Softwareentwicklung II (IB)

Blatt 3

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 04.05.2018 21:46

Abgabe der Aufgabe auf diesem Blatt: bis 23.05.18, 08:00 Uhr durch Pushen in das zur Aufgabe gehörende GitHub-Repository.

Aufgabe 0 — Javadoc

Lesen Sie sich das Kapitel Javadoc (Slides, Handout) durch.

Ab diesem Blatt ist die Dokumentation **aller** `public`-Sourcen mit Javadoc-Kommentaren obligatorisch. Andernfalls müssen Sie mit Punktabzug rechnen. Ab Blatt 3 führt fehlendes Javadoc zum Nichtbestehen der entsprechenden Aufgabe.

Aufgabe 1 — Autovermietung

Das Repository für diese Aufgabe bekommen Sie unter `https://classroom.github.com/a/-PXCXQrd`.

a) **Personen**

Implementieren Sie die Klasse `Person` wie folgt:

- private Objektvariablen mit öffentlichen Gettern

  `name` Vor- und Nachname in einer Zeichenkette

  `drivingLicences` Eine Menge von Fahrerlaubnissen (nutzen Sie zur Implementierung ein `HashSet<String>`). Damit der Getter eine **Kopie** der Menge zurück geben kann, erzeugen Sie diese einfach mit

```
new HashSet<>(drivingLicences)
```

`activeRentals` Anzahl der im Moment gemieteten Fahrzeuge

- öffentliche Konstruktoren

  Implementieren Sie einen Custom-Konstruktor der Name und **eine** Fahrerlaubnis (ein `String`) als Parameter hat. Fügen Sie die eine Fahrerlaubnis in die Menge ein.

  Setzen Sie `activeRentals` auf einen sinnvollen Wert.

- öffentliche Methoden
  - um `activeRentals` zu inkrementieren und dekrementieren (darf aber nicht negativ werden)

    ```
    public void addActiveRental()
    public void removeActiveRental()
    ```

  - um eine Fahrerlaubnis hinzuzufügen oder zu entfernten

    ```
    public void addDrivingLicence(final String drivingLicence)
    public void removeDrivingLicence(final String drivingLicence)
    ```

b) **Autos**

Implementieren Sie die Klasse `Car` wie folgt:

- private Objektvariablen mit öffentlichen Gettern

  `manufacturer` Hersteller
  `type` Modell
  `licencePlate` Nummernschild
  `neededDrivingLicence` die zum Fahren notwendige Fahrerlaubnis (eine Zeichenkette)
  `commissioned` zeigt an ob das Auto zur Zeit vermietet ist

- Konstruktoren

  Implementieren Sie zwei Konstruktoren: Einen für Hersteller, Modell und notwendige Fahrerlaubnis und einen bei dem zusätzlich (als dritter Parameter zwischen Modell und Fahrerlaubnis) das Nummernschild übergeben wird.

- öffentliche Methoden
  - Nummernschild setzen

    ```
    public void setLicencePlate(final String licencePlate)
    ```

  - vermieten und zurück geben

    ```
    public void commissionIt()
    public void resetCommission()
    ```

c) **Mietverträge**

Implementieren Sie die Klasse `RentalAgreement` wie folgt:

- private Objektvariablen mit öffentlichen Gettern

  `id` Identifier (`int`)
  `car` das gemietete Auto
  `person` der Mieter/die Mieterin
  `duration` Mietdauer = Anzahl der Tage (1 Tag bedeutet von heute bis morgen) Die Mietdauer wird im Laufe eines Mietvertrages mit der Methode `nextDay` (siehe unten) heruntergezählt.
  `lastday` Anzeige ob der letzte Tag der Mietdauer erreicht ist
  `active` Anzeige ob zum Mietvertrag gerade ein Mietverhältnis besteht oder ob das Auto bereits abgegeben wurde

- Konstruktoren

  Implementieren Sie die folgenden **verketteten** Konstruktoren:

  ```
  public RentalAgreement(final int id, final Car car,
      final Person person)
  public RentalAgreement(final int id, final Car car,
      final Person person, final int duration)
  ```

  Beim ersten Konstruktor soll die Mietdauer auf `1` gesetzt werden.

- öffentliche Methoden

  – Auto zum Vertrag hinzufügen:

    ```
    public void setCar(final Car car)
    ```

    Nutzen Sie diese Methode auch aus dem Konstruktor heraus. Ein Auto kann nur erfolgreich zu einem Mietvertrag hinzugefügt werden, wenn es existiert ($\neq$ `null`), wenn es nicht gerade verliehen ist und wenn die Person über die notwendige Fahrerlaubnis verfügt.

  – Weiterschalten des Vertrags zum nächsten Tag

    ```
    public boolean nextDay()
    ```

    Die Verträge sind so, dass als Dauer immer der Wert den das Mietverhältnis noch läuft festgehalten ist.

    D.h. wenn ein Mietvertrag heute noch 5 Tage läuft, läuft er am nächsten Tag nur noch 4 Tage.

    Die Methode soll zurück geben ob der letzte Tag erreicht ist und in diesem Fall `lastday` auf wahr setzen.

    Wird der Tag über den letzten Tag hinaus weiter gezählt, soll `lastDay` wieder auf `false` gesetzt werden.

  - Beenden des Mietverhältnisses mit

    ```
    public void finish()
    ```

  - Canceln des Mietverhältnisses mit

    ```
    public void cancel()
    ```

    Canceln ist genau dasselbe wie Beenden. Schreiben Sie daher möglichst keinen doppelten Code. Da sich in einer zukünftigen Version beides unterscheiden könnte, werden aber von Beginn an beide Methoden zur Verfügung gestellt.

d) **Autovermietung**

Implementieren Sie die Klasse `CarRentalAgency` wie folgt:

- private Objektvariablen **ohne** Gettern

  `persons` Eine `HashMap<String, Person>`; Schlüssel ist der Name der Person.
  `cars` Eine `HashMap<String, Car>`; Schlüssel ist das Nummernschild des Autos.
  `rentalAgreements` Eine `HashMap<Integer, RentalAgreement>`; Schlüssel ist die `id` des Mietvertrages.
  `lastId` Speichert die zuletzt vergebene `id` für Mietverträge. Begonnen werden soll mit `1` und dann fortlaufend inkrementiert.

- Konstruktoren

  Sie benötigen keinen extra Konstruktor. Stellen Sie aber sicher, dass alle Objektvariablen initialisiert werden.

- öffentliche Methoden

  - Person zum Bestand hinzufügen:

    ```
    public boolean addPerson(final String name,
        final String drivingLicence)
    ```

    Wenn eine Person mit dem selben Namen bereits vorhanden ist, soll die neue Person **nicht** eingefügt werden und `false` zurück gegeben werden. Sonst einfügen und Ergebnis `true`.

  - Auto zum Bestand hinzufügen:

    ```
    public boolean addCar(final String manufacturer, final String type,
        final String licencePlate, final String neededDrivingLicence)
    ```

    Wenn ein Auto mit dem selben Nummernschild bereits vorhanden ist, soll das neue Auto **nicht** eingefügt werden und `false` zurück gegeben werden. Sonst einfügen und Ergebnis `true`.

  - Mietvertrag erstellen

```
public int rent(final String name, final String licencePlate,
    final int days)
```

Diese Methode soll einen neuen Mietvertrag mit der nächsten zu vergebenden `id` (siehe `lastId`) erzeugen und die `id` als Ergebnis zurück liefern.

In folgenden Fehlerfällen wird als Ergebnis eine negative Zahl zurück gegeben:

* Person mit dem Namen ist nicht gespeichert: `-1`
* Auto mit dem Nummernschild ist nicht gespeichert: `-2`
* Auto ist derzeit verliehen: `-3`
* Die Person verfügt nicht über die notwendige Fahrerlaubnis: `-4`

– Die Methode

```
public int rent(final String name, final String licencePlate)
```

ermöglicht einen Mietvertrag für einen Tag abzuschließen und nutzt die bereits implementierte `rent`-Methode

– Rückgabe des Autos

```
public boolean returnCar(final int id)
```

Gibt es keinen Mietvertrag mit der `id` soll `false` zurück gegeben werden. Andernfalls wird der Mietvertrag beendet, aber nicht aus der `HashMap` entfernt.

– Weiterschalten aller Verträge zum nächsten Tag

```
public Set<Integer> nextDay()
```

Alle Mietverträge sollen um einen Tag weiter geschaltet werden. Ergebnis der Methode ist eine Menge von `id`s der **aktiven** Mietverträge die an diesem (neuen) Tag den letzten Tag gesetzt haben. Diese Info bekommen Sie beim Weiterschalten der Verträge als Ergebnis! Fügen Sie sie in dem Fall zu der Menge hinzu und geben Sie die Menge am Ende einfach mit `return` zurück.

In der gesamten Aufgabe kommt keine Eingabe (`Scanner`) oder Ausgabe (`System.out.print`) vor. Sie können sich selbst eine `CarRentalApp` zum Ausprobieren schreiben.

Aufgabe 2 — (fast) Unveränderliche Flaschen

Das Repository für diese Aufgabe bekommen Sie unter https://classroom.github.com/a/ZdTEOoWm.

Implementieren Sie eine Klasse `Bottle` für Flaschen mit einem *unveränderlichem* Fassungsvermögen (`max`) und einem *unveränderlichem* Inhalt (`content`). Darüber hinaus soll es einen Indikator `broken` geben, der anzeigt ob die Flasche zerbrochen ist oder nicht. Eine neue Flasche ist nie zerbrochen (`broken == false`). Die Objektvariable `broken` ist die einzige veränderliche Objektvariable der Klasse `Bottle`.

Sobald die Flasche zerbrochen ist, sind Inhalt und Fassungsvermögen gleich 0. Nachdem dies nicht in den unveränderlichen Objektvariablen verändert werden kann, müssen die Getter das entsprechend kapseln.

Implementieren Sie folgende zwei Konstruktoren:

```
public Bottle(final int content, final int max)
public Bottle(final int content, final Bottle bottle)
```

für die folgendes gilt:

- ist einer der beiden Parameter des ersten Konstruktors negativ soll eine `IllegalArgumentException` geworfen werden:

  ```
  if ( ... )
      throw new IllegalArgumentException();
  ```

- ist der Parameter für den Inhalt größer als das Fassungsvermögen, soll der Inhalt auf das Maximum gesetzt werden
- der zweite Konstruktor enthält nur einen Aufruf des anderen (Verkettung, `this( ... )`) und eine Anweisung zum Zerbrechen der übergebenen Flasche
- der zweite Konstruktor soll dasselbe Fassungsvermögen wie die übergebene Flasche haben, aber den übergebenen Inhalt

Implementieren Sie die folgenden Methoden:

```
public boolean isFull()
```

Gibt zurück ob die Flasche voll ist.

```
public boolean isEmpty()
```

Gibt zurück ob die Flasche leer ist.

```
public int fillingLevel()
```

Gibt den Füllstand der Flasche in Prozent (ganzzahlig) an. Beispiel: Der Aufruf der Methode der Flasche `new Bottle(20,50)` gibt 40 zurück.

```
public Bottle empty()
```

Gibt eine neue leere Flasche zurück, die genauso groß ist wie diese, und besteht aus nur einer Anweisung. Diese Flasche ist danach zerbrochen.

```
public Bottle fill()
```

Gibt eine neue ganz volle Flasche zurück, die genauso groß ist wie diese, und besteht aus nur einer Anweisung. Diese Flasche ist danach zerbrochen.

```
public Bottle fill(final int delta)
```

Gibt eine neue Flasche mit dem Inhalt dieser Flasche plus dem `delta` zurück und enthält nur eine Anweisung. Diese Flasche ist danach zerbrochen.

```
public Bottle fill(final Bottle other)
```

Gibt eine neue Flasche mit dem Inhalt dieser Flasche plus dem Inhalt der anderen Flasche (bis zum Maximum) zurück und enthält nur eine Anweisung. Diese Flasche ist danach zerbrochen.

Schreiben Sie sich zum Ausprobieren einen eigene `BottleApp`.