

Aufgabenkatalog Programmieren (Java)

Nane Kratzke

03.09.2019

Inhaltsverzeichnis

1	Einleitung	3
2	UNIT-01	4
2.1	Hello World	4
2.2	Zeichenketten klammern	4
2.3	Zeichen zählen	5
2.4	Zeichenketten formatieren mit <code>endUp()</code>	6
2.5	Verflixtes Zeichenketten formatieren mit <code>stringX()</code>	6
2.6	Zeichen in Zeichenketten zählen	7
3	UNIT-02	8
3.1	HTML (und andere) Tags erzeugen	8
3.2	Letzte Ziffer	9
3.3	Armstrongzahlen	9
3.4	Zeichenketten "rotieren"	10
3.5	Tripple in Zeichenketten finden	10
3.6	Addieren mit der <code>luckySum()</code>	11
3.7	Selbstteilende Zahlen	11
3.8	Vorkommen von Zeichenketten zählen	12
3.9	Zeichenketten verarbeiten mit <code>everyNth()</code>	13
3.10	Ein Passwort-Generator	13
3.11	Zeichenketten prüfen mittels <code>sameStarChar()</code>	14
3.12	Klammerungen prüfen mit <code>checkBrackets()</code>	14
3.13	Sternchen in Zeichenketten tilgen	14
4	UNIT-03	16
4.1	Sortierung von Arrays prüfen	16
4.2	Listen nach geraden und ungeraden Zahlen ordnen	16
4.3	Räume	17
4.4	Primzahlen bestimmen	17
4.5	Armstrongzahlen gruppieren	18
4.6	Aufsteigend sortierte Tripple finden	19
4.7	Toppings	19
4.8	Worthäufigkeit in Zeichenketten bestimmen	20
4.9	Listen von Zeichenketten mittels <code>wordAppend()</code> verarbeiten	20
5	UNIT-04	22
5.1	Zeichenhäufigkeiten in Textdateien bestimmen	22
6	UNIT-05	24
6.1	Rekursives <code>allStar()</code>	24
6.2	Zeichenketten rekursiv bereinigen	24
6.3	Listen rekursiv verarbeiten	25
6.4	Vorkommen von Zeichenketten rekursiv zählen	25
6.5	Geratene Zeichen rekursiv "blanken"	26
6.6	Quersumme rekursiv berechnen	26
6.7	Binärbäume rekursiv verarbeiten	27

6.8	Binärbäume rekursiv prettyprinten	28
6.9	Maximum auf einer verketteten Liste rekursiv bestimmen	30
6.10	Filtern von Listen von Zeichenketten mittels Lambdas	31
6.11	Bestimmen einer Stelle einer Zahl mittels Lambdas	32
6.12	Tabellarische Konsolenausgabe mittels Lambdas	33
6.13	Geratene Zeichen mittels Lambdas "blanken"	33
6.14	Primzahlen mit Lambdas bestimmen (und ausgeben)	34
6.15	Vollkommene Zahlen mittels Lambdas bestimmen	34
7	UNIT-06	36
7.1	Auto-Klasse	36
7.2	Raum-Verwaltung	37
7.3	Possible Chessmen	38
7.4	Fruchtkorb	40
8	UNIT-09	42
8.1	Generische Warteschlange	42
8.2	Generisches Convertable Interface	43
8.3	Eigenständig generische selector() Methode	46
9	Dank an ...	48

Kapitel 1

Einleitung

Programmieren lernt man – wie Fahrrad fahren oder Klavier spielen – nicht vom zusehen oder hören einer Vorlesung. Donald Knuth spricht nicht umsonst von *“The Art of Programming”*. Eine Vorlesung ist wertvoll, um wesentliche Inhalte zu strukturieren und einen Pfad vorzuschlagen, der einem das Erwerben von Programmierfähigkeiten erheblich erleichtern kann. Jedoch ist eine Vorlesung nur eine Seite der Medaille und reicht alleine nicht aus.

Programmieren lernt man nur durch programmieren.
Wer hat schon Rad fahren aus einem Physikbuch gelernt?

Dieser Aufgabenkatalog beinhaltet mehrere Aufgabenblätter, die im Rahmen verschiedener Lehrveranstaltungen durch Studierende im Rahmen praktischer Übungen und Praktika zu bearbeiten sind. Dieser Aufgabenkatalog wird in folgenden Lehrveranstaltungen der Technischen Hochschule Lübeck genutzt:

- **Programmieren I**, Informatik/Softwaretechnik, 1. Semester
- **Grundlagen der Programmierung**, Informationstechnologie und Design, 1. Semester
- **Programmieren II**, Informatik/Softwaretechnik, 2. Semester

Der Aufgabenkatalog wird kontinuierlich fortgeschrieben und beinhaltet auf die einzelnen Units der oben genannten Vorlesungen abgestimmte Aufgaben. Alle Aufgaben sind für den Einsatz in Moodle mit dem Testingframework [JEdUnit](#) im Virtual Programming Lab [VPL](#) konzipiert.

Damit die Aufgaben auch außerhalb einer VPL Umgebung als Übungen bearbeitet werden können, wird dieser Katalog aus diesen VPL Aufgaben automatisch generiert. Die automatische Evaluierung der Aufgaben entfällt dadurch natürlich.

Alle nicht im Rahmen der Lehrveranstaltung oder Übung/Praktika bearbeiteten Aufgaben können im Selbststudium durch die Studierenden bearbeitet werden. Dies wird auch für eine zielgerichtete Klausurvorbereitung empfohlen.

Ich wünsche Ihnen viel Erfolg und Erkenntnisgewinn bei der Bearbeitung der nachfolgenden *“Fingerübungen”*.

Lübeck im März 2019

Nane Kratzke

Kapitel 2

UNIT-01

2.1 Hello World

Entwickeln Sie bitte eine Methode `hello()`, die eine Begrüßung generiert. `hello()` soll ein Begrüßungstext nach folgendem Muster erzeugen: "Hello <name>!"

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Denken Sie über den Einsatz folgender String-Methoden nach. Die Wirkungsweise finden Sie im Handout der Unit 02 oder unter diesem [API Link](#).
- `trim()`

Main.java:

```
class Main {  
  
    public static String hello(String name) {  
        return null;  
    }  
  
    public static void main(String[] args) {  
        String greet = hello("Max");  
        System.out.println(greet); // => "Hello Max!"  
        System.out.println(hello("Moritz")); // => "Hello Moritz!"  
  
        // Achten sie auf die Leerzeichen  
        System.out.println(hello("Maren ")); // => "Hello Maren!"  
        System.out.println(hello(" Tessa")); // => "Hello Tessa!"  
  
        System.out.println(hello("")); // => "Hello!"  
    }  
}
```

2.2 Zeichenketten klammern

Schreiben Sie eine Methode `embedCenter()`, die eine payload Zeichenkette in die Mitte einer anderen Klammer-Zeichenkette setzt.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Beachten Sie Sonderfälle wie leere Klammer und Payload Zeichenketten.
- Denken Sie über den Einsatz folgender String-Methoden nach. Die Wirkungsweise finden Sie im Handout der Unit 02 oder unter diesem [API Link](#).
- substring()
- length()

Main.java:

```
class Main {  
  
    public static String embedCenter(String embed, String payload) {  
        return null;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(embedCenter("<<>>", "Yay")); // => <<Yay>>  
        System.out.println(embedCenter("()", "Yay")); // => (Yay)  
        System.out.println(embedCenter(":-)", "Example")); // :Example-  
    }  
}
```

2.3 Zeichen zählen

Entwickeln Sie bitte eine Methode `countChar()`, die zählt wie oft ein Zeichen in einer Zeichenkette vorkommt. Das Zählen soll case-insensitiv erfolgen (d.h. 'a' ist wie 'A' zu zählen).

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Denken Sie über den Einsatz folgender String-Methoden nach. Die Wirkungsweise finden Sie im Handout der Unit 02 oder unter diesem [API Link](#).
- length()
- toLowerCase() und toUpperCase()
- replaceAll()

Main.java:

```
class Main {  
  
    public static int countChars(char c, String s) {  
        return -1;  
    }  
  
    public static void main(String[] args) {  
        int n = countChars('a', "Abc");  
        System.out.println(n); // => 1  
        System.out.println(countChars('A', "abc")); // => 1  
        System.out.println(countChars('x', "ABC")); // => 0  
        System.out.println(countChars('!', "!!!")); // => 3  
    }  
}
```

2.4 Zeichenketten formatieren mit endUp()

Schreiben Sie eine Methode `endUp()`, die die letzten drei Zeichen einer Zeichenkette groß schreibt.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Denken Sie über den Einsatz folgender String-Methoden nach. Die Wirkungsweise finden Sie im Handout der Unit 02 oder unter diesem [API Link](#).
- `substring()`
- `toUpperCase()`
- `length()`
- Der Einsatz der Kontrollanweisung `if` kann hilfreich sein.

Main.java:

```
class Main {  
  
    public static String endUp(String s) {  
        return null;  
    }  
  
    public static void main(String[] args) {  
        String result = endUp("Hello");  
        System.out.println(result); // => "HeLLo"  
        System.out.println(endUp("Hi there")); // => "Hi thERE"  
        System.out.println(endUp("hi")); // => "HI"  
    }  
}
```

2.5 Verflixtes Zeichenketten formatieren mit stringX()

Entwickeln Sie nun bitte eine Methode `stringX()`, die alle 'x' aus einer Zeichenkette entfernt, es sei denn sie stehen am Anfang oder am Ende der Zeichenkette.

Aufruf Beispiele finden Sie in der `main()`-Methode.

Hinweise:

- Denken Sie über den Einsatz folgender String-Methoden nach. Die Wirkungsweise finden Sie im Handout der Unit 02 oder unter diesem [API Link](#).
- `substring()`
- `replaceAll()`

Main.java:

```
class Main {  
  
    public static String stringX(String s) {  
        return null;  
    }  
  
    public static void main(String[] args) {  
        String result = stringX("xxHix");  
        System.out.println(result); // => xHix  
  
        System.out.println(stringX("abxxxxcd")); // => abcd  
        System.out.println(stringX("xabxxxxcdx")); // => xabcdx  
    }  
}
```

```
}  
}
```

2.6 Zeichen in Zeichenketten zählen

Schreiben Sie eine Methode `stringE()`, die prüft, ob eine beliebige Zeichenkette mindestens ein aber maximal drei 'E' beinhaltet. Die Prüfung soll case-insensitiv erfolgen.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Sie haben bereits Vorkommen von Zeichen gezählt.
- Nutzen Sie den logischen `&&`-Operator (AND).
- Nutzen Sie die Vergleichsoperatoren `>` und `<=`.

Main.java:

```
class Main {  
  
    public static boolean stringE(String s) {  
        return true;  
    }  
  
    public static void main(String[] args) {  
        boolean result = stringE("Earth");  
        System.out.println(result); // => true  
  
        System.out.println(stringE("Nonsense")); // => true  
        System.out.println(stringE("This is nuts")); // => false  
        System.out.println(stringE("This example contains nonsense")); // => false  
    }  
}
```

Kapitel 3

UNIT-02

3.1 HTML (und andere) Tags erzeugen

Entwickeln Sie nun bitte eine Methode `makeTags()`, die HTML-artige Zeichenketten wie "`Yay`" aus zwei Zeichenketten erzeugen kann.

- Eine Zeichenkette beschreibt ein Tag und eine Zeichenkette den Inhalt, der durch dieses Tag gekennzeichnet werden soll.
- Tags werden grundsätzlich klein geschrieben.
- Wird kein Tag angegeben (leere Zeichenkette "" oder null) soll nur der Inhalt zurückgegeben werden.
- Der Inhalt in Tags hat nie führende oder abschließende Leerzeichen.
- Tags wie Inhalte sollen in der Ausgabe keine führenden oder abschließenden Whitespaces haben.

Aufruf Beispiele finden Sie in der `main()`-Methode.

Hinweise:

- Die String Methoden `trim()`, `isEmpty()` und `toLowerCase()` sind vermutlich hilfreich.
- Dadurch letztlich viel einfacher als es aussieht.

Main.java:

```
class Main {

    public static String makeTags(String tag, String content) {
        return null;
    }

    public static void main(String[] args) {
        String result = makeTags("em", " Yay ");
        System.out.println(result); // => "<em>Yay</em>"

        System.out.println(
            makeTags("CITE ", "Programmieren lernt man nur durch programmieren.")
        );
        // => "<cite>Programmieren lernt man nur durch programmieren.</cite>"

        System.out.println(makeTags("", "No tags")); // => No tags
    }
}
```

3.2 Letzte Ziffer

Entwickeln Sie nun bitte eine Methode `lastDigit()`, die für zwei Zahlen (Dezimalnotation) prüft, ob diese dieselbe letzte Ziffer haben.

Aufruf Beispiele finden Sie in der `main()`-Methode.

Hinweise:

- Der Modulo Operator `%` ist sicher hilfreich (siehe Unit 2, arithmetische Operatoren)
- Einen Betrag können Sie mittels `Math.abs()` bestimmen.

Main.java:

```
class Main {  
  
    public static boolean lastDigit(int a, int b) {  
        return false;  
    }  
    public static void main(String[] args) {  
        boolean result = lastDigit(21, 12);  
        System.out.println(result); // => false  
        System.out.println(lastDigit(121, 2001)); // => true  
    }  
}
```

3.3 Armstrongzahlen

Ja, so etwas gibt es. Eine Armstrongzahl ist eine Zahl, deren Summe ihrer Stellen, jeweils potenziert mit ihrer Stellenanzahl, wieder die Zahl selbst ergibt.

Z.B.: $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$

Entwickeln Sie nun bitte eine Methode `isArmstrong()`, die prüft, ob eine Zahl eine Armstrongzahl ist.

Entwickeln Sie zusätzlich eine Methode `countArmstrongs()` die angibt, wieviele Armstrongzahlen es bis zu einer oberen Schranke gibt.

Aufruf Beispiele finden Sie in der `main()`-Methode.

Hinweise:

- https://de.wikipedia.org/wiki/Narzisstische_Zahl
- Mittels `% 10` koennen Sie den Wert der letzten Ziffer bestimmen.
- Mittels `/ 10` koennen Sie eine Zahl um eine Stelle nach rechts "rausschieben".

Main.java:

```
class Main {  
  
    public static boolean isArmstrong(int x) {  
        return false;  
    }  
  
    public static int countArmstrongs(int n) {  
        return -1;  
    }  
  
    public static void main(String[] args) {  
  
        boolean result = isArmstrong(153);  
  
    }  
}
```

```

System.out.println(result); // => true
System.out.println(isArmstrong(999)); // => false

int n = countArmstrongs(100);
System.out.println(n); // => 10
System.out.println(countArmstrongs(153)); // => 11
System.out.println(countArmstrongs(1000)); // => 14
}
}

```

3.4 Zeichenketten "rotieren"

Entwickeln Sie nun bitte eine Methode `rotate()`, die eine Zeichenkette nach links oder rechts "rotiert". Zeichen die links oder rechts aus der Zeichenkette "geschoben" werden, sollen rechts bzw. links wieder "hineingeschoben" werden.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Beachten Sie, dass eine Rotation positiv und negativ sein kann.
- Beachten Sie, dass eine Rotation länger als die eigentliche Zeichenkette sein kann.
- Das Problem lässt sich tatsächlich ohne Schleife lösen (sicherlich aber auch mit ;-).

Main.java:

```

class Main {

    public static String rotate(int n, String s) {
        return null;
    }

    public static void main(String[] args) {
        String result = rotate(2, "Hello");
        System.out.println(result); // => "loHel"
        System.out.println(rotate(3, "Hello")); // => "lloHe"
        System.out.println(rotate(6, "Hello")); // => "oHell"
    }
}

```

3.5 Tripple in Zeichenketten finden

Entwickeln Sie nun bitte eine Methode `noMultiples()`, die prüft, ob in einer Zeichenkette niemals drei (oder mehr) gleiche Zeichen aufeinander folgen.

Verallgemeinern Sie `noMultiples()` nun so, dass die Anzahl der zu wiederholenden Zeichen parametrisiert ist.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Sehen Sie sich noch einmal überladene Methoden in Unit 2 (Methoden) an.

Main.java:

```

class Main {

    public static boolean noMultiples(int n, String s) {
        return true;
    }

    public static boolean noMultiples(String s) {
        return false;
    }

    public static void main(String[] args) {
        boolean result = noMultiples("Hello World");
        System.out.println(result); // => true
        System.out.println(noMultiples("faaantastic")); // => false
        System.out.println(noMultiples(2, "Hello World")); // => false
    }
}

```

3.6 Addieren mit der luckySum()

Entwickeln Sie nun bitte eine Methode `luckySum()`, die eine variable Anzahl von ganzzahligen Parametern solange aufaddiert bis der Wert 13 in einem Parameter auftaucht.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Sehen Sie sich noch einmal variable Parameter in Unit 2 (Methoden) an.

Main.java:

```

class Main {

    public static int luckySum(int... values) {
        return 42;
    }

    public static void main(String[] args) {
        int result = luckySum(5, 6, 13, 8);
        System.out.println(result); // => 11
        System.out.println(luckySum(1, 2, 3, 4, 5)); // => 15
        System.out.println(luckySum(1, 2)); // => 3
        System.out.println(luckySum(13)); // => 0
    }
}

```

3.7 Selbstteilende Zahlen

Entwickeln Sie nun bitte eine Methode `dividesSelf()`, die prüft, ob eine Zahl selbstteilend ist. Eine Zahl ist selbstteilend, wenn alle ihre Stellen die Zahl ganzzahlig teilt.

Da durch den Wert 0 bekanntlich nicht geteilt werden kann, können alle Zahlen mit einer Nullziffer (Dezimalnotation) nicht selbstteilend sein.

z.B.: $128 = 128 \% 1 == 0 \ \&\& \ 128 \% 2 == 0 \ \&\& \ 128 \% 8 == 0$

Entwickeln Sie zusätzlich eine Methode `countSelfDivides()` die angibt, wieviele sich selbstteilende Zahlen es ab 0 bis zu einer oberen Schranke gibt.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Die Methodensignaturen sind Ihnen nicht mehr vorgegeben. Leiten Sie diese aus der Aufgabenstellung bitte selber ab.
- Mittels `% 10` können Sie den Wert der letzten Ziffer bestimmen.
- Mittels `/ 10` können Sie eine Zahl um eine Stelle nach rechts "rausschieben".

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
  
        boolean result = dividesSelf(128);  
        System.out.println(result); // => true  
        System.out.println(dividesSelf(12)); // => true  
        System.out.println(dividesSelf(102)); // => false  
  
        int n = countDividesSelf(10);  
        System.out.println(n); // => 9  
        System.out.println(countDividesSelf(100)); // => 23  
        System.out.println(countDividesSelf(1000)); // => 79  
    }  
}
```

3.8 Vorkommen von Zeichenketten zählen

Schreiben Sie nun eine Methode `countOccurrences()` die zählt, wie häufig eine Zeichenkette `a` in einer anderen Zeichenkette `b` vorkommt. Sich überlagernde Zeichenketten sind erlaubt. D.h. "xx" ist als zweimal in "xxx" vorhanden zu zählen. Leere Zeichenketten sind nicht zu zählen.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Beachten Sie, dass leere Zeichenketten schnell eine Endlosschleife erzeugen können.
- Liefert VPL eine Out-of-Memory Fehlermeldung ist dies vermutlich auf eine Endlosschleife zurückzuführen.
- Die String-Methoden `indexOf()` oder `startsWith()` könnten hilfreich sein.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        System.out.println(countOccurrences("Hello", "Hello World")); // => 1  
        System.out.println(countOccurrences("abc", "abc abc abc")); // => 3  
        System.out.println(countOccurrences("xx", "xxx")); // => 2  
    }  
}
```

3.9 Zeichenketten verarbeiten mit everyNth()

Entwickeln Sie nun bitte eine Methode `everyNth()`, die für eine Zeichenkette `s` nur jedes `n`.te Zeichen zurück liefert. Bei $n = 3$ sollen also nur die Zeichen 0, 3, 6, ... und so weiter zurückgegeben werden. Für $n \leq 0$ soll die leere Zeichenkette zurückgegeben werden.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Achtung: Diese Aufgabe ist anfällig für Endlosschleifen.
- Erhalten Sie in VPL eine Out-of-Memory Fehlermeldung, haben Sie vermutlich eine Endlosschleife gebaut.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        String result = everyNth("Miracle", 2);  
        System.out.println(result); // => "Mrce"  
        System.out.println(everyNth("abcdefg", 2)); // => "aceg"  
        System.out.println(everyNth("abcdefg", 3)); // => "adg"  
    }  
}
```

3.10 Ein Passwort-Generator

Schreiben Sie nun bitte eine Methode `pwdgen()` zum Generieren von Passwörtern.

- Passwörter sollen dabei aus einem Satz gebildet werden.
- Worte in dem Satz sind durch ein oder mehrere Leerzeichen voneinander getrennt.
- Für jedes Wort soll abwechselnd der erste oder letzte Buchstabe des Wortes genommen werden.
- Die Anzahl an Worten soll an den Anfang des Passworts gesetzt werden.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Die String-Methode `split()` ist sicher hilfreich.
- Beachten Sie, dass Sätze mit Leerzeichen beginnen oder enden können.

Achtung: Ab sofort werden keine Methodenköpfe mehr vorgegeben. Diese aus der Problemformulierung zu bestimmen, ist Teil der Aufgabe.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        String pwd = pwdgen("Dies ist nur ein doofes Beispiel");  
        System.out.println(pwd); // => "6Dtndl"  
        System.out.println(pwdgen("a b c")); // => 3abc  
    }  
}
```

3.11 Zeichenketten prüfen mittels sameStarChar()

Entwickeln Sie nun bitte eine Methode `sameStarChar()`, die für eine Zeichenkette prüft, ob bei allen '*' Zeichen, das jeweils linke und rechte gleich sind.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Achten Sie auf Sonderfälle wie beginnende und abschließende '*'-Zeichen.
- Was passiert bspw. bei Zeichenketten wie "*", "***" oder "****"?

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        boolean result = sameStarChar("xy*yz");  
        System.out.println(result); // => true  
        System.out.println(sameStarChar("xy*zzz")); // => false  
        System.out.println(sameStarChar("*xa*az")); // => false  
    }  
}
```

3.12 Klammerungen prüfen mit checkBrackets()

Eine vollständige Klammerung bedeutet: Jeder geöffneten Klammer muss eine schließende Klammer folgen. Darüber hinaus müssen die runden Klammern korrekt verschachtelt sein. Andere Zeichen sind zu ignorieren.

Schreiben Sie nun eine Methode `checkBrackets()`, die prüft, ob eine Zeichenkette den oben angegebenen Regeln einer vollständigen Klammerung entspricht oder nicht.

Main.java:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        boolean check = checkBrackets("(");  
        System.out.println(check); // => true  
        System.out.println(checkBrackets("(")(a)((c))))"); // => true  
  
        System.out.println(checkBrackets("(")); // => false  
        System.out.println(checkBrackets("a ((())a)")); // => false  
    }  
}
```

3.13 Sternchen in Zeichenketten tilgen

Schreiben Sie nun eine Methode `starOut()`, die in einer Zeichenkette jeweils das Zeichen links und rechts eines '*' löscht.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Auch wenn Kontrollanweisungen wie `continue` und `break` meist nicht schön sind, ist diese Aufgabe ein Beispiel wie zumindest `continue` den Code sogar einfacher lesbar machen kann.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        String result = starOut("ab*cd");  
        System.out.println(result); // => "ad"  
        System.out.println(starOut("ab**cd")); // => "ad"  
        System.out.println(starOut("sm*eilly")); // => "silly"  
    }  
}
```

Kapitel 4

UNIT-03

4.1 Sortierung von Arrays prüfen

Sie sollen nun eine Methode `scoresIncreasing()` entwickeln, die für ein gegebenes Array prüft, ob alle Werte in diesem Array aufsteigend sortiert sind oder das Array leer ist.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Sehen Sie sich Arrays in Unit 03 noch einmal an.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        int[] d1 = {1, 3, 4};  
  
        boolean increasing = scoresIncreasing(d1);  
        System.out.println(increasing); // => true  
  
        int[] d2 = {1, 3, 2};  
        System.out.println(scoresIncreasing(d2)); // => false  
  
        int[] d3 = {1, 1, 4};  
        System.out.println(scoresIncreasing(d3)); // => true  
  
        int[] d4 = {1};  
        System.out.println(scoresIncreasing(d4)); // => true  
    }  
}
```

4.2 Listen nach geraden und ungeraden Zahlen ordnen

Entwickeln Sie nun bitte eine Methode namens `evenOdd()`, die eine Liste auf Basis einer bestehenden Liste von Integern erzeugt.

- In der neuen Liste müssen erst alle geraden Werte der ursprünglichen Liste stehen, erst dann sollen die ungeraden Werte folgen.
- Die Reihenfolge der ursprünglichen Liste soll innerhalb der geraden und ungeraden Werte aber erhalten bleiben.

Aufrufbeispiele finden sich in der `main()`-Methode.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        List<Integer> result = evenOdd(Arrays.asList(1, 2, 3, 4, 5, 6));  
        System.out.println(result); // => [2, 4, 6, 1, 3, 5]  
        System.out.println(evenOdd(Arrays.asList(5, 1, 3))); // => [5, 1, 3]  
        System.out.println(evenOdd(Arrays.asList(4, 2, 6))); // => [4, 2, 6]  
    }  
}
```

4.3 Räume

Sie sollen nun eine Klasse anlegen, mit denen man Räume verwalten kann, die dem Benennungsschema der TH Lübeck entsprechen.

Ein Raum liegt

- in einem Gebäude,
- auf einer Etage
- und hat eine Raumnummer (< 100).

Wie Räume angelegt und ausgegeben werden können, lässt sich den Beispielen in der `main()`-Methode entnehmen.

Ergänzend sollen Räume noch gem. Java Konventionen inhaltlich verglichen und geklont werden können.

Hinweise:

- Sehen Sie sich die relevanten Punkte in Unit 03 noch einmal an.
- Sehen Sie sich nochmal `String.format()` in Unit 01 an.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        Room office = new Room(17, 0, 10);  
        Room lecture = new Room(2, 0, 10);  
        Room lab = new Room(18, 1, 1);  
  
        System.out.println(office); // => "17-0.10"  
        System.out.println(lecture); // => " 2-0.10"  
        System.out.println(lab); // => "18-1.01"  
    }  
}
```

4.4 Primzahlen bestimmen

Eine Primzahl ist eine natürliche Zahl > 1, die nur durch sich selbst und 1 teilbar ist. Sie sollen nun Primzahlen generieren und tabellarisch auf der Konsole ausgeben.

Entwickeln Sie hierzu bitte die folgenden Methoden:

- `isPrim()` prüft, ob eine gegebene Zahl eine Primzahl ist.
- `primsUntil()` erzeugt eine Liste aller aufsteigen sortierten Primzahlen bis zu einer oberen Schranke.
- `columnize()` erzeugt aus einer Liste eine Zeichenkette in dem jedes Element mit einem Tabulator `\t` getrennt wird. Jeder `n`.te Tabulator wird jedoch durch ein `\n` ersetzt (solche Zeichenketten erscheinen tabellarisch auf der Konsole).

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- <https://de.wikipedia.org/wiki/Primzahl>
- Kennen Sie noch die String Methode `trim()`?

Main.java:

```
class Main {

    public static void main(String[] args) {
        boolean prim = isPrim(7);
        System.out.println(prim); // => true

        List<Integer> prims = primsUntil(20);
        System.out.println(prims); // => [2, 3, 5, 7, 11, 13, 17, 19]

        String output = columnize(prims, 3);
        System.out.println(output);
        // 2  3  5
        // 7  11 13
        // 17 19

        // Entspricht der Zeichenkette: "2\t3\t5\n7\t11\t13\n17\t19"
    }
}
```

4.5 Armstrongzahlen gruppieren

Entwickeln Sie nun bitte die Methoden

- `armstrongs()`, die eine Liste aller aufsteigend sortierter Armstrongzahlen bis zu einer oberen Schranke erzeugt,
- und `groupByLength()`, die eine Liste von Integer Werten mittels einer Map nach der Anzahl ihrer Stellen gruppiert. Die Map soll das Ordnungskriterium Stellenanzahl bei einer sequentiellen Verarbeitung der Schlüssel erhalten.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Sie haben bereits einmal eine Lösung entwickelt, die prüft, ob eine Zahl eine Armstrongzahl ist.
- Beachten Sie die Wahl ihrer Map-Implementierung.

Main.java:

```
class Main {

    public static void main(String[] args) {
        List<Integer> lance = armstrongs(500);
        System.out.println(lance);
        // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
    }
}
```

```

    Map<Integer, List<Integer>> grouped = groupByLength(lance);
    System.out.println(grouped);
    // {1=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 3=[153, 370, 371, 407]}
}
}

```

4.6 Aufsteigend sortierte Tripple finden

Schreiben Sie nun bitte eine Methode `trippleUp()` die in einem Integer-Array prüft, ob dieses drei aufsteigende, benachbarte Werte wie bspw. 1, 2, 3, oder auch 24, 25, 26 beinhaltet.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Main.java:

```

class Main {

    public static void main(String[] args) {
        int[] a1 = {1, 4, 5, 6, 2};
        int[] a2 = {1, 2, 3};
        int[] a3 = {1, 2, 4};
        int[] a4 = {3, 2, 1};

        System.out.println(trippleUp(a1)); // => true
        System.out.println(trippleUp(a2)); // => true
        System.out.println(trippleUp(a3)); // => false
        System.out.println(trippleUp(a4)); // => false
    }
}

```

4.7 Toppings

Sie sollen nun eine Methode `topping()` schreiben, die für einen Bringdienst ein paar Bereinigungen auf Bestellungen vornimmt.

Bestellungen werden als Map codiert.

- Taucht in der Bestellung "ice cream" auf, soll derselbe Wert auch für "yoghurt" gesetzt werden.
- Taucht in der Bestellung "spinach" auf, soll dieser Eintrag gelöscht werden (niemand mag Spinat).

`topping()` soll die ursprüngliche Bestellung nicht modifizieren, sondern eine neue modifizierte Bestellung erzeugen.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Beachten Sie den Unterschied von Referenztypen und primitiven Datentypen.

Main.java:

```

class Main {

    public static void main(String[] args) {
        Map<String, String> order1 = new TreeMap<>();
        order1.put("ice cream", "cherry");
    }
}

```

```

Map<String, String> order2 = new TreeMap<>();
order2.put("spinach", "dirt");
order2.put("ice cream", "cherry");

Map<String, String> order3 = new TreeMap<>();
order3.put("yoghurt", "salt");

System.out.println(topping(order1));
// => { "ice cream"="cherry", "yoghurt"="cherry" }
System.out.println(order1);
// => { "ice cream"="cherry" }

System.out.println(topping(order2));
// => { "ice cream"="cherry", "yoghurt"="cherry" }
System.out.println(order2);
// => { "spinach"="dirt", "ice cream"="cherry" }

System.out.println(topping(order3));
// => { "yoghurt"="salt" }
System.out.println(order3);
// => { "yoghurt"="salt" }
}
}

```

4.8 Worthäufigkeit in Zeichenketten bestimmen

Entwickeln Sie nun eine Methode `wordCount()`, die die absolute Häufigkeit von Worten in einem Text mittels eines Mappings zählt. Worte sind durch ein oder mehrere Whitespace Zeichen (Leerzeichen, Tabulatoren, Linebreaks, etc.) voneinander getrennt. Worte sollen case-insensitiv gezählt werden.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- In regulären Ausdrücken können Sie alle Whitespace Zeichen mit `"\\s"` selektieren.

Main.java:

```

class Main {

    public static void main(String[] args) {
        Map<String, Integer> result = wordCount("aa BB cC Aa Cc Bb aA AA");
        System.out.println(result); // => { "aa": 4, "bb": 2, "cc": 2 }
        System.out.println(wordCount("Ein kleines Beispiel"));
        // => { "ein": 1, "kleines": 1, "beispiel": 1 }
    }
}

```

4.9 Listen von Zeichenketten mittels `wordAppend()` verarbeiten

Entwickeln Sie nun bitte eine Methode `wordAppend()`. Diese soll eine Liste von Strings durchlaufen, um einen Ausgabestring zu erzeugen. Gehen Sie dabei wie folgt vor:

Immer wenn ein String zum 2., 4., 6., usw. mal in der Liste auftaucht, soll der String an den Ausgabestring gehängt werden. Wenn kein String doppelt vorkommt, soll die leere Zeichenkette zurückgegeben werden.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Denken Sie über den Einsatz einer geeigneten Datenstruktur (Collection) nach.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        List<String> example = Arrays.asList("a", "b", "a");  
        String result = wordAppend(example);  
        System.out.println(result); // -> "a"  
        System.out.println(wordAppend(  
            Arrays.asList("a", "b", "a", "c", "a", "d", "a")  
        )); // -> "aa"  
        System.out.println(wordAppend(Arrays.asList("a", "", "a"))); // -> "a"  
    }  
}
```

Kapitel 5

UNIT-04

5.1 Zeichenhäufigkeiten in Textdateien bestimmen

Sie sollen nun zwei Methoden entwickeln.

- `readFrom()` soll eine Text-Datei als Zeichenkette einlesen. Im Fehlerfall ist die `null`-Referenz zurückzugeben.
- `countChars()` soll in einer Zeichenkette gegebene Zeichen case-insensitiv zählen (d.h. 'a' ist wie 'A' und umgekehrt zu zählen). Werden keine Zeichen angegeben, sollen standardmäßig die Zeichen von 'a' bis 'z' gezählt werden.

Aufrufbeispiele beider Methoden finden Sie in der `main()`-Methode. Die Rückgabe von `countChars()` soll dazu geeignet sein, alphabetisch aufsteigend ausgegeben zu werden.

Hinweise:

- Denken Sie über den Einsatz von Exceptions nach (siehe Unit 02, spezielle Kontrollanweisungen).
- Sehen Sie sich noch einmal das *Überladen von Methoden in Unit 02 an.
- Sehen Sie sich noch einmal mögliche Map-Implementierungen in Unit 03 an.

Main.java:

```
class Main {

    public static void main(String[] args) {
        // Aufruf von readFrom()
        File f = new File("lorem.txt");
        String content = readFrom(f);
        String lorem = content.substring(0, 21);
        System.out.println(lorem);
        // => Lorem ipsum dolor sit

        // Rückgabe von readFrom() im Fehlerfall
        System.out.println(readFrom(new File("gibtesnicht.txt")));
        // => null

        // Default-Aufruf von countChars()
        Map<Character, Integer> occurrences = countChars(lorem);
        System.out.println(occurrences);
        // => {d=1, e=1, i=2, l=2, m=2, o=3, p=1, r=2, s=2, t=1, u=1}

        // Aufruf von countChars() mit vorgegebenen Zeichen
        System.out.println(countChars(lorem, "AeIoU"));
        // => {e=1, i=2, o=3, u=1}
```

```
}  
}
```

lorem.txt:

Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa
qui officia deserunt mollit anim id est laborum.

Kapitel 6

UNIT-05

6.1 Rekursives allStar()

Entwickeln Sie nun bitte eine **rekursive** Methode `allStar()`, die in einer Zeichenkette jedes Zeichen durch ein '*' trennt.

Aufrufbeispiele für `allStar()` finden Sie in der `main()`-Methode.

Verallgemeinern Sie dann `allStar()` so, dass ein beliebiges Zeichen anstelle des '*' als Trennzeichen genutzt werden kann.

Hinweise:

- Berücksichtigen Sie die Hinweise wie sich sequenzbasierte Rekursionen formulieren lassen (Unit 05).
- Sehen Sie sich ggf. noch einmal überladene Methoden aus Unit 02 an.

Verbote:

- Die Lösung ist rekursiv zu lösen, d.h. Schleifen aller Art sind verboten.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        String result = allStar("Hello");  
        System.out.println(result); // => H*e*l*l*o  
        System.out.println(allStar("abc")); // => a*b*c  
        System.out.println(allStar("ab")); // => a*b  
        System.out.println(allStar("ab", '-')); // => a-b  
    }  
}
```

6.2 Zeichenketten rekursiv bereinigen

Schreiben Sie bitte eine **rekursive** Methode `cleanString()`, die mehrfache benachbarte Zeichenfolgen in einem String löscht.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Verbote:

- Schleifen aller Art sind verboten.

- Lambdafunktionen sind verboten
- Datenfelder (globale Variablen) sind verboten.

Main.java:

```
class Main {

    public static void main(String[] args) {
        System.out.println(cleanString("yyzzza")); // => yza
        System.out.println(cleanString("aabbcbdd")); // => abcd
        System.out.println(cleanString("Hello")); // => Helo
    }
}
```

6.3 Listen rekursiv verarbeiten

Schreiben Sie nun bitte zwei **rekursive** Methoden `join()` und `sum()`.

`join()` soll eine Liste von Integer Werten mit einem Trennzeichen verknüpfen und eine Zeichenkette zurückgeben.

`sum()` soll eine Liste von Integer Werten aufaddieren und einen `int` Wert zurückgeben.

Sie finden Aufrufbeispiele für beide Methoden in der `main()`-Methode.

Hinweis:

- Sehen Sie sich noch einmal in Unit 05 an, wie sequenzbasierte Rekursionen formuliert werden können.
- Wiederholen Sie noch einmal das Überladen von Methoden.

Verbote:

- Sie sollen rekursiv programmieren, d.h. Schleifen aller Art sind verboten.

Main.java:

```
class Main {

    public static void main(String[] args) {
        List<Integer> values = Arrays.asList(1, 2, 3);
        String operation = join(values, "+");
        System.out.println(operation); // => 1+2+3

        int sum = sum(values);
        System.out.println(operation + "=" + sum); // => 1+2+3=6
    }
}
```

6.4 Vorkommen von Zeichenketten rekursiv zählen

Bitte entwickeln Sie nun eine **rekursive** Methode `countSubstring()`, die zählt wie oft ein String Teilstring eines anderen Strings ist.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweis:

- Sehen Sie sich noch einmal an wie man sequenzbasierte Rekursionen formulieren kann (Unit 05).

Verbote:

- Sie sollen rekursiv programmieren, d.h. Schleifen aller Art sind verboten.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        int n = countSubstring("Hello World", "Hello");  
        System.out.println(n); // => 1  
        System.out.println(countSubstring("Hello World", "l")); // => 3  
        System.out.println(countSubstring("xxx", "xx")); // => 2  
    }  
}
```

6.5 Geratene Zeichen rekursiv "blanken"

Entwickeln Sie nun bitte eine **rekursive** Methode `blank()`.

`blank()` soll zwei Zeichenketten `a` und `b` nehmen und eine neue Zeichenkette wie folgt generieren. Alle Zeichen die in `a` aber nicht in `b` sind, sollen in der generierten Zeichenkette durch ein `'_'` ersetzt werden. Leerzeichen bleiben ebenfalls in der generierten Zeichenkette erhalten, auch wenn sie nicht in `b` vorkommen.

Der Vergleich soll case-insensitiv erfolgen, also `'a'` ist wie `'A'` zu werten. Die Rückgabe soll in Upper-case erfolgen.

Hinweise:

- Aufruf-Beispiele finden Sie in der `main()`-Methode.

Verbote:

- Es sind keine Schleifen erlaubt.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        String result = blank("Hello World", "ell");  
        System.out.println(result); // => _ELL_ ___L_  
        System.out.println(blank("abc def ghj", "a")); // => A__ ___ ___  
    }  
}
```

6.6 Quersumme rekursiv berechnen

Entwickeln Sie nun bitte eine **rekursive** Methode `sumDigits()`, die die Quersumme einer natürlichen Zahl berechnet.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Die ganzzahlige Division durch 10 und modulo 10 können sehr hilfreich sein.
- <https://de.wikipedia.org/wiki/Quersumme>
- Die Aufgabe kann als Einzeiler gelöst werden.

Verbote:

- Sie sollen rekursiv programmieren, d.h. Schleifen aller Art sind verboten.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        int sum = sumDigits(123);  
        System.out.println(sum); // => 6  
        System.out.println(sumDigits(99996)); // => 42  
        System.out.println(sumDigits(-123)); // => -6  
    }  
}
```

6.7 Binärbäume rekursiv verarbeiten

Gegeben sei die Klasse `Node` mit der Binärbäume gebildet werden können, die Zeichenketten als Werte speichern können.

Entwickeln Sie nun bitte die folgenden Methoden:

- `serialize()`: Erzeugt eine Liste in dem der Baum inorder durchlaufen wird.
- `count()`: Liefert die Anzahl an Knoten in einem Baum.
- `depth()`: Liefert die maximale Tiefe eines Baums.
- `longest()`: Findet das längste (im Baum am tiefsten rechts stehende) Wort.

Sie finden Aufrufbeispiele in der `main()`-Methode.

Hinweis:

- Sehen Sie sich noch einmal die Inhalte zu rekursiven Datenstrukturen der Unit 05 an.

Verbote:

- Sie sollen rekursiv programmieren, d.h. Schleifen aller Art sind verboten.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        Node tree = new Node("f",  
            new Node("o",  
                new Node("C",  
                    new Node("tasty"),  
                    null),  
                new Node("F")  
            ),  
            new Node("E",  
                null,  
                new Node("e")  
            )  
        );  
        // Hinweis, der Evaluator wird diesen Baum in folgender  
        // Notation ausgeben (Pattern: %value[%left,%right])  
        // <f[o[C[tasty,null],F],E[null,e]]>  
  
        List<String> serialized = serialize(tree);  
        System.out.println(serialized); // => [tasty, C, o, F, f, E, e]  
    }  
}
```

```

        System.out.println(count(tree)); // => 7
        System.out.println(longest(tree)); // => tasty
        System.out.println(depth(tree)); // => 4
    }
}

```

Node.java:

```

public class Node {

    public String value;
    public Node left = null;
    public Node right = null;

    public Node(String v) { this.value = v; }

    public Node(String v, Node l, Node r) {
        this.value = v;
        this.left = l;
        this.right = r;
    }

    public void insert(String content, double shift) {
        if (Math.random() < shift) {
            // insert left
            if (this.left != null) this.left.insert(content, shift);
            else this.left = new Node(content);
        } else {
            // insert right
            if (this.right != null) this.right.insert(content, shift);
            else this.right = new Node(content);
        }
    }

    public Node insert(List<String> contents, double shift) {
        for (String content : contents) this.insert(content, shift);
        return this;
    }

    public String toString() {
        if (this.left == null && this.right == null) return this.value;
        return String.format("%s[%s,%s]", this.value, this.left, this.right);
    }
}

```

6.8 Binärbäume rekursiv prettyprinten

Gegeben sei die Klasse `Node` mit der Binärbäume gebildet werden können, die Zeichenketten als Werte speichern können.

Entwickeln Sie nun bitte eine **rekursive** Methode `prettyPrint()`, die eine Zeichenkettenrepräsentation für solche Binärbäume generiert, die auf der Konsole ausgegeben gut als Baumstruktur interpretierbar sind.

Sie finden Aufrufbeispiele in der `main()`-Methode.

`prettyPrint()` soll

- jeden Knoten des Baums mit einem vorangestellten "- " kennzeichnen,
- jede Ebene des Baums um zwei Leerzeichen " " einrücken und
- für den null Baum, die leere Zeichenkette "" zurückgeben.

Hinweis:

- Die Datenstruktur Node findet sich in der Datei Node.java.
- Es bietet sich ggf. an das Einrückungsproblem mit einer überladenen Methode anzugehen, der eine Einrückungstiefe als zusätzlicher Parameter übergeben wird.

Verbote:

- Das Problem ist rein rekursiv zu lösen. Schleifen sind nicht gestattet.

Main.java:

```
class Main {

    public static void main(String[] args) {
        Node tree = new Node("f",
            new Node("o",
                new Node("C",
                    new Node("tasty"),
                    null),
                new Node("F")
            ),
            new Node("E",
                null,
                new Node("e")
            )
        );
        // Hinweis, der Evaluator wird diesen Baum in folgender
        // Notation ausgeben (Pattern: %value[%left,%right])
        // <f[o[C[tasty,null],F],E[null,e]]>

        String output = prettyPrint(tree);
        System.out.println(output); // =>
        // - f
        //   - o
        //     - C
        //       - tasty
        //     - F
        //   - E
        //     - e

        System.out.println(prettyPrint(new Node("A"))); // => "- A"
        System.out.println(prettyPrint(null)); // => ""
    }
}
```

Node.java:

```
public class Node {

    public String value;
    public Node left = null;
    public Node right = null;

    public Node(String v) { this.value = v; }

    public Node(String v, Node l, Node r) {
        this.value = v;
    }
}
```

```

        this.left = l;
        this.right = r;
    }

    public void insert(String content, double shift) {
        if (Math.random() < shift) {
            // insert left
            if (this.left != null) this.left.insert(content, shift);
            else this.left = new Node(content);
        } else {
            // insert right
            if (this.right != null) this.right.insert(content, shift);
            else this.right = new Node(content);
        }
    }

    public Node insert(List<String> contents, double shift) {
        for (String content : contents) this.insert(content, shift);
        return this;
    }

    public String toString() {
        if (this.left == null && this.right == null) return this.value;
        return String.format("%s[%s,%s]", this.value, this.left, this.right);
    }
}

```

6.9 Maximum auf einer verketteten Liste rekursiv bestimmen

Gegeben sei eine einfach verkettete Liste (RList). Implementieren Sie nun bitte auf dieser Datenstruktur eine **rekursive** Methode `max()`, die das Maximum in einer solchen Liste findet.

Ist eine RList leer, soll `max()` die `null`-Referenz zurückgeben.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Die Datenstruktur finden Sie in der Datei `RList.java`.
- Die Datenstruktur `RList` ist gegeben und muss durch Sie **nicht** implementiert werden.
- Wenn ein Rückgabewert im Fehlerfall eine `null`-Referenz sein soll, kann man bei primitiven Datentypen mit der Referenztypentsprechung arbeiten (also bspw. `Integer` statt `int`).

Verbote:

- Sie sollen rekursiv programmieren, d.h. Schleifen aller Art sind verboten.

Main.java:

```

class Main {

    public static void main(String[] args) {
        RList ls = RList.build(0, 3, 2, 4, 7, 1);
        RList empty = RList.build();
        System.out.println(ls); // => [0, 3, 2, 4, 7, 1]
        System.out.println(empty); // => []
        System.out.println(max(ls)); // => 7
        System.out.println(max(empty)); // => 7
    }
}

```

```
}  
}
```

RList.java:

```
public class RList {  
  
    public int value;  
    public RList next;  
    public boolean isEmpty = false;  
  
    public RList() { this.isEmpty = true; }  
    public RList(int v) { this.value = v; }  
  
    public static RList build(List<Integer> values) {  
        if (values.isEmpty()) return new RList();  
        RList first = new RList(values.get(0));  
        RList last = first;  
        for (int i = 1; i < values.size(); i++) {  
            last.next = new RList(values.get(i));  
            last = last.next;  
        }  
        return first;  
    }  
  
    public static RList build(Integer... values) {  
        return RList.build(Arrays.asList(values));  
    }  
  
    public String serialize() {  
        return value + (next == null ? "" : ", " + next.serialize());  
    }  
  
    public String toString() {  
        return isEmpty ? "[]" : "[" + serialize() + "];"  
    }  
}
```

6.10 Filtern von Listen von Zeichenketten mittels Lambdas

Diese Aufgabe haben Sie bereits exakt oder in sehr ähnlicher Form an anderen Stellen im Praktikum gelöst. Diesmal dürfen Sie allerdings nur Streams und Lambdafunktionen nutzen. Innerhalb von Lambdafunktionen sind jegliche Kontrollanweisungen wie Schleifen oder bedingte Anweisungen untersagt. Sie dürfen aber natürlich Teilprobleme in andere Lambdafunktionen, Operatoren oder Prädikate auslagern.

Entwickeln Sie nun bitte eine Lambdafunktion `without` und machen Sie diese in `public static` Datenfeldern außerhalb der `main()`-Methode für die Auto-Evaluierung bekannt.

`without` soll aus einer Liste von Zeichenketten eine neue Zeichenkette ohne eine vorgebene Zeichenkette generieren.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Nutzen Sie jeweils den konkretesten Lambda Typ (also bspw. `Predicate<String>` anstatt `Function<String, Boolean>`).
- Achten Sie darauf, dass sie Streams limitieren (ansonsten haben Sie Endlosberechnungen).

Verbote:

- Es sind nur Lambda Funktionen erlaubt, keine Methoden (bis auf die `main()`).
- Es sind keine Schleifen oder bedingten Anweisungen erlaubt.

Main.java:

```
public class Main {  
  
    public static void main(String[] args) {  
        List<String> examples = Arrays.asList(  
            "Dies", "ist", "nur", "ein", "Beispiel"  
        );  
        List<String> result = without.apply(examples, "nur");  
        System.out.println(result); // => ["Dies", "ist", "ein", "Beispiel"]  
    }  
}
```

6.11 Bestimmen einer Stelle einer Zahl mittels Lambdas

Diese Aufgabe haben Sie bereits exakt oder in sehr ähnlicher Form an anderen Stellen im Praktikum gelöst. Diesmal dürfen Sie allerdings nur Streams und Lambdafunktionen nutzen. Innerhalb von Lambdafunktionen sind jegliche Kontrollanweisungen wie Schleifen oder bedingte Anweisungen untersagt. Sie dürfen aber natürlich Teilprobleme in andere Lambdafunktionen, Operatoren oder Prädikate auslagern.

Entwickeln Sie nun bitte eine Lambdafunktion `nthDigit` und machen Sie diese in `public static` Datenfeldern außerhalb der `main()`-Methode für die Auto-Evaluierung bekannt.

`nthDigit` soll aus einer ganzen Zahl (Dezimalnotation) den Wert der `n`-ten Stelle zurückgeben.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Nutzen Sie jeweils den konkretesten Lambda Typ (also bspw. `Predicate<String>` anstatt `Function<String, Boolean>`).

Verbote:

- Es sind nur Lambda Funktionen erlaubt, keine Methoden (bis auf die `main()`).
- Es sind keine Schleifen oder bedingten Anweisungen erlaubt.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        int result = nthDigit.apply(4321, 1);  
        System.out.println(result); // => 3  
        System.out.println(nthDigit.apply(4321, 5)); // => null  
    }  
}
```

6.12 Tabellarische Konsolenausgabe mittels Lambdas

Diese Aufgabe haben Sie bereits exakt oder in sehr ähnlicher Form an anderen Stellen im Praktikum gelöst. Diesmal dürfen Sie allerdings nur Streams und Lambdafunktionen nutzen. Innerhalb von Lambdafunktionen sind jegliche Kontrollanweisungen wie Schleifen oder bedingte Anweisungen untersagt. Sie dürfen aber natürlich Teilprobleme in andere Lambdafunktionen, Operatoren oder Prädikate auslagern.

Entwickeln Sie nun bitte eine Lambdafunktion `columnize` und machen Sie diese in `public static` Datenfeldern außerhalb der `main()`-Methode für die Auto-Evaluierung bekannt.

`columnize` soll aus einer Liste von Zahlen eine Zeichenkette in dem jedes Element mit einem Tabulator `\t` getrennt wird. Jeder `n`.te Tabulator wird jedoch durch ein `\n` ersetzt (solche Zeichenketten erscheinen tabellarisch auf der Konsole). Im Fehlerfall soll `columnize` die `null` Referenz zurueckgeben.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- Nutzen Sie jeweils den konkretesten Lambda Typ (also bspw. `Predicate<String>` anstatt `Function<String, Boolean>`).

Verbote:

- Es sind nur Lambda Funktionen erlaubt, keine Methoden (bis auf die `main()`).
- Es sind keine Schleifen oder bedingten Anweisungen erlaubt.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
  
        String result = columnize.apply(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8), 3);  
        System.out.println(result);  
        // 1      2      3  
        // 4      5      6  
        // 7      8  
  
        // Entspricht der Zeichenkette:  
        // "1\t2\t3\n4\t5\t6\n7\t8"  
    }  
}
```

6.13 Geratene Zeichen mittels Lambdas "blanken"

Entwickeln Sie nun bitte eine Lambda-Funktion und machen Sie diese in einem `public static` Datenfeld namens `blank` außerhalb der `main()`-Methode für die Auto-Evaluierung bekannt.

`blank()` soll zwei Zeichenketten `a` und `b` nehmen und eine neue Zeichenkette wie folgt generieren. Alle Zeichen die in `a` aber nicht in `b` sind, sollen in der generierte Zeichenkette durch ein `'_'` ersetzt werden. Leerzeichen bleiben ebenfalls in der generierten Zeichenkette erhalten, auch wenn sie nicht in `b` vorkommen.

Hinweise:

- Aufruf-Beispiele finden Sie in der `main()`-Methode.
- Nutzen Sie jeweils den konkretesten Lambda Typ (also bspw. `Predicate<String>` anstatt `Function<String, Boolean>`).

Verbote:

- Es sind keine Methoden erlaubt (bis auf die `main()`).
- Es sind keine Schleifen erlaubt.
- Es sind keine Blöcke innerhalb von Lambda Funktionen erlaubt.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        String result = blank.apply("Hello World", "ell");  
        System.out.println(result); // => _ell_ ___l_  
        System.out.println(blank.apply("abc def ghj", "a")); // => a__ ___ ___  
    }  
}
```

6.14 Primzahlen mit Lambdas bestimmen (und ausgeben)

Entwickeln Sie nun bitte mehrere Lambda-Funktionen und machen Sie diese in `public static` Datenfeldern außerhalb der `main()`-Methode für die Auto-Evaluierung bekannt.

- `isPrim` (bestimmt eine Primzahl)
- `primes` (liefert eine Liste aller Primzahlen bis zu einer oberen Schranke, exklusiv)

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- <https://de.wikipedia.org/wiki/Primzahl>
- Nutzen Sie jeweils den konkretesten Lambda Typ (also bspw. `Predicate<String>` anstatt `Function<String, Boolean>`).
- Achten Sie darauf, dass Sie Streams limitieren (ansonsten haben Sie Endlosberechnungen).

Verbote:

- Es sind nur Lambda Funktionen erlaubt, keine Methoden (bis auf die `main()`).
- Es sind keine Schleifen oder bedingten Anweisungen erlaubt.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        System.out.println(isPrime.test(4)); // => false  
        System.out.println(isPrime.test(7)); // => true  
  
        List<Integer> result = primes.apply(20);  
        System.out.println(result); // => [2, 3, 5, 7, 11, 13, 17, 19]  
        System.out.println(primes.apply(3)); // => [2]  
    }  
}
```

6.15 Vollkommene Zahlen mittels Lambdas bestimmen

Eine natürliche Zahl n wird vollkommene Zahl (auch perfekte Zahl) genannt, wenn sie gleich der Summe aller ihrer (positiven) Teiler außer sich selbst ist.

Die kleinsten drei vollkommenen Zahlen sind:

- $6 = 1 + 2 + 3 = 6$
- $28 = 1 + 2 + 4 + 7 = 28$
- $496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248 = 496$

Entwickeln Sie nun bitte mehrere Lambda-Funktionen, um perfekte Zahlen bis zu einer oberen Schranke zu bestimmen und machen Sie diese in `public static` Datenfeldern außerhalb der `main()`-Methode für die Auto-Evaluierung bekannt.

Zerlegen Sie das Problem so in folgende Lambdafunktionen (Teilprobleme).

- Entwickeln Sie ein Prädikat `perfect`, das bestimmt, ob eine Zahl eine vollkommene Zahl ist.
- Entwickeln Sie eine Lambdafunktion `perfectNumbers`, die `perfect` nutzt, um eine Liste aller aufsteigend sortierten vollkommenen Zahlen bis zu einer gegebenen Schranke zu bestimmen.

Entsprechende Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- https://de.wikipedia.org/wiki/Vollkommene_Zahl
- Vergleichen Sie den Wert von `Integer` (nicht `int`) in Streams mittels `equals()` oder casten Sie mittels `(int)` auf den primitiven Datentyp `int`, der mittels `==` verglichen werden kann.
- Sie können auch `IntStream` nutzen, um einen Stream primitiver Datentypen zu erzeugen und diese Vergleichsprobleme zu vermeiden.
- Nutzen Sie jeweils den konkretesten Lambda Typ (also bspw. `Predicate<String>` anstatt `Function<String, Boolean>`).
- Achten Sie darauf, dass Sie Streams limitieren (ansonsten haben Sie Endlosberechnungen).

Verbote:

- Es sind nur Lambda Funktionen erlaubt, keine Methoden (bis auf die `main()`).
- Es sind keine Schleifen oder bedingten Anweisungen erlaubt.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println(perfect.test(4)); // => false  
        System.out.println(perfect.test(496)); // => true  
  
        int[] perfects = perfectNumbers.apply(10000);  
        System.out.println(Arrays.toString(perfects)); // => [6, 28, 496]  
    }  
}
```

Kapitel 7

UNIT-06

7.1 Auto-Klasse

Für ein Autohaus ist eine Klasse `Auto` zu entwickeln. Folgende Datenfelder sind bei einem `Auto` zu verwalten.

- Fabrikat (`String`)
- Laufleistung (km, `int`)
- Preis (EUR, `double`)
- Farbe (`String`)
- Unfallwagen (`boolean`)
- Kraftstoff (`String`)
- Leistung (PS, `double`)

Wie ein `Auto` angelegt und auf der Konsole ausgegeben werden soll, finden Sie in der `main()`-Methode. Die Ausgabe `"!!! UNFALLFREI !!!"` soll tatsächlich nur für unfallfreie Wagen erfolgen.

Ein Autobestand ist nichts weiter als eine Liste von solchen Autos. Auf diesem Autobestand sollen Sie nun dem Verkaufsleiter ein paar Fragen beantworten indem Sie hierfür sinnvoll definierte Methoden definieren:

- Wieviel Prozent der Autos sind Unfallwagen? (`accidents()`)
- Wieviel Prozent der Autos haben eine spezifische Kraftstoffart? (`fuel()`)
- Wie hoch ist der Verkaufserlös wenn im Schnitt `n%` Nachlass auf einen unfallfreien und `m%` Nachlass auf einen Unfallwagen gegeben werden? (`revenue()`)

Entsprechende Aufrufbeispiele finden Sie in der `main()`-Methode. Aus diesen Aufrufbeispielen können Sie sich auch die Methodensignaturen ableiten.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        Auto a = new Auto("VW Touran", 88888, 7999.99, "weiß", false, "Benzin", 101.0);  
        System.out.println(a);  
        // VW Touran (Benzin; 101,0 PS; weiß; 88888 km) für 7999,99 EUR !!! UNFALLFREI !!!  
  
        Auto b = new Auto("Ford Focus", 139000, 3999.99, "metallic", true, "Diesel", 103.5);  
        System.out.println(b);  
        // Ford Focus (Diesel; 103,5 PS; metallic; 139000 km) für 3999,99 EUR  
  
        List<Auto> cars = Arrays.asList(  
            new Auto("VW Touran", 88888, 7999.99, "weiß", false, "Benzin", 101.0),  
            new Auto("Ford Focus", 139000, 3999.99, "metallic", true, "Diesel", 103.5),  
            new Auto("C-Klasse", 25000, 23999.99, "metallic", false, "Diesel", 153),
```

```

new Auto("VW Golf", 39000, 33999.99, "blue racing", false, "Benzin", 193.5),
new Auto("Citroen Clio", 19000, 23999.99, "silber", true, "Gas", 103.5),
new Auto("VW Up", 31000, 23999.99, "post yellow", false, "Elektro", 73.5)
);

System.out.println(accidents(cars));           // => 33.3333
System.out.println(fuel(cars, "Elektro"));     // => 16.6666
System.out.println(fuel(cars, "elektro"));     // => 16.6666
System.out.println(fuel(cars, "Diesel"));      // => 33.3333
System.out.println(revenue(0.25, 0.1, cars)); // => 101999.949
System.out.println(revenue(0.1, 0.1, cars));  // => 106199.946

}
}

```

7.2 Raum-Verwaltung

In dieser Aufgabe sollen Mitarbeiter Räume reservieren können. Reservierungen für einen Raum von einem Mitarbeiter sollen durch zwei Uhrzeiten (Beginn und Ende) begrenzt sein und ggf. eine Anmerkung haben.

Der Zusammenhang ist in beigelegtem UML-Diagramm erläutert.

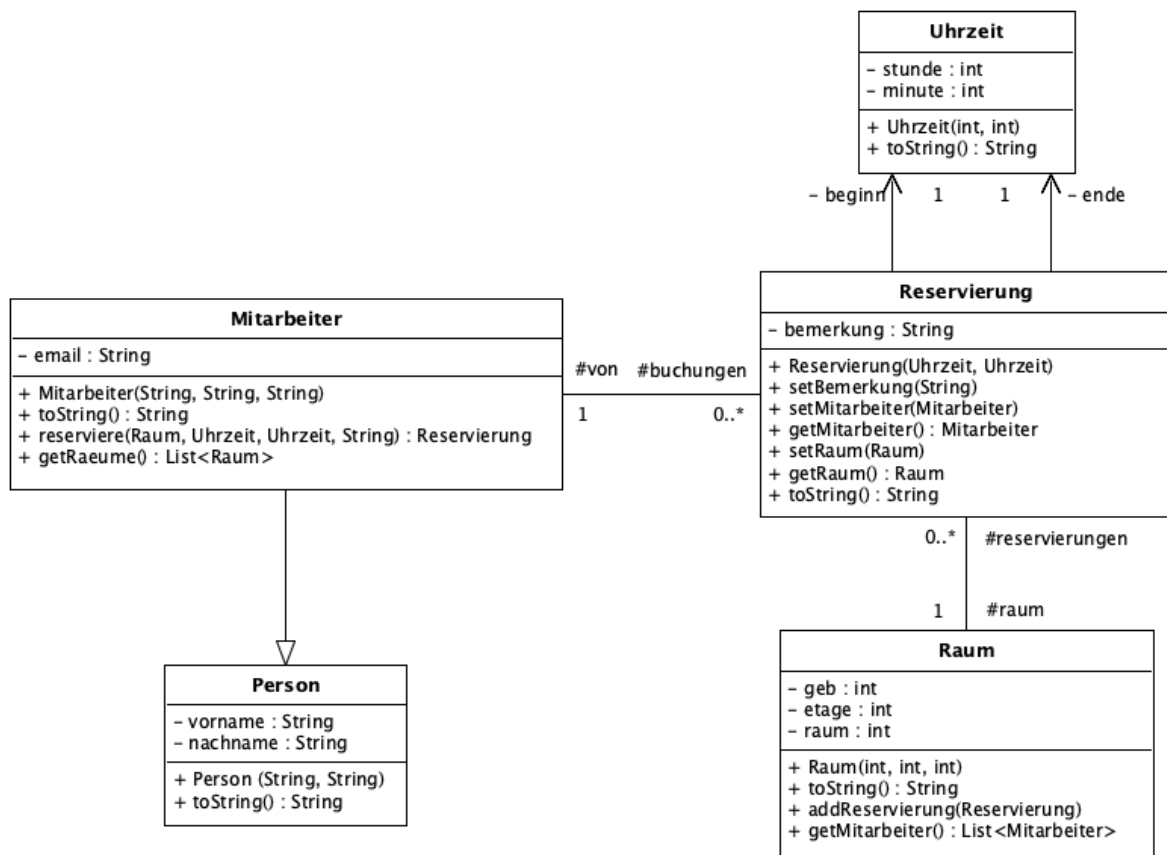


Abbildung 7.1: UML

In der `main()`-Methode sehen Sie wie die entsprechenden Objekte angelegt werden können und auf der Konsole ausgegeben werden können.

Sie sehen ferner wie sich mittels `reserviere()` von einem Mitarbeiter Räume buchen lassen und die zugehörigen Reservierungen ausgegeben werden sollen.

Achten Sie ferner darauf das ein Mitarbeiter mittels `getRaeume()` auf die für ihn reservierten Räume zugreifen kann. Auch von einem Raum können mittels `getMitarbeiter()` die buchenden Mitarbeiter ermittelt werden. In beiden Fällen sollen Mitarbeiter oder Räume nicht doppelt auftauchen (also z.B. in Fällen wenn ein Mitarbeiter einen Raum mehrmals zu unterschiedlichen Zeiten gebucht hat).

Implementieren Sie nun die Klassen dieses UML-Diagramms.

Main.java:

```
class Main {

    public static void main(String[] args) {
        Uhrzeit t = new Uhrzeit(12, 7);
        System.out.println(t); // 12:07 Uhr

        Raum r = new Raum(2, 0, 1);
        System.out.println(r); // 2-0.01

        Person p = new Person("Max", "Musterfrau");
        System.out.println(p); // Max Musterfrau

        Mitarbeiter ma = new Mitarbeiter("Maren", "Mustermann", "ceo@example.com");
        System.out.println(ma); // Maren Mustermann (ceo@example.com)

        Reservierung res = ma.reserviere(r, new Uhrzeit(12, 0), new Uhrzeit(13, 15), "");
        System.out.println(res);
        // 2-0.01 gebucht von 12:00 Uhr bis 13:15 Uhr für Maren Mustermann (ceo@example.com)

        System.out.println(r.getMitarbeiter());
        // [Maren Mustermann (ceo@example.com)]

        System.out.println(ma.getRaeume());
        // [2-0.01]
    }
}
```

7.3 Possible Chessmen

Ein Schachbrett ist waagrecht in acht Reihen (A bis H) und senkrecht in acht Linien (1 bis 8) aufgebaut. Auf so einem Schachbrett seien nun mehrere Zugfolgen gegeben. Z.B.:

B2 - C3 - E5 - E8 - G6

Gem. den Schachregeln kann diese Zugfolge nicht jede Figur ziehen. Sie sollen in dieser Aufgabe nun eine Methode `possibleChessmen()` entwickeln, die bestimmen kann, welche Schachfiguren (König, Dame, Turm, Läufer, Springer) in der Lage sind, eine **beliebig vorgegebene Zugfolge** zu ziehen.

Zur Hilfe sei Ihnen folgendes UML Klassendiagramm an die Hand gegeben, mit denen das Problem objektorientiert strukturiert werden kann.

Implementieren Sie nun bitte die Methode `possibleChessmen()` in der `Main`-Klasse und die Klassen des UML-Klassendiagramm um das Problem zu lösen.

Hinweise:

- Die Bauern werden bewusst nicht berücksichtigt.
- Beachten Sie, dass die Klasse `Figur` bewusst als abstrakte Klasse konzipiert wurde.

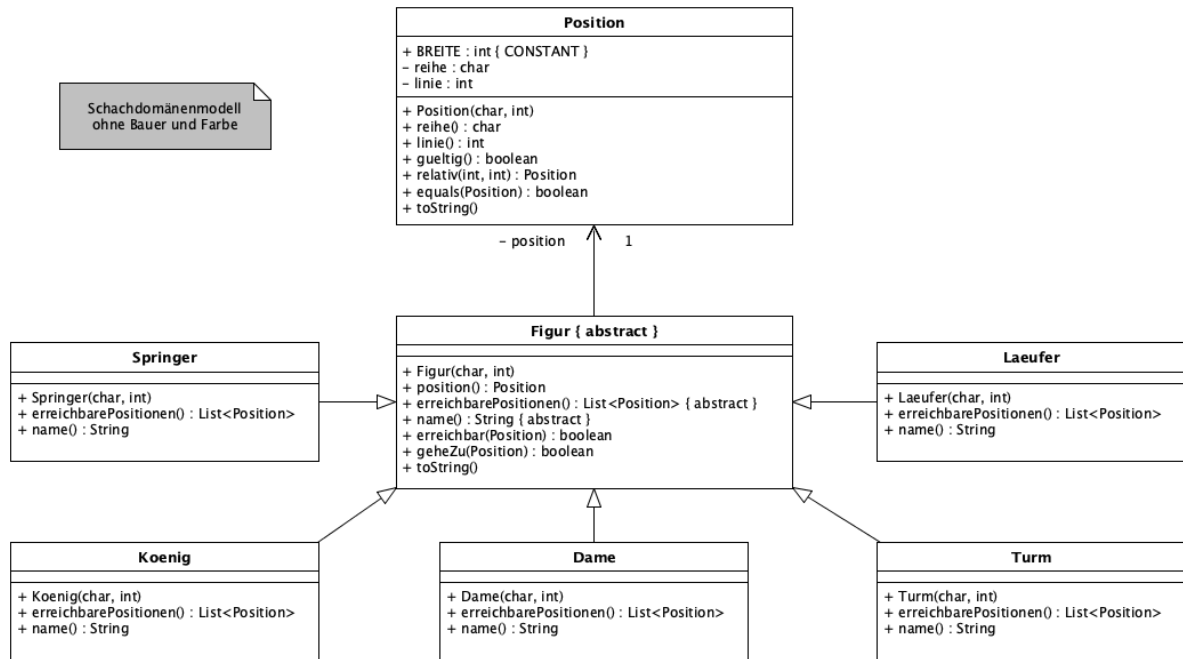


Abbildung 7.2: UML

- Nehmen Sie sich ein Schachbrett (oder mindestens ein Blatt Papier) zur Hand, um das Problem nachzuvollziehen.

Main.java:

```

class Main {

    public static void main(String[] args) {

        // Wirkungsweise eines Positionsobjekts
        Position pos = new Position('c', 2);
        System.out.println(pos); // => C2
        System.out.println(pos.gueltig()); // => true;
        System.out.println(pos.relativ(1, -1)); // => D1
        System.out.println(pos.relativ(1, -2).gueltig()); // => false
        System.out.println(pos.relativ(-3, 0).gueltig()); // => false

        // Wirkungsweise einer Schachfigur
        Figur f = new Springer('c', 2);
        System.out.println(f); // => Springer [C2]
        System.out.println(f.erreichbarePositionen()); // => [B4, D4, A3, E3, A1, E1]
        System.out.println(f.geheZu(new Position('b', 3))); // => false
        System.out.println(f); // => Springer [C2]
        System.out.println(f.geheZu(new Position('e', 3))); // => true
        System.out.println(f); // => Springer [E3]

        // Demonstration der Wirkungsweise von possibleChessmen()
        List<Position> moves = Arrays.asList(
            new Position('B', 2),
            new Position('C', 3),
            new Position('E', 5),
            new Position('E', 8),
            new Position('G', 6)
        );
    }
}
  
```



```

for (int i = 0; i < moves.size(); i++) {
    List<Position> submoves = moves.subList(0, i + 1);
    String men = possibleChessmen(submoves)
                .stream()
                .map(fig -> fig.name())
                .collect(Collectors.joining(", "));
    System.out.println(moves.get(i) + ": " + men);
}
// Ergibt folgende Ausgabe:
// B2: König, Dame, Turm, Laeufer, Springer
// C3: König, Dame, Läufer
// E5: Dame, Laeufer
// E8: Dame
// G6: Dame
}
}

```

7.4 Fruchtkorb

In dieser Aufgabe sollen Früchte Körben zugeordnet werden können. Zu den folgenden Früchten ist bekannt:

- Apfel: Gewicht zwischen 100 g und 200 g (gleichverteilt), 52 kcal pro 100 g
- Banane: Gewicht von 75 g bis 200 g (gleichverteilt), 93 kcal pro 100 g
- Pfirsich: Gewicht von 100 g bis 160 g (gleichverteilt), 39 kcal pro 100 g

Werden Früchte ohne weitere Angaben erzeugt, so sollen Sie ein zufälliges Gewicht aus den angegebenen Bereichen erhalten.

Der Zusammenhang ist in beigelegtem UML-Diagramm erläutert.

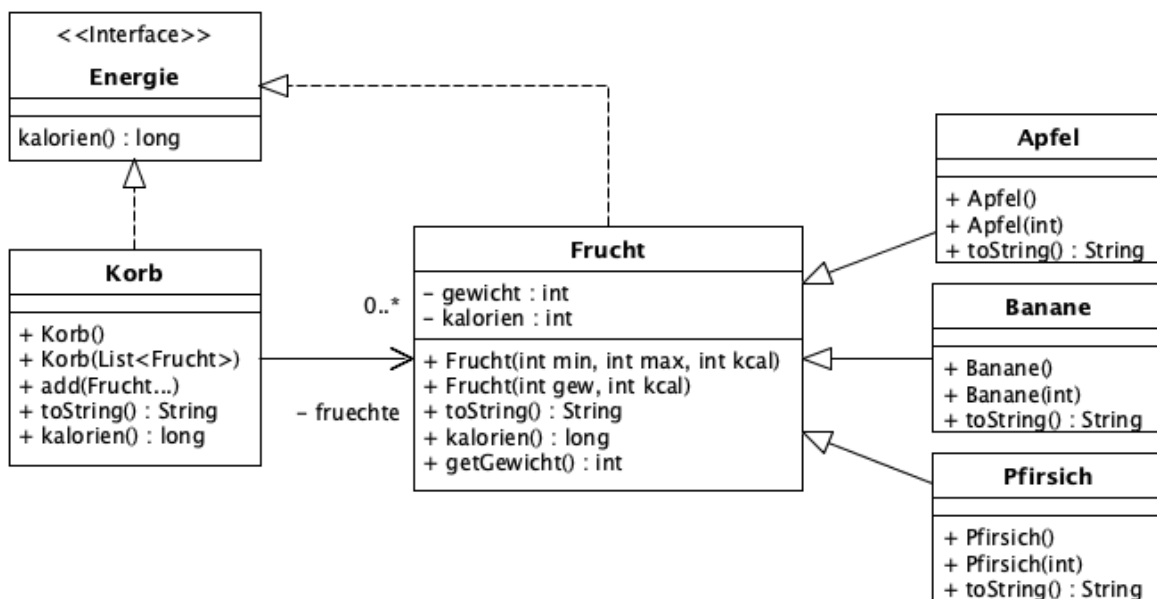


Abbildung 7.3: UML

In der `main()`-Methode sehen Sie wie die entsprechenden Objekte angelegt werden können und auf der Konsole ausgegeben werden können. Implementieren Sie nun bitte das UML Diagramm.

Hinweise:

- Mit der `Math.random()` Methode können Sie eine gleichverteilte Zufallszahl `[0.0, 1.0[` erzeugen.
- Mit der `Math.round()` Methode können Sie runden.

Main.java:

```
class Main {  
  
    public static void main(String[] args) {  
        RFrucht apple = new RApfel();  
        RFrucht banana = new RBanane();  
        RFrucht peach = new RPfirsich();  
        System.out.println(apple); // Apfel, 127 g, 66 kcal  
        System.out.println(banana); // Banane, 105 g, 98 kcal  
        System.out.println(peach); // Pfirsich, 117 g, 46 kcal  
  
        RKorb korb = new RKorb();  
        korb.add(banana, apple);  
        korb.add(peach);  
        System.out.println(korb);  
        // Fruchtkorb mit insgesamt 210 kcal  
        // - Banane, 105 g, 98 kcal  
        // - Apfel, 127 g, 66 kcal  
        // - Pfirsich, 117 g, 46 kcal  
  
    }  
}
```

Kapitel 8

UNIT-09

8.1 Generische Warteschlange

Lesen Sie sich bitte in das Konzept der Datenstruktur [Warteschlange](#) ein und entwickeln Sie eine generische Warteschlange `MyQueue` mit dem generischen Typparameter `T`. Die Datenstruktur `MyQueue` soll die folgenden Methoden implementieren:

- `enter()` fügt ein neues Element der Warteschlange hinzu. Liefert `true`, wenn das Element der Warteschlange hinzugefügt werden konnte, andernfalls `false`.
- `leave()` entnimmt das erste Element der Warteschlange. Ist die Warteschlange leer wirft die Methode eine `java.util.NoSuchElementException`.
- `isEmpty()` prüft ob die Warteschlange leer ist.
- `front()` liest das erste Element der Warteschlange, belässt es aber in der Warteschlange. Liefert `null`, wenn sich kein Element in der Warteschlange befindet.
- `toString()` liefert eine textuelle Repräsentation der Warteschlange in folgender Form: "[e1, e2, e3, e4]" wenn `e1, ..., e4` Elemente der Warteschlange sind und `e1` das hinterste Element und `e4` das vorderste Element der Warteschlange ist.

Aufrufbeispiele finden Sie in der `main()`-Methode. Ergänzend ist Ihnen noch folgendes UML-Diagramm an die Hand gegeben.

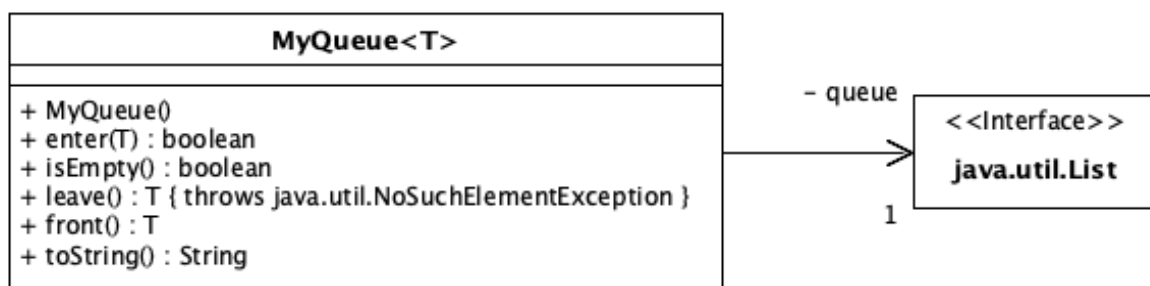


Abbildung 8.1: UML

Main.java:

```
class Main {

    public static void main(String[] args) {

        MyQueue<String> myq = new MyQueue<String>();
        System.out.println(myq);
        // => []
    }
}
```

```

myq.enter("Max");
myq.enter("Maren");
myq.enter("Tessa");
myq.enter("Hans");

System.out.println(myq);
// => [Hans, Tessa, Maren, Max]

System.out.println(myq.front());
// => Max

System.out.println(myq);
// => [Hans, Tessa, Maren, Max]

System.out.println(myq.leave());
// => Max

System.out.println(myq);
// => [Hans, Tessa, Maren]
}
}

```

8.2 Generisches Convertable Interface

Die in dieser Aufgabe zu implementierende `Convertable` Schnittstelle dient dazu einen Datentyp `F` in einen Datentyp `T` zu konvertieren und die Konvertierung auch wieder invertieren zu können. Der Datentyp `T` kann dabei als interne Zwischenrepräsentation (z.B. Listen oder Bäume) für Verarbeitungen verwendet werden. Die Schnittstelle definiert dazu folgende Methoden:

- `transform()`: Konvertiert Daten vom Typ `F` (from) in den Datentyp `T` (to). Als Rückgabe liefert es eine Selbstreferenz auf das eigene `Convertable`, um so z.B. [Method-Chaining](#) zu ermöglichen.
- `get()`: Liefert die transformierten Daten im Datentyp `T`.
- `restore()`: Wandelt die konvertierten Daten wieder in den Ursprungstyp `F`. Das Resultat muss dabei nicht 1:1 identisch mit den Eingabedaten sein. Auf diese Weise lässt sich ein `Convertable` `c` zur Normalisierung von Daten verwenden (`c.transform(data).reverse()`).

Zur besseren Übersicht sei Ihnen für diese Aufgabe folgendes UML-Klassendiagramm an die Hand gegeben.

Die Klasse `Replace` implementiert `Convertable` und dient zur Veranschaulichung. In der `main()`-Methode finden Sie Aufrufbeispiele wie `Replace` dazu genutzt werden kann in Texten Worte, die durch ein oder mehrere Whitespaces getrennt sind, so zu normalisieren, dass Worte immer nur durch ein Leerzeichen getrennt sind.

Sie sollen nun eine Methode `sort()` und die Klasse `Node` (sortierter Binärbaum) implementieren und diese Datenstruktur in einer eigenständig generischen Methode `sort()` als interne Datenstruktur verwenden, um Listen von `Comparable`-Objekten aufsteigend sortieren zu können.

Hinweise: Bitte durchlesen und berücksichtigen bevor Sie beginnen!!!

- Sie müssen aus dem UML-Diagramm die Schnittstelle `Convertable`, die Klasse `Node` und die Methode `sort()` der Klasse `Main` implementieren.
 - Die Klasse `Replace` ist Ihnen als Beispiel für ein `Convertable` bereits gegeben und muss **nicht** durch Sie entwickelt werden. Sie können die Klasse `Replace` als Anschauungsobjekt nutzen und auf `Node` übertragen.
- Die Methode `sort()` ist tatsächlich extrem kurz. Vermutlich ist der Methodenkopf länger als Ihre Implementierung, wenn Sie die Klasse `Node` im Sinne eines `Convertable` implementieren.

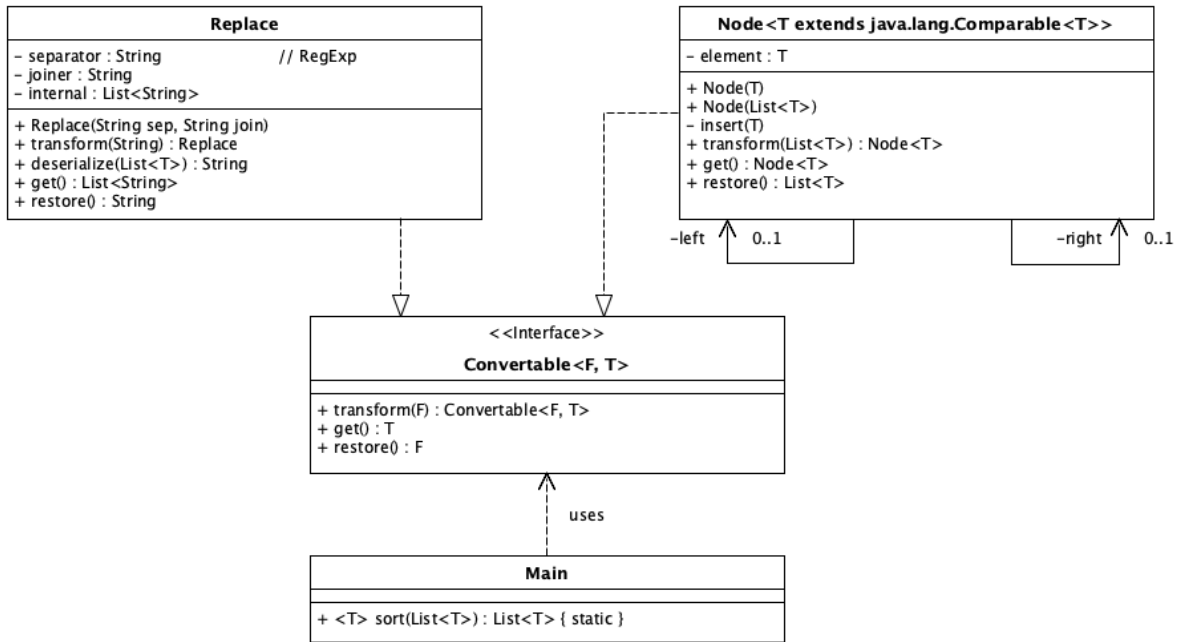


Abbildung 8.2: IMG

- Im Gegensatz zu Replace ist die Klasse Node generisch. Die Klasse Node kann jedoch nicht beliebige Datentypen verarbeiten, sondern nur Objekte, die die Comparable Eigenschaft haben, da diese zum Sortieren benötigt wird.
- Das Comparable Interface gehört zum Java Standard Sprachumfang und ist durch Sie **nicht** zu entwickeln.
- Sehen Sie sich noch einmal Binsort aus dem ersten Semester (Unit 05).

Main.java:

```

class Main {

    public static void main(String[] args) {

        // (1) Veranschaulichung des Replace Convertable
        Convertable<String, List<String>> s = new Replace("\\s+", "+");
        s.transform("Dies ist nur\n ein Beispiel.");
        System.out.println(s.restore());
        // => "Dies+ist+nur+ein+Beispiel."

        // (2) Veranschaulichung der `sort()`-Methode

        System.out.println(sort(Arrays.asList("Dies", "ist", "nur", "ein", "Beispiel")));
        // => [Beispiel, Dies, ein, ist, nur]

        System.out.println(sort(Arrays.asList(1, 7, 3, 6, 2, 4, 5)));
        // => [1, 2, 3, 4, 5, 6, 7]

        System.out.println(sort(Arrays.asList('a', 'b', 'B', 'A')));
        // => [A, B, a, b]
    }
}
  
```

Node.java:

```
public class Node {
}

```

Replace.java:

```
public class Replace implements Convertable<String, List<String>> {

    /**
     * Regulärer Ausdruck an dem eine Zeichenkette in Teilzeichenkette
     * getrennt werden soll (z.B. "\\s+", trennen an einem oder mehreren
     * Whitespace-Zeichen).
     * Wird im Konstruktur definiert.
     */
    private String separator;

    /**
     * Zeichenkette die zwischen alle Zeichenkette einer Liste von
     * Zeichenketten (internes Format) bei der `restore()`-Operation
     * gesetzt werden soll.
     * Wird im Konstruktur definiert.
     */
    private String joiner;

    /**
     * Interne Datenrepräsentation (T). Liste von Zeichenketten.
     */
    private List<String> internal = new LinkedList<>();

    /**
     * Konstruktor zum Anlegen des `Convertible`-Objekts.
     * @param sep Regulärer Ausdruck zum Trennen einer Zeichenkette (z.B. "\\s+",
     *           trennen an einem oder mehreren Whitespace-Zeichen)
     * @param join Trennzeichenkette, die zwischen jedes Zeichen bei der restore()
     *           gesetzt werden soll (z.B. " ")
     */
    public Replace(String sep, String join) {
        this.separator = sep;
        this.joiner = join;
    }

    /**
     * Transformiert eine Zeichenkette in eine Liste von Zeichenketten
     * indem die Zeichenkette an Separatorzeichenkette `separator` gesplittet wird.
     * `separator` wird im Konstruktor des `Convertible` gesetzt.
     * @param from Zu transformierende Zeichenkette.
     */
    public Replace transform(String from) {
        internal.addAll(Arrays.asList(from.split(separator)).stream()
            .map(s -> s.trim())
            .filter(s -> !s.isEmpty())
            .collect(Collectors.toList()));
        return this;
    }

    /**
     * Liefert das Resultat der Transformation.
     * @return Liste von Zeichenketten
     */
}

```

```

public List<String> get() {
    return this.internal;
}

/**
 * Wandelt die Transformation wieder in das ursprüngliche
 * Datenformat (hier: String) um, indem zwischen jede
 * Zeichenkette des internen Formats (Liste von Zeichenketten)
 * eine Trennzeichenkette `join` gesetzt wird.
 * `join` wird im Konstruktor gesetzt.
 * @return Gejointe Zeichenkette
 */
public String restore() {
    return this.get().stream().collect(Collectors.joining(joiner));
}
}

```

8.3 Eigenständig generische selector() Methode

Entwickeln Sie nun bitte eine **generische** Methode `selector()`, die aus einer Liste von Werten beliebigen Typs `T` diejenigen Werte selektiert, die einer booleschen Bedingung genügen.

- Wird in `selector()` keine boolesche Bedingung als zweiter Parameter angegeben, so soll `selector()` alle von `null` verschiedenen Werte zurückgeben.
- Ist der zweite Parameter `null`, so soll `selector()` ebenfalls alle von `null` verschiedenen Werte zurückgeben.
- Ist der erste Parameter `null`, so soll `selector()` eine leere Liste zurückgeben.

Aufrufbeispiele finden Sie in der `main()`-Methode.

Hinweise:

- **ALLE** `selector()`-Methoden müssen eigenständig generisch implementiert sein.
- Erinnerung: Optionale Parameter (hier der zweite Parameter) lassen sich mittels überladenen Methoden (Methoden gleichen Namens mit unterschiedlichen Parametersätzen) realisieren.

Main.java:

```

class Main {

    public static <T> List<T> selector(List<T> vs) {
        return vs.stream().filter(b -> b != null).collect(Collectors.toList());
    }

    public static <T> List<T> selector(List<T> vs, Predicate<T> p) {
        if (vs == null) return Arrays.asList();
        if (p == null) selector(vs);
        return selector(vs).stream().filter(p).collect(Collectors.toList());
    }

    public static void main(String[] args) {

        List<Integer> values = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
        List<Integer> result = selector(values, i -> i % 2 == 0);
        System.out.println(result);
        // => [2, 4, 6, 8]

        List<String> words = Arrays.asList(

```

```
        "Dies", "ist", "nur", "ein", "dummes", "Beispiel"
    );
    System.out.println(selector(words, w -> w.length() <= 3));
    // => [ist, nur, ein]

    List<Boolean> withNulls = Arrays.asList(
        true, null, false, null, true, null, false, null
    );
    System.out.println(selector(withNulls));
    // => [true, false, true, false]
}
}
```

Kapitel 9

Dank an ...

Auch dieser Katalog ist nicht perfekt und wird nur mit jedem Semester besser. Studierende oder Mitarbeiter/innen finden Fehler in Aufgabenstellungen oder entwickeln Lösungen, die einfach genialer als die Musterlösungen sind. Vielen Dank für alle Hinweise und Ideen zu Lösungen oder Aufgaben, die häufig dem Prinzip "Think outside the box" folgen.

- WiSe 2012/13: Eric Massenbergr und Rebecca Wunderlich
- SoSe 2013: Georg Schnabel und Michael Breuker
- WiSe 2013/14: Patrick Schuster, Chris Deter und René Kremer
- SoSe 2014: Finn Kothe
- WiSe 2014/15: Florian Löhden
- SoSe 2015: Dario Lehmhus, Tim Faltin und Torge Tönnies
- WiSe 2015/16: Marco Torge Gabrecht, Jan-Marco Bruhns und Robert Vagt und Max Sternitzke
- SoSe 2016: Duc Tu Le Anh, Yevhenii Vasylenko, Marco Gabrecht, Robert Vagt
- WiSe 2016/17: Alenka Rixen
- SoSe 2017 und SoSe 2018: David Engelhardt
- WiSe 2018/19: Patrick Willnow (der den Anstoß für den Einsatz von VPL gegeben hat)
- SoSe 2019: Eric De Ron (Bug in Unit 06 Auto-Klasse Evaluation gefunden)
- SoSe 2019: Tom Hüttmann, Denzel Kuhlemaan (Bugs in Possible Chessmen gefunden)

Ich entschuldige mich hiermit bei allen, die ich vergessen haben sollte und verspreche sie in diese Liste aufzunehmen.