALL.stream().intermediate1(…).intermediate2(…).terminal()
```

Terminale Operationen:

1. Gib alle Informationen über Helden im CSV-Format auf dem Bildschirm aus.
2. Frage, ob alle Helden nach 1900 eingeführt wurden.
3. Frage, ob irgendein weiblicher Held nach 1950 (inklusiv) eingeführt wurde.
4. Welcher Held taucht als erster auf?
5. Welcher Held liegt beim Erscheinungsjahr am nächsten an 1960? Es soll auf dem Stream nur eine einzige terminale Operation genutzt werden.
6. Ein StringBuilder soll entstehen, der alle Jahreszahlen kommasepariert enthält. Das Ergebnis soll mit einer einzigen terminalen Stream-Methode entstehen, keiner

intermediären Operation dazwischen. Die Reihenfolge der Jahreszahlen im String spielt keine Rolle.
7. Teile die männlichen und weiblichen Helden in zwei Gruppen auf. Das Ergebnis soll vom Typ Map<Sex, List<Hero>> sein.
8. Bilde zwei Partitionen mit Helden, die vor und nach 1970 eingeführt wurden. Das Ergebnis soll vom Typ Map<Boolean, List<Hero>> sein.

Intermediäre (nichtterminale) Operationen:

1. Wie viele weibliche Helden gibt es insgesamt?
2. Sortiere alle Helden nach dem Erscheinungsdatum, und gib dann alle Helden aus.
3. Gehe folgende Schritte durch:
   - Erzeuge einen kommaseparierten String mit den Namen aller weiblichen Helden.
   - Im Hero gibt es keinen Setter, weil der Hero immutable ist. Aber mit dem Konstruktor können wir neue Helden aufbauen. Konvertiere die Helden in eine Liste anonymer Helden, in denen der Klarname in Klammern zusammen mit den Klammern selbst entfernt wird.
   - Erzeuge ein int[] mit allen Jahren, in denen Helden eingeführt wurden – ohne doppelte Einträge.
4. Gehe über UNIVERSES und nicht über ALL, um die Namen aller Helden auszugeben.

16

16.1.2 Quiz: Doppelt ausgegeben ★

Wenn folgende drei Zeilen in der main(…)-Methode stehen und das Programm startet, was wird die Ausgabe sein?

```
Stream<Integer> numbers = Stream.of( 1, 2, 3, 4, 5 );
numbers.peek( System.out::println );
numbers.forEach( System.out::println );
```

16.1.3 Den geliebten Captain aus einer Liste ermitteln ★

Am Ende des Jahres stimmt die Schiffsbesatzung darüber ab, welche Kandidatin oder welcher Kandidat für die Position des Captains ihnen in Zukunft den Weg zu reicher Beute ebnen sollen. Gewinner ist die Person mit den meisten Nennungen.

Aufgabe:

- Gegeben ist ein Array von Strings mit Namen. Welcher Name wurde wie häufig genannt? Groß-/Kleinschreibung spielt bei den Namen keine Rolle.
- Viele nennen Captain CiaoCiao einfach nur CiaoCiao, das soll gleichbedeutend sein mit Captain CiaoCiao.

Beispiel:

`{ "Anne", "Captain CiaoCiao", "Balico", "Charles", "Anne", "CiaoCiao", "CiaoCiao", "Drake", "Anne", "Balico", "CiaoCiao" }` → `{charles=1, anne=3, drake=1, ciaociao=4, balico=2}`

16.1.4 Bilder einrahmen (Java 11) ★

Captain CiaoCiao ist als bester Captain gewählt worden, die Freude ist groß. Er möchte sein Bild eingerahmt haben.

Gegeben ist ein mehrzeiliger String, wie

```
     ______
_.-':::::::`.
\::::::::::::`.-._
 \:::''   `::::`-.`.
  \         `:::::`.\
   \          `-::::`:
    \______       `:::`.
    .|_.-'__`._     `:::\
   ,'`|:::|  )/`.     \:::
  /. -.`--'  : /.\     ::|
  `-,-'  _,'/| \|\\    |:|
   ,'`::.    |/>`;'\   |:|
   (_\ \:.:.:`((_));`. ;:|
   \.:\ ::_:_:_`-','  `-:|
    `:\\|     SSt:
       )`__...---'
```

Dieser soll in einen Rahmen gesetzt werden:

```
+------------------------------+
|                              |
|       ______                 |
|  _.-':::::::`.               |
|  \::::::::::::`.-._          |
|   \:::''   `::::`-.`.        |
|    \         `:::::`.\       |
|     \          `-::::`:      |
|      \______       `:::`.    |
|      .|_.-'__`._     `:::\   |
|     ,'`|:::|  )/`.     \:::  |
|    /. -.`--'  : /.\     ::|  |
|    `-,-'  _,'/| \|\\    |:|  |
|     ,'`::.    |/>`;'\   |:|  |
|     (_\ \:.:.:`((_));`. ;:|  |
|     \.:\ ::_:_:_`-','  `-:|  |
|      `:\\|     SSt:          |
|         )`__...---'          |
|                              |
+------------------------------+
```


Aufgabe:

Schreibe eine Methode `frame(String)`, die einen mehrzeiligen String umrahmt. Nutze die `String`-Methoden `lines()` und `repeat()`. Beide Methoden sind neu in Java 11; wer die Aufgabe unter Java 8 realisieren möchte, muss ein wenig umdenken.

- Die horizontalen Striche bestehen aus `-`.
- Die vertikalen Striche bestehen aus `|`.
- In den Ecken sitzen Pluszeichen `+`.
- Der Abstand rechts und links vom Rahmen beträgt 2 Leerzeichen.
- Der innere Abstand oben und unten ist eine freie Zeile.
- Zeilenumbrüche sind `\n`, sie sollen sich aber relativ leicht im Programm ändern lassen.

16.1.5 Schau und sag ★★

Captain CiaoCiao träumt vor sich hin und schreibt auf ein Stück Papier:

`1`

Er sieht die Eins und sagt sich: »Oh, 1-mal die 1!« Das schreibt er hin:

`1 1`

Jetzt sieht er zwei Einsen und kann es so vorlesen:

`2 1`

»Arrr! Das ist doch einmal die Zwei und einmal die Eins!« Er schreibt es hin:

`1 2 1 1`

Er liest die Zahlen erneut vor und sagt:

`1 1 1 2 2 1`

Jetzt kommt die Eins sogar dreimal vor:

`3 1 2 2 1 1`

Captain CiaoCiao findet, dass die Zahlen aber schnell groß werden. Er ist neugierig, ob nach ein paar Durchläufen nur die 1, 2 und 3 als Ziffern vorkommen.

Aufgabe:

- Erzeuge mit einem `Stream.iterate(…)` einen unendlichen Stream von Schau-und-sag-Zahlen.
- Begrenze den Stream auf 20 Elemente
- Gib die Zahlen auf der Konsole aus; sie können auch kompakt als String wie `111221` ausgegeben werden.

> **Tipp**
> Die Aufgabe lässt sich mit einem geschickten regulären Ausdruck mit einer Back-Reference lösen. Diese Lösungsvariante ist aber anspruchsvoll, und wer diesen Weg einschlagen möchte, findet unter *https://regular-expressions.mobi/backref.html* weitere Details.

> **Hinweis**
> Was hier gefragt wird, ist die *Look-and-Say-Sequenz*, die *https://oeis.org/A005150* mit vielen Verweisen ausführlicher erklärt.

16.1.6 Doppelte Inseln mit Metallen der Seltenen Erden entfernen (Java 9) ★★★

Das Geschäft mit Metallen der Seltenen Erden ist für Bonny Brain besonders attraktiv. Ihre Crew stellt eine Liste zusammen, auf welchen Inseln welche Metalle der Seltenen Erden vorkommen. Das Ergebnis kommt in eine Textdatei, die so aussieht:

```
Balancar
Erbium
Benecia
Yttrium
Luria
Thulium
Kelva
Neodym
Mudd
Europium
Tamaal
Erbium
Varala
Gadolinium
Luria
Thulium
```


In einer Zeile steht die Insel, in der nächsten Zeile stehen die Metalle der Seltenen Erden. Allerdings kann es passieren, dass verschiedene Crewmitglieder gleiche Paare in die Textdatei eintragen. Im Beispiel ist es das Paar `Luria` und `Thulium`.

Aufgabe:

- Schreibe ein Programm, das alle doppelten Zeilenpaare aus dem Text löscht.
- Das Programm muss so flexibel sein, dass die Eingabe aus einem `String`, `File`, `InputStream` oder `Path` kommen kann.
- Die Zeilen sind immer nur mit einem `\n` getrennt. Auch die letzte Zeile endet mit einem `\n`.

> **Tipp**
> Für die Lösung helfen die Typen `Pattern`, `Scanner` und `MatchResult` sowie die `Scanner`-Methode `findAll(…)` und weitere `Stream`-Methoden. `findAll(…)` ist neu in der Java-Version 9.

16.1.7 Wo gibt es die Segel? ★★

Bonny Brain benötigt für das Schiff ein neues Hochleistungssegel. Die Sachbearbeiter aus der Materialwirtschaft bereiten eine Liste mit Koordinaten von geeigneten Tuchmachern vor:

```
Point.Double[] targets = { // Latitude, Longitude
   new Point.Double( 44.7226698,  1.6716612 ),
   new Point.Double( 50.4677807, -1.5833018 ),
   new Point.Double( 44.7226698,  1.6716612 )
};
```

Aufgabe:

- In der Liste kommen einige Koordinaten doppelt vor, diese können ignoriert werden.
- Am Ende soll eine `Map<Point.Double, Integer>` stehen mit der Koordinate und dem Abstand in Kilometer zum aktuellen Standort von Bonny Brain (40.2390577, 3.7138939).

Eine Beispielausgabe könnte wie folgt aussehen:

```
{Point2D.Double[50.4677807, -1.5833018]=1209, Point2D.Double[
44.7226698, 1.6716612]=525}
```

Der Abstand in Kilometer berechnet sich mit der Haversine-Formel so:

```
private static int distance( double lat1, double lng1,
    double lat2, double lng2 ) {
  double earthRadius = 6371; // km
  double dLat = Math.toRadians( lat2 - lat1 );
  double dLng = Math.toRadians( lng2 - lng1 );
  double a = Math.sin( dLat / 2 ) * Math.sin( dLat / 2 ) +
    Math.cos( Math.toRadians( lat1 ) ) * Math.cos( Math.toRadians( lat2 ) ) *
    Math.sin( dLng / 2 ) * Math.sin( dLng / 2 );
  double d = 2 * Math.atan2( Math.sqrt( a ), Math.sqrt( 1 - a ) );
  return (int) (earthRadius * d);
}
```

16.1.8 Das beliebteste Auto kaufen ★★★

Captain CiaoCiao muss seinen Fuhrpark vergrößern, und so fragt er die Crew, welche gepanzerten Autos empfohlen werden. Er bekommt ein Array von Modellnamen der folgenden Art:

```
String[] cars = {
  "Gurkha RPV", "Mercedes-Benz G 63 AMG", "BMW 750", "Toyota Land Cruiser",
  "Mercedes-Benz G 63 AMG", "Volkswagen T5", "BMW 750", "Gurkha RPV",
  "Dartz Prombron", "Marauder", "Gurkha RPV" };
```

Aufgabe:

- Schreibe ein Programm, das ein Array mit Modellnamen verarbeitet und am Ende eine `Map<String, Long>` erzeugt, das die Modellnamen mit der Anzahl Vorkommen assoziiert. Dieser Aufgabenteil kann gut mit der Stream-API gelöst werden.
- Es soll keine Modelle geben, die nur einmal genannt wurden; erst Modelle ab zwei Nennungen sollen in der Datenstruktur auftauchen. Bei diesem Aufgabenteil können wir besser auf die Stream-API verzichten und eine andere Variante heranziehen.

Eine Beispielausgabe könnte so aussehen:

```
{Mercedes-Benz G 63 AMG=2, BMW 750=2, Gurkha RPV=3}
```

Modifiziere die Abfrage so, dass zwar alle Modelle in einer `Map` stehen, aber die Namen mit `false` assoziiert sind, wenn es weniger als zwei Nennungen gibt. Eine Ausgabe könnte so aussehen:

```
{Marauder=false, Dartz Prombron=false, Mercedes-Benz G 63 AMG=true, Toyota
Land Cruiser=false, Volkswagen T5=false, BMW 750=true, Gurkha RPV=true}
```


16.2 Primitive Ströme

Neben den Streams für Objekte bietet die Java-Standard-Bibliothek drei besondere Ströme für primitive Datentypen: `IntStream`, `LongStream` und `DoubleStream`. Viele Methoden sind ähnlich, wichtige Unterschiede sind Bereiche (engl. *range*) und spezielle Reduktionen, zum Beispiel auf die Summe oder den Durchschnitt.

16.2.1 NaN in einem Array erkennen ★

Java unterstützt beim Fließkommatyp `double` drei besondere Werte: `Double.NaN`, `Double.NEGATIVE_INFINITY` und `Double.POSITIVE_INFINITY`; entsprechende Konstanten gibt es für `float` in `Float`. Bei mathematischen Operationen muss geprüft werden, ob das Ergebnis gültig ist und kein `NaN` ist. Durch die arithmetischen Operationen wie Addition, Subtraktion, Multiplikation, Division ist NaN nicht zu erreichen, es sei denn, ein Operand ist `NaN`, aber diverse Methoden aus der Klasse `Math` liefern bei ungültigen Eingaben als Ergebnis `NaN`. Ist zum Beispiel bei den Methoden `log(double a)` oder `sqrt(double a)` das Argument `a` echt kleiner als Null, ist das Ergebnis `NaN`.

Aufgabe:

- Schreibe eine Methode `containsNan(double[])`, die `true` zurückliefert, wenn das Array ein `NaN` enthält, andernfalls `false`.
- Ein einziger Ausdruck soll im Rumpf der Methode reichen.

16

Beispiel:

```
double[] numbers1 = { Math.sqrt( 2 ), Math.sqrt( 4 ) };
System.out.println( containsNan( numbers1 ) );           // false

double[] numbers2 = { Math.sqrt( 2 ), Math.sqrt( -4 ) };
System.out.println( containsNan( numbers2 ) );           // true
```

16.2.2 Jahrzehnte erzeugen ★

Ein Jahrzehnt ist ein Zeitraum von zehn Jahren, egal, wann es anfängt und endet. Üblicherweise werden Jahrzehnte anhand ihrer gemeinsamen Zehnerstelle gruppiert. Die 1990er beginnen am 1. Januar 1990 und enden am 31. Dezember 1999. Diese Interpretation nennt sich *0-bis-9-Dekade*. Es gibt auch die *1-bis-0-Dekade*, bei der die Zählung der Jahrzehnte mit einer 1 auf der Einerstelle beginnt. Nach dieser Interpretation gehen die 1990er vom 1. Januar 1991 und enden am 31. Dezember 2000.

Aufgabe:

- Schreibe eine Methode `int[] decades(int start, int end)`, die alle Jahrzehnte von einem Startjahr bis zu einem Endjahr als Array liefert.
- Es soll die 0-bis-9-Dekade verwendet werden.

Beispiele:

- `Arrays.toString( decades( 1890, 1920 ) )` → `[1890, 1900, 1910, 1920]`
- `Arrays.toString( decades( 0, 10 ) )` → `[0, 10]`
- `Arrays.toString( decades( 10, 10 ) )` → `[10]`
- `Arrays.toString( decades( 10, -10 ) )` → `[]`

16.2.3 Array mit konstantem Inhalt über Stream erzeugen ★

Aufgabe:

Schreibe eine Methode

`fillNewArray(int size, int value)`

Beispiel:

`Arrays.toString( fillNewArray( 3, -1 ) )` → `[-1, -1, -1]`

16.2.4 Pyramiden zeichnen (Java 11) ★

Aufgabe:

- Erzeuge aus einer geschickten Kombination von `range(…)`, `mapToObj(…)` und `forEach(…)` die folgende Ausgabe:

```
    /\
   /\/\
  /\/\/\
 /\/\/\/\
/\/\/\/\/\
```

  Die Pyramide ist fünf Zeilen hoch.
- Versuche, die Aufgabe in nur einer Anweisung zu lösen, vom Aufbau der Pyramide bis zur Konsolenausgabe.

16.2.5 Buchstabenhäufigkeit eines Strings ermitteln ★

Eine Voraussetzung für eine Kompression ist, häufig vorkommende Folgen möglichst kurz zu repräsentieren. Wenn in einer Datei etwa 0 0 0 0 1 1 1 vorkommt, dann

wird später hinterlegt: viermal eine 0, dann dreimal eine 1. Die Information über die Zahlen 0 und 1 versucht ein Kompressionsalgorithmus in sehr wenigen Bits auszudrücken. Von Vorteil ist, wenn bekannt ist, wie oft ein Symbol oder eine Folge insgesamt vorkommt, um einschätzen zu können, ob sich eine Kompression dieser Folge überhaupt lohnt. Eine Schleife könnte vorher über die Eingabe laufen und Häufigkeiten zählen.

Aufgabe:

- Die Eingabe ist ein String. Generiere mithilfe einer geschickten Stream-Verkettung einen neuen String, der jeden Buchstaben des Ursprung-Strings enthält, gefolgt von der Häufigkeit dieses Buchstabens im gegebenen String.
- Die Pärchen aus Buchstabe und Häufigkeit sollen durch einen Schrägstrich im Ergebnis-String getrennt werden.
- Performance spielt keine zentrale Rolle.

Beispiele:

- `"eclectic"` → `"e2/c3/l1/e2/c3/t1/i1/c3"`
- `"cccc"` → `c4/c4/c4/c4`
- `""` → `""`

16.2.6 Von 1 auf 0, von 10 auf 9 ★ ★

Bonny Brain möchte ein neues Boot erwerben und schickt Elaine in den Hafen, um Boote zu bewerten. Elaine schreibt ihre Bewertungen von 1 bis 10 hintereinander auf ein Papier, etwa so:

`102341024`

Bonny bekommt die Zahlenfolge, ist aber mit der Anordnung und den Zahlen nicht zufrieden. Die Zahlen sollen erstens durch ein Komma getrennt werden und zweitens bei `0` anfangen, nicht bei `1`.

Aufgabe:

- Schreibe eine Methode `String decrementNumbers(Reader)`, die aus einer Eingabequelle eine Zeichenfolge mit Ziffern liest und diese in einen kommaseparierten String konvertiert; alle Zahlen sollen um 1 vermindert werden. Was keine Ziffer ist, soll auch nicht in das Ergebnis kommen.

Beispiele:

- `102341024` → `"9, 1, 2, 3, 9, 1, 3"`
- `-1` → `"0"`
- `abc123xyz456` → `"0, 1, 2, 3, 4, 5"`

16.2.7 Zwei int-Arrays zusammenführen ★★

Aufgabe:

- Gesucht ist eine Methode, die zwei `int`-Arrays zusammenführt.
- Es soll zwei überladene Methoden geben:
  `static int[] join( int[] numbers1, int[] numbers2)` und
  `static int[] join( int[] numbers1, int[] numbers2, long maxSize)`.

  Mit dem optionalen dritten Parameter kann die maximale Anzahl Elemente des Ergebnisses reduziert werden.

Beispiele:

```
int[] numbers1 = { 7, 12 };
int[] numbers2 = { 51, 56, 0, 2 };
int[] result1 = join( numbers1, numbers2 );
int[] result2 = join( numbers1, numbers2, 3 );
System.out.println( Arrays.toString( result1 ) );   // [7, 12, 51, 56, 0, 2]
System.out.println( Arrays.toString( result2 ) );   // [7, 12, 51]
```

16.2.8 Gewinnkombinationen ermitteln ★★

Bonny Brain plant die nächste Party und bereitet ein Ringwurfspiel vor. Als Erstes stellt sie unterschiedliche Objekte auf, zum Beispiel diese zwei:

▨ ▧

Dann gibt sie den Spielern zwei Ringe in die Hand und lässt sie werfen. Geht der Ring über ein Objekt, zählt das als Gewinn. Wie viele Möglichkeiten zum Gewinnen gibt es, und wie sehen die Möglichkeiten aus? Nimmt man die beiden Objekte ▨ ▧, könnte es sein, dass ein Spieler ▨ oder ▧ oder auch beide – ▨ und ▧ – »trifft«; kein Treffer ist kein Gewinn.

Aufgabe:

- Gegeben ist ein String mit beliebigen Zeichen aus der Basic Multilingual Plane (BMP), also `U+0000` bis `U+D7FF` und `U+E000` bis `U+FFFF`.
- Erzeuge eine Liste mit allen Möglichkeiten, wie ein Spieler gewinnen kann.

Beispiel:

- ▨▧ → [▧, ▨, ▨▧], aber nicht [▧, ▨, ▨▧, ▧▨]!
- ▣▢▲ → [▣▢▲, ▲, ▢, ▣, ▢▲, ▣▲, ▣▢]
- `MOON` → `[OO, MN, MO, MOON, MOO, MON, M, N, OON, ON, O]`

16.3 Statistiken

Die Ströme `IntStream`, `LongStream` und `DoubleStream` haben terminierende Methoden wie `average()`, `count()`, `max()`, `min()` und `sum()`. Falls allerdings nicht nur eine dieser statistischen Informationen interessant ist, sondern mehrere, lassen sich diverse Informationen in einem `IntSummaryStatistics`, `LongSummaryStatistics` oder `DoubleSummaryStatistics` sammeln.

16.3.1 Die schnellsten und langsamsten Paddler ★

Bonny Brain richtet auf der Partyinsel X Æ A-12 den jährlichen Paddelwettbewerb »Venomous Paddle Open« aus. Am Ende sollen die beste, schlechteste und Durchschnittszeit ausgegeben werden. Die Ergebnisse der Paddler sind durch folgenden Datentyp repräsentiert:

```
class Result {
  String name;
  double time;

  Result( String name, double time ) {
    this.name = name;
    this.time = time;
  }
}
```

Aufgabe:

- Erzeuge einen `Stream` von `Result`-Objekten. Belege einige `Result`-Objekte mit ausgewählten Werten zum Testen vor.
- Gib am Ende eine kleine Statistik der Zeiten aus.

Beispiel:

Aus dem folgenden Stream …

```
Stream<Result> stream =
    Stream.of( new Result( "Bareil Antos", 124.123 ),
      new Result( "Kimara Cretak", 434.22 ),
      new Result( "Keyla Detmer", 321.34 ), new Result( "Amanda Grayson",
        143.99 ),
      new Result( "Mora Pol", 122.22 ), new Result( "Gen Rhys", 377.23 ) );
```

… kann die Ausgabe so aussehen:

```
count:   6
min:     122,22
max:     434,22
average: 253,85
```

16.3.2 Median berechnen ★ ★

Die Typen `XXXSummaryStatistics` liefern mit `getAverage()` den *arithmetischen Mittelwert*. Der arithmetische Mittelwert berechnet sich aus der Summe der gegebenen Werte geteilt durch die Anzahl der Werte. Es gibt eine Reihe weiterer Mittelwerte, etwa den *geometrischen Mittelwert* oder den *harmonischen Mittelwert*.

Mittelwerte werden häufig in der Statistik genutzt, doch sie haben das Problem, dass sie anfälliger für Ausreißer sind. Die Statistik arbeitet oft mit dem *Median*. Der Median ist der Zentralwert, also der Wert, der in der sortieren Liste »in der Mitte« steht. Zu kleine oder zu große Zahlen stehen am Rand und sind Ausreißer und gehen in den Median nicht ein.

Ist die Anzahl der Werte ungerade, dann gibt es eine natürliche Mitte.

Beispiele:

- In der Liste 9, 11, **11**, 11, 12 steht in der Mitte der Median 11. Wenn die Anzahl der Werte gerade ist, lässt sich der Median aus dem arithmetischen Mittel der beiden mittleren Zahlen definieren.
- In der Liste 10, **10, 12**, 12 ist der Median das arithmetische Mittel aus den Werten 10 und 12, also 11.

Aufgabe:

- Gegeben ist ein `double[]` mit Messwerten. Schreibe eine Methode `double median (double... values)`, die den Median des Arrays mit gerader und ungerader Anzahl berechnet.
- Nutze zur Lösung einen `DoubleStream`, und überlege, ob `limit(…)` und `skip(…)` helfen.

16.3.3 Temperaturstatistiken berechnen und Charts zeichnen ★ ★ ★

Bonny Brain ist gut mit Zahlen, allerdings mag sie Charts viel lieber. Grafisch lassen sich die Daten viel einfacher erfassen, als wenn sie in Textform vorliegen.

Sie bekommt eine Tabelle mit Temperaturdaten und möchte auf den ersten Blick ablesen, wann es am wärmsten ist und ein Urlaub mit der Familie gut möglich ist.

Aufgabe:

Gesucht ist ein Programm, das Temperaturen verarbeiten und darstellen kann. Genauer gesagt:

- Generiere eine Liste von Zufallszahlen, die im besten Fall dem Temperaturverlauf des Jahres folgen, etwa in Form einer Sinuskurve von 0 bis π.
- Generiere für mehrere Jahre zufällige Temperaturwerte, und speichere die Jahre mit den Werten in einem Assoziativspeicher. Nutze den Datentyp `Year` als Schlüssel für eine nach Jahren sortierte `Map`. Bonus: Die Anzahl der Tage entspricht wirklich der Anzahl der Tage in dem Jahr, also 365 oder 366.
- Schreibe eine ASCII-Tabelle mit den Temperaturen aller Jahre auf die Konsole.
- Gib die höchste und niedrigste Jahrestemperatur eines Jahres aus.
- Gib die höchste, niedrigste und durchschnittliche Temperatur für einen Monat eines Jahres aus.
- Generiere eine Datei, in der von einem Jahr die zwölf Durchschnittstemperaturen eines Monats aggregiert und visualisiert werden. Nimm folgendes HTML-Dokument als Grundlage, und fülle das `data`-Array entsprechend:

```
<!DOCTYPE html>
<html lang="de">
<body>
<canvas id="chart" width="500" height="200"></canvas>
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.9.3/
Chart.bundle.min.js"></script>
<script>
const cfg = {
    type: 'bar',
    data: {
        labels: ["Jan", "Feb", "Mrz", "Apr", "Mai", "Jun", "Jul", "Aug",
                 "Sep", "Okt", "Nov", "Dez"],
        datasets: [ {
            label: "Durchschnittswerte", fill: false,
            data: [11.9,17.0,21.3,25.1,27.8,29.1,29.2,27.9,25.6,21.6,
                   17.5,12.5],
        } ]
    },
    options: {
        responsive: true,
        title: { display:true, text:'Temperaturverlauf' },
        tooltips: { mode: 'index', intersect: false },
        hover: { mode: 'nearest', intersect: true },
        scales: {
            xAxes: [ { display: true, scaleLabel: { display: true,
```

16

```
                        labelString: 'Monat' } } ],
              yAxes: [ { display: true, scaleLabel: { display: true,
                        labelString: 'Temperatur' } } ]
          }
      }
  };
  window.onload = () =>
    new Chart(document.getElementById("chart").getContext("2d"), cfg);
</script>
</body>
</html>
```

Kapitel 20

XML, JSON und weitere Datenformate mit Java verarbeiten

Zwei wichtige Datenformate für den Austausch von Dokumenten sind *XML* und *JSON*. XML ist historisch gesehen der ältere Datentyp, JSON finden wir heutzutage oft in der Kommunikation zwischen einem Server und einer JavaScript-Anwendung. JSON-Dokumente werden auch gerne für Konfigurationsdateien verwendet.

Während die Java SE unterschiedliche Klassen zum Lesen und Schreiben von XML-Dokumenten bietet, ist die Unterstützung von JSON nur in der Java Enterprise Edition oder durch ergänzende Open-Source-Bibliotheken gegeben. Viele der Aufgaben in diesem Kapitel greifen daher zu externen Bibliotheken.

Eine wichtige Kategorie der Dokumentenformate bilden Beschreibungssprachen. Sie definieren die Struktur der Daten. Zu den wichtigsten Formaten zählen HTML, XML, JSON und PDF.

Java bringt bis auf die Unterstützung der Property-Dateien und der Möglichkeit, ZIP-Archive zu verarbeiten, keine Unterstützung für andere Datenformate mit. Das gilt insbesondere auch für CSV-Dateien, PDFs oder Office-Dokumente. Glücklicherweise füllen Dutzende Open-Source-Bibliotheken diese Lücke, sodass man diese Funktionalität nicht selbst programmieren muss.

Voraussetzungen

- Maven-Dependencies hinzunehmen können
- StAX kennen
- XML-Dokumente schreiben können
- JAXB-Beans aus XML-Schema-Dateien erzeugen können
- Objekt-XML-Mapping mit JAXB einsetzen können
- Einarbeitung in JSON-Bibliothek Jackson
- ZIP-Archive auslesen können

Java SE 9 Standard-Bibliothek, Kapitel 9 und Kapitel 10

In der »2. Insel« führen Kapitel 9, »Datenformate«, und Kapitel 10 speziell zu XML und JSON in das Thema ein, lassen aber Open-Source-Bibliotheken außen vor.

Verwendete Datentypen in diesem Kapitel:

- `javax.xml.stream.XMLOutputFactory` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/stream/XMLOutputFactory.html*)
- `javax.xml.stream.XMLStreamWriter` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/stream/XMLStreamWriter.html*)
- `javax.xml.stream.XMLStreamException` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/stream/XMLStreamException.html*)
- `javax.xml.stream.XMLInputFactory` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/stream/XMLInputFactory.html*)
- `javax.xml.stream.XMLStreamReader` (*https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/stream/XMLStreamReader.html*)
- `javax.xml.bind.JAXB` (*https://docs.oracle.com/javaee/7/api/javax/xml/bind/JAXB.html*)
- `javax.xml.bind.DataBindingException` (*https://docs.oracle.com/javaee/7/api/javax/xml/bind/DataBindingException.html*)
- *https://github.com/FasterXML/jackson*

20.1 XML-Verarbeitung mit Java

Es gibt unterschiedliche Java-APIs zum Umgang mit XML-Dokumenten. Eine Möglichkeit ist das Halten von kompletten XML-Objekten im Speicher, die andere Lösung erinnert an Datenströme. *StAX* ist eine Pull-API, mit der die Elemente aktiv aus dem Datenstrom gezogen und auch geschrieben werden können. Das Verarbeitungsmodell eignet sich optimal für große Dokumente, die nicht komplett im Speicher stehen müssen.

JAXB bietet eine einfache Möglichkeit, Java-Objekte in XML zu konvertieren und XML später wieder in Java-Objekte. Mithilfe von Annotationen oder externen Konfigurationsdateien lässt sich die Abbildung präzise steuern.

20.1.1 XML-Datei mit Rezept schreiben ★

Captain CiaoCiao hat so viele Rezepte, dass er eine Datenbank benötigt. Es liegen ihm verschiedene Angebote für Datenbankmanagementsysteme vor, und er möchte schauen, ob sie alle seine Rezepte importieren können.

Seine eigenen Rezepte liegen im RecipeML-Format vor, einem XML-Format, das lose spezifiziert ist: *http://www.formatdata.com/recipeml/*. Unter *https://dsquirrel.tripod.com/recipeml/indexrecipes2.html* gibt es eine große Datenbank. Ein Beispiel von »Key Gourmet«:


```
<?xml version="1.0" encoding="UTF-8"?>
<recipeml version="0.5">
  <recipe>
    <head>
      <title>11 Minute Strawberry Jam</title>
      <categories>
        <cat>Canning</cat>
        <cat>Preserves</cat>
        <cat>Jams & jell</cat>
      </categories>
      <yield>8</yield>
    </head>
    <ingredients>
      <ing>
        <amt>
          <qty>3</qty>
          <unit>cups</unit>
        </amt>
        <item>Strawberries</item>
      </ing>
      <ing>
        <amt>
          <qty>3</qty>
          <unit>cups</unit>
        </amt>
        <item>Sugar</item>
      </ing>
    </ingredients>
    <directions>
      <step>Put the strawberries in a pan.</step>
      <step>Add 1 cup of sugar.</step>
      <step>Bring to a boil and boil for 4 minutes.</step>
      <step>Add the second cup of sugar and boil again for 4 minutes.</step>
      <step>Then add the third cup of sugar and boil for 3 minutes.</step>
      <step>Remove from stove, cool, stir occasionally.</step>
      <step>Pour in jars and seal.</step>
    </directions>
  </recipe>
</recipeml>
```

Aufgabe:

Schreibe ein Programm, das ein XML-Dokument im RecipeML-Format ausgibt.

20.1.2 Prüfen, ob alle Bilder ein alt-Attribut haben ★

Bilder in HTML-Dokumenten sollten immer ein `alt`-Attribut haben.

Aufgabe:

- Implementiere einen XHTML-Prüfer, der meldet, ob jedes `img`-Tag ein Attribut `alt` gesetzt hat.
- Nimm als XHTML-Datei z. B. *http://tutego.de/download/index.xhtml*.

20.1.3 Java-Objekte mit JAXB schreiben ★

JAXB vereinfacht den Zugriff auf XML-Dokumente, denn es erlaubt eine praktische Abbildung von einem Java-Objekt auf ein XML-Dokument und umgekehrt.

JAXB wurde in der Java 6 in die Standard Edition aufgenommen und in Java 11 wieder entfernt. Um für aktuelle Java-Versionen vorbereitet zu sein, binde Folgendes in die POM-Datei ein:

```
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.3</version>
  <scope>runtime</scope>
</dependency>
```

Aufgabe:

- Schreibe JAXB-Beans, damit wir folgendes XML erzeugen können:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ingredients>
    <ing>
        <amt>
            <qty>3</qty>
            <unit>cups</unit>
        </amt>
        <item>Sugar</item>
    </ing>
    <ing>
        <amt>
            <qty>3</qty>
            <unit>cups</unit>
```



```
        </amt>
    </ing>
</ingredients>
```

- Lege die Klassen `Ingredients`, `Ing`, `Amt` an.
- Gibt den Klassen entsprechende Objektvariablen; es ist in Ordnung, wenn diese `public` sind.
- Überlege, welche Annotation eingesetzt werden muss.

20.1.4 Witze einlesen und herzlich lachen ★★

Bonny Brain lacht auch über einfache Witze, wovon sie nie genug haben kann. Sie findet im Internet die Seite *https://sv443.net/jokeapi/v2/joke/Any?format=xml*, die ihr immer neue Witze liefert.

Das Format ist XML, das gut für den Datentransport ist, aber wir sind Java-Entwickler und wünschen uns alles in Objekten! Mit JAXB sollen die XML-Dateien eingelesen und in Java-Objekt konvertiert werden, sodass wir später eine individuelle Ausgabe entwickeln können.

Im ersten Schritt sollen aus einer XML-Schema-Datei JAXB-Beans automatisch generiert werden. Das Schema für die Joke-Seite ist wie folgt – keine Angst, man muss das nicht verstehen.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="data">
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element type="xs:string" name="category" />
        <xs:element type="xs:string" name="type" />
        <xs:element name="flags">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:boolean" name="nsfw" />
              <xs:element type="xs:boolean" name="religious" />
              <xs:element type="xs:boolean" name="political" />
              <xs:element type="xs:boolean" name="racist" />
              <xs:element type="xs:boolean" name="sexist" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element type="xs:string" name="setup" />
        <xs:element type="xs:string" name="delivery" />
        <xs:element type="xs:int" name="id" />
        <xs:element type="xs:string" name="error" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Der Anbieter bietet kein Schema, daher ist es mithilfe von *https://www.freeformatter.com/xsd-generator.html* aus dem XML generiert.

Aufgabe:

- Lade die XML-Schema-Definition unter *http://tutego.de/download/jokes.xsd*, und setze die Datei in das Maven-Verzeichnis */src/main/resources*.
- Ergänze die POM-Datei um folgendes Element:

```
<build>
<plugins>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxb2-maven-plugin</artifactId>
    <version>2.5.0</version>
    <executions>
      <execution>
        <id>xjc</id>
        <goals>
          <goal>xjc</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <packageName>com.tutego.exercise.xml.joke</packageName>
      <sources>
        <source>src/main/resources/jokes.xsd</source>
      </sources>
      <generateEpisode>false</generateEpisode>
      <outputDirectory>${basedir}/src/main/java</outputDirectory>
      <clearOutputDir>false</clearOutputDir>
      <noGeneratedHeaderComments>true</noGeneratedHeaderComments>
      <locale>en</locale>
    </configuration>
  </plugin>
</plugins>
</build>
```

Die Plugin-Sektion bindet `org.codehaus.mojo:jaxb2-maven-plugin` ein und konfiguriert es; alle Optionen sind unter *https://www.mojohaus.org/jaxb2-maven-plugin/Documentation/v2.5.0/index.html* erklärt.

- Starte von der Kommandozeile `mvn generate-sources`. Es entstehen zwei Klassen im Paket `com.tutego.exercise.xml.joke`:
  - `Data`
  - `ObjectFactory`
- Nutze JAXB, um von der URL *https://sv443.net/jokeapi/v2/joke/Any?format=xml* einen Witz zu beziehen und in ein Objekt zu konvertieren.

20.2 JSON

Die Java SE bringt keine Unterstützung für JSON mit, die Jakarta EE schon. Eine beliebte Implementierung ist *Jackson*, das auch Java-Objekte in JSON abbilden und aus JSON-Objekten wieder Java-Objekte rekonstruieren kann.

Jackson ist gut modularisiert. Es gibt drei Kernmodule – *Streaming*, *Annotations* und *Databind* – und mehrere Drittmodule, insbesondere für speziellere Datentypen wie von Guava, `javax.money` und Ergänzungen etwa zur Performanceoptimierung, die zum Beispiel Bytecode generieren, statt auf Reflection zu setzen.

Das Modul Databind hat eine Abhängigkeit auf Streaming und Annotations, sodass Entwickler damit alle Kernfunktionalitäten bekommen. Nimm in das Maven-POM folgende Dependency mit auf:

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.12.3</version>
</dependency>
```

20.2.1 Hacker News: JSON auswerten ★

Die Seite Hacker News (*https://news.ycombinator.com*) wurde in Kapitel 19, »Netzwerkprogrammierung«, kurz vorgestellt.

Die URL *https://hacker-news.firebaseio.com/v0/item/24857356.json* liefert ein JSON-Objekt der Nachricht mit der ID 24857356. Die Antwort sieht (formatiert und bei den kids etwas gekürzt) so aus:

```
{
   "by":"luu",
   "descendants":257,
   "id":24857356,
   "kids":[
      24858151,
      24857761,
      24858192,
      24858887
   ],
   "score":353,
   "time":1603370419,
   "title":"The physiological effects of slow breathing in the healthy human",
   "type":"story",
   "url":"https://breathe.ersjournals.com/content/13/4/298"
}
```

Mit Jackson lässt sich dieses JSON in eine Map konvertieren:

```
ObjectMapper mapper = new ObjectMapper();
Map map = mapper.readValue( src, Map.class );
```

Es sind src verschiedene Quellen für die Daten, etwa vom Typ String, File, Reader, InputStream, URL ...

Aufgabe:

Schreibe eine neue Methode `Map<?, ?> news(long id)`, die mithilfe von Jackson das JSON-Dokument unter `"https://hacker-news.firebaseio.com/v0/item/" + id + ".json"` bezieht und in eine Map konvertiert und zurückliefert.

Beispiel:

- `news(24857356).get("title")` → `"The physiological effects of slow breathing in the healthy human"`
- `news(111111).get("title")` → `null`

20.2.2 Editor-Konfigurationen als JSON lesen und schreiben ★★

Die Entwickler arbeiten für Captain CiaoCiao an einem neuen Editor, und die Konfigurationen sollen in einer JSON-Datei gesichert werden.

Aufgabe:

- Schreibe eine Klasse Settings, sodass sich die folgenden Konfigurationen abbilden lassen:

```
{
  "editor" : {
    "cursorStyle" : "line",
    "folding" : true,
    "fontFamily" : [ "Consolas, 'Courier New', monospace" ],
    "fontSize" : 22, "fontWeight" : "normal"
  },
  "workbench" : {
    "colorTheme" : "Default Dark+"
  },
  "terminal" : {
    "integrated.unicodeVersion" : "11"
  }
}
```

- Die JSON-Datei lässt die Datentypen gut erkennen:

  cursorStyle ist String, folding ist boolean, fontFamily ist ein Array oder List.
- Wenn ein Attribut nicht gesetzt ist, also null ist, soll es nicht geschrieben werden.
- Bei terminal sind die enthaltenen Schlüsselwerte unbekannt, sie sollen in einer `Map<String, String>` enthalten sein.

20.3 HTML

HTML ist eine wichtige Auszeichnungssprache. Die Java-Standardbibliothek bringt keine Unterstützung für HTML-Dokumente mit, sieht man einmal von dem ab, was die javax.swing.JEditorPane kann, nämlich HTML 3.2 und eine Teilmenge von CSS 1.0 darstellen.

Damit Java-Programme in der Lage sind, HTML-Dokumente korrekt und valide zu schreiben und einzulesen und Knoten auszulesen, müssen wir zu (Open-Source-) Bibliotheken greifen.

20.3.1 Mit jsoup Wikipedia-Bilder laden ★★

Die beliebte quelloffene Bibliothek *jsoup* (*https://jsoup.org/*) lädt den Inhalt von Webseiten und repräsentiert den Inhalt in einem Baum im Speicher.

Nimm folgende Dependency mit in das POM auf:

```
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.13.1</version>
</dependency>
```

Aufgabe:

- Studiere die Beispiele unter *https://jsoup.org/cookbook/extracting-data/dom-navigation* und *https://jsoup.org/cookbook/extracting-data/selector-syntax*.
- Erfrage von der Wikipedia-Hauptseite alle Bilder, und speichere sie im eigenen Dateisystem.

20.4 Office-Dokumente

Microsoft Office steht weiterhin ganz oben, wenn es um Textverarbeitung und Tabellenkalkulation geht. Seit vielen Jahren ist das binäre Dateiformat wohlbekannt, und es gibt Java-Bibliotheken zum Lesen und Schreiben. Die Verarbeitung von Microsoft-Office-Dokumenten ist deutlich einfacher geworden, seitdem die Dokumente im Kern XML-Dokumente sind, die in einem ZIP-Archiv zusammengefasst werden. Die Unterstützung in Java ist sehr gut.

20.4.1 Word-Dateien mit Screenshots generieren ★★

Lies den Wikipedia-Eintrag zu POI: *https://de.wikipedia.org/wiki/Apache_POI*.

Aufgabe:

1. Ergänze für Maven im POM Folgendes, damit Apache POI und die nötigen Abhängigkeiten für DOCX eingebunden werden:

```
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-ooxml</artifactId>
```



```
  <version>5.0.0</version>
</dependency>
```

2. Studiere den Quellcode von *SimpleImages.java* unter *http://svn.apache.org/repos/asf/poi/trunk/poi-examples/src/main/java/org/apache/poi/examples/xwpf/usermodel/SimpleImages.java*.
3. Mit Java lassen sich Screenshots aufnehmen, und zwar so:

```
private static byte[] getScreenCapture() throws AWTException, IOException {
  BufferedImage screenCapture = new Robot().createScreenCapture(
    SCREEN_SIZE );
  ByteArrayOutputStream os = new ByteArrayOutputStream();
  ImageIO.write( screenCapture, "jpeg", os );
  return os.toByteArray();
}
```

4. Schreibe ein Java-Programm, das 20 Sekunden lang alle 2 Sekunden einen Screenshot macht und das Bild in das Word-Dokument hängt.