Softwareentwicklung II (IB)

Blatt 1

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 28.01.2020 17:34

Git und GitHub

Die Abgaben in diesem Semester erfolgen wieder über GitHub. Sofern Sie noch keine Erfahrungen mit Git und GitHub haben, sehen Sie sich Blatt 0 der Lehrveranstaltung Softwareentwicklung I (IB) und `https://ob.cs.hm.edu/git.html` an.

Abgabe der Aufgabe auf diesem Blatt: bis 11.04.18, 08:00 Uhr durch Pushen in das zur Aufgabe gehörende GitHub-Repository.

Bewertung: Eine Aufgabe ist korrekt gelöst, wenn

1. Sie **fristgerecht** den Code in das zur Aufgabe gehörende GitHub-Repository gepusht haben und
2. der zugehörige Jenkins-Job **bei Fristende** grün ist und
3. Sie bis spätestens eine Woche nach Fristende **kein** Feedback erhalten haben, dass etwas **falsch** ist.

Im Praktikum zur Lehrveranstaltung "Softwareentwicklung II (IB)" sollen Sie das in der Vorlesung gelernte anwenden und Sie sollen auch üben sich in neue Aspekte hineinzudenken und einzuarbeiten. Für Fragen stehe ich Ihnen während der Praktikumstermine sowie außerhalb über ein Issue oder im Gitter-Channel zur Verfügung.

Gradle und IntelliJ IDEA

Die Aufgaben sind, wie in Softwareentwicklung I (IB), als Gradle-Projekte vorbereitet und als solche abzugeben. Sie können die Aufgaben lokal mit einem beliebigen Editor

oder einer IDE bearbeiten. Ich empfehle, wie in Softwareentwicklung I (IB), die Nutzung von IntelliJ IDEA oder dem Atom Editor.

Sollten Sie mit den genannten Tools nicht vertraut sein, sehen Sie sich die Aufgabenblätter von Softwareentwicklung I (IB) an.

Ab diesem Blatt ist das Gradle-Projekt so konfiguriert, dass Sie mit

```
./gradlew idea
```

ein Projekt für IntelliJ IDEA erzeugen können. Das heißt der Workflow zu Beginn jeder Aufgabe sollte sein:

1. Repository über GitHub-Classroom-URL erzeugen.
2. Repository auf der Kommandozeile oder mit einem Git-Tool (z.B. SourceTree oder GitHub Desktop) clonen.
3. Auf der Kommandozeile in das Repository wechseln und

   ```
   ./gradlew idea
   ```

   eingeben.
4. IntelliJ IDEA starten und das Projekt **öffnen**, nicht importieren.
5. Wenn Sie von IntelliJ IDEA gefragt werden, ob Sie das Gradle-Projekt importieren wollen, machen Sie das.
6. Beim Import setzen Sie zu den ausgewählten Punkten noch einen Haken bei `Use auto-import`.

Aufgabe 1 — SkiShop

Das Repository für diese Aufgabe bekommen Sie unter `https://classroom.github.com/a/VlomPUPI`. Wählen Sie zunächst Ihre HM-E-Mail-Adresse aus um mir eine Zuordnung zu ermöglichen. Sollte Ihre E-Mail-Adresse nicht in der Liste stehen, sind Sie nicht für diesen Kurs eingeschrieben.

Sie finden darin schon eine Anwendung `SkiShopApp`. Um diese erfolgreich zu compilieren und zu starten, fehlen aber noch zwei Klassen, `Ski` und `SkiShop`, die Sie im Rahmen dieser Aufgabe implementieren müssen.

In dieser Aufgabe verwenden wir erstmalig eine sogenannte `Map` um viele gleichartige Objekte darin zu speichern und zu verwalten. Eine `Map` ist so etwas ähnliches wie eine Liste (Listen werden wir demnächst auch verwenden).

Wir nutzen die `Map` um darin Ski zu speichern. Wenn wir sie in eine Liste speichern, dann haben die Ski eine Reihenfolge: Das erste Paar Ski, dann das zweite Paar, das dritte, …

bis zum letzten Paar. Wenn wir ein Paar löschen, rücken die anderen nach vorne, d.h. wenn wir das zweite Paar löschen, wird das dritte Paar zum neuen zweiten, das vierte zum neuen dritten usw.

Wenn wir die Ski in einer `Map` speichern, haben sie keine Reihenfolge, d.h. es gibt kein erstes Paar usw. Um die verschiedenen Ski zu identifizieren, gibt es einen sogenannten Schlüssel. Ein Schlüssel ist ein eindeutiger Wert, der Objekte identifiziert. Zum Beispiel ist die Autonummer ein Schlüssel für zugelassene Autos und die Matrikelnummer ein Schlüssel für Studierende.

D.h. wir müssen unseren Ski jeweils einen eindeutigen Schlüssel zuordnen, den wir zum Speichern in einer `Map` nutzen können. Wir nutzen dafür einfach eine Objektvariable `id` vom Typ `int`. Damit sieht unsere Ski-Klasse folgendermaßen aus:

```
class Ski {
  final int id; // (eindeutiger) Schlüssel
  final String manufacturer;
  final String model;
  final int price; // Preis in Cent
  // hier fehlt noch ein bisschen etwas...
}
```

Es gibt verschiedene Implementierungen von `Map`s in Java, z.B. die `HashMap`, die ein sog. Hashingverfahren zur Verteilung und zum schnellen Auffinden der Werte verwendet. Wir verwenden eine `TreeMap`. Vorteil der `TreeMap` ist, dass sie die Werte nach den Schlüsseln sortiert speichert und zurück gibt. Das macht die `HashMap` beispielsweise nicht, dafür ist das Speichern und Zugreifen bei einer `HashMap` viel schneller.

Um eine Variable für eine `TreeMap` und eine `TreeMap` zu erzeugen, müssen wir mit dem Typ `TreeMap` angeben, welchen Typ die Schlüssel und welchen die Werte haben. Angegeben wird das in der Reihenfolge `Schlüsseltyp, Werttyp` in spitzen Klammern (Größer- bzw. Kleinerzeichen). Für die `Map` für die Ski deklarieren wir also folgendermaßen eine `TreeMap`:

```
java.util.TreeMap<Integer, Ski> skis;
```

Damit ist `skis` eine Variable die eine `TreeMap` referenziert, deren Schlüssel `Integer` und deren Werte `Ski` sind. In Java lassen sich als Schlüssel- oder Werttypen keine primitiven Typen, wie z.B. `int`, verwenden. Daher nehmen wir den Referenztyp `Integer`, der einen `int` kapselt. Die Umwandlung von einem `int` in einen `Integer` und zurück, funktioniert aber automatisch. Daher können wir in der `Ski`-Klasse einfach bei dem `int` bleiben. Um eine neue, leere `TreeMap` zu erzeugen, verwenden wir:

```
new java.util.TreeMap<Integer, Ski>();
```

Wenn wir diese `TreeMap` in der Klasse `SkiShop` als Objektvariable verwenden wollen, können wir das so zusammensetzen.

```
import java.util.TreeMap;

class SkiShop {
  final TreeMap<Integer, Ski> skis = new TreeMap<>();
  ...
}
```

Wir habe dabei zwei kleine Verbesserungen eingebaut.

1. Um nicht jedesmal `java.util.TreeMap` schreiben zu müssen, können wir dem Compiler **vor** der Klassendefinition mit dem `import`-Statement sagen, dass er `TreeMap` doch bitte durch `java.util.TreeMap` ersetzt, wenn er es sonst nicht findet.
2. Wenn wir eine `TreeMap`-Variable unter Angabe von Schlüssel- und Wertetypen definieren und sofort mit einer `TreeMap` initialisieren, müssen wir die Schlüssel- und Wertetypen nicht noch einmal angeben. Es reichen die leeren spitzen Klammern.

In der Java-Dokumentation finden Sie unter https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html die Beschreibung einer `TreeMap` und der Methoden, d.h. Sie können dort nachlesen was man alles mit einer `TreeMap` machen kann und, vor allem, wie.

Beispielsweise finden Sie die Info zur Methode `put` mit der Sie ein Schlüssel-Wert-Paar in die `Map` einfügen können. Wenn wir beispielsweise eine Referenzvariable `ski` haben, der wir ein `Ski`-Objekt zugewiesen haben, können wir es folgendermaßen zu den `skis` hinzufügen:

```
skis.put(ski.id, ski);
```

Falls es bereits einen Eintrag mit dem Schlüssel `ski.id` in der `TreeMap` gab, wird dieser ersetzt. D.h. mit einer `Map` können wir Werte ganz einfach *updaten* in dem wir einen neuen Wert *darüber schreiben*. `put` gibt zwar ein Ergebnis zurück, da wir es aber nicht weiterverwenden wollen, müssen wir es keiner Variablen zuweisen.

Um aus einer `TreeMap` etwas zu entfernen, müssen wir einfach nach einer Methode suchen, die einen englischen Namen hat, der etwas mit *Entfernen* zu tun hat. Um zu Überprüfen ob die `TreeMap` leer ist, finden Sie sicherlich genauso eine Methode. Und um einen Wert aus der `TreeMap` zu bekommen, gibt es die Methode `get`. Sie bekommt den Wert zurück oder `null`, wenn es keinen Eintrag mit dem übergebenen Schlüssel gibt.

Beispiel: Wir haben Ski mit den `ids` `1` und `2` gespeichert:

```
Ski ski2 = skis.get(2); // Ski mit der id 2
Ski ski3 = skis.get(3); // ski3 == null
```

Wenn Sie nur überprüfen wollen, ob es einen Wert mit einem bestimmten Schlüssel gibt, Sie aber am konkreten Wert gar nicht interesiert sind, gibt es z.B. die Methode `containsKey`.

Die Klasse `Ski`

Die Klasse `Ski` hat folgende **unveränderliche** Objektvariablen:

- `id`: Ein eindeutiger Schlüssel
- `manufacturer`: Name des Herstellers
- `model`: Modell des Skis
- `price`: Preis des Skis in Cent
- `stringRepresentation`: Ein `String`, der den Ski folgendermaßen repräsentiert:

  `<id>: <manufacturer> <model> <preis in € x,xx>`

  `id`, `manufacturer` und `model` können einfach genutzt werden. Der Preis muss so angegeben werden: `€` gefolgt von einem Leerzeichen, dann der €-Anteil, gefolgt von einem Komma und dann der Cent-Anteil zweistellig. Berechnen Sie dazu den Centanteil. Wenn dieser einstellig ist, geben Sie erst eine `0` aus und dann den einstelligen Centbetrag. Beispiel:

  `13: Dynastar Powertrack 89 € 419,95`

- `jsonRepresentation`: Ein `String`, der den Ski in der Form eines JSON-Objektes repräsentiert, z.B.

```
{
  "id": 13,
  "manufacturer": "Dynastar",
  "model": "Powertrack 89",
  "price": 41995
}
```

Die Klasse `Ski` hat außerdem einen Custom-Konstruktor mit folgender Signatur:

```
Ski(final int id, final String manufacturer,
    final String model, final int price);
```

In diesem Konstruktor müssen **alle** sechs oben angegebenen Objektvariablen auf den korrekten Wert gesetzt werden.

Die Klasse `SkiShop`

Die Klasse `SkiShop` verwaltet die `TreeMap` mit den Ski, wie bereits oben angegeben.

Darüber hinaus gibt es 5 Methoden, die Sie implementieren müssen:

`printStock`

Die Methode

```
void printStock()
```

gibt alle Ski aus, die in der `TreeMap` gespeichert sind. Jeder Ski wird auf einer eigenen Zeile ausgegeben. Zur Ausgabe verwenden Sie die Objektvariable `stringRepresentation` aus der Klasse `Ski`.

Nutzen Sie zum Zusammensetzen der endgültigen Zeichenkette eine `StringBuilder` und verwenden Sie nur eine `print`-Anweisung. Denken Sie daran, dass es der Idee des `StringBuilder` widerspricht, wenn Sie `String`s mit `+` konkatenieren.

Um alle Werte aus der `TreeMap` zu bekommen, nutzen Sie die Methode `values`.

Neben den in Softwareentwicklung I (IB) behandelten Schleifen, gibt es noch die sog. foreach-Schleife, die erst im Zusammenhang mit Collections, wie `Map`s und Listen, Sinn macht. Statt mit einem Index durch die `Map` zu gehen, können Sie mit der foreach-Schleife direkt durch die Elemente iterieren. Zum Beispiel Zeilenweise Ausgabe aller `id`s:

```
for (Ski ski : skis.values()) {
  System.out.println(ski.id);
}
```

Die foreach-Schleife nutzt auch das Schlüsselwort `for`. In den runden Klammern gibt es aber nur eine Deklaration einer Variablen, z.B. `Ski ski`, gefolgt von einem Doppelpunkt. Hinter dem Doppelpunkt steht eine Collection, im Beispiel eine Collection aller Werte aus der `TreeMap` `skis`. In jedem Schleifendurchlauf referenziert die Variable `ski`, dann einen der Ski. Die Schleife läuft automatisch so oft durch, wie es Werte in der Collection gibt.

Obwohl die Methode `printStock` mit Hilfe der foreach-Scheife einfach **nichts** bei einer leeren `TreeMap` ausgeben würde, müssen Sie es explizit prüfen und bei einer leeren `TreeMap` die Zeichenkette `empty` ausgeben.

`printStockJson`

Die Methode

```
void printStockJson()
```

gibt alle Ski aus der `TreeMap` als sog. JSON-Array aus. Ein JSON-Array entspricht einer Liste. Diese enthält für jedes Paar Ski ein JSON-Objekt. Das Array wird durch eckige Klammern notiert, die einzelnen Elemente werden durch Kommata getrennt. Beispielsweise wird ein Array bestehend aus den Zahlen 1, 3, 2 und 9 so geschrieben:

```
[ 1, 3, 2, 9]
```

Ein leeres JSON-Array wird durch leere eckige Klammern dargestellt, d.h. wenn die `TreeMap` leer ist, sollen Sie folgendes ausgeben:

```
[]
```

Obwohl JSON selbst kein besonderes Layout benötigt, sollen Sie die Ausgabe aber gut lesbar machen. Dazu sollen die JSON-Objekte, die je ein Paar Ski repräsentieren, um zwei Leerzeichen eingerückt in den eckigen Klammern stehen, die jeweils auf einer eigenen Zeile stehen. Ausgabe des Bestands mit einem Paar Ski sieht im Beispiel so aus:

```
[
  {
    "id": 13,
    "manufacturer": "Dynastar",
    "model": "Powerttrack 89",
    "price": 41995
  }
]
```

Nutzen Sie unbedingt die Objektvariable `jsonRepresentation` um die Ski auszugeben. Um jede Zeile von `jsonRepresentation` um zwei Leerzeichen mehr einzurücken, müssen Sie die Zeichenkette manipulieren. Mit Hilfe der Methode `replace` aus der Klasse `String` können Sie z.B. alle Vorkommen eines Teilstrings mit einem anderen Teilstring ersetzen, z.B.

```
"Hallo Welt!".replace("l", "_l_") // ergibt Ha_l__l_o We_l_t!
```

Das Komma, dass zwischen zwei Paar Ski stehen muss, schreiben Sie direkt hinter die schließende geschweifte Klammer des ersten Paar Ski, also z.B.

```
[
  {
    "id": 5,
    "manufacturer": "Head",
    "model": "Jerry",
    "price": 39900
  },
  {
    "id": 13,
    "manufacturer": "Dynastar",
    "model": "Powertrack 89",
    "price": 41995
  }
]
```

Wenn Sie wieder eine foreach-Schleife nutzen, was sich auf jeden Fall anbietet, müssen Sie aber wegen der Kommata mit zählen, wann Sie beim vorletzten Paar Ski angekommen sind, weil nach dem letzten natürlich kein Komma mehr stehen darf. Dazu können

Sie einen Zählvariable nutzen, die Sie wie bei `while`-Schleifen selbst initialisieren und im Schleifenrumpf verändern. Für die Anzahl der Elemente in einer `Map` gibt es die dafür nützliche Methode `size`. Selbstverständlich sollen Sie aus Effizienzgründen auch in `printStockJson` wieder einen `StringBuilder` nutzen.

`add`

Mit Hilfe der Methode

```
void add(final Scanner)
```

sollen neue Ski zum Bestand hinzugefügt werden können. Der Methode wird zum Einlesen der Eingaben der Kommandozeile einen Scanner übergeben. Die exakten Ausgaben sind durch die mitgelieferten Tests vorgegeben.

Die Methode `add` soll folgendermaßen ablaufen:

- Frage nach und Eingabe einer Seriennummer.
- Frage nach und Eingabe eines Herstellers.
- Frage nach und Eingabe eines Modells.
- Frage nach und Eingabe eines Preises in Cent.
- Ausgabe eines neu mit den Eingaben erzeugten Skis und Frage ob dieser zum Bestand hinzugefügt werden soll.

  Antwort `y` oder `Y`:

  - Überprüfung ob es bereits einen Ski mit der eingegebenen Seriennummer gibt.

    Wenn es einen gibt, frage ob der existierende Ski ersetzt werden soll

    * Antwort `y` oder `Y`: Alten Ski durch neuen mit der Methode `replace` ersetzen. Ausgabe `Ski added.`
    * Alle anderen Antworten: Ausgabe `Ski not added.`

    Wenn es keinen mit der selben Seriennummer gibt, einfügen und Ausgabe `Ski added.`

  Alle anderen Antworten: Ausgabe `Ski not added.`

Nutzen Sie zum Vergleich von Zeichenketten entweder eine `switch`-Anweisung oder `equals`.

`remove`

Mit Hilfe der Methode

```
void remove(final Scanner)
```

sollen Ski aus dem Bestand gelöscht werden können. Der Methode wird zum Einlesen der Eingaben der Kommandozeile einen Scanner übergeben. Die exakten Ausgaben sind durch die mitgelieferten Tests vorgegeben.

Die Methode `remove` soll folgendermaßen ablaufen:

- Frage nach und Eingabe einer Seriennummer.
- Überprüfung ob es einen Ski mit der eingegebenen Seriennummer im Bestand gibt.

  Wenn ja: Ausgabe `Ski with serial <serial> not in stock.`. Die Zeichenkette `<serial>` muss dabei selbstverständlich durch die eingegebene Seriennummer ersetzt werden.

  Wenn nein:

  – Ausgabe der Ski und Frage ob diese gelöscht werden sollen.

    Antwort `y` oder `Y`: Entfernen des Eintrags aus der `TreeMap` und Ausgabe `Ski removed.`.

    Alle anderen Antworten: Ausgabe `Ski not removed.`

`cheapen`

Mit Hilfe der Methode

```
void cheapen(final Scanner)
```

soll der Preis für ein Paar Ski aus dem Bestand heruntergesetzt werden können. Der Methode wird zum Einlesen der Eingaben der Kommandozeile einen Scanner übergeben. Die exakten Ausgaben sind durch die mitgelieferten Tests vorgegeben.

Die Methode `cheapen` fragt zunächst wieder nach der Seriennummer und sucht in der `TreeMap` nach den Ski. Werden keine gefunden, erfolgt eine Ausgabe wie bei `remove`. Werden die Ski mit der `id` gefunden, so wird ausgegeben:

```
Cheapen "<stringRepresentation der Ski>"?
Input: -x              --- for reducing by x cent,
       x%              --- for reducing by x %,
       everything else --- for not cheapening.
```

Wie die Ausgabe nahelegt soll es zwei verschiedene Möglichkeiten geben, die Ski zu reduzieren. Zum Einen um `x` Cent und zum Anderen um `x`%.

Erkannt werden muss das durch das `-` zu Beginn des Eingabestrings bzw. das `%` am Ende. Um das festzustellen, sind die Methoden `startsWith` und `endsWith` hilfreich.

Sie müssen also für die weitere Verarbeitung drei Fälle unterscheiden:

1. Die Eingabe beginnt mit `-`.
2. Die Eingabe endet mit `%`.
3. Alles sonst.

In den Fällen 1. und 2. dürfen Sie falsche Eingaben ignorieren, d.h. es ist okay wenn ihr Programm z.B. bei der Eingabe `-Hallo` abstürzt.

Im Fall 1. können Sie die gesamte Eingabe einfach mit `Integer.parseInt` in eine Zahl umwandeln. Im Fall 2. brauchen Sie einen Zwischenschritt. Sie müssen das `%`-Zeichen los werden, bevor Sie `Integer.parseInt` nutzen können. Das geht z.B. in dem Sie mit der Methode `substring` nur die ersten Zeichen extrahieren. Die Länge des `String`s können Sie übrigens mit `lenghth` berechnen. Das erste Zeichen von `str` hat den Index 0, das letze den Index `str.length - 1`.

In beiden Fällen, 1. und 2., soll als erstes der neue Preis berechnet werden. Die Eingabe von 1. ist der Wert in Cent der **addiert** wird. Beispiel: Ein Ski kostet `50000` Cent und es wird `-10000` eingegeben, dann ist der neue Preis: `50000 + -10000 = 40000` Cent . Die Eingabe von 2. ist die Prozentzahl **um** den der Skipreis verringert wird. Beispiel: Ein Ski kostet `40000` Cent und es wird `25%` eingegeben, dann ist der neue Preis: `30000` Cent. Wird der neue Preis in einem der beiden Fälle negativ, so soll ausgegeben werden:

```
New price <newPrice> not acceptable.
```

Wobei `<newPrice>` durch den Centbetrag ersetzt werden soll, also z.B. `-10000`.

Anderenfalls wird ein neuer Ski erzeugt und ausgegeben:

```
New ski "<stringRepresentation des neuen Ski>" ok? [y/N]
```

Lautet die Antwort `y` oder `Y` wird der alte Ski in der `TreeMap` durch den neuen ersetzt und ausgegeben `Ski cheapened.`. Sonst wird ausgegeben `Ski not cheapened.`. Letzteres soll auch im Fall 3 ausgegeben werden.