Aufgaben Termin 2: Klassen & Objekte

Level 1

Aufgabe 1

Definieren Sie eine Klasse `Pet`. Diese Klasse soll ein virtuelles Haustier (wie ein Tamagotchi) repräsentieren. Zur Repräsentation auf der `Stage` soll die Klasse eine `Actor`-Eigenschaft besitzen. Dabei kann es sich um das erste `Actor`-Objekt aus der letzten Aufgabe handeln (z.B. Kodee).

Definieren Sie in dieser Klasse zudem grundlegende Eigenschaften wie `name` (z.B. "Kodee"), `health` (z.B. 80) und `happiness` (z.B. 50) mit den passenden Datentypen.

Erstellen Sie eine weitere Klasse `Game` für Ihre Spiellogik, in der Sie die `Stage`, das `Pet` und später noch weitere Aktoren speichern. Die Klasse wird es später anderen Klassen ermöglichen, von dort auf die `Stage` und einzelne Aktoren zuzugreifen.

Aufgabe 2

Definieren Sie eine Klasse `Health` bestehend aus numerischen Werten für `energy` und `fitness`. Wählen Sie jeweils den passenden Datentyp.

Passen Sie zudem den Datentyp der Eigenschaft `health` in der Klasse `Pet` an, indem Sie die Gesundheit nun in einem `Health`-Objekt speichern.

Aufgabe 3

Definieren Sie eine Methode `lifeGoesOn()` für das `Pet`, in der die Energie mit einer zufälligen Wahrscheinlichkeit reduziert wird. Die Energieanzeige soll auf der `Stage` sichtbar sein. Hierzu können Sie einen weiteren `Actor` verwenden und den `text.content` setzen. Denken sie bei den folgenden Aufgaben daran, den Textinhalt zu aktualisieren, wenn sich der Energiewert ändert.

Aufgabe 4

Legen Sie im `init {}`- Block der Klasse `Pet` fest, wie oft die Methode `lifeGoesOn()` aufgerufen werden soll. Sie können über die Eigenschaft

```
animation.everyNsteps.reactionForTimePassed = {…}
```

festlegen, welcher Code nach einer bestimmten Zeit ausgeführt werden soll. Die Zeitspanne können Sie über die Eigenschaft

```
animation.everyNsteps.timeSpan = …
```

festlegen. Die `timeSpan` legt fest, wie viele Zeitschritte vergehen, bis der `reactionForTimePassed`-Code das nächste Mal ausgeführt wird. Jeder Zeitschritt dauert 25 ms. Eine Sekunde entspricht als 40 Zeitschritten.

Aufgabe 5

Erstellen Sie eine Klasse `Item` mit den Eigenschaften `name`, `category`, `amount`, `energyImpact` und `happinessImpact`. Diese Klasse soll einen Gegenstand repräsentieren, den das `Pet` besitzt. Erstellen Sie für den Typ der Eigenschaft `category` zusätzlich ein Enum-Klasse `ItemCategory` mit den Ausprägungen `FOOD`, `TOY`, `OTHER` hinzu. Fügen Sie in der Klasse `Pet` eine Eigenschaft `inventory` hinzu, die eine **Liste** von `Items` speichert.

Ergänzen Sie `Pet` außerdem um grundlegende Methoden zum Hinzufügen, Entfernen und Zählen von `Items`, so dass z.B. folgender Aufruf möglich wird:

```
val item :Item  = …
myPet.addItem(item)
```

Beim Hinzufügen eines `Item` soll sich die `happiness` des `Pet` um den in `happinessImpact` festgelegten Wert erhöhen: Wenn ein Pet ein Buch oder einen Ball besitzt, wird es glücklicher. Das `Item` soll zudem im `inventory` gespeichert werden. Wenn ein `Item` entfernt wird, reduziert sich die `happiness`.

Die veränderten `happiness`-Werte können Sie auf der Konsole ausgeben oder im `text.content` eines `Actors` anzeigen (Aktualisierung des Textinhalts bei Änderungen nicht vergessen!)

Aufgabe 6

Dem `Pet` sollen verschiedene Gegenstände im Inventar hinzugefügt werden.

Fügen Sie der `Stage` mehrere Snacks als `Actors` hinzu, wie in der letzten Aufgabe.

Wenn auf der `Stage` ein Snack angeklickt wird, soll dem `inventory` ein passendes `Item` hinzugefügt werden, das diesen Snack repräsentiert.

In gleicher Weise können Spielsachen ins `inventory` hinzugefügt werden, z.B. ein Ball oder Ufo. Sie können die Assets `Assets.misc.BALL` und `Assets.misc.UFO` nutzen.

Aufgabe 7

Definieren Sie für `Pet` eine weitere Methode `feed(item : Item)`, in der die Energy und die Happiness abhängig vom jeweiligen `item` verändert werden können.

Passen Sie den `addItem(…)`-Code so an, dass eingesammelte Items sofort mit `feed(…)` gegessen werden, wenn sie zur Kategorie `FOOD` gehören und ansonsten ins Inventar hinzugefügt werden. In beiden Fällen sollen sich `energy` und `happiness` verändern und die Textanzeige aktualisiert werden. Vermeiden Sie Code-Redundanz, indem Sie ggf. weitere Methoden schreiben.

Aufgabe 8

Erstellen Sie eine berechnete Eigenschaft `hungry` (Boolean) für das `Pet`, die wahr ist, wenn die Energie unter 20 liegt.

Aufgabe 9

Erstellen Sie eine weitere Klasse `Activity` mit den Eigenschaften `energyImpact`, `happinessImpact` und `description`.

Definieren Sie für die Klasse eine Methode `execute(…)` , die ein `Pet` entgegen nimmt. In `execute(…)` sollen für das entgegengenommen `Pet` die Eigenschaften verändert werden, abhängig von `engergyImpact` und `happinessImpact`. Eine Aktivität hat also ebenfalls Auswirkungen auf den Energielevel und das Glücksgefühl des Haustiers.

Damit eine Activity ausgeführt wird, ergänzen Sie die Klasse `Pet` um eine Methode `doActivity`, die eine `Activity` entgegennimmt. In `doActivity()` soll dann die `execute(…)`−Methode der übergebenen `Activity` aufgerufen werden.

Aufgabe 10

Auf der Stage soll es drei Buttons für verschiedene Aktivitäten geben (z.B. Kekse backen, Laufen, Fußball spielen).

Erstellen Sie für jede der Aktivitäten ein passendes `Activity`-Objekt. Beim Klick auf einen der Buttons soll das entsprechende `Activity`-Objekt an die `doActivity()`-Methode des Pets übergeben werden.

Aufgabe 11

Überlegen Sie sich anhand des bisher geschriebenen Codes, welche Sichtbarkeitsmodifikatoren an welchen Stellen sinnvoll sind und warum.

Level 2

Aufgabe 1

Implementieren Sie für die Klasse `Pet` die Eigenschaften `minutesAwake` und `hoursAwake`. `minutesAwake` soll ein Backing-Field haben, also den Datenwert speichern. `hoursAwake` soll in der `set`- und `get`-Methode auf `minutesAwake` zugreifen, damit der Wert der verstrichenen Zeit nicht doppelt gespeichert wird. Zusätzlich soll es beim Festlegen der Minuten den Seiteneffekt geben, dass die Energie alle 10 Minuten sinkt. Ergänzen Sie den `reactionForTimePassed`-Code, so dass im Spiel bei jeder Ausführung 1 Minute verstreicht (auch wenn dies tatsächlich nur ein paar ms sind).

Aufgabe 2

Ändern Sie den Code in `execute(…)` so ab, dass für die Aktivität "Fußball spielen" ein Ball im Inventar die Voraussetzung ist. Stellen Sie dazu in `Pet` eine Methode `hasItem` bereit. Die Methode erhält einen String als Beschreibung und prüft, ob es ein Item mit diesem Namen im `inventory` gibt.

Aufgabe 3

Ändern Sie den Code so ab, dass eine `Exception` geworfen wird, wenn ein `Item` verfüttert wird, das nicht zur Kategorie `FOOD` gehört. Wenn das benötigte `Item` für eine Aktivität nicht im Inventar vorhanden ist, soll ebenfalls eine `Exception` geworfen werden. Arbeiten Sie auch mit `try` und `catch`, um auf solche Ausnahme-Situationen zu reagieren.

Aufgabe 4

Passen Sie die Aktivitäten so an, dass es passende Effekte auf der `Stage` gibt.

Verzweigen Sie in der `execute(…)`-Methode von `Activity` über die verschiedenen Aktivitäten, um zu entscheiden, was auf der Bühne geschehen soll.

Beispiele für die Darstellung einer Aktivität:

- Bei „Backen“ erscheinen Kekse auf der Stage
- Bei „Laufen“ bewegt sich Kodee ein bisschen (z.B. 2 Hin- und Herlaufen)
- Bei „Fußball spielen“ bewegt sich der Ball vom Pet zu einer zufälligen Position

Tipp: Um aus einer `Activity` heraus auf die `Stage` zuzugreifen, müssen die `Stage` und ggf. weiter Aktoren entweder jeder `Activity` übergeben werden (als weitere Eigenschaftem) oder Sie statten die `Game`-Klasse mit einem **companion object** aus, in dem das `Stage`-Objekt und ggf. weitere Aktoren dauerhaft gespeichert und anderen Objekten zur Verfügung gestellt werdenn. Was sind die Vor- und Nachteile der beiden Lösungsansätze?

Aufgabe 5

Ergänzen sie `Pet` um eine private Eigenschaft `lastActivity`, die die zuletzt ausgeführte Aktivität speichert. Wenn die gleiche Aktivität zwei Mal hintereinander ausgeführt wird, soll sich die `happiness` um einen zufälligen Wert zwischen 10-30 reduzieren. Zufallswerte könne mit `Random.nextInt(maxValue)` generiert werden. Die Ausführung der Aktivität kann dabei unverändert die `happiness` wieder erhöhen.

Level 3

Implementieren Sie als weitere Aktivität das Starten eines Spiels auf der Stage.

Dazu müssen zusätzliche Actors auf der Bühne hinzugefügt werden, um das Spiel und die Spiellogik abzubilden. Das Pet wird immer glücklicher, wenn es gewinnt.

Denken Sie sich ein eigenes Spiel aus oder nehmen Sie einen dieser Vorschläge:

- Setzen Sie das Spiel Tic Tac Toe um.
- Entwickeln Sie ein Quiz-Spiel mit einer Frage (z.B. in einem Sprechblase-Actor) und vier Antwortmöglichkeiten (z.B. vier Button-Actors). Definieren Sie eine Klasse `QuizQuestion` mit einer Frage, einer richtigen Antwort und drei falschen Antworten. Erzeugen Sie mehrere `QuizQuestion`-Objekte und speichern Sie diese in einer Liste, aus der zufällig Fragen gewählt werden können.
- Eine einfache Umsetzung von Memory, wobei sich für das „Umdrehen“ der Karten das Bild eines Actors wechselt.
- Eine Umsetzung von Schere-Stein-Papier. Legen sie für Schere, Stein und Papier jeweils einen Actor als Icon fest. Wenn User eine Wahl treffen, entscheidet sich das `Pet` zeitgleich (per Zufallsgenerator) für eine der Möglichkeiten. Wer gewinnt, hängt von den Regeln ab: Schere schlägt Papier, Papier schlägt Stein, Stein schlägt Schere
- Lassen Sie Items "vom Himmel fallen". Erstellen Sie für mehrere Actors eine Animation, so dass ein Gegenstand von oben herabfällt. Das Pet kann sich zu diesen Gegenständen hinbewegen, z.B. durch Steuerung über die Tastatur oder durch Aktivierung des Draggings. Wenn das Pet dort steht, wo der Gegenstand ankommt, kann dieser eingesammelt werden. Dazu kann ein `Item`-Objekt an das Pet übergeben werden.

Ein `Actor` kann mit der Maus oder über die Tastatur bewegt werden.

Dragging kann so festgelegt werden:

```
yourActor.setDraggingOptions(true, DragRestriction.NONE);
```

Die Steuerung über die Tastatur kann so festgelegt werden:

```
yourActor.motion.keyMotionControl =
      KeyboardMotionControl.MOTION_BY_ARROWS_AUTOSTOP
```