Kommentierte Vorlesungsbeispiele (Java)

Nane Kratzke

02.10.2018

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[] 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.");
}
```

```java
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 in einem Satz (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.

```java
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.

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

```java
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 double average(String s) {
    String[] worte = s.split(" +");
    int sum = 0;
    for (String w : worte) {
        sum += w.length();
    }
    return sum / (double) worte.length;
    // Ohne cast würde die ganzzahlige Division
    // erfolgen (der Nachkommaanteil würde nicht
    // berücksichtigt werden).
}
```

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.

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.

```java
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 Teilme-

thoden 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.