Programmieren II (Java)

2. Praktikum: Grundlagen Objektorientierung

Sommersemester 2023
Christopher Auer, Tobias Lehner

Abgabetermine

Lernziele

- Implementieren nach einer Spezifikation
- Aufbau von Klassen: Attribute und Methoden
- Konstruktoren: Verkettung, Kopier-Konstruktor
- Getter und Setter
- Java-Standard-Methoden: `equals`, `toString`
- Dokumentation: `javadoc`
- Testen mit JUnit
- **`enum`**s: definieren, erweitern und verwenden

Hinweise

- Sie dürfen die Aufgaben *alleine* oder zu *zweit* bearbeiten und abgeben
- Sie müssen *4 der 5* Praktika bestehen
- *Kommentieren* Sie Ihren Code
  - Jede *Methode* (wenn nicht vorgegeben)
  - *Wichtige* Anweisungen/Code-Blöcke
  - *Nicht kommentierter* Code führt zu *Nichtbestehen*
- Bestehen Sie eine Abgabe *nicht* haben Sie einen *zweiten Versuch*, in dem Sie Ihre Abgabe *verbessern müssen*.
- *Wichtig*: Sie sind einer *Praktikumsgruppe* zugewiesen, *nur* in dieser werden Ihre Abgaben *akzeptiert*!

Bevor es losgeht

Bevor wir starten, lesen Sie sich *aufmerksam* folgende Hinweise durch:

- *Importieren* Sie zunächst das beigefügte *Gradle-Projekt* unter `SupportMaterial/virtualpet` in Ihre bevorzugte Entwicklungsumgebung. Erstellen Sie Ihre Klassen im Projekt-Verzeichnis unter:

  `app/src/main/java`

  Wenn Sie Ihre Klassen *woanders* erstellen, wird Ihr Projekt *nicht funktionieren*!
- *Dokumentieren* Sie *alle Klassen* und *öffentlichen Methoden* mit *JavaDoc*! Zusätzlich müssen Sie weiterhin Ihren Quellcode *kommentieren*.
- Verwenden Sie zum Prüfen auf *Gleichheit* von `Strings` und anderen Objekten die Methode `equals`!
- Prüfen Sie die Parameter jeder Methode auf *Gültigkeit*! Sollte ein Parameter einen ungültigen Wert haben, erzeugen Sie eine `IllegalArgumentException` wie folgt:

  ```
  throw new IllegalArgumentException("Aussagekräftige (!) Fehlermeldung");
  ```

  Geben Sie eine *aussagekräftige Fehlermeldung* an!
- Achten Sie auf die korrekte *Sichbarkeit* (**`public`**, **`private`**, **`protected`**) der Klassen, Attribute und Methoden! Es ist im Folgenden *nicht korrekt* einfach *keine Sichtbarkeit* anzugeben — und der Übungsleiter wird Sie darauf ansprechen!
- Testen Sie Ihre Klassen mit den mitgelieferten *JUnit*-Tests in den beigefügten Gradle-Projekten! Führen Sie dazu den Gradle-Task `test` aus. Die JUnit-Tests finden Sie im Projektverzeichnis unter:

  `app/src/test/java`

  Machen Sie sich außerdem mit der Unterstützung von JUnit-Tests in Ihrer Entwicklungsumgebung vertraut.
- *Tipp*: Kommentieren Sie zunächst alle Methoden in den JUnit-Tests aus. Haben Sie die Implementierung einer Methode abgeschlossen, kommentieren Sie die Tests mit entsprechenden Namen wieder ein. Diese sind nach dem Schema `testMethodenName...` benannt.
- Solange ein Test scheitert, ist *Ihre Implementierung nicht korrekt*. Betrachten Sie in diesem Fall die Fehlermeldung und den Quellcode des gescheiterten Tests. Für die Abgabe müssen *alle Testfälle* erfolgreich durchlaufen — der Übungsleiter wird Sie dazu auffordern, die Tests laufen zu lassen und die Abgabe *erst akzeptieren* wenn alle Tests *erfolgreich* sind!
- *Verändern Sie nicht die Inhalte der JUnit-Tests* um das Problem zu „lösen“!

Aufgabe: Virtuelle Haustiere

In dieser Aufgaben implementieren wir *virtuelle Haustiere und ihre Besitzer*. Dazu implementieren wir im Folgenden drei Klassen:

- `Pet` modelliert ein *virtuelles Haustier* und dessen *Stimmung* (`happiness`, `hungriness`, `sleepiness`).
- `PetType` ist ein **`enum`**, das *vier Tierarten* definiert.
- `PetOwner` modelliert den *Besitzer* von einem oder zwei virtuellen Tieren, der die Tiere *bei Laune halten* muss.

Hauptprogramm

Erstellen Sie ein Java-Hauptprogramm in einer Klasse `VirtualPetMain`! Die `main`-Methode ist zunächst leer und wird im Laufe der Übung erweitert. Natürlich dürfen/sollten Sie die `main`-Methode verwenden um Ihre Implementierung zu *testen*.

Tierarten: Das `enum PetType`

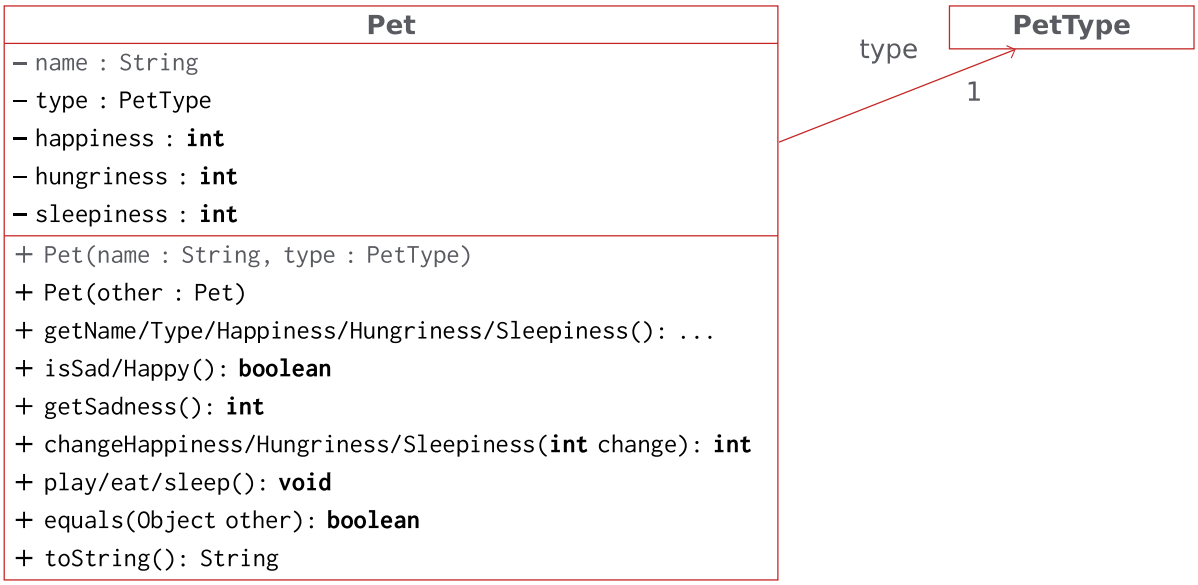
In unserer Simulation gibt es *vier Tierarten*, die durch ein **`enum`** `PetType` mit folgenden *Eigenschaften* modelliert sind:

Wert	`double happinessMultiplier`	`double hungrinessMultiplier`	`double sleepinessMultiplier`
`DOG`	2	2	2
`CAT`	1	3	2
`BIRD`	0.75	1	0.75
`RABBIT`	0.5	0.5	1

Die genaue Bedeutung der Attribute wird im Laufe der nächsten Aufgaben klar. *Implementieren* Sie `PetType` wie angegeben und statten Sie es mit den drei *Gettern* für die drei **`double`**-Attribute aus!

Die Klasse `Pet`

Die Klasse `Pet` modelliert ein *virtuelles Haustier*: [1]



Die Attribute von `Pet` modellieren:

- `String name` — der Name des Tiers (*unveränderlich*, darf *nicht* `null` oder *leer* sein)
- `PetType type` — der Typ des Tiers (*unveränderlich*, darf *nicht* `null` sein)
- `int happiness` — gibt an wie *zufrieden* das Tier gerade ist; ein *veränderlicher* Wert von 0 (*unglücklich*) bis 10 (*überglücklich*)
- `int hungriness` — gibt an wie *hungrig* das Tier gerade ist; ein *veränderlicher* Wert von 0 (*satt*) bis 10 (*sehr hungrig*)
- `int sleepiness` — gibt an wie *müde* das Tier gerade ist; ein *veränderlicher* Wert von 0 (*sehr wach*) bis 10 (*sehr müde*)

Deklarieren Sie die Klasse `Pet` und die *Attribute* wie angegeben! Achten Sie auf *korrekte Sichtbarkeit* und *Modifizierer*. Implementieren Sie den Konstruktor `Pet(String name, PetType type)`, der die beiden Attribute `name` und `type` *initialisiert*. Den „*Gemütszustand* “ des Tiers *initialisieren* sie mit `happiness=5`, `hungriness=3` und `sleepiness=1`.

Testen Sie Ihre Implementierung mit dem Test `PetTest.testInitConstructor`.

JUnit-Test: `PetTest.testCopyConstructor`.

Kopier-Konstruktor

Statten Sie `Pet` mit einem *Kopier-Konstruktor* `Pet(Pet other)` aus!

JUnit-Test: `PetTest.testCopyConstructor`.

Getter

Implementieren Sie die Getter für alle Attribute! Für das Attribut `happiness` implementieren *zusätzlich* noch folgende *öffentliche* Methoden:

[1] Aus *Platzgründen* sind manche Methoden, durch *Schrägstriche* getrennt, in einer Zeile aufgelistet.

- **boolean** `isHappy()` gibt **true** zurück, wenn `happiness` einen Wert ≥ 8 hat; sonst **false**
- **boolean** `isSad()` gibt **true** zurück, wenn `happiness` einen Wert ≤ 2 hat; sonst **false**
- **int** `getSadness()` gibt die *Traurigkeit* zurück, die dem Wert `10-happiness` entspricht.

JUnit-Tests: `PetTest.testIsSadHappy/testGetSadness`.

Die Methoden `changeHappiness/Hungriness/Sleepiness`

Die öffentliche Methode **int** `changeHappiness(int change)` zählt zum Attribut `happiness` den Wert `change` hinzu. Dabei darf `happiness` *mindestens* der Wert `0` und *höchstens* der Wert `10` annehmen. Die Methoden `changeHungriness` und `changeSleepiness` funktionieren nach dem gleichen Prinzip. Alle drei Methoden geben den *neuen Wert* zurück. Implementieren Sie die drei Methoden!

JUnit-Tests: `PetTest.testChangeHappiness/Hungriness/Sleepiness`.

Gleichheit mit `equals`

Implementieren Sie die `equals`-Methode in `Pet`! Zwei `Pet`-Instanzen sind *gleich* wenn alle ihre Attribute *gleich* sind. Gehen Sie dabei nach dem in der Vorlesung gezeigten „*Kochrezept* “ vor. Vergessen Sie nicht `@Override` zu verwenden und beachten Sie, dass das Argument der Methode `equals` vom *Typ* `Object` und *nicht vom Typ* `Pet` ist.

JUnit-Test: `PetTest.testEquals`

String-Repräsentation mit `toString`

Implementieren Sie die Methode `String toString()`, die eine `String`-Repräsentation des Tiers nach Folgendem Schema zurückgibt:

```
Rosco :-) (DOG): Ha: 8, Hu: 3, Sl: 7
Morgana :-| (CAT): Ha: 6, Hu: 3, Sl: 3
Rabbit of Caerbannog :-( (RABBIT): Ha: 2, Hu: 1, Sl: 3
```

In dem Beispiel ist der Name des Hundes `Rosco` mit `happiness=8`, `hungriness=3` und `sleepiness=7`. Das Emoji `:-)` heißt, dass `isHappy()` **true** zurückgibt. Für den Fall `isSad` verwenden Sie `:-(` und sonst `:-|`. Sie können für die Erstellung der `String`-Repräsentation *Konkatenation* oder die Methode `String.format` verwenden.

JUnit-Test: keiner

`main`-Methode: Teil 1

Deklarieren Sie in Ihrer `main`-Methode folgende *drei Tiere* als *lokale Variablen* und *initialisieren* Sie diese mit `Pet`-Instanzen mit folgenden Eigenschaften:

Variablenname	Name	Tierart
rosco	"Rosco"	DOG
morgana	"Morgana"	CAT
rabbit	"Rabbit of Caerbannog"	RABBIT

Geben Sie dann die `String`-Repräsentation der drei `Pet`-Objekte auf dem Terminal aus. Deklarieren Sie eine Variable `roscoClone` und initialisieren Sie diese, indem Sie den *Kopier-Konstruktor* von `Pet` auf `rosco` aufrufen. Rufen Sie `roscoClone.equals(rosco)` auf und geben Sie das Ergebnis aus! Unterscheidet sich das Ergebnis vom Wert des *boole'schen Ausdrucks* `rosco == roscoClone` und warum? Was ist der Wert des boole'schen Ausdrucks, wenn Sie vorher die Anweisung `roscoClone = rosco` ausführen und warum?

Die Methoden `play`, `eat`, `sleep`

Die Klasse `Pet` enthält die drei Methoden `play`, `eat` und `sleep`, die die Werte `happiness`, `hungriness` und `sleepiness` je nach Aktion (z.B. „spielen") beeinflussen:

Methode	happinessChange	hungrinessChange	sleepinessChange
play	+2	+2	+2
eat	+1	−2	+1
sleep	−2	+2	−2

Die drei **`double`**-Attribute in `PetType` definieren dabei, wie stark die jeweilige Änderung ausfällt. Die neuen Attributwerte berechnen sich dann wie folgt (kein gültiger Java-Code):

```
happiness = happiness + happinessMultiplier * happinessChange
hungriness = hungriness + hungrinessMultiplier * hungrinessChange
sleepiness = sleepiness + sleepinessMultiplier * sleepinessChange
```

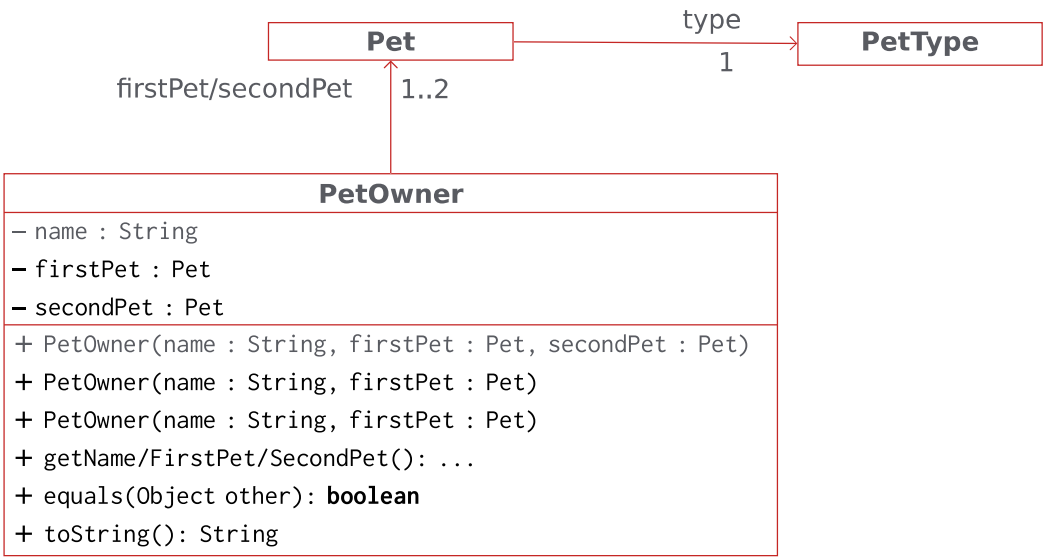
Die Ergebnisse auf der rechten Seite sollen dabei *gerundet* werden. Suchen Sie sich dazu eine passende Methode in der Klasse `Math` und achten Sie außerdem auf die korrekte Typkonversion.

Beispielrechnung: Starten wir mit einem Kaninchen mit `happiness=5`, `hungriness=3`, `sleepiness=1` und `happinessMultiplier=0.5`, `hungrinessMultiplier=0.5`, `sleepinessMultiplier=1` (s. `PetType`) und geben ihm was zu fressen (siehe Tabelle oben), so ergibt sich:

```
happiness = happiness + happinessMultiplier * happinessChange = 5 + 0.5 * 1 = 5.5
hungriness = hungriness + hungrinessMultiplier * hungrinessChange = 3 + 0.5 * (-2) = 2
sleepiness = sleepiness + sleepinessMultiplier * sleepinessChange = 1 + 1 * 1 = 2
```

Und damit gerundet `happiness=6`, `hungriness=2`, `sleepiness=2`. Implementieren Sie `play`, `eat` und `sleep`! Vermeiden Sie *Codeduplikation*, indem Sie **`private`**-Methoden verwenden. Testen Sie Ihre Implementierungen in der `main`-Methode.

JUnit-Tests: `testPlay/Eat/Sleep`



Die Klasse `PetOwner` modelliert den Besitzer von `Pet`s mit folgenden Attributen:

- `String name` — Name des Besitzers (*unveränderlich*, *nicht* `null`, *nicht leer*)
- `Pet firstPet` — erstes Tier (*unveränderlich*, *nicht* `null`)
- `Pet secondPet` — zweites Tier (*unveränderlich*, *darf* `null` *sein*)

Ein `PetOwner` kann also *ein* oder *zwei* Tiere besitzen.

Attribute und Konstruktoren von `PetOwner`

Deklarieren Sie die Klasse `PetOwner` mit den *drei Attributen* wie oben definiert! Achten Sie wieder auf die *Sichtbarkeit* und *weitere Modifizierer*. Implementieren Sie die ersten beiden *Konstruktoren*:

- `PetOwner(name : String, firstPet : Pet, secondPet : Pet)` — initialisiert alle drei Attribute. Prüfen Sie auf *gültige Parameterwerte*!
- `PetOwner(name : String, firstPet : Pet)` — ruft obigen Konstruktor mit `secondPet==null` auf (*Konstruktor-Verkettung*)

JUnit-Test: `PetOwner.testInitConstructor`

Kopier-Konstruktor

Statten Sie auch `PetOwner` mit einem *Kopier-Konstruktor* aus! Dabei sollen die Attribute `firstPet` und `secondPet` (wenn *nicht* `null`) *tief kopiert* werden. Verwenden Sie dabei den Kopier-Konstruktor von `Pet`.

JUnit-Test: `PetOwner.testCopyConstructor`

Getter

Implementieren Sie die *Getter* für die drei Attribute! Warum machen *Setter* für diese Klasse *keinen Sinn*?

Gleichheit über `equals`

Implementieren Sie die `equals`-Methode in `PetOwner`! Zwei `PetOwner` sind *gleich*, wenn alle ihre Attribute *gleich* sind. Dies gilt insbesondere für `firstPet` *und* `secondPet`.

JUnit-Test: `PetOwnerTest.testEquals`

String-Repräsentation mit `toString`

Implementieren Sie die Methode `String toString()`, die eine `String`-Repräsentation des Besitzers nach Folgendem Schema zurückgibt:

```
Jimmy
- First pet: Rosco :-) (DOG): Ha: 9, Hu: 7, Sl: 5
- Second pet: Rabbit of Caerbannog :-| (RABBIT): Ha: 6, Hu: 4, Sl: 3
Timmy
- First pet: Morgana :-| (CAT): Ha: 7, Hu: 9, Sl: 5
- Second pet: none
```

D.h. zuerst kommt der *Name* und dann in zwei Zeilen die beiden Tiere. Sollte der Besitzer nur *ein Tier* haben, wird `none` für das zweite Tier ausgegeben. Verwenden Sie für die Ausgabe der Tiere die Methode `Pet.toString`. Sie können für die Erstellung der `String`-Repräsentation *Konkatenation* oder die Methode `String.format` verwenden.

JUnit-Test: keiner

`main`-Methode: Teil 2

Deklarieren Sie zwei lokale Variablen vom Typ `PetOwner` mit folgenden Werten:

Variablenname	Name	firstPet	secondPet
jimmy	"Jimmy"	rosco	rabbit
timmy	"Timmy"	morgana	—

Führen Sie noch folgende Schritte hinzu:

- Geben Sie die `String`-Repräsentationen von `jimmy` und `timmy` aus.
- Erstellen Sie eine *tiefe Kopie* von `jimmy` mit dem Namen `jimmyClone` mit Hilfe des *Kopier-Konstruktors*.
- Geben Sie folgende *boole'schen Werte* aus und überlegen Sie sich, warum die Ergebnisse so sind:
  - `jimmyClone.equals(jimmy)`
  - `jimmyClone == jimmy`
  - `jimmyClone.getFirstPet().equals(jimmy.getFirstPet())`
  - `jimmyClone.getFirstPet()== jimmy.getFirstPet()`

Versorgen der Tiere mit `takeCareOfPets`

Was noch fehlt ist eine Methode, mit der sich ein `PetOwner` um seine *Tiere kümmert*: *Implementieren* Sie die *öffentliche* Methode `void takeCareOfPets()` in der sich ein `PetOwner` um *alle seine Tiere* nach folgenden Regeln kümmert:

- Gilt „Hunger $\geq$ Schläfrigkeit“ *und* „Hunger $\geq$ Traurigkeit“ (`getSadness`), dann bekommt das Tier was zu *fressen* (`eat`).
- Gilt obige Bedingung *nicht* und gilt „Schläfrigkeit $\geq$ Hunger“ *und* „Schläfrigkeit $\geq$ Traurigkeit“ (`getSadness`), dann darf das Tier *schlafen* (`sleep`).
- Gelten die beiden obigen Bedingungen nicht, so *spielt* der Besitzer mit dem Tier (`play`).

Vermeiden Sie *Codeduplikation*!

JUnit-Test: `PetOwnerTest.testTakeCareOfPets`

`main`-Methode: Teil 3

Vervollständigen Sie die `main`-Methode, mit einer `for`-Schleife über *zehn Iterationen* mit folgendem Inhalt:

- Rufen sie `takeCareOfPets` auf `jimmy` und `timmy` auf.
- Geben Sie die aktuelle Iteration aus (`Iteration 1`, etc.)
- Geben Sie die `String`-Repräsentationen von `jimmy` und `timmy` aus.