1 Einleitung

Herzlich willkommen zu diesem Übungsbuch! Bevor Sie loslegen, möchte ich kurz darstellen, was Sie bei der Lektüre erwartet.

Dieses Buch behandelt verschiedene praxisrelevante Themengebiete und deckt diese durch Übungsaufgaben unterschiedlicher Schwierigkeitsstufen ab. Die Übungsaufgaben sind (größtenteils) voneinander unabhängig und können je nach Lust und Laune oder Interesse in beliebiger Reihenfolge gelöst werden.

Neben den Aufgaben finden sich die jeweiligen Lösungen inklusive einer kurzen Beschreibung des zur Lösung verwendeten Algorithmus sowie dem eigentlichen, an wesentlichen Stellen kommentierten Sourcecode.

1.1 Aufbau der Kapitel

Jedes Kapitel ist strukturell gleich aufgebaut, sodass Sie sich schnell zurechtfinden werden.

Einführung

Ein Kapitel beginnt jeweils mit einer Einführung in die jeweilige Thematik, um auch diejenigen Leser abzuholen, die mit dem Themengebiet vielleicht noch nicht so vertraut sind, oder aber, um Sie auf die nachfolgenden Aufgaben entsprechend einzustimmen.

Aufgaben

Danach schließt sich ein Block mit Übungsaufgaben und folgender Struktur an:

Aufgabenstellung Jede einzelne Übungsaufgabe besitzt zunächst eine Aufgabenstellung. Dort werden in wenigen Sätzen die zu realisierenden Funktionalitäten beschrieben. Oftmals wird auch schon eine mögliche Methodensignatur als Anhaltspunkt zur Lösung angegeben.

Beispiele Ergänzend finden sich fast immer Beispiele zur Verdeutlichung mit Eingaben und erwarteten Ergebnissen. Nur für einige recht einfache Aufgaben, die vor allem zum Kennenlernen eines APIs dienen, wird mitunter auf Beispiele verzichtet.

Oftmals werden in einer Tabelle verschiedene Wertebelegungen von Eingabeparameter(n) sowie das erwartete Ergebnis dargestellt, etwa wie folgt:

Eingabe A	Eingabe B	Ergebnis
[1, 2, 4, 7, 8]	[2, 3, 7, 9]	[2, 7]

Für die Angaben gelten folgende Notationsformen:

- "AB" – steht für textuelle Angaben
- true / false – repräsentieren boolesche Werte
- 123 – Zahlenangaben
- [value1, value2, ...] – steht für Collections wie Sets oder Listen, aber auch Arrays
- { key1 : value1, key2 : value3, ... } – beschreibt Maps

Lösungen

Auch der Teil der Lösungen besitzt die nachfolgend beschriebene Struktur.

Aufgabenstellung und Beispiele Zunächst finden wir nochmals die Aufgabenstellung, sodass wir nicht ständig zwischen Aufgaben und Lösungen hin- und herblättern müssen, sondern das Ganze in sich abgeschlossen ist.

Algorithmus Danach folgt eine Beschreibung des gewählten Algorithmus zur Lösung. Aus Gründen der Didaktik zeige ich bewusst auch einmal einen Irrweg oder eine nicht so optimale Lösung, um daran dann Fallstricke aufzudecken und iterativ zu einer Verbesserung zu kommen. Tatsächlich ist die eine oder andere Brute-Force-Lösung manchmal sogar schon brauchbar, bietet aber Optimierungspotenziale. Exemplarisch werde ich immer wieder entsprechende, mitunter verblüffend einfache, aber oft auch sehr wirksame Verbesserungen vorstellen.

Prüfung Teilweise sind die Aufgaben recht leicht oder dienen nur dem Kennenlernen von Syntax oder API-Funktionalität. Dafür scheint es mir oftmals ausreichend, ein paar Aufrufe direkt in der JShell auszuführen. Deshalb verzichte ich hierfür auf Unit Tests. Gleiches gilt auch, wenn wir bevorzugt eine grafische Aufbereitung einer Lösung, etwa die Darstellung eines Sudoku-Spielfelds zur Kontrolle nutzen und der korrespondierende Unit Test vermutlich schwieriger verständlich wäre.

Je komplizierter allerdings die Algorithmen werden, desto mehr lauern auch Fehlerquellen, wie falsche Indexwerte, eine versehentliche oder unterbliebene Negation oder ein übersehener Randfall. Deswegen bietet es sich an, Funktionalitäten mithilfe von Unit Tests zu überprüfen – in diesem Buch kann das aus Platzgründen natürlich nur exemplarisch für wichtige Eingaben geschehen. Insgesamt existieren jedoch über 90 Unit Tests mit rund 750 Testfällen. Ein ziemlich guter Anfang. Trotzdem sollte in der Praxis das Netz an Unit Tests und Testfällen wenn möglich noch umfangreicher sein.

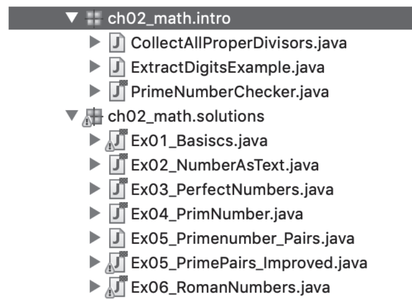
1.2 Grundgerüst des Eclipse-Projekts

Auch das mitgelieferte Eclipse-Projekt orientiert sich in seinem Aufbau an demjenigen des Buchs und bietet für die Kapitel mit Übungsaufgaben jeweils ein eigenes Package pro Kapitel, z. B. `ch02_math` oder `ch08_recursion_advanced`. Dabei weiche ich ausnahmsweise von der Namenskonvention für Packages ab, weil ich die Unterstriche in diesem Fall für eine lesbare Notation halte.

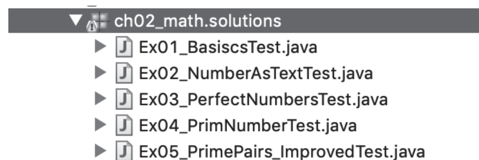
Einige der Sourcecode-Schnipsel aus den jeweiligen Einführungen finden sich in einem Subpackage `intro`. Die bereitgestellten (Muster-)Lösungen werden in jeweils eigenen Subpackages namens `solutions` gesammelt und die Klassen sind gemäß Aufgabenstellung wie folgt benannt: `Ex<Nr>_<Aufgabenstellung>`.

Das gesamte Projekt folgt dem Maven-Standardverzeichnisaufbau und somit finden siche die Sourcen unter `src/main/java` und die Tests unter `src/test/java`.

Sourcen – `src/main/java` Nachfolgend ist ein Ausschnitt für das Kapitel 2 gezeigt:



Test-Klassen – `src/test/java` Exemplarisch hier einige dazugehörige Tests:



Utility-Klassen Alle in den jeweiligen Kapiteln entwickelten nützlichen Utility-Methoden sind im bereitgestellten Eclipse-Projekt in Form von Utility-Klassen enthalten. Beispielsweise implementieren wir in Kapitel 5 einige hilfreiche Methoden, unter anderem `swap()` und `find()` (alle in Abschnitt 5.1.1). Diese kombinieren wir dann in einer Klasse `ArrayUtils`, die in einem eigenen Subpackage `util` liegt – für das Kapitel zu Arrays im Subpackage `ch05_arrays.util`. Gleiches gilt für die anderen Kapitel und Themengebiete.

1.3 Grundgerüst für die Unit Tests

Um den Rahmen des Buchs nicht zu sprengen, zeigen die abgebildeten Unit Tests jeweils nur die Testmethoden, jedoch nicht die Testklasse und die Imports. Damit Sie ein Grundgerüst haben, in das Sie die Testmethoden einfügen können sowie als Ausgangspunkt für eigene Experimente, ist nachfolgend eine typische Testklasse gezeigt:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.time.LocalDate;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.ValueSource;
import org.junit.jupiter.params.provider.MethodSource;

public class SomeUnitTests
{
    @ParameterizedTest(name = "value at pos {index} ==> {0} should be perfect")
    @ValueSource(ints = { 6, 28, 496, 8128 } )
    void testIsPerfectNumberSimple(int value)
    {
        assertTrue(Ex03_PerfectNumbers.isPerfectNumberSimple(value));
    }

    @ParameterizedTest
    @CsvSource({"2017-01-01, 2018-01-01, 53", "2019-01-01, 2019-02-07, 5"})
    void testAllSundaysBetween(LocalDate start, LocalDate end, int expected)
    {
        var result = Ex09_CountSundaysExample.allSundaysBetween(start, end);

        assertEquals(expected, result.count());
    }

    @ParameterizedTest(name = "calcPrimes({0}) = {1}")
    @MethodSource("argumentProvider")
    void testCalcPrimesBelow(int n, List<Integer> expected)
    {
        List<Integer> result = Ex04_PrimNumber.calcPrimesBelow(n);

        assertEquals(expected, result);
    }

    // Parameter sind Listen von Werten => Stream<Arguments>
    static Stream<Arguments> argumentProvider()
    {
        return Stream.of(Arguments.of(2, List.of(2)),
                         Arguments.of(3, List.of(2, 3)),
                         Arguments.of(10, List.of(2, 3, 5, 7)),
                         Arguments.of(15, List.of(2, 3, 5, 7, 11, 13)));
    }
}
```

Neben den Imports und den ausgiebig genutzten parametrisierten Tests, die das Prüfen mehrerer Wertkombinationen auf einfache Weise erlauben, ist hier die Bereitstellung der Testeingaben über `@CsvSource` und `@MethodSource` in Kombination mit einem `Stream<Arguments>` gezeigt. Für Details schauen Sie bitte in Anhang B.

1.4 Anmerkung zum Programmierstil

In diesem Abschnitt möchte ich noch vorab etwas zum Programmierstil sagen, weil in Diskussionen ab und an einmal die Frage aufkam, ob man gewisse Dinge nicht kompakter gestalten sollte.

Gedanken zur Sourcecode-Kompaktheit

In der Regel sind mir beim Programmieren und insbesondere für die Implementierungen in diesem Buch vor allem eine leichte Nachvollziehbarkeit sowie eine übersichtliche Strukturierung und damit später eine vereinfachte Wartbarkeit und Veränderbarkeit wichtig. Deshalb sind die gezeigten Implementierungen möglichst verständlich programmiert und vermeiden etwa den ?-Operator für komplexere Ausdrücke. Dadurch ist vielleicht nicht jedes Konstrukt maximal kompakt, dafür aber in der Regel gut verständlich. Diesem Aspekt möchte ich in diesem Buch den Vorrang geben. Auch in der Praxis kann man damit oftmals besser leben als mit einer schlechten Wartbarkeit, dafür aber einer kompakteren Programmierung.

Beispiel 1

Schauen wir uns zur Verdeutlichung ein kleines Beispiel an. Zunächst betrachten wir die lesbare, gut verständliche Variante zum Umdrehen des Inhalts eines Strings, die zudem sehr schön die beiden wichtigen Elemente des rekursiven Abbruchs und Abstiegs verdeutlicht:

```
static String reverseString(final String input)
{
    // rekursiver Abbruch
    if (input.length() <= 1)
        return input;

    final char firstChar = input.charAt(0);
    final String remaining = input.substring(1);

    // rekursiver Abstieg
    return reverseString(remaining) + firstChar;
}
```

Die folgende deutliche kompaktere Variante bietet diese Vorteile nicht:

```
static String reverseStringShort(final String input)
{
    return input.length() <= 1 ? input :
           reverseStringShort(input.substring(1)) + input.charAt(0);
}
```

Überlegen Sie kurz, in welcher der beiden Methoden Sie sich sicher fühlen, eine nachträgliche Änderung vorzunehmen. Und wie sieht es aus, wenn Sie noch Unit Tests ergänzen wollen: Wie finden Sie passende Wertebelegungen und Prüfungen?

Außerdem sollte man bedenken, dass die obere Variante entweder bereits während der Kompilierung (Umwandlung in den Bytecode) oder später während der Ausführung und Optimierung automatisch in etwas Ähnliches wie die untere Variante konvertiert wird – eben mit dem Vorteil der besseren Lesbarkeit beim Programmieren.

Beispiel 2

Lassen Sie mich noch ein weiteres Beispiel anbringen, um meine Aussage zu verdeutlichen. Nehmen wir folgende Methode `countSubstrings()`, die die Anzahl der Vorkommen eines Strings in einem anderen zählt und für die beiden Eingaben "halloha" und "ha" das Ergebnis 2 liefert.

Zunächst implementieren wir das einigermaßen geradeheraus wie folgt:

```
static int countSubstrings(final String input, final String valueToFind)
{
    // rekursiver Abbruch
    if (input.length() < valueToFind.length())
        return 0;

    int count;
    String remaining;

    // startet der Text mit der Suchzeichenfolge?
    if (input.startsWith(valueToFind))
    {
        // Treffer: Setze die Suche nach dem gefundenen
        // Begriff nach der Fundstelle fort
        remaining = input.substring(valueToFind.length());
        count = 1;
    }
    else
    {
        // entferne erstes Zeichen und suche erneut
        remaining = input.substring(1);
        count = 0;
    }

    // rekursiver Abstieg
    return countSubstrings(remaining, valueToFind) + count;
}
```

Schauen wir uns an, wie man dies kompakt zu realisieren versuchen könnte:

```
static int countSubstringsShort(final String input, final String valueToFind)
{
    return input.length() < valueToFind.length() ? 0 :
           (input.startsWith(valueToFind) ? 1 : 0) +
           countSubstringsShort(input.substring(1), valueToFind);
}
```

Würden Sie lieber in dieser Methode oder der zuvor gezeigten ändern?

Übrigens: Die untere enthält noch eine subtile funktionale Abweichung! Bei den Eingaben von "XXXX" und "XX" »konsumiert« die erste Variante immer die Zeichen und findet zwei Vorkommen. Die untere bewegt sich aber jeweils nur um ein Zeichen weiter und findet somit drei Vorkommen.

Und weiter: Die Integration der oben realisierten Funktionalität des Weiterschiebens um den gesamten Suchstring in die zweite Variante wird zu immer undurchsichtigerem Sourcecode führen. Dagegen kann man oben das Weiterschieben um nur ein Zeichen einfach durch Anpassen des oberen `substring(valueToFind.length())`-Aufrufs umsetzen und diese Funktionalität dann sogar aus dem `if` herausziehen.

Gedanken zu `final` und `var`

Normalerweise bevorzuge ich, unveränderliche Variablen als `final` zu markieren. In diesem Buch verzichte ich mitunter darauf, vor allem in Unit Tests, um diese so kurz wie möglich zu halten. Ein weiterer Grund ist, dass die JShell das Schlüsselwort `final` nicht überall unterstützt, glücklicherweise aber an den wichtigen Stellen, nämlich für Parameter und lokale Variablen.

Local Variable Type Inference – `var` Seit Java 10 existiert die sogenannte Local Variable Type Inference, besser bekannt als `var`. Diese erlaubt es, auf die explizite Typangabe auf der linken Seite einer Variablendefinition zu verzichten, sofern sich der konkrete Typ für eine lokale Variable anhand der Definition auf der rechten Seite der Zuweisung vom Compiler ermitteln lässt:

```
var name = "Peter";                // var => String
var chars = name.toCharArray();    // var => char[]

var mike = new Person("Mike", 47); // var => Person
var hash = mike.hashCode();        // var => int
```

Insbesondere im Zusammenhang mit generischen Containern spielt die Local Variable Type Inference ihre Vorteile aus:

```
// var => ArrayList<String>
var names = new ArrayList<String>();
names.add("Tim");
names.add("Tom");
names.add("Jerry");

// var => Map<String, Long>
var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L,
                              "Michael", 47L, "Max", 25L);
```

Konvention: `var` falls lesbarer Sofern die Verständlichkeit darunter nicht leidet, werde ich `var` an geeigneter Stelle verwenden, um den Sourcecode kürzer und klarer zu halten. Ist jedoch eine Typangabe für das Nachvollziehen von größerer Wichtigkeit, bevorzuge ich den konkreten Typ und vermeide `var` – die Grenzen sind aber fließend.

Konvention: `final` oder `var` Eine weitere Anmerkung noch: Zwar kann man `final` und `var` kombinieren, ich finde dies jedoch stilistisch nicht schön und verwende entweder das eine oder das andere.

Anmerkungen zu Methodensichtbarkeiten

Vielleicht fragen Sie sich, wieso die vorgestellten Methoden nicht als `public` markiert sind. Die in diesem Buch vorgestellten Methoden sind oftmals ohne Sichtbarkeitsmodifier dargestellt, da sie vor allem in demjenigen Package und Kontext aufgerufen werden, wo sie definiert sind. Dadurch sind sie sowohl für die begleitenden Unit Tests als auch für Experimente in der JShell problemlos aufrufbar. Manchmal extrahiere ich aus der Implementierung der Übungsaufgabe zur besseren Strukturierung und Lesbarkeit einige Hilfsmethoden. Diese sind dann in der Regel `private`, um diesen Umstand entsprechend auszudrücken.

Tatsächlich fasse ich die wichtigsten Hilfsmethoden in speziellen Utility-Klassen zusammen, wo diese für andere Packages natürlich `public` und `static` sind, um den Zugriff zu ermöglichen.

Blockkommentare in Listings

Beachten Sie bitte, dass sich in den Listings diverse Blockkommentare finden, die der Orientierung und dem besseren Verständnis dienen. In der Praxis sollte man derartige Kommentierungen mit Bedacht einsetzen und lieber einzelne Sourcecode-Abschnitte in Methoden auslagern. Für die Beispiele des Buchs dienen diese Kommentare aber als Anhaltspunkte, weil die eingeführten oder dargestellten Sachverhalte für Sie als Leser vermutlich noch neu und ungewohnt sind.

```
// Prozess erzeugen
final String command = "sleep 60s";
final String commandWin = "cmd timeout 60";
final Process sleeper = Runtime.getRuntime().exec(command);
//    ...

// Process => ProcessHandle
final ProcessHandle sleeperHandle = ProcessHandle.of(sleeper.pid()).
                                    orElseThrow(IllegalStateException::new);
//    ...
```

Gedanken zur Formattierung

Die in den Listings genutzte Formatierung weicht leicht von den Coding Conventions von Oracle[1] ab. Ich orientiere mich an denjenigen von Scott Ambler[2], der insbesondere (öffnende) Klammern in jeweils eigenen Zeilen vorschlägt. Dazu habe ich ein spezielles Format namens `Michaelis_CodeFormat` erstellt. Dieses ist im Projekt-Download integriert.

[1] http://www.oracle.com/technetwork/java/codeconv-138413.html
[2] http://www.ambysoft.com/essays/javaCodingStandards.html

1.5 Ausprobieren der Beispiele und Lösungen

Grundsätzlich verwende ich möglichst nachvollziehbare Konstrukte und keine ganz besonders ausgefallenen Syntax- oder API-Features spezieller Java-Versionen. Sofern nicht explizit im Text erwähnt, sollten Sie die Beispiele und Lösungen daher mit der aktuellen LTS-Version Java 11 ausprobieren können. In wenigen Ausnahmen setze ich allerdings Syntaxerweiterungen aus Java 14 ein, weil diese das tägliche Programmiererleben deutlich einfacher und angenehmer gestalten.

Besonderheiten für Java-14-Previews Durch die mittlerweile kurzen Abstände von 6 Monaten zwischen den Java-Releases werden der Entwicklergemeinde einige Features als Previews vorgestellt. Möchte man diese nutzen, so sind in den IDEs und Build-Tools gewisse Parametrierungen sowohl beim Kompilieren als auch beim Ausführen nötig, etwa wie im folgenden Beispiel:

```
java --enable-preview -cp build/libs/Java14Examples.jar \
     java14.RecordExamples
```

Weitere Details rund um Java 14 stelle ich in meinem Buch »Java – die Neuerungen in Version 9 bis 14: Modularisierung, Syntax- und API-Erweiterungen« [3] vor.

Ausprobieren mit JShell, Eclipse und als JUnit-Test Vielfach können Sie die abgebildeten Sourcecode-Schnipsel einfach in die JShell kopieren und ausführen. Alternativ finden Sie alle relevanten Sourcen in dem zum Buch mitgelieferten Eclipse-Projekt. Dort lassen sich die Programme durch eine `main()`-Methode starten oder – sofern vorhanden – durch korrespondierende Unit Tests überprüfen.

Los geht's: Entdeckungsreise Java Challenge

So, nun ist es genug der Vorrede und Sie sind bestimmt schon auf die ersten Herausforderungen durch die Übungsaufgaben gespannt. Deshalb wünsche ich Ihnen nun viel Freude mit diesem Buch sowie einige neue Erkenntnisse beim Lösen der Übungsaufgaben und beim Experimentieren mit den Algorithmen.

Wenn Sie zunächst eine Auffrischung Ihres Wissens zu JUnit, zur JShell oder der O-Notation benötigen, bietet sich ein Blick in die Anhänge an.