Kommentierte Vorlesungsbeispiele (Java)

Nane Kratzke

16.09.2019

Zusammenfassung

Dieses Dokument führt alle im Rahmen der Vorlesung gezeigten Beispiele Unit für Unit auf. Das Dokument wird mit dem Vorlesungsverlauf kontinuierlich fortgeschrieben. Es soll ihnen ein Überblick über die in der Vorlesung gemachten Beispiele geben und die Beispiele mit ergänzenden Erläuterungen und Kommentaren versehen. Es wird mit einem einfachen `Hello World` Beispiel begonnen, welches kontinuierlich ausgebaut wird. Nutzen Sie dieses Dokument im Sinne des **Computational Thinkings** um ihre **Pattern Recognition** Fähigkeiten zu trainieren. Häufig lassen sich diese Beispiele adaptieren, um Lösungsansätze für Probleme im begleitenden **Trimm-Dich-Pfad** zu finden. Am Ende der jeweiligen Kapitel finden Sie eine Übersichtsliste welche Konzepte und Methoden in den Beispielen erstmalig behandelt wurden.

Inhaltsverzeichnis

1 Unit 01 (Einleitung)

1.1 Hello World

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

Wir beginnen mit dieser Variante des klassischen `Hello World` Beispiels als Einstieg. Dieses Beispiel soll häufig erforderliche Programmiermuster veranschaulichen und "greifbarer" machen, die erforderlich sind, schnell erste kleine Programme zu schreiben. Es ist nicht das Ziel bereits alle Einzelheiten im Detail zu erläutern. Exemplarisch vermittelt werden sollen vor allem die folgenden und permanent wiederkehrenden Grundlagen:

1. Ausgabe auf der Konsole mittels `System.out.println("Hello World")`.
2. Eingabe von der Konsole mittels `Scanner in = new Scanner(System.in)`.
3. Ausführung von Sequenzen von Anweisungen.
4. Generierung von Zeichenketten mittels eines Patterns, wie bspw. `"Hello <name>!"`.
5. Auslagern von Logik in Methoden (hier `greet()`).
6. Hinweis auf Methoden, die die Klasse `String` anbietet, wie bspw. `trim()`.

Ausgaben auf der Konsole kann man so erzeugen.

```
public class Xample {
    public static void main(String[] args) {
        System.out.println("Hello World"); // => Hello World
    }
}
```

Eingaben über die Konsole und Ausführung von Sequenzen von Anweisungen.

```
import java.util.Scanner;

public class Xample {
    public static void main(String[] args) {
        System.out.println("Wie heißt Du?");
        Scanner in = new Scanner(System.in);
        String name = in.nextLine();
```

```
        System.out.println("Hello " + name + "!");
    }
}
```

Der nächste Schritt ist das Auslagern der Grußgenerierungslogik in eine Methode namens `greet()`. Methoden sind von Anfang an insbesondere für VPL-Umgebung (Virtual Programming Lab) in Moodle erforderlich. VPL nutzt Methoden als Einstiegspunkte für eine Autoevaluierung im Rahmen des **Trimm-Dich-Pfads** und der **Präsenzaufgaben** (Klausurbonus).

Ohne Methoden werden Sie nicht einen Punkt im Praktikum bekommen.

Methoden ermöglichen es (komplexe) Logik, z.B. zur Behandlung von Sonderfällen, zu kapseln und zu parametrisieren (**Dekomposition** im Sinne des Computational Thinkings). Der Einsatz von `trim()` behandelt Fälle, wenn der Nutzer ggf. gar keinen Namen oder unsinnige führende oder folgende Leerzeichen (Whitespaces) eingegeben hat, die beseitigt werden müssen, um eine "ansprechende" Ausgabe zu generieren.

```
import java.util.Scanner;

public class Xample {

    public static String greet(String n) {
        String normalized = n.trim();
        String hello = ("Hello " + name).trim();
        return hello + "!";
    }

    public static void main(String[] args) {
        System.out.println("Wie heißt Du?");
        Scanner in = new Scanner(System.in);
        String name = in.nextLine();
        String sayHello = greet(name);
        System.out.println(sayHello);
    }
}
```

1.2 Zeichen und Leerzeichen zählen

Dieses Wiederholungsbeispiel zu Unit 01 am Anfang der Unit 02 verfeinert das `Hello World` Beispiel aus Unit 01 etwas. Es soll nun nicht gegrüßt, sondern Zeichen und Leerzeichen einer Eingabe gezählt werden. Das Grundmuster (Eingabe -> Zeichenkettengenerierung) bleibt aber bestehen.

Wir beginnen mit einer einfachen Interaktion:

```
import java.util.Scanner;

public class Xample {
    public static void main(String[] args) {
        System.out.println("Bitte geben Sie einen Satz ein: ");
        Scanner in = new Scanner(System.in);
        String satz = in.nextLine();
        System.out.println(satz);
```

```
    }
}
```

Es soll nun aber nicht nur der Satz ausgegeben werden, sondern berechnet werden wieviele Zeichen der Satz hat. Die Logik soll in einer Methode `zeichen()` realisiert werden.

```
import java.util.Scanner;

public class Xample {
    public static void main(String[] args) {
        System.out.println("Bitte geben Sie einen Satz ein: ");
        Scanner in = new Scanner(System.in);
        String satz = in.nextLine();
        int zs = zeichen(satz);
        System.out.println("Der Satz hat " + zs + " Zeichen.");
    }

    public static int zeichen(String s) {
        return s.length();
    }
}
```

Die Methode `zeichen()` kann also trivial auf die Methode `length()` abgebildet werden.

Wir verkomplizieren daher das Beispiel. Es sollen nun nur noch Leerzeichen gezählt werden. Da das Konzept von Schleifen noch nicht eingeführt wurde, wird darauf hingewiesen, dass es eine Methode `replaceAll()` gibt, mit der Zeichenvorkommen in einer Zeichenkette ersetzt werden können. Diese kann dazu genutzt werden, um zu zählen.

```
import java.util.Scanner;

public class Xample {
    public static void main(String[] args) {
        System.out.println("Bitte geben Sie einen Satz ein: ");
        Scanner in = new Scanner(System.in);
        String satz = in.nextLine();
        int zs = zeichen(satz);
        int lzs = leerzeichen(satz);
        System.out.println("Der Satz hat " + zs + " Zeichen.");
        System.out.println("Der Satz hat " + lzs + " Leerzeichen.");
    }

    public static int zeichen(String s) {
        return s.length();
    }

    public static int leerzeichen(String s) {
        int original = s.length();
        String ohneLeerzeichen = s.replaceAll(" ", "");
        return original - ohneLeerzeichen.length();
    }
}
```

Neue Logik kann also aus bestehender Logik "zusammengesetzt" werden.

Das ist Dekomposition im Sinne des Computational Thinkings.

1.3 Eingeführte Konzepte und Methoden der Unit 01

In dieser Unit wurden die folgenden Dinge exemplarisch erläutert.

- Alle Java Programme beginnen in der `main()`-Methode
- Konsolenausgabe mit `System.out.println()`
- Konsoleneingabe mit `Scanner` und `nextLine()`
- Datentyp `String` und `int`
- Zeichenkettenkonkatenation mit +
- Die Stringmethoden `trim()`, `replaceAll()` und `equals()`
- Variablendeklaration (am Beispiel von `String` und `int`)
- Methodendeklaration (Dekomposition) als Erfordernis für VPL
- `import` Anweisung um Java-Klassenbibliotheken zu nutzen (meist nur aus dem `java.util`-Package).

Wir haben u.a. zwei Datentypen `String` (Zeichenketten) und `int` (natürliche Zahlen) kennengelernt. In der Unit 02 werden nun weitere Datentypen eingeführt.

2 Unit 02 (Datentypen, Operatoren, Kontrollstrukturen, Methoden)

Hinweis: *Auf Klassenangaben wie `public class Xample {}` wird ab sofort aus Platzgründen verzichtet (sofern die Beispiele nur eine Klasse benötigen und es daher keine Verwechslung geben kann). Es werden nur noch die Methoden und die `import` Anweisungen angegeben.*

2.1 Worte zählen (Datentypen)

Dieses Beispiel am Ende des **Datentypen-Teils der Unit 02** soll das Zeichen-zählen-Beispiel aus Unit 01 vertiefen. Das Grundmuster (Eingabe -> Zeichenkettengenerierung) bleibt aber weiterhin bestehen. Es sollen nun nicht nur Zeichen gezählt werden, sondern Worte. Das Problem bei Worten ist, dass Worte ungleich lang sein können, aber jedes Wort (egal ob kurz oder lang) dieselbe Wertigkeit für das Zählen hat.

Wir müssen also eine Zeichenkette in Teilzeichenketten (Worte) aufteilen (`splitten`) und nur die Anzahl der Worte zählen. Hierfür gibt es die Methode `split()`, die eine Zeichenkette anhand eines "Trennmusters" trennt und in einer "Liste" (`Array`) zurückgibt. Arrays von Zeichenketten lassen sich mittels `String[]` deklarieren und Worte sind in Sätzen durch ein oder mehrere Leerzeichen getrennt (`" +"`).

```
import java.util.Scanner;
import java.util.Arrays;

public static void main(String[] args) {
    System.out.println("Bitte geben Sie einen Satz ein: ");
    Scanner in = new Scanner(System.in);
    String satz = in.nextLine();

    String[] worte = satz.split(" +");
    System.out.println(Arrays.toString(worte));
```

```
    // Ergibt bei Eingabe: "Dies ist nur ein Beispiel"
    // [Dies, ist, nur, ein, Beispiel]
}
```

Wie Zeichenketten eine `length()` Methode haben, so hat auch ein Array ein Datenfeld `length`, dass Auskunft über die Anzahl an Elementen in diesem Array gibt.

Vergleichbar zur Unit 01 (vgl. die Methoden `zeichen()` und `leerzeichen()`) soll nun mit dieser Kenntnis eine Methode `worte()` entwickelt werden.

```
import java.util.Scanner;
import java.util.Arrays;

public static void main(String[] args) {
    System.out.println("Bitte geben Sie einen Satz ein: ");
    Scanner in = new Scanner(System.in);
    String satz = in.nextLine();
    int ws = worte(satz);
    System.out.println("Der Satz hat " + ws + " Worte.");
}

public static int worte(String s) {
    String clean = "[,;\.:!?]"; // Satzzeichen die gecleaned werden sollen.
    String[] ws = s.split(" +");
    return ws.length;
}
```

2.2 Gerade oder ungerade Anzahl an Worten in einem Satz (Operatoren)

Dieses Beispiel am Ende des **Operator-Teils der Unit 02** soll das Worte-zählen-Beispiel vertiefen. Das Grundmuster (Eingabe -> Zeichenkettengenerierung) bleibt aber weiterhin bestehen. Es sollen nun nicht nur Worte gezählt werden, sondern bestimmt werden, ob ein Satz eine gerade oder ungerade Anzahl an Worten hat, dabei kann natürlich auf die Methode `worte()` zurückgegriffen werden **(Decomposition)**.

Erfahrene Programmierer würden vermutlich auf die bedingte Anweisung (`if`) zurückgreifen. Diese ist aber noch nicht eingeführt. Das Beispiel soll daher auch zeigen, dass es andere Möglichkeiten gibt einfache Wenn-dann-Situationen programmiertechnisch auszudrücken.

```
import java.util.Scanner;
import java.util.Arrays;

public static void main(String[] args) {
    System.out.println("Bitte geben Sie einen Satz ein: ");
    Scanner in = new Scanner(System.in);
    String satz = in.nextLine();
    String g = gerade(satz) ? "gerade" : "ungerade";
    System.out.println("Der Satz hat eine " + g + " Anzahl an Worten.");
}

public static boolean gerade(String s) {
    boolean g = worte(s) % 2 == 0;
    return g;
```

```
    // Kann vereinfacht werden zu:
    // return worte(s) % 2 == 0;
}

public static int worte(String s) {
    String[] ws = s.split(" +");
    return ws.length;
    // Kann vereinfacht werden zu:
    // return s.split(" +").length;
}
```

> **Die bedingte Auswertung `cond ? a : b` ist häufig ausreichend,**
> *um einfache Wenn-Dann-Situation abzubilden und macht – geschickt eingesetzt – den Code meist kompakter.*

2.3 Durchschnittliche Wortlänge, längestes Wort und Worte bestimmer Länge in einem Satz bestimmen (Kontrollanweisungen)

Dieses Beispiel am Ende des **Kontrollanweisungs-Teils der Unit 02** soll das Worte-zählen-Beispiel abwandeln. Das Grundmuster (Eingabe -> Zeichenkettengenerierung) bleibt weiterhin bestehen. Es sollen nun aber nicht nur Worte gezählt werden, sondern die durschschnittliche Wortlänge eines Satzes bestimmt werden.

Zur Bestimmung der durchschnittlichen Wortlänge müssen wir tatsächlich jedes Wort auswerten (das war in den vorhergehenden Beispielen nicht erforderlich). Die Anzahl der Worte ist uns im Vorfeld nicht bekannt, d.h. das Problem ist nicht effektiv mit einer statischen Anzahl von Anweisungen lösbar. Wir müssen daher abhängig von der Wortanzahl diverse Analyseanweisungen pro Wort wiederholend durchführen.

Schleifen sind hierfür sinnvoll. Wir beginnen mit einer einfachen Auflistung aller Worte auf der Konsole.

```
public static void main(String[] args) {
    System.out.println("Bitte geben Sie einen Satz ein: ");
    Scanner in = new Scanner(System.in);
    String satz = in.nextLine();
    String[] worte = satz.split(" +");
    for (String wort : worte) {
        System.out.println(wort);
    }
    // Beliebte Variante mit zählender for-Schleife
    // aber eigentlich viel fehleranfälliger
    //
    // for (int i = 0; i < wort.length; i++) {
    //    System.out.println(wort[i]);
    // }
}
```

> **Die foreach Schleife `for(:)` ist meist ausreichend**
> *und weniger fehleranfällig als die zählende for-Schleife.*

Nun soll die durchschnittliche Wortlänge eines Satzes mittels einer Methode `average()` bestimmt

werden. Ergänzend wollen wir das längste Wort eines Satzes mittels `longest()` und alle Worte einer gegebenen Länge mittels `wordsOfLength()`bestimmen, um Ähnlichkeiten (`for` Schleife) zwischen allen genannten Problemen zu erkennen.

Zu Beginn soll das Gesamtproblem in die folgenden Teilprobleme gegliedert werden:

- Alle Worte sollen mittels `words()` aus einer Zeichenkette bestimmt werden.
- Auf Worten soll die durchschnittliche Wortlänge mittels `average()` berechnet werden.
- Auf Worten soll ein längstes Wort mittels `longest()` bestimmt werden.
- Aus Worten sollen alle Wörter einer gegebenen Länge mitels `wordsOfLength()` bestimmt werden.

```
import java.util.Scanner;

public static void main(String[] args) {
    System.out.println("Bitte geben Sie einen Satz ein: ");
    Scanner in = new Scanner(System.in);
    String satz = in.nextLine();
    double avg = average(satz);
    System.out.printf("Durchschnittliche Wortlänge von %.2f Zeichen.", avg);
    // Oder auch:
    // String out = String.format(
    //     "Durchschnittliche Wortlänge von %.2f Zeichen.", avg
    // );
    // System.out.println(out);
}

public static String[] words(String s) {
    String clean = "[,;\\.:!?\"\']"; // Satzzeichen die gecleaned werden sollen.
    String[] worte = s.replaceAll(clean, " ").trim().split("\s+");
    return worte;
}

public static double average(String s) {
    int sum = 0;
    for (String w : words(s)) {
        sum += w.length();
    }
    return sum / (double) worte.length;
    // Ohne cast würde die ganzzahlige Division
    // erfolgen (der Nachkommaanteil würde nicht
    // berücksichtigt werden).
}

public static String longest(String s) {
    String longest = "";
    for (String w : words(s)) {
        if (w.length() > longest.length()) longest = w;
    }
    return longest;
}

public static String[] wordsOfLength(String s, int n) {
```

```
    String result = "";
    for (String w : words(s)) {
        if (w.length() == n) result += w + ",";
    }
    return result.split(",");
}
```

> **Die unbeabsichtigte ganzzahlige Division passiert recht schnell**
> *und ist auf den ersten Blick schwer zu erkennen, weil in Formeln meist keine expliziten Datentypen stehen!*

2.4 Städte raten (Methoden und Kontrollanweisungen)

Dieses Wiederholungsbeispiel am Anfang des **Methoden-Teils der Unit 02** soll das Wortlängen-Beispiel abwandeln. Es soll ein kleines Konsolen-basiertes Spiel entwickelt werden. Aus einer gegebenen Liste von Städten, soll das Programm eines per Zufall auswählen und der Nutzer soll es in einer vorgegebenen Anzahl an Fehlversuchen erraten. Alle bislang nicht geratenen Buchstaben sollen "ausgeblankt" werden, so dass sich Ausgaben wie bspw. `"L_BE_K"` (für die zu ratende Stadt LÜBECK) ergeben, wenn bereits `"Ü"` und `"C"` eingegeben wurden.

Das Problem soll – um **Dekomposition** zu üben – dahingehend untersucht werden, welche Teilprobleme identifizierbar sind, die implementiert werden können, bevor man sich der eigentlichen Spiellogik widmet. In diesem Beispiel bieten sich folgende Methoden an:

- `randomTown()` soll eine Stadt zufällig bestimmen.
- `blank()` soll noch nicht geratene Zeichen aus einer Zeichenkette ausblanken.

```
public static String randomTown() {
    String[] towns = {
        "San Francisco", "Boston", "Berlin", "Barcelona",
        "Funchal", "London", "Paris", "Gatineau", "Rom",
        "Ottawa", "Montreal",  "Tunis", "Kairo", "Venedig",
        "Los Angeles", "Johannesburg", "Lübeck", "Hamburg",
        "München", "Stockholm", "Kopenhagen", "Nizza"
    };
    int n = (int)(Math.random() * towns.length);
    return towns[n].toUpperCase();
}

public static String blank(String word, String guessed) {
    String ret = "";
    for (String c : word.toUpperCase().split("")) {
        boolean show = guessed.toUpperCase().contains(c) || c.equals(" ");
        ret += show ? c : "_";
    }
    return ret;
}
```

Unter Nutzung dieser beiden Methoden lässt sich die `main()` Methode recht einfach formulieren. Es bietet sich eine `while` Schleife an, für die eine logische Bedingung formuliert werden muss, die angibt, ob der Nutzer weiter Rateversuche abgeben darf.

```
import java.util.Scanner;
```

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    String town = randomTown();
    String guessed = "";
    int tries = 5;

    // Solange noch nicht alles geraten wurde und noch
    // Fehlversuche erlaubt sind
    while (tries > 0 && blank(town, guessed).contains("_")) {
        String blanked = blank(town, guessed);
        System.out.println("Raten Sie diese Stadt: " + blanked);
        System.out.printf("Noch %d Fehlversuche: ", tries);
        String g = in.nextLine().toUpperCase();
        if (town.contains(g) && g.length() == 1)
            guessed += g; // Erfolgreicher Rateversuch
        else
            tries--;      // Nicht erfolgreicher Rateversuch
    }

    // Spielauswertung
    if (blank(town, guessed).contains("_"))
        System.out.println("Verloren: " + town.toUpperCase());
    else
        System.out.println("Gewonnen: " + town.toUpperCase());
}
```

Methoden sind ein wichtiges Hilfsmittel zur Dekomposition.
In Methoden gekapselte Logik ist auch in anderen Kontexten wiederverwendbar.

2.5 Eine sehr einfache Suchmaschine (Methoden mit variabler Parameteranzahl)

Dieses Beispiel am Ende des **Methoden-Teils der Unit 02** soll das Wortlängen-Beispiel weiterentwickeln. Es sollen nun nicht mehr nur durchschnittliche Wortlängen eines Satzes bestimmt werden, sondern aus einem Satz die Worte bestimmt werden, die einer variablen Anzahl an Suchwörtern entsprechen. Dieses Beispiel greift viele Einzelaspekte vorhergehender Beispiele auf und ist komplexer, um die **Dekomposition** in Teilmethoden zu motivieren.

Hierzu soll eine Methode `searchFor()` entwickelt werden, die in einer Zeichenkette die Worte selektiert, die die Suchwörter beinhalten. Die Suchworte sollen mit eckigen Klammern `[]` in den Treffern markiert werden (z.B. das Suchwort `"spiel"` im Wort `"Bei[spiel]"`).

Die Suchworte werden als **variable Parameteranzahl** übergeben, d.h. `searchFor()` kann nach beliebig vielen – d.h. ein, zwei, drei, oder mehr – Suchworten suchen.

Viele Studierende vergessen die Bedeutung des Parameter-Modifiers . . .
und werden in der Klausur davon "überrascht". Gehören Sie nicht dazu ;-)

```
import java.util.*;

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

```
        System.out.println("Bitte geben Sie einen Satz ein: ");
        Scanner in = new Scanner(System.in);
        String satz = in.nextLine();
        for (String wort : searchFor(satz, "nur", "ei", "spiel")) {
            System.out.println("Treffer: " + wort);
        }
    }
```

Das Problem soll ergänzend in Teilprobleme mittels Methoden gegliedert werden, um **Dekomposition** zu üben:

- Worte sollen mittels `words()` bestimmt und "normalisiert" (d.h. u.a. Satzzeichen und unnötige Leerzeichen entfernt) werden.
- Suchbegriffe sollen in einem Wort mittels `markHit()` markiert werden.

Für eine gegebene Zeichenkette bestimmt die Methode `words()` normalisierte Einzelwörter ohne Satzzeichen und Whitespaces. Die Satzzeichen werden mittels eines regulären Ausdrucks durch Leerzeichen ersetzt.

```
    public static String[] words(String s) {
        String zeichen = "[.,;:!?'\"]";
        return s.replaceAll(zeichen, " ").trim().split(" +");
        // Dies kann noch kompakter ausgedrückt werden:
        // return s.replaceAll("[.,;:!?'\"]", " ").trim().split(" +");
    }
```

Alle Suchbegriffe (`terms`) in einem normalisierten Wort können mittels der Methode `markHit()` markiert werden.

```
    public static String markHit(String hit, String... terms) {
        String marked = hit;
        for (String t : terms) {
            marked = marked.replaceAll(t, "[" + t + "]");
        }
        return marked;
    }
```

Mit diesen beiden Methoden kann die Methode `searchFor()` dann wie folgt formuliert werden.

```
    public static String[] searchFor(String s, String... terms) {
        // Worte selektieren
        String hits = "";
        for (String word : words(s)) {
            for (String term : terms) {
                if (hits.contains(word)) continue;
                if (word.contains(term)) hits += word + ",";
            }
        }
        // Selektierte Worte markieren
        String result = "";
        for (String hit : hits.split(",")) {
            result += mark(hit, terms) + ",";
        }
        return result.split(",");
    }
```

In einem weiteren Schritt kann die `searchFor()`-Methode in zwei weitere Methoden `select()` und `markAll()` gegliedert werden.

```
public static String[] select(String s, String... terms) {
    String hits = "";
    for (String word : words(s)) {
        for (String term : terms) {
            if (hits.contains(word)) continue;
            if (word.contains(term)) hits += word + ",";
        }
    }
    return hits.split(",");
}

public static String[] markAll(String[] hits, String... terms) {
    String marked = "";
    for (String hit : hits) marked += markHit(hit, terms) + ",";
    return marked.split(",");
}
```

Dadurch vereinfacht sich die `searchFor()`-Methode weiter zu:

```
public static String[] searchFor(String s, String... terms) {
    String[] hits = select(s, terms);
    return markAll(hits, terms);
    // Dies kann noch kompakter ausgedrückt werden:
    // return markAll(select(s, terms), terms);
}
```

Die gesamte Suchlogik von `searchFor()` wurde in folgende Methoden aufgespalten. Die Einrückung zeigt an, wie die Methoden einander aufrufen.

```
-> String[] searchFor(String s, String... terms)    // 1 Zeile
   |
   +-> String[] select(String s, String... terms)   // 8 Zeilen
   |   |
   |   +-> String[] words(String s)                 // 1 Zeile
   |
   +-> String[] markAll(String[], String... terms)  // 3 Zeilen
       |
       +-> String markHit(String, String... terms)  // 5 Zeilen
```

2.6 Eingeführte Konzepte und Methoden der Unit 02

- `String`-Methode `split()` zum Trennen von Zeichenketten mittels Mustern
- `System.out.printf()` und `String.format()` zum Formatieren von Daten (finden Sie in Unit 01)
- Einfache **reguläre Ausdrücke** wie `" +"` (Sequenz von Zeichen, hier ein oder mehrere Leerzeichen) und `"[.,;!?]"` (Auswahl von Zeichen, hier eines der angegeben Satzzeichen)
- Deklaration von Zeichenketten-**Arrays** mittels `String[]`
- `Arrays.toString()` für eine lesbare Konsolenausgabe von Arrays
- Datentypen `boolean` und `double`
- Vergleichsoperator == und die bedingte Auswertung `cond ? expr1 : expr2`

- Arithmetischer Modulzo Operator %
- Integer-Division versus Fließkomma-Division mit dem /-Operator
- `Math.random()` für gleichverteilte Zufallszahlen zwischen 0.0 und 1.0
- Casting am Beispiel `int` -> `double` mittels `(double)`
- Die for-each Schleife `for(variable : liste)` als häufig einfachere und weniger fehleranfällige Variante der zählenden for-Schleife `for(index; bedingung; update)`
- `while` Schleife
- `if/else` Anweisung
- `continue`Anweisung
- **Variable Parameteranzahl** in Methoden (`...`-Parametermodifier)
- **Dekomposition** komplexer Logik in kleinere überschaubare Teilmethoden

Wir haben mit dem "Suchmaschinen"-Beispiel gesehen, dass wir nicht nur Einzelberechnungen vornehmen können, sondern auch größere Mengen von Daten bearbeiten können. Arrays ermöglichen dies bereits in begrenztem Umfang. Wesentlich flexiblere Datenstrukturen (z.B. Listen und Mappings) werden in der nun folgenden Unit 03 eingeführt werden.

3 Unit 03 (Arrays, Klassen, Collections – insb. `List` und `Map`)

3.1 Rauten (Diamonds) erzeugen und ausgeben

Da wir aufgrund der Nutzung der `String.split()` Methode in Unit 02 bereits viel mit eindimensionalen Arrays zu tun hatten, soll dieses Beispiel vor allem den Umgang mit zweidimensionalen Arrays veranschaulichen. Wir beginnen mit einem einfachen zweidimensionalen Array (oder präziser einem Array von Arrays), wie z.B.

```
char[][] raute = {
    { ' ', ' ', '*', ' ', ' ' },
    { ' ', '*', '*', '*', ' ' },
    { '*', '*', '*', '*', '*' },
    { ' ', '*', '*', '*', ' ' },
    { ' ', ' ', '*', ' ', ' ' }
};
```

Diese soll mittels einer Funktion `toString()` wie folgt

```
System.out.println(toString(raute));
```

auf der Konsole ausgegeben werden können.

```
  *
 ***
*****
 ***
  *
```

Die folgende Implementierung der Methode `toString()` realisiert dies, indem das zweidimensionale Array Zeile für Zeile (`row`) und jede Zeile Spalte für Spalte (`col`) durchlaufen wird. Am Ende einer jeden Zeile (d.h. nach der inneren `for`-Schleife) wird ein Zeilenumbruch `"\n"` eingefügt.

```
public static String toString(char[][] diamond) {
    String s = "";
    for (char[] row : diamond) {
        for (char col : row) s += col;
```

```
            s += "\n";
        }
        return s;
    }
```

Wir erweitern die Aufgabe nun so, dass das Array `raute` mittels einer Methode `diamond()` erstellt werden können soll. Die Methode `diamond()` soll dabei das Problem parameterisieren und Rauten beliebiger Größe `n` und mit beliebigen Zeichen `c` erzeugen können.

Ist `n` gerade soll die nächst kleinere Raute `n - 1` erzeugt werden.

```
char[][] raute = diamond(5, '*'); // oder auch diamond(6, '*');
```

Die Lösungsidee lautet wie folgt. Die mittlere Spalte des zu erzeugenden Arrays ist immer mit einem Zeichen belegt. Mit jeder Zeile r bis zur mittleren Zeile $m = n/2$ der Raute kommen jeweils ein Zeichen links und rechts dazu. Die Breite w der Zeichen wächst bis zur Mitte m also jeweils um zwei Zeichen, ab der Mitte sinkt die Breite allerdings wieder mit jeder Zeile um zwei Zeichen.

Für jede Position (r, c) kann also durch folgende Bedingung

$p(r, c) = c >= m - w/2 \wedge c <= m + w/2$

geprüft werden, ob ein Zeichen `c` oder ein Leerzeichen gesetzt werden muss.

Wenn wir Bedingungsvariablen m, w, r, c wie folgt Java-Bezeichnern zuordnen

- m: `int mid` (Mitte, mittlere Spalte und Zeile)
- w: `int width` (Breite zu setzender Zeichen, abhängig von `row`)
- r: `int row` (Zeile)
- c: `int col` (Spalte)

kommen wir zu folgender Lösung für das Problem.

```
public static char[][] diamond(int n, char c) {
    int size = n % 2 == 0 ? n - 1 : n;
    char[][] data = new char[size][size];
    int mid = size / 2;
    int width = 1;

    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            data[row][col] = (col >= mid-width/2 && col <= mid+width/2) ? c : ' ';
        }
        width += row >= mid ? -2 : 2;
    }

    return data;
}
```

3.2 Eigene Datentypen definieren (am Beispiel der Klasse `Person`)

Bislang haben wir im Wesentlichen nur primitive Datentypen kennengelernt. Diese bilden die Basis für Datentypen höherer Ordnung. Dies soll am Beispiel von Personen deutlich gemacht werden.

Personen haben einen Vor- und einen Nachnamen (und ein Geburtsjahr). Bislang könnten wir das in Java z.B. durch drei Variablen ausdrücken:

```
String vorname;
String nachname;
int geburtsjahr;
```

Das Problem hierbei ist, dass diese Variablen beliebig sind und nicht klar ist, ob sich bspw. die Variablen `vorname` und `nachname` wirklich auf dieselbe Person beziehen. Um daher ausdrücken zu können, dass sich ein Satz an Variablen immer auf dasselbe Objekt bezieht, kann man diese in Klassen zusammenfassen. Die **Datenfelder** solcher Klassen (also in diesem Beispiel `vorname`, `nachname` und `geburtsjahr`) haben nicht den Modifier `static`. Das fehlende `static` drückt aus, dass diese Datenfelder zu einem Objekt gehören (und nicht statisch an eine Klasse gebunden sind). Objekte können beliebig häufig aus einer Klasse erzeugt werden, eine Klasse kann es aber immer nur einmal in einem Programm geben.

```
public class Person {
    public String vornamne;
    public String nachname;
    public int geburtsjahr;
}
```

Von dieser Klasse `Person` können dann mittels des `new`-Operators Objekte angelegt werden. Auf die einzelnen Komponenten dieses Objekts kann mittels der `.`-Notation lesend wie schreibend zugegriffen werden.

```
Person p = new Person();
p.vorname = "Max";
p.nachname = "Mustermann";
p.geburtsjahr = 1976;
System.out.println(p.vorname); // => Max
```

Auf diese Weise ist es möglich, einen Satz an Datenfeldern eindeutig einem Objekt zuzuordnen. Diese Zuordnung kann zur Laufzeit des Programms nie "verloren gehen".

3.2.1 Konstruktoren

Es ist dennoch recht umständlich ein Objekt erst anlegen, und dann alle Datenfelder des Objekts einzeln setzen zu müssen. Eleganter wäre, wenn man eine Person wie folgt anlegen könnte:

```
Person p = new Person("Max", "Mustermann", 1976);
System.out.println(p.vorname); // => Max
```

Hierzu kann man die Klasse mit einem **Konstruktor** versehen. Ein Konstruktor ist eine Art "Spezialmethode", die durch den `new`-Operator beim Anlegen eines Objekts aufgerufen wird. Innerhalb der Klasse kann ein Objekt auf die "eigenen" Datenfelder mittels der `this`-Referenz zugreifen. Der Konstruktor nutzt dass, um das Objekt beim Erstellen mit initialen Datenwerten zu belegen.

```
public class Person {
    public String vornamne;
    public String nachname;
    public int geburtsjahr;

    public Person(String vn, String nn, String gj) {
        this.vorname = vn;       // auch möglich: vorname = vn;
        this.nachname = nn;      // auch möglich: nachname = nn;
        this.geburtsjahr = gj;   // auch möglich: geburtsjahr = gj;
    }
}
```

Wenn die Datenfelder eindeutig bezeichnet sind kann auf die Angabe von `this` verzichtet werden.
Es wird aber empfohlen immer die `this`-Referenz zu verwenden, wenn auf Datenfelder innerhalb eines Objekts zugegriffen wird. Das macht es eindeutiger für den Leser der Klasse.

3.2.2 Zeichenkettenrepräsentationen von Objekten mittels `toString()`

Häufig benötigt man eine textuelle Repräsentation eines Objektzustands. Stellen Sie sich z.B. vor, Sie wollen eine `Person` ausgeben. In Java kann man dann so etwas schreiben.

```
Person p = new Person("Max", "Mustermann", 1976);
System.out.println(p);
```

Die Ausgabe ist aber meist nicht hilfreich:

```
Person@7a46a697
```

Das sagt nur aus, von welcher Klasse das Objekt ist und an welcher Stelle im Hauptspeicher es zu finden ist. Üblicherweise hilft einem das nicht weiter. In jeder Klasse in Java kann man jedoch eine **`toString()`**-Methode ergänzen, die in Fällen aufgerufen wird, in denen ein Objekt in einem Zeichenketten-Kontext genutzt wird (z.B. in einer `System.out.println()`-Anweisung). Eine `toString()`-Methode für die Klasse `Person` kann bspw. so aussehen und im Klassenblock (wie auch bereits der Konstruktor) ergänzt werden.

```
public String toString() {
    return this.vorname + " " + this.nachname + "(Geb: " + this.geburtsjahr + ")");
}
```

Mit dieser Methode erzeugt folgender Aufruf

```
Person p = new Person("Max", "Mustermann", 1976);
System.out.println(p);
```

dann auch eine besser interpretierbare Ausgabe:

```
Max Mustermann (Geb: 1976)
```

3.2.3 Objekte inhaltlich vergleichen mittels `equals()`

Stellen wir uns nun ergänzend folgenden Fall vor

```
Person p = new Person("Max", "Mustermann", 1976);
Person q = new Person("Maren", "Musterfrau", 1975);
Person r = new Person("Max", "Mustermann", 1976);
```

und stellen uns die Frage welche Objekte gleich sind. `p` und `q` sicherlich nicht. Aber `p` und `r` sehr wohl. Wenn wir den Gleichheitsoperator == nutzen, erhalten wir jedoch die Antwort, das weder `p`, `q` oder `r` gleich sind.

```
System.out.println(p == q); // => false
System.out.println(p == r); // => false
```

Das hat damit zu tun, wie der ==-Operator in Java definiert ist. Bei Referenztypen prüft == nicht, ob die Inhalte eines Objekts gleich sind, sondern ob die **Referenzen** gleich sind (d.h. beide Objekte an **derselben** Stelle im Hauptspeicher stehen). Der **new**-Operator erzeugt aber immer ein **neues** Objekt

an einer neuen Stelle im Hauptspeicher. Zwei Objekte mit den (zufällig) gleichen Werten ihrer Datenfelder sind also gem. == nicht gleich.

Will man die Inhalte zweier Objekte (und nicht deren Hauptspeicheradressen) vergleichen, muss man hierzu in Java die `equals()`-Methode nutzen. Die `equals()`-Methode muss aber erst definiert werden.

Üblicherweise wird Gleichheit zweier Objekte derart definiert, dass die Datenfelder zweier Objekte paarweise gleich sein sollten. Eine `equals()`-Methode für die Klasse `Person` kann im Klassenblock ergänzt werden.

```
public boolean equals(Person other) {
    return this.vorname.equals(other.vorname) &&      // Strings vergleichen
           this.nachname.equals(other.nachname) &&    // Strings vergleichen
           this.geburtsjahr == other.geburtsjahr;     // ints vergleichen
}
```

Und mit dieser Gleichheitsdefinition können wir unseren Vergleich nun inhaltlich durchführen (und erhalten das initial erwartete Ergebnis):

```
System.out.println(p.equals(q)); // => false
System.out.println(p.equals(r));  // => true
```

Beachten Sie also immer den Unterschied zwischen == und `equals()`.
*Wollen Sie zwei Objekte inhaltlich vergleichen, nutzen Sie `equals()` (**GLEICHHEIT**).*
*Wollen Sie prüfen ob sich zwei Referenzen auf dasselbe Objekt beziehen, nutzen Sie == (**IDENTITÄT**).*
*Primitive Datentypen werden **immer** mit == verglichen!*

3.2.4 Objekte duplizieren mittels `clone()`

Eine weitere Standardmethode bei Klassen ist die `clone()`-Methode. Sie erzeugt eine Wertekopie (ein Duplikat) eines Objekts. Ein Klon (eine Kopie) ist nie identisch mit dem Original aber inhaltlich gleich.

Wie bereits bei `toString()` und `equals()` kann eine `clone()`-Methode für die Klasse `Person` ganz einfach im Klassenblock ergänzt werden.

```
public Person clone() {
    return new Person(this.vorname, this.nachname, this.geburtsjahr);
}
```

3.2.5 Komponenten mittels `static` an Klassen binden (und nicht an Objekte)

Wir wollen unser Personenbeispiel nun so erweitern, dass jede `Person` eine laufende Nummer erhält. Ergänzend soll bei der Ausgabe einer `Person` auf der Konsole auch angegeben werden, die wievielte von insgesamt vorhandenen Personen diese Person ist. Folgendes Beispiel soll dies deutlicher machen:

```
Person p = new Person("Max", "Mustermann", 1976);
Person q = new Person("Maren", "Musterfrau", 1975);
Person r = new Person("Max", "Mustermann", 1976);
System.out.println(p);
System.out.println(q);
System.out.println(r);
```

Wir haben also insgesamt drei Personen angelegt, die Person `p` als erstes und die Person `r` als letztes. Die Ausgabe sollte daher lauten:

```
[1/3] Max Mustermann (Geb: 1976)
[2/3] Maren Musterfrau (Geb: 1976)
[3/3] Max Mustermann (Geb: 1976)
```

Auffällig ist, dass dem Konstruktor weder die Gesamzzahl noch die laufende Nummer der Person mitgeteilt wurde. Der Konstruktor muss diese also offenbar selber verwalten. Wir stellen ferner fest, dass

- die laufende Nummer für jedes Objekt unterschiedlich ist,
- die Gesamtanzahl aber für alle Objekte gleich ist.

Wir haben also zwei "Arten" von Datenfeldern, eines das pro Objekt gilt (laufende Nummer) und eines das pro Klasse von Objekten gilt (Gesamtanzahl). Wie wir schon gesehen haben, werden Datenfelder automatisch pro Objekt angelegt. Will man das nicht, kann man Datenfelder auch nur einmal pro Klasse anlegen lassen (man sagt auch: *"statisch an die Klasse binden"*), in dem man diese Datenfelder mit dem Schlüsselwort `static` kennzeichnet. Statische Datenfelder nennt man daher auch **Klassenvariablen**.

```
public class Person {
    public String vornamne;       // pro Objekt
    public String nachname;       // pro Objekt
    public int geburtsjahr;       // pro Objekt

    public int lfdNr;             // pro Objekt
    public static int gesamt = 0; // pro Klasse
}
```

Den Konstruktor können wir dann wie folgt erweitern, damit er mit jeder neu angelegten `Person` die Gesamtanzahl erhöht und eine neue laufende Nummer vergibt.

```
public Person(String vn, String nn, String gj) {
    this.vorname = vn;
    this.nachname = nn;
    this.geburtsjahr = gj;
    this.lfdNr = ++Person.gesamt; // Erweiterung
}
```

Auch die `toString()`-Methode muss entsprechend angepasst werden. Da die zu generierende Zeichenkette nun bereits einem komplexeren Muster folgt, greifen wir hierzu auf die `String.format()` Methode zurück, die bereits in Unit 02 eingeführt wurde.

```
public String() toString() {
    return String.format("[%d/%d] %s %s (Geb: %d)",
        this.lfdNr, Person.gesamt,
        this.vorname, this.nachname,
        this.geburtsjahr
    );
}
```

Beachten Sie folgende Konvention bei Klassenvariablen und Datenfeldern:
*- Auf **Klassenvariablen** mit `Klasse.bezeichner` zugreifen.*
*- Auf **Datenfelder** mit `this.bezeichner` zugreifen.*
Wenn die Datenfelder/Klassenvariablen eindeutig bezeichnet sind, kann auf die Angabe von `this`/`Klasse` grundsätzlich verzichtet werden.

Es wird aber aus Gründen der Nachvollziehbarkeit empfohlen dies dennoch zu tun.

3.3 Sprache eines Satzes (besser eines längeren Textes) bestimmen (nur INF)

Wir haben bislang in Sätzen bereits die Anzahl an Zeichen, Leerzeichen, und Wörtern bestimmt sowie das längste Worte, Worte einer bestimmten Länge, oder die durchschnittliche Wortlänge in einem Satz ermittelt.

Mit weiteren Programmierkonzepten können wir aber weitere erstaunliche Dinge mit unseren bislang immer noch recht begrenzten Programmiermöglichkeiten erreichen. Dieses Beispiel soll nun zeigen, wie man `Maps` zum Zählen von Vorkommen nutzen kann und dies u.a. dafür nutzen kann, die (wahrscheinliche) Sprache eines Satzes (oder besser eines längeren Textes) zu ermitteln.

Texte in verschiedenen Sprachen haben eine charakteristische Häufung ihrer Zeichen. Im Deutschen ist der häufigste Buchstabe das e (genauso wie in vielen anderen Sprachen auch). Das e macht aber etwa 17.4% aller Buchstaben in deutschen Texten aus, im Englischen sind es nur etwa 12.7%, im Schwedischen sogar nur 10%. Solche Häufigkeiten wurden für diverse Sprachen ermittelt und man kann sie als `Map` in Java ausdrücken.

Im Weiteren werden wir mit der Konstanten `LANGUAGES` die Buchstabenhäufigkeiten der Sprachen Deutsch (`"de"`) und Englisch (`"en"`) angeben. Die Daten sind aus folgender Häufigkeitstabelle entnommen (die auch weitere Sprachen umfasst).

```
import java.util.Map;
import static java.util.Map.entry;

// Achtung: Map.of und Map.ofEntries gibt erst ab Java 9!
public static final Map<String, Map<Character, Double>> LANGUAGES = Map.of(
    "de", Map.ofEntries(
        entry('a', 0.065), entry('b', 0.019), entry('c', 0.030), entry('d', 0.051),
        entry('e', 0.174), entry('f', 0.017), entry('g', 0.030), entry('h', 0.048),
        entry('i', 0.076), entry('j', 0.03),  entry('k', 0.012), entry('l', 0.034),
        entry('m', 0.025), entry('n', 0.098), entry('o', 0.025), entry('p', 0.008),
        entry('q', 0.0002), entry('r', 0.07), entry('s', 0.073), entry('t', 0.062),
        entry('u', 0.044), entry('v', 0.007), entry('w', 0.019), entry('x', 0.0003),
        entry('y', 0.0004), entry('z', 0.011)
    ),
    "en", Map.ofEntries(
        entry('a', 0.082), entry('b', 0.015), entry('c', 0.028), entry('d', 0.043),
        entry('e', 0.127), entry('f', 0.022), entry('g', 0.020), entry('h', 0.061),
        entry('i', 0.070), entry('j', 0.002), entry('k', 0.008), entry('l', 0.040),
        entry('m', 0.024), entry('n', 0.067), entry('o', 0.075), entry('p', 0.019),
        entry('q', 0.001), entry('r', 0.060), entry('s', 0.063), entry('t', 0.091),
        entry('u', 0.028), entry('v', 0.010), entry('w', 0.024), entry('x', 0.002),
        entry('y', 0.020), entry('z', 0.001)
    )
);
```

Das folgende Verfahren funktioniert um so besser, je länger die zu analysierenden Texte sind (je mehr Daten vorliegen und je weniger Abweichungen damit vom "Mittelwert" der Buchstabenhäufigkeit einer Sprache existieren). Da sich Englisch und Deutsch an einigen Stellen deutlich genug unterscheiden, funktioniert dies für beide Sprachen bereits meist bei kurzen Sätzen. Das ist der Grund warum

wir diese beiden Sprachen als Beispiel nehmen. Das Verfahren kann grundsätzlich für alle Sprachen angewendet werden, benötigt dann allerdings meist längere Texte.

Mit dem Wissen der Buchstabenhäufigkeit einer Sprache können wir das Problem nun in Teilprobleme zerlegen (**Dekomposition**) und folgendes Verfahren aufbauen (**Algorithm Design** im Sinne des **Computational Thinkings**):

1. Für einen gegebenen Text wird dessen relative Buchstabenhäufigkeit bestimmt (`countChars()`).
2. Es wird jeweils der Abstand der Buchstabenhäufigkeit des gegebenen Textes zu einer Sprache bestimmt (`distance()`).
3. Die Sprache mit der kleinsten Distanz wird als (wahrscheinliche) Sprache angegeben (`language()`).

Als erstes müssen wir somit das Problem lösen, die relative Buchstabenhäufigkeit in Texten zu zählen. Uns interessieren dabei nur die Zeichen `'a'` bis `'z'` (und keine Satz- oder sonstigen Sonderzeichen, auch keine Groß-/Kleinschreibung). Mittels der folgenden Methode `countChars()` lässt sich die relative Häufigkeit der Zeichen von `'a'` bis `'z'` in einer Zeichenkette `text` zählen und mittels einer `Map` ausdrücken.

```
public static Map<Character, Double> countChars(String text) {
    Map<Character, Double> chars = new TreeMap<>();
    int total = 0;
    for (char c : text.toLowerCase().toCharArray()) {
        if (c > 'a' && c < 'z') {
            chars.put(c, chars.getOrDefault(c, 0.0) + 1);
            total++;
        }
    }
    // Aus absoluter Häufigkeit relative Häufigkeit machen
    for (char c : chars.keySet()) chars.put(c, chars.get(c) / total);
    return chars;
}
```

Wir benötigen **als zweites** eine Möglichkeit den "Abstand" zwischen zwei Häufigkeiten (also zwei Mappings des Typs `Map<Character, Double>`) zu messen, um die Frage beantworten zu können, wie weit weg ein Satz (bzw. seine Buchstabenhäufigkeit p) von der Buchstabenhäufigkeit einer Sprache q ist. Hierzu verwendet man oft den euklidischen Abstand, den sie vielleicht noch aus der Vektorrechnung Ihrer Schulmathematik kennen.

$$d(p,q) = \sqrt{(q_{'a'} - p_{'a'})^2 + \ldots + (q_z - p_z)^2} = \sqrt{\sum_{i='a'}^{'z'} (q_i - p_i)^2} \tag{1}$$

Mittels der folgenden Methode `distance()` lässt sich der euklidische Abstand zwischen zwei Mappings `p` und `q` ausdrücken.

```
public static double distance(Map<Character, Double> p, Map<Character, Double> q) {
    Set<Character> chars = new HashSet<>(p.keySet());
    chars.addAll(q.keySet());
    double sum = 0.0;
    for (char c : chars) {
        sum += Math.pow(p.getOrDefault(c, 0.0) - q.getOrDefault(c, 0.0), 2);
    }
    return Math.sqrt(sum);
}
```

Als drittes fehlt nun nur noch die Bestimmung der Sprache mittels der Methode `language()`. Hierzu wird für eine zu untersuchende Zeichenkette `text` deren relative Buchstabenhäufigkeit mittels `countChars()` bestimmt und die Distanz zu allen bekannten Sprachen `LANGUAGES` bestimmt. Es wird die Sprache mit der kleinsten Distanz als Vorschlag gewählt.

```
public static String language(String text) {
    Map<Character, Double> vector = countChars(text);
    double min = distance(vector, new TreeMap<>());
    String language = "";
    for (String lang : LANGUAGES.keySet()) {
        double dist = distance(vector, LANGUAGES.get(lang));
        if (dist < min) {
            min = dist;
            language = lang;
        }
    }
    return language;
}
```

Das Problem lässt sich also als im Kern auf die Suche nach einem Minimum zurückführen (**Abstraktion** im Sinne des **Computational Thinkings**). Probieren Sie den Ansatz einmal mit folgenden Codezeilen aus und spielen sie mit den Daten herum. Ergänzen Sie gerne auch weitere Sprachen aus folgender Häufigkeitstabelle.

```
    String deutsch = "Ein zufälliger deutscher Satz. "
        + "Beachten Sie, dass längere Texte meist bessere Resultate "
        + "aufgrund von mehr Daten erzeugen.";
    String english = "A random English sentence. "
        + "But consider that it works much better with longer texts "
        + "that provide simply more data.";

    Map<Character, Double> d = countChars(deutsch);
    Map<Character, Double> e = countChars(english);

    System.out.println(distance(d, LANGUAGES.get("en")));
    System.out.println(distance(d, LANGUAGES.get("de")));

    System.out.println(distance(e, LANGUAGES.get("en")));
    System.out.println(distance(e, LANGUAGES.get("de")));

    System.out.println(language(deutsch));
    System.out.println(language(english));
```

Wir können so mit nur knapp 30 Zeilen Code in den Methoden `countChars()`, `distance()` und `language()` die (wahrscheinliche) Sprache einer Zeichenkette bestimmen. Eigentlich erstauntlich – insbesondere wenn wir bedenken, dass wir mit Zeichenzählen angefangen haben. *Pretty amazing result for a freshman student …*

3.4 Worte in einer Zeichenkette nach Wortlängen gruppieren

Im Sprachbestimmungsbeispiel haben wir gesehen, dass wir `Maps` dazu nutzen können, um Vorkommen von Entities zu zählen. Das kennen Sie ggf. aus dem Modul *Datenbanken* und SQL-Aggregationsfunktionen wie `COUNT()`.

Maps können aber auch genutzt werden, um die Entities nach Eigenschaften zu gruppieren. Das Prinzip könnten Sie in *Datenbanken* als GROUP-BY-SQL-Statement bereits kennen gelernt haben.

Wie (fast) immer lesen wir erst einmal einen Satz ein:

```
Scanner in = new Scanner(System.in);
System.out.println("Bitte geben Sie einen Satz ein: ");
String sentence = in.nextLine();
```

In solchen Sätzen wollen wir nun mit einer Methode `groupBy()` die Wörter des Satzes aufsteigend nach ihrer Länge sortiert ausgeben.

```
Map<Integer, List<String>> byLength = groupBy(sentence);
for (int l : byLength.keySet()) {
    System.out.printf("Wortlänge %d: %s \n", l, byLength.get(l));
}
```

Das sollte für die Eingabe des Satzes *"Dies ist nur ein dummes Beispiel"* dann die folgende Ausgabe erzeugen:

```
Wortlänge 3: [ist, nur, ein]
Wortlänge 4: [Dies]
Wortlänge 6: [dummes]
Wortlänge 8: [Beispiel]
```

Das Gruppieren ist sogar meist einfacher (oder kürzer) als das Zählen oder aggregieren.

- Hierzu wird eine `Map` des Typs der zu gruppierenden Eigenschaften (Key) auf eine Liste der zu gruppierenden Entities (Values) angelegt. Man kombiniert also einfach die Collections `List` und `Map`.
- `Listen` werden genutzt, um (ggf. mehrere) Entities (zu einer Eigenschaft) zu speichern. `Maps` werden genutzt um die Eigenschaften auf die Entities abzubilden (abbilden, engl. *to map*).
- Die zu gruppierenden Entities werden sequenziell durchlaufen. Für jedes Entity wird die Eigenschaft bestimmt, nach der gruppiert werden soll.
- In der zu erzeugenden `Map` wird erst für jede Eigenschaft eine neue leere Liste angelegt.
- Anschließend wird jedes Entity gemäß seiner Eigenschaft an die der Eigenschaft zugeordneten Liste gehängt.

Dieser Vorgang lässt sich mit der `get()`-Methode von `Maps` und der `add()`-Methode von `Listen` erstaunlich kompakt ausdrücken (vgl. Sie die Implementierung einmal mit der `countChars()`-Methode aus dem Sprachbestimmungsbeispiel):

```
import java.util.List;
import java.util.Map;
import java.util.LinkedList;
import java.util.TreeMap;

public static Map<Integer, List<String>> groupBy(String s) {
    String[] words = s.trim().split("\\s+");
    Map<Integer, List<String>> grouped = new TreeMap<>();
    for (String word : words) grouped.put(word.length(), new LinkedList<>());
    for (String word : words) grouped.get(word.length()).add(word);
    return grouped;
}
```

Hinweise zur Wahl der Datentypen: Wir nutzen die `TreeMap` als Implementierung für `Map`, da wir die Eigenschaften aufsteigend sortiert ausgeben wollen und die `TreeMap` das Ordnungskriterium der Schlüs-

sel erhält (anders als bspw. eine `HashMap`). Wir nutzen als Implementierung für `List` eine `LinkedList`, da diese schnellere An-/Einfügeoperationen erlaubt und wir beim sequenziellen Durchlauf des Gruppierungsvorgangs viele Anfügeoperationen erzeugen.

In Klausuren werden Sie immer begründen müssen,
warum Sie welche Collection-Implementierung nutzen.
Lassen Sie diese als Geschenk gemeinten Punkte nicht liegen!!!

3.5 Äpfel selektieren und gruppieren (nur ITD)

Dieses Beispiel im Anschluss des List-Collection Teils der Vorlesung soll zeigen, wie Listen erzeugt und genutzt werden können. Ferner werden selbst definierte Klassen wiederholt.

Für dieses Beispiel sei daher die folgende Klasse `Apple` gegeben. Ein Apfel hat eine zufällige Farbe `"red"`, `"green"`, `"yellow"` und ein Gewicht zwischen 200g und 400g.

```
public class Apple {

    public static final String[] COLORS = { "red", "green", "yellow" };

    public String color;
    public double weight;

    public Apple() {
        this.weight = Math.random() * 200 + 200;
        this.color = COLORS[(int)(Math.random() * COLORS.length)];
    }

    public String toString() {
        return String.format("%s apple (%.2fg)", this.color, this.weight);
    }
}
```

Die Aufgabe ist nun eine zufällige Anzahl an Äpfeln zu erzeugen und aus diesen Äpfeln einige Äpfel anhand von Farbe und Gewicht zu selektieren und auszugeben.

```
List<Apple> apples = createApples(20);
for (Apple apple : select(apples, "red", 250, 350)) {
    System.out.println("- " + apple);
}
```

Dies sollte bspw. folgende Ausgabe auf der Konsole ergeben:

```
- red apple (262,06 g)
- red apple (344,78 g)
- red apple (292,98 g)
```

3.5.1 Selektieren in Listen

Als erstes bauen wir eine Methode `createApples()`, die `n` zufällige Äpfel erzeugt und diese als `Liste` zurückgibt.

```
public static List<Apple> createApples(int n) {
    List<Apple> apples = new LinkedList<>();
    while(n-- > 0) apples.add(new Apple());
    return apples;
}
```

Auf solchen Listen von Äpfeln wollen wir nun Äpfel nach drei Kriterien

- Farbe
- minimales Gewicht
- maximales Gewicht

mittels einer Methode `select()` selektieren. Selektierte Äpfel sollen wieder als Liste zurückgegeben werden, die ursprüngliche Liste der Äpfel soll **NICHT** verändert werden.

```
public static List<Apple> select(List<Apple> as, String col, double min, double max) {
    List<Apple> selected = new LinkedList<>();
    for (Apple apple : as) {
        // Wir "skippen" Eintraege die nicht unseren Kriterien entsprechen
        if (!apple.color.equals(col)) continue;
        if (apple.weight < min || apple.weight > max) continue;
        selected.add(apple);
    }
    return selected;
}
```

Die Methode `select()` entspricht einem häufig vorkommenden Filter-Pattern.
*Filtern bedeutet eine Menge von Werten auf Kriterien zu prüfen und diese bei Erfüllung in eine (reduzierte) Zielmenge für eine weitere Verarbeitung zu übernehmen. Dieses Pattern taucht sehr häufig in der Programmierung auf und Sie sollten es im Sinne des **Computational Thinkings** verinnerlichen.*

3.5.2 Gruppieren mittels Maps

Im Weiteren sollen nun die Gewichte von Äpfeln nach Farbe gruppiert werden. Z.B. um rauszufinden ob "rote" Äpfel schwerer sind als "grüne" (nicht Gegenstand des Beispiels).

Folgende Codezeilen

```
List<Apple> apples = createApples(9);
Map<String, List<Double>> grouped = groupByColor(apples);
for (String color : grouped.keySet()) {
    System.out.println(color + ": " + grouped.get(color));
}
```

sollen bspw. die folgende Konsolenausgabe erzeugen:

```
red: [286.2031, 251.7000, 343.6424, 249.9332]
green: [271.6452, 220.0805]
yellow: [287.0421, 248.2274, 269.5802]
```

Wie wir sehen, können wir `Maps` nutzen, um zu kategorisieren. Unsere Schlüssel/Wert-Paare können wir dabei nutzen, um die Gewichte unserer Äpfel nach Farben (Kategorie) zu gruppieren. Abstrakt gesprochen: Wir mappen Zeichenketten (Farben) auf Liste von numerischen Werten (Gewichte).

```java
public static Map<String, List<Double>> groupByColor(List<Apple> apples) {
    Map<String, List<Double>> grouped = new HashMap<>();
    for (Apple apple : apples) {
        grouped.put(apple.color, new LinkedList<Double>());
    }
    for (Apple apple : apples) {
        grouped.get(apple.color).add(apple.weight);
    }
    return grouped;
}
```

Die Methode `groupByColor()` entspricht einem häufig vorkommenden Gruppierungs-Pattern.
Gruppieren bedeutet eine Menge von Werten anhand eines Kriteriums zu gliedern.
Nach einer Gruppierung können Werte mit bestimmten Eigenschaften sehr schnell selektiert werden.
Dieses Pattern taucht sehr häufig in der Programmierung auf und Sie sollten es
*im Sinne des **Computational Thinkings** verinnerlichen.*

3.6 Eingeführte Konzepte und Methoden der Unit 03

- Mehrdimensionale Arrays (`[] []`).
- `new`-Operator um Arrays (und weitere Objekte von Referenzdatentypen) anzulegen.
- Schlüsselwort `class` zur Definition eigener Datentypen (Klassen, noch im Sinne strukturierter Datentypen).
- Die Methoden `equals()` und `clone()` um Objekte miteinander inhaltlich zu vergleichen bzw. zu kopieren.
- Den Unterschied zwischen den == Operator **(prüft auf Identität zweier Objekte)** und der `equals()`-Methode **(prüft auf inhaltliche Gleichheit zweier Objekte)**.
- Die Bedeutung des Schlüsselwortes `static` bei Datenfeldern (Klassenvariablen).
- Die Begrifflichkeiten **Datenfeld** (pro Objekt) und **Klassenvariable** (pro Klasse).
- Den Modifier `public static final` zur Definition unveränderlicher **Konstanten**.
- Das Interface `Map` zur Referenzierung von Key-Value Paaren in Form von Mappings.
- Die `Map`-Implementierung `TreeMap` zur Speicherung von Key-Value-Paaren bei dem die Schlüsselordnung erhalten bleibt.
- Die `Map`-Implementierung `HashMap` zur Speicherung von Key-Value-Paaren bei dem die Schlüsselordnung **nicht** erhalten bleibt.
- `Map.of` und `Map.ofEntries` zur Literal-artigen Deklarationen von Mappings in Java (ab Version 9).
- `keySet()` zur Bestimmung aller Schlüssel eines Mappings.
- `get()` und `getOrDefault()` um den Wertes eines Schlüssels in einem Mapping zu bestimmen (bzw. Nutzung eines Default-Wertes falls der Schlüssel nicht existent ist).
- Das Interface `Set` zur Referenzierung von Mengen (Listen ohne doppelte Werte).
- Die Mengen-Implementierung `HashSet` zur Speicherung von unsortierten Mengen.
- Das Interface `List` zur Speicherung von Werten in Listen.
- Die `List`-Implementierung `LinkedList` für Listen, die schnelle Anfügeoperationen ermöglichen.

4 Unit 04 (File I/O, nur INF)

4.1 Faust vs. Mephisto

Vermutlich haben Sie alle einmal Goethes Faust in der Schule gelesen. Haben Sie sich je gefragt wer von den beiden Charakteren *Faust* und *Mephisto* eigentlich mehr sagt?

Das wollen wir in diesem Beispiel ermitteln und gleichzeitig sehen, wie man trickreich die File I/O für solche Aufgaben einsetzen kann. Unter anderem der `Scanner`, den wir bislang immer nur für Texteingaben von der Konsole genutzt haben, kann viel mehr als sie vielleicht denken.

Als erstes benötigen wir für unsere Analyse eine gut zu verarbeitende Datei mit den Inhalten.

Eine Textdatei von Faust finden Sie beim Projekt Gutenberg unter diesem Link.

`https://www.gutenberg.org/cache/epub/2229/pg2229.txt`

Da das Projekt Gutenberg leider gerade einen Urheberrechtsstrei führt, sind deutsche IP-Adressen aktuell geblockt. Laden Sie die Datei daher ggf. mit dem TOR-Browser, um ihre deutsche IP-Adresse zu verschleiern. Der Rechtstreit geht um Werke (u.a. von Thomas Mann, nicht ganz unbekannt in Lübeck ;-) die nach dt. Urheberrecht 70 Jahre, nach US-Urheberrecht allerdings nur 25 Jahre geschützt sind. Goethes Faust ist damit in keinem Fall von Urheberrechtsproblemen betroffen. Also keine Angst beim Download!

Diese Datei ist in etwa wie folgt aufgebaut:

```
[...]

DER HERR:
Du darfst auch da nur frei erscheinen;
Ich habe deinesgleichen nie gehaßt.
Von allen Geistern, die verneinen,
ist mir der Schalk am wenigsten zur Last.
[...]
Das Werdende, das ewig wirkt und lebt,
Umfass euch mit der Liebe holden Schranken,
Und was in schwankender Erscheinung schwebt,
Befestigt mit dauernden Gedanken!
(Der Himmel schließt, die Erzengel verteilen sich.)

MEPHISTOPHELES (allein):
Von Zeit zu Zeit seh ich den Alten gern,
Und hüte mich, mit ihm zu brechen.
Es ist gar hübsch von einem großen Herrn,
So menschlich mit dem Teufel selbst zu sprechen.

[...]
```

Die Figur des Dr. Faust wird als "FAUST" gekennzeichnet, die des Mephisto als "MEPHISTOPHELES".

Die **Idee** ist jetzt, die Datei zeilenweise einzulesen, alle Zeilen zu selektieren, die Charakterangaben haben und in diesen die Figur zu selektieren. Alle so bestimmten Vorkommen zählen wir mit einer `Map`, um so gleich eine Analyse aller Figuren aus Goethes Faust zu bestimmen.

Der Ihnen für Konsoleneingaben bekannte `Scanner` ist dabei sehr hilfreich. Da das Java I/O Stream System sehr flexibel ist, können wir einen `Scanner` nicht nur für `System.in` sondern für beliebige

`InputStreams` nutzen, wie z.B. einen `FileInputStream`. Ein `Scanner` bietet ferner eine Methode `findInLine()` mit der nach beliebigen regulären Ausdrücken in Zeilen gesucht werden kann.

- Eine Figurenangabe ist immer "GROß GESCHRIEBEN" und endet mit einem Doppelpunkt ":".
- Eine Figurenbezeichner kann aus einem (z.B. "FAUST") oder mehreren Worten (z.B. "DER HERR") bestehen.
- Eine Figurenangabe kann in Klammern ergänzende *'Regieanweisungen'* vor dem Doppelpunkt beinhalten, z.B. "(allein)". Regieanweisungen können mehrere Worte beinhalten, z.B. "ALTE (zu den Bürgermädchen):"

`EIN BEZEICHNER (ggf. weitere Angaben):`

Ein regulärer Ausdruck kann dieses Muster ausdrücken:

`"[A-ZÖÄÜ ]+ +([(].*[)])*:"`

Damit können wir das Problem in nicht einmal zwanzig Zeilen lösen

```
Scanner in = new Scanner(new FileInputStream("/path/to/your/faust.txt"));
Map<String, Integer> count = new TreeMap<>();
while (in.hasNextLine()) {
    String hit = in.findInLine("[A-ZÖÄÜ ]+ *([(].*[)])*:");
    in.nextLine();
    if (hit == null) continue;
    hit = hit.trim();

    // Ein paar Fehltreffer aus dem Gutenberg-Vorspann und Abspann
    // müssen wir skippen.
    if (hit.startsWith("START")) continue;
    if (hit.startsWith("END")) continue;
    if (hit.startsWith("PG")) continue;

    // Wir trennen Bezeichner von Regieanweisungen
    String name = hit.split("[(:]")[0].trim();

    // Und nutzen eine Map zum Zählen aller Rollenbezeichner
    count.put(name, count.getOrDefault(name, 0) + 1);
}
System.out.println("Faust: " + count.get("FAUST"));
System.out.println("Mephisto: " + count.get("MEPHISTOPHELES"));
```

und erhalten folgende Ausgabe:

```
Faust: 220
Mephisto: 250
```

Mephisto ist also der Geschwätzigere von den Beiden ;-) Diese Erkenntnis hat Ihnen im Deutschunterricht aber vermutlich nie jemand vorgerechnet.

Die `Scanner`-Klasse kann mehr als man vermutet.
Und Java I/O-Streams sind flexibler als man denkt.

4.2 Eingeführte Konzepte und Methoden der Unit 04

- **`Scanner`**-Klasse zum Verarbeiten beliebiger **`InputStreams`**.

- Komplexere **reguläre Ausdrücke** zum Selektieren von Daten in Texten mittels `Scanner.findInLine()`.

5 Unit 05 (Rekursive Programmierung und Lambdafunktionen)

5.1 Zeichenketten wiederholen

Das erste Beispiel soll einfache Rekursionen veranschaulichen. Einfache Rekursionen nennen wir in diesem Modul solche Rekursionen, in denen ein Parameter meist "einfach" nach oben oder unten gezählt wird.

Exemplarisch wollen wir eine in Java irgendwie fehlende Methode rekursiv implementieren, die eine Zeichenkette *n*-mal wiederholt (und diese als `Liste` zurückgibt).

Folgendes Beispiel, sollte die Wirkungsweise deutlich machen.

```
List<String> repeated = repeat(4, "Hello");
System.out.println(repeated);           // ["Hello", "Hello", "Hello", "Hello"]
System.out.println(repeat(0, "Hello")); // []
System.out.println(repeat(3, ""));      // ["", "", ""]
```

Mit Schleifen (also imperativ) würden Sie das vermutlich wie folgt lösen:

```
public static List<String> repeat(int n, String s) {
    List<String> ret = new LinkedList<>();
    for (int i = 0; i < n; n++) {
        ret.add(s);
    }
    return ret;
}
```

Eine rekursive Lösung könnte wie folgt aussehen. Entwickelt nach dem bewährten Prinzip:

1. Methodenkopf definieren
2. Triviale Fälle beantworten (Rekursionsabbruch)
3. Rückführung des Falls *n* auf den kleineren Fall *n - 1*

```
public static List<String> repeat(int n, String s) { // (1)
    if (n < 1) return new LinkedList<>();  // (2) Rekursionsabbruch
    List<String> ret = repeat(n - 1, s);   // (3) Rekursionsaufruf
    ret.add(s);                            // (3) n mit n - 1 verknüpfen
    return ret;
}
```

> **Rekursionen kann man also dazu nutzen, um zu wiederholen (Schleifen)**
> *wie dieses Beispiel eindrucksvoll zeigt…*

5.2 Zeichenketten `joinen`

In Java fehlt ferner eine weitere Methode, die mehrere Zeichenketten mit einem Trennzeichen verknüpft. So eine Funktion `join()` benötigen wir erstauntlich häufig. Wir wollen diese daher rekursiv implementieren.

Folgendes Beispiel, sollte die Wirkungsweise deutlich machen.

```
List<String> words = Arrays.asList("Dies", "ist", "nur", "ein", "Beispiel");
String joined = join(" ", words);
System.out.println(joined);  // => "Dies ist nur ein Beispiel"
System.out.println(join("-", words)); // "Dies-ist-nur-ein-Beispiel"
System.out.println(join("...", words)); // "Dies...ist...nur...ein...Beispiel"
```

Eine rekursive Lösung könnte wie folgt aussehen. Entwickelt nach dem bewährten Prinzip:

1. Methodenkopf definieren
2. Triviale Fälle beantworten (Rekursionsabbruch)
3. Rückführung des Falls *n* auf den kleineren Fall *n - 1*
    1. Dabei **Kopf** der Sequenz bestimmen
    2. **Rest** der Sequenz
    3. Kopf und Rest geeignet verknüpfen **(Rekursionsaufruf)**

```
public static String join(String s, List<String> seq) { // (1)
    if (seq.isEmpty()) return "";                        // (2)
    if (seq.size() == 1) return seq.get(0);              //
    String head = seq.get(0);                            // (3.1) Kopf
    List<String> rest = seq.subList(1, seq.size());      // (3.2) Rest
    return head + s + join(s, rest);                     // (3.3) Rekursionsaufruf
}
```

Wir wandeln nun unser Problem leicht ab. Es sollen nun keine Listen von Strings verknüpft werden, sondern eine variable Anzahl an Parametern. Wir wollen `join()` also auch wie folgt aufrufen können.

```
System.out.println(join("+", "Beispiel"));
// => "Beispiel"

System.out.println(join("+", "Nur", "ein", "Beispiel"));
// => "Nur+ein+Beispiel"

System.out.println(join("+", "Dies", "ist", "nur", "ein", "Beispiel"));
// => "Dies+ist+nur+ein+Beispiel"
```

Wie sie aus Unit 02 (Variable Parameteranzahl und Methoden überladen) wissen, können wir hierzu die `join()` Methode wie folgt überladen.

```
public static String join(String s, String... seq) {
    // TO BE DONE
}
```

Das Problem ist nun, dass der variable Parameter `seq` auf ein Array abgebildet wird. Anders als Listen, lassen sich Arrays aber nicht "kürzer" machen (es gibt keine `subList()`-Methode für Arrays). Rekursionen beruhen aber auf dem Prinzip Probleme mit jedem Rekursionsaufruf kleiner zu machen (bis man bei trivialen Fällen – Rekursionsabbrüchen – gelandet ist). Muss man also Sequenzen durchlaufen, die nicht kleiner gemacht werden können, kann man sich folgenden Tricks bedienen und eine sequenzbasierte Rekursion auf eine einfache Rekursion zurückführen.

Man führt einfach eine überladene Methode mit einem Indexparameter `i` mehr ein. Dieser Indexparameter dient zum Zählen (vglb. einer zählenden `for`-Schleife) einer Index-Position in der Sequenz. Diese überladene Methode kann wie eine einfache Rekursion formuliert weren.

Das führt dann zu folgender rekursiven Lösung:

```
public static String join(String s, String... seq) {
    return join(0, s, seq); // Ab 0 die einfache Indexrekursion starten
```

```
}

// Überladene Methode mit zusätzlichem Indexparameter
// Sequenzbasierte Rekursion wird so auf einfache Rekursion zurückgeführt
public static String join(int i, String s, String... seq) {
    if (seq.length == 0) return "";           // Abbruch
    if (seq.length - 1 == i) return seq[i];   // Abbruch
    return seq[i] + s + join(i + 1, s, seq);  // Rekursionsaufruf
}
```

Rekursionen müssen Probleme mit jedem Aufruf kleiner machen.
Für Sequenzen bedeutet dies, diese müssen kürzer werden. Geht dies Typ-bedingt nicht, z.B. bei Arrays, kann man einen Indexparameter einführen und diesen wie bei einer einfachen Rekursion von 0 bis zur Länge der zu verarbeitenden Sequenz durchlaufen lassen.

5.3 Rekursiv auf vollständige Klammerung prüfen (`checkBrackets()`, nur INF)

Dieses Einstiegsbeispiel zur Wiederholung von Rekursionen greift die imperative Version der `checkBrackets()`-Aufgabe des Trimm-Dich-Pfads auf. Wir erinnern uns:

*"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."*

Im Trimm-Dich-Pfad haben Sie dies vermutlich mit einer Schleife gelöst. Wir wollen diesselbe Aufgabe nun aber **rekursiv** (d.h. ohne Schleifen) lösen.

Der Einfachheit halber nehmen wir uns die folgenden in der Trimm-Dich-Pfad Aufgabe gegebenen Testfälle, um unsere rekursive Version zu testen.

```
List<String> testfaelle = Arrays.asList(
    "()",               // => true
    "(()(a)(()((c))))", // => true
    "a (()())a)",       // => false
    "(()",              // => false
    ")("                // => false
);

for (String bs : testfaelle) {
    System.out.println(bs + ": " + checkBrackets(bs));
}
```

Was uns auffällt ist, dass es sich hierbei um eine Sequenz-basierte Rekursion handelt (Zeichenketten). Andererseits reicht eine Methode mit folgender Signatur

```
boolean checkBrackets(String bs)
```

nicht aus, da wir uns nirgendwo die "Ebene" (`level`) der Klammerung merken können *(es sei denn, wir speichern uns die Klammerebene außerhalb der Methode global in einem Datenfeld, was aber nicht schön ist)*.

In solchen Fällen können wir immer zusätzliche Parameter mittels überladenen Methoden einführen, um Zwischenzustände von Aufruf zu Aufruf weiterzugeben. Als erstes führen wir unsere

`checkBrackets()` Methode also auf eine überladene Version zurück, die die aktuelle Klammerungsebene (des bereits analysierten vorderen Teils der Zeichenkette) mitzählt. Logischerweise beginnen wir bei der Ebene 0 zu zählen (da uns beim Start der Rekursion bislang noch keine Klammer *"über den Weg gelaufen ist"*).

```
public static boolean checkBrackets(String bs) {
    return checkBrackets(0, bs);
}
```

Folgenden Trick wenden wir an: Wann immer wir eine öffnende Klammer "(" finden, inkrementieren wir den `level` Parameter. Bei schließenden Klammern ")" dekrementieren wir logischerweise den `level` Parameter. Haben wir die Zeichenkette rekursiv durchgearbeitet, muss am Ende der `level` Parameter im Falle einer korrekten Klammerung auf 0 stehen (andernfalls gab es mehr öffnende als schließende Klammern). Wird der `level`-Parameter während der Verarbeitung einmal negativ, gab es mehr schließende als öffende Klammern bis zu dieser Stelle der Zeichenkette, d.h. die Klammerung kann nicht korrekt sein.

Die eigentliche Rekursion entwickeln wir dann nach dem bewährten Muster einer Sequenz-basierten Rekursion:

- Methodenkopf definieren (1)
- Abbruchfälle definieren, bzw. nur die trivialen Antworten geben (2)
- Kopf (3.1) und Rest (3.2) des Problems bestimmen und die Rekursion formulieren (3.3)

Es ergibt sich damit recht schematisch die folgende rekursive Implementierung.

```
public static boolean checkBrackets(int level, String bs) {  // (1)
    if (bs.isEmpty()) return level == 0;                      // (2)
    char head = bs.charAt(0);                                 // (3.1)
    String rest = bs.substring(1);                            // (3.2)
    if (head == '(') level++;                                 // Level zählen
    if (head == ')') level--;
    return (level >= 0) && checkBrackets(level, rest);        // (3.3)
}
```

Zustände kann man zwischen Rekursionsaufrufen durch zusätzliche Parameter weiterreichen.
Hierzu können Sie einfach eine Methode überladen und einen zusätzlichen Parameter definieren. Diese überladene Methode wird dann von der ursprünglichen Methode mit dem initialen Startzustand aufgerufen.

5.4 Rekursive Datenstrukturen am Beispiel von BinSort

Dieses Beispiel soll die Wirkungsweise von rekursiven Datenstrukturen am Beispiel von BinSort veranschaulichen. Es sei hierfür die folgende rekursiv definierte Datenstruktur `Node` gegeben, die zum Aufbau eines Binärbaums genutzt werden kann. Die Methode `insert()` der Klasse `Node` fügt einen Wert in einen sortierten Binärbaum ein.

```
public class Node {
    public Node left;
    public Node right;
    public int value;

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

```
    public Node insert(int v) {
        if (v < this.value) {
            if (this.left != null) left.insert(v);
            else this.left = new Node(v);
        }
        if (v >= this.value) {
            if (this.right != null) right.insert(v);
            else this.right = new Node(v);
        }
        return this;
    }
}
```

Es sollen nun zwei Methoden `buildTree()` und `serialize()` **rekursiv** entwickelt werden, die in `sort()` genutzt werden.

```
    public static List<Integer> sort(List<Integer> values) {
        Node tree = buildTree(values);
        return serialize(tree);
    }
```

Fangen wir mit `buildTree()` an. Diese Methode generiert aus einer Liste von Integer Werten einen sortierten Binärbaum vom Typ `Node`. Ähnlich wie in der `checkBrackets()` Methode benötigen wir einen zusätzlichen Parameter der sich den Zustand unseres gebildeten Baums merken kann.

```
    public static Node buildTree(List<Integer> values) {
        if (values.isEmpty()) return null;
        Node tree = new Node(values.get(0));
        return buildTree(tree, values.subList(1, values.size()));
    }
```

Diese überladene Methode mit zusätzlichen Parameter kann dann gem. dem Pattern einer Sequenz-basierten Rekursion wie folgt formuliert werden:

```
    public static Node buildTree(Node tree, List<Integer> values) {
        if (values.isEmpty()) return tree;
        int head = values.get(0);
        List<Integer> rest = values.subList(1, values.size());
        tree.insert(head);
        return buildTree(tree, rest);
    }
```

So ein Baum kann dann inorder (LKR) durchlaufen werden, um aus dem Baum eine Liste zu serialisieren, die dann automatisch aufsteigend sortiert ist.

```
    public static List<Integer> serialize(Node tree) {
        if (tree == null) return new LinkedList<>();
        List<Integer> serialized = serialize(tree.left); // L
        serialized.add(tree.value);                      // K
        serialized.addAll(serialize(tree.right));        // R
        return serialized;
    }
```

Rufen wir `sort()` nun mit Testdaten auf, sehen wir, dass sortieren über den "Umweg" eines Binärbaums tatsächlich funktioniert.

```
List<Integer> data = Arrays.asList(10, 0, 20, 15, 15, 35, 25, 75, 55);
System.out.println(sort(data));
// => [0, 10, 15, 15, 20, 25, 35, 55, 75]
```

Die folgende **rekursive** Methode `prettyPrint()` serialisiert die Struktur des Binärbaums als Zeichenkette. Damit können wir das Zwischenprodukt des BinSort-Algorithmus (den Binärbaum) besser in Augenschein nehmen. `prettyPrint()` liegt ein preorder (KLR) Durchlauf zu Grunde. Die Tiefeneinrückung erfolgt über einen gesonderten `indent` Parameter.

```
public static String prettyPrint(Node tree) {
    return prettyPrint("", tree);
}

public static String prettyPrint(String indent, Node tree) {
    if (tree == null) return "";
    return indent + "|_" + tree.value + "\n" +       // K
           prettyPrint(indent + "  ", tree.left) +   // L
           prettyPrint(indent + "  ", tree.right);   // R
}
```

Folgender Aufruf

```
List<Integer> data = Arrays.asList(10, 0, 20, 15, 15, 35, 25, 75, 55);
System.out.println(prettyPrint(buildTree(data)));
```

erzeugt dann folgende Konsolenausgabe:

```
|_10
  |_0
  |_20
    |_15
      |_15
    |_35
      |_25
      |_75
        |_55
```

5.5 Wiederholungen vermeiden mit Lambdas am Beispiel von `filter()` (nur INF)

Dieses Beispiel zu **Beginn der Lambda Unit** soll Lambda-Funktionen motivieren. Lambda Funktionen sind eine Möglichkeit um Wiederholungen (Copy-Paste-Programming) zu vermeiden.

Für dieses Beispiel sind die beiden folgenden Methoden `readText()` und `words()` gegeben.

```
public static String readText(String f) throws IOException {
    File file = new File(f);
    return new String(Files.readAllBytes(file.toPath()));
}

public static List<String> words(String txt) {
    return Arrays.asList(txt.split("[\\p{Punct}\\s]+"));
}
```

`readText()` ließt eine Datei als Zeichenkette ein. `words()` trennt Wörter in einer Zeichenkette an Satzzeichen wie `".,;!?..."` und Whitespaces wie `" \n\t\r..."`.

Wenn wir das Beispiel des Faust Textes aus Unit 04 nehmen, können wir mit folgenden Zeilen also alle Worte in Goethes Faust bestimmen.

```
String text = readText("/Path/to/your/faust.txt");
List<String> words = words(text);
```

Die erste Aufgabe soll es nun sein eine Methode `filterByTerm()` zu entwickeln, die alle Worte selektiert, die einem Begriff (case-insensitive) entsprechen. Das kann man dazu nutzen, herauszufinden wie häufig die Zeichenketten `"Faust"` oder `"Mephistopheles"` in Goethes Faust auftauchen *(also nicht nur in Rollenangaben, sondern im gesamten Text)*.

```
System.out.println(filterByTerm(words, "Faust").size());          // => 272
System.out.println(filterByTerm(words, "Mephistopheles").size()); // => 283
```

Die Methode lässt sich recht einfach wie folgt implementieren.

```
public static List<String> filterByTerm(List<String> ws, String needle) {
    List<String> filtered = new LinkedList<>();
    for (String w : ws) {
        if (w.equalsIgnoreCase(needle)) filtered.add(w);
    }
    return filtered;
}
```

Wenn wir die Aufgabenstellung nur leicht abwandeln, wie z.B. alle Worte mit zwei oder drei Buchstaben zu selektieren *(z.B. um rauszufinden welche kurzen Worte in deutschen Texten üblicherweise auftreten)*, sind wir allerdings gezwungen eine weitere Methode `filterByLength()` zu entwickeln. Diese sieht recht ähnlich aus:

```
public static List<String> filterByLength(List<String> ws, int len) {
    List<String> filtered = new LinkedList<>();
    for (String w : ws) {
        if (w.length() == len) filtered.add(w);
    }
    return filtered;
}
```

Eigentlich unterscheiden sich `filterByTerm()` und `filterByLength()` nur in den beiden folgenden logischen Ausdrücken:

```
filterByTerm: w.equalsIgnoreCase(needle)
filterByLength: w.length() == len
```

Der Rest der Methodenimplementierungen ist vollkommen gleich. Das würde sich auch nicht ändern, wenn wir nach anderen Kriterien selektieren wollten, wie bspw.

- Worte innerhalb einer minimalen und einer maximalen Länge,
- Worte, die mit großen Buchstaben anfangen,
- Worte, die GROSS geschrieben sind,
- ...

Wir stellen also fest, dass es bei dieser Art von "Filterproblemen" einen unveränderlichen Teil der Lösung gibt, in den eigentlich immer nur eine problemspezifische Bedingung eingebaut wird. Es wäre also gut, wenn wir die oben angegebenen logischen Bedingungen außerhalb einer `filter()` Methode definieren könnten. Auf diese lässt sich die Implementierung einer allgemeinen Filterlogik von der konkreten und problemspezifischen Filterbedingung trennen und wiederverwenden.

Seit Java 8 kann man das mit sogenannten Lambda-Funktionen auch machen. Anstelle eines konreten Filterwertes können wir nun eine Filterbedingung (oder ganz allgemeine sonstige Logik) in Form einer sogenannten Lambda-Funktion definieren und einer Methode als Parameter übergeben, den diese Methode in ihrem inneren ausführen kann.

In Mathe bzw. Informatik I haben Sie die oben genannten logischen Bedingungen vermutlich bereits als Prädikate der Prädikatenlogik kennengelernt. Folgerichtig können in Java solche logischen Bedingungen mittels des Datentyps `Predicate` ausgedrückt werden.

Ein `Predicate` kann einen Wert mittels einer Methode `test()` auf Erfüllung einer logischen Bedingung prüfen. Unsere Filtermethode sieht dann allgemeiner mit einem `Predicate` definiert wie folgt aus:

```
import java.util.function.*;

public static List<String> filter(List<String> ws, Predicate<String> p) {
    List<String> filtered = new LinkedList<>();
    for (String w : ws) {
        if (p.test(w)) filtered.add(w); // <= All the magic is here
    }
    return filtered;
}
```

Unsere Vorkommen von Faust und Mephisto lassen sich dann wie folgt bestimmen:

```
System.out.println(filter(words, w -> w.equalsIgnoreCase("Faust")).size());
System.out.println(filter(words, w -> w.equalsIgnoreCase("Mephistopheles")).size());
```

Dasselbe funktioniert natürlich auch für Worte mit einer bestimmten Länge:

```
System.out.println(filter(words, w -> w.length() == 2));
```

Die Notationen

```
w -> w.equalsIgnoreCase("Faust")
w -> w.length() == 2
```

bezeichnen dabei Lambda-Funktionen, die die zu prüfenden logischen Bedingungen ausdrücken.

Vorteil des Ganzen ist es, dass wir nun auch recht schnell in der Lage sind eher ungewöhnlich Dinge zu machen, wie bspw. alle Palindrome *(also Wörter die vorwärts wie rückwärts gelesen gleich sind)* in Goethes Faust zu bestimmen.

Das ginge dann bspw. so:

```
System.out.println(
    filter(words,
        w -> w.length() > 1 &&
             w.equalsIgnoreCase(new StringBuffer(w).reverse().toString())
    )
);
```

5.6 Mit Lambdas Faust vs. Mephisto lösen (nur INF)

Erinneren Sie sich daran, dass wir bestimmt haben, wer mehr Redeanteile in Goethes Faust hat – Faust oder Mephisto? Schauen Sie sich noch einmal die Lösung in Unit 04 an.

Ein `BufferedReader` hat eine `lines()` Methode, die eine Textdatei in einem Stream von Zeichenketten wandelt. Mit dieser Methode können wir das Faust vs. Mephisto Problem nach dem bewährten `filter()`, `map()`, `collect()` Pattern auch wie folgt formulieren (reguläre Ausdrücke sind 1:1 aus dem Beispiel der Unit 04 übernommen).

```
BufferedReader in = new BufferedReader(
    new FileReader("/Path/to/your/faust.txt")
);
Map<String, Long> roles = in.lines()
    .filter(s -> s.matches("[A-ZÖÜÄ ]+ *([(].*[)])*:"))
    .map(s -> s.split("[(:]")[0].trim())
    .collect(Collectors.groupingBy(s -> s, Collectors.counting()));
System.out.println("Faust: " + roles.get("FAUST"));
System.out.println("Mephisto: " + roles.get("MEPHISTOPHELES"));
```

Durch den Einsatz von Streams wird die Lösung damit noch kompakter und erzeugt folgende Ausgabe:

```
Faust: 220
Mephisto: 250
```

5.7 Die häufigsten Palindrome am Beispiel der Buddenbrooks bestimmen (nur INF)

In diesem Beispiel nehmen wir uns nun einfach ein anderes Werk: *Die Buddenbrooks* von Thomas Mann, in Lübeck kein ganz unbekanntes Werk ;-)

`https://www.gutenberg.org/ebooks/34811`

Wir bedienen uns also wieder einer Textdatei des Projekts Gutenberg und wissen bereits, dass diesen Dateien meist ein Projekt-Gutenberg-spezifischer Kopf voran und Fuß hintan gestellt ist (Disclaimer etc.), der nichts mit dem Werk an sich zu tun hat.

Wir haben ferner einen generischen Workflow kennengelernt, mit dem sich viele Datenverarbeitungsprozesse ausdrücken lassen.

1. Selektieren von Daten (ggf. mehrstufig)
2. Verarbeiten der Daten (ggf. mehrstufig)
3. Aggregrieren der Daten zu einem Result

Alle kennen gelernten Streammethoden wollen wir nun anwenden, um eine Aufstellung der häufigsten Palindrome in deutschen Texten zu erhalten.

Das Werk beginnt nach einer Textzeile die

```
"THOMAS MANN + BUDDENBROOKS"
```

enthält und endet mit einer Textzeile, die

```
"=Ende="
```

enthält. Alles dazwischen ist Text des Werks und soll auf Palindrome analysiert werden.

Folgende Codezeile würden also die *"Buddenbrooks"* ausgeben.

```
BufferedReader text = new BufferedReader(
    new FileReader("/Path/to/your/buddenbrooks.txt")
);
text.lines()
    .dropWhile(s -> !s.contains("THOMAS MANN + BUDDENBROOKS"))
```

```
    .takeWhile(s -> !s.contains("=Ende="))
    .forEach(System.out::println);
```

Als erste Aufgabe müssen wir die Bedingung formulieren, ob eine Zeichenkette ein Palindrom ist.

```
Predicate<String> palindrom = s -> s.length() > 1 &&
    (new StringBuffer(s)).reverse().toString().equalsIgnoreCase(s);
```

In einem nächsten Schritt müssen wir alle Worte in dem Text als Stream von Zeichenketten erhalten. Leider wird die Datei zeilenweise verarbeitet. In jeder Zeile können wir mit der `split()`-Methode recht einfach alle Worte trennen, jedoch erhalten wir dann einen Stream von Streams (die nicht unbedingt einfach zu verarbeiten sind). Wir bedienen uns daher anstelle der `map()`-Methode einer Variante, nämlich der `flatMap()`-Methode. Diese können Sie sich so vorstellen, dass alle Streams (also Zeilen) hintereinander gehängt werden. Folgende Zeilen generieren also alle Worte der *Buddenbrooks*.

```
text.lines()
    .dropWhile(s -> !s.contains("THOMAS MANN + BUDDENBROOKS"))
    .takeWhile(s -> !s.contains("=Ende="))
    .flatMap(z -> Stream.of(z.split("[\\p{Punct}\\s]+"))) // Worte bestimmen
    .filter(palindrom)                                    // Palindrome filtern
    .forEach(System.out::println);
```

Damit haben wir unsere Daten selektiert und aufbereitet, wir müssen diese so bestimmten Palindrome jetzt nur noch zählen, z.B. mit der `collect()`-Methode und dem `groupingBy()`-Kollektor.

```
Map<String, Long> palindrome = text.lines()
    .dropWhile(s -> !s.contains("THOMAS MANN + BUDDENBROOKS"))
    .takeWhile(s -> !s.contains("=Ende="))
    .flatMap(z -> Stream.of(z.split("[\\p{Punct}\\s]+"))) // Worte bestimmen
    .filter(palindrom)                                    // Palindrome filtern
    .collect(Collectors.groupingBy(s -> s.toLowerCase(), Collectors.counting()));
```

Diese Map können wir wie folgt auf der Konsole ausgeben:

```
palindrome.entrySet().stream().forEach(e ->
    System.out.printf("%s: %d\n", e.getKey(), e.getValue())
);
```

Dies würde eine Ausgabe wie die hier ergeben:

```
hoh: 1
ehe: 39
tät: 2
tat: 78
nun: 447
...
```

Wie wir sehen ist diese Ausgabe leider nicht absteigend sortiert. Um einen Überblick über die Häufigkeit von Palindromen zu bekommen, wollen wir die Ausgabe absteigend nach Häufigkeit (also nach dem **Value** und nicht nach dem **Key**) sortiert ausgeben lassen. Wir schieben dazu einfach nur einen `sorted()`-Schritt in das Stream-Processing ein.

```
palindrome.entrySet().stream()
    .sorted((e1, e2) -> (int)(e2.getValue() - e1.getValue()))
    .forEach(e ->
        System.out.printf("%s: %d\n", e.getKey(), e.getValue())
    );
```

5.8 Buchstabenhäufigkeiten in Texten am Beispiel der *Buddenbrooks* bestimmen (nur INF)

Erinnern Sie sich an die Unit 03 und die Aufgabe 3.3 (Sprache eines Satzes bestimmen)? Im Kern haben wir die Buchstabenhäufigkeit innerhalb eines Satzes mit der Buchstabenhäufigkeit einer Referenzverteilung einer Sprache verglichen. Diese Referenzverteilung haben wir in Aufgabe 3.3 einfach "irgendwo abgeschrieben". Die Frage ist jedoch, wie diese Referenzverteilungen eigentlich bestimmt werden können.

Das ist eigentlich ganz einfach: Man nimmt einfach viele Texte einer Sprache (einen sogenannten Korpus) und zählt in diesen Texten die Buchstaben. Genau dies wollen wir nun am Beispiel der *Buddenbrooks* machen.

Sie werden sehen, dass die Verarbeitung sich natürlich von der Palindrom Verarbeitung unterscheidet, aber auch erhebliche Parallelen aufweist. Wie im vorherigen Beispiel lesen wir erstmal den Text ein und filtern ihn auf den vom Autor verfassten Umfang.

```
text.lines()
    .dropWhile(s -> !s.contains("THOMAS MANN + BUDDENBROOKS"))
    .takeWhile(s -> !s.contains("=Ende="))
    .forEach(System.out::println);
```

Während wir bei den Palindromen an Worten interessiert waren, sind wir aber nun an den Zeichen interessiert. D.h. unsere Vorverarbeitung wirft erst einmal alle Trenn- und Interpunktionszeichen raus, damit wir einen Stream von Zeichen (und nicht von Worten) erhalten. Ferner sind wir nur an den Zeichen von `'a'` bis `'z'` interessiert.

```
text.lines()
    .dropWhile(z -> !z.contains("THOMAS MANN + BUDDENBROOKS"))
    .takeWhile(z -> !z.contains("=Ende="))
    .flatMap(z -> z.toLowerCase().chars().mapToObj(c -> (char)c)) // Stream von Zeichen
    .filter(z -> z >= 'a' && z <= 'z')                            // Filtern
    .forEach(System.out::print);
```

Dies ergibt eine Ausgabe wie die folgende auf der Konsole …

```
thomasmannbuddenbrooksverfalleinerfamilie...
```

Wie bei den Palindromen müssen wir nun mittels `collect()` aggregieren, um die Buchstaben zählen. Anschließend sortieren wir absteigend nach der Häufigkeit der Vorkommen und geben diese auf der Konsole aus.

```
text.lines()
    .dropWhile(z -> !z.contains("THOMAS MANN + BUDDENBROOKS"))
    .takeWhile(z -> !z.contains("=Ende="))
    .flatMap(z -> z.toLowerCase().chars().mapToObj(c -> (char)c))  // Stream von Zeichen
    .filter(z -> z >= 'a' && z <= 'z')                             // Filtern
    .collect(Collectors.groupingBy(c -> c, Collectors.counting())) // Zählen
    .entrySet().stream()
    .sorted((e1, e2) -> (int)(e2.getValue() - e1.getValue()))      // Sortieren
    .forEach(e ->
        System.out.printf("%s: %d\n", e.getKey(), e.getValue())    // Ausgabe
    );
```

Dies ergibt eine Ausgabe wie diese hier:

```
e: 201705
n: 126254
```

```
i: 94597
r: 84885
s: 73547
...
```

Und wir erkennen, dass e und n erwartungsgemäß die häufigsten Buchstaben in den *Buddenbrooks* sind.

Lambdaausdrücke können schnell lang und übersichtlich werden.
Es bietet sich daher an, Sequenzen von Arbeitsschritten durch Formatierung zu veranschaulichen, indem man diese untereinander schreibt.
Dennoch lassen sich in nicht einmal 10 Zeilen die Häufigkeit von Palindromen oder Zeichen in Texten problemlos bestimmen.
Unterschätzen Sie daher nicht die Mächtigkeit von Lambda-Funktionen. Diese können Ihnen viel Arbeit abnehmen.

5.9 Eingeführte Konzepte und Methoden der Unit 05

- Einfache Rekursionen (hoch/runter zählen)
- Sequenzbasierte Rekursionen (Kopf, Rest, Rekursionsaufruf auf Rest)
- Sequenzbasierte Rekursionen durch Einführung einer überladenen Methode mit zusätzlichem Indexparameter auf einfache Rekursionen zurückführen.
- `subList()`-Methode um Teillisten von Listen zu bestimmen (Sequenzen für Rekursionsaufruf kleiner machen).
- Ggf. erforderliche Zustände zwischen Rekursionsaufrufen mittels zusätzlicher Parameter in überladenen Methoden weiterreichen.
- Rekursive Datenstrukuren am Beispiel eines Binärbaums (`Node`).
- Inorder (LKR) und preorder (KLR) Durchläufe auf Binärbäumen.
- Komplexerer rekursiver Algorithmus am Beispiel von BinSort.
- Lambda Funktionen am Bsp. von `Predicate` und einer `filter` Methode.
- `BufferedReader.lines()` Methode um Dateien zeilenweise als Strings per Stream zu verarbeiten.
- `dropWhile(), takeWhile()` um Daten vorzuselektieren,
- Maps mittels `entrySet()` und `stream()` als Stream von Key/Value-Paaren zu verarbeiten,
- `sorted()`-Methode um Streams zu sortieren.
- `forEach()`-Methode (als Pendant zur `foreach`-Schleife)
- `filter(), map(), collect()` Pattern, um Daten funktional per Streams zu verarbeiten.
- `flatMap()`, um Streams von Streams "plattzudrücken" (engl. to flatten) und als einen durchgängigen Stream zu verarbeiten.
- `Collectors.groupingBy()` zum Gruppieren von Streams als Maps.
- `Collectors.counting()` vglb. dem count Aggreator bei SQL.