Themen: Interfaces - Definition, Implementierung und Nutzung

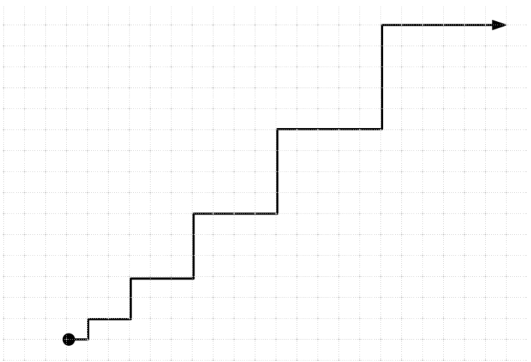
Aufgabe 11.1
Gegeben sei das folgende Interface

```
public interface Turtle extends Beweglich {

        void step(); // macht einen Schritt in die aktuelle Blickrichtiung
        void step(int n); // n Schritte in aktuelle Blickrichtung
        void turnL(); // Vierteldrehung nach links (gegen den Uhrzeigersinn), ohne
            die Position zu aendern
        void turnR(); // Vierteldrehung nach rechts (mit dem Uhrzeigersinn), ohne
            die Position zu aendern
        void setDirection(int r); // setzt die Blickrichtung auf Wert (0 ,1, 2,
            oder 3)
        int direction(); //liefert die aktuelle Blickrichtung (0 = oben, 1 = rechts,
             2 = unten, 3 = rechts)

}
```

Nehmen Sie an, die Klasse `TurtleRoboter` implementiere dieses Interface.
Die Klasse verfüge über einen parameterlosen Konstruktor, der einen Roboter mit Startposition $(0, 0)$ und Blickrichtung „oben“ erzeugt.

a) Definieren Sie ein Interface `TreppenTurtle`, das das Interface `Turtle` um eine Methode `void treppe(int n)` erweitert:

b) Die Methode `void treppe(int n)` ermöglicht den Objekten, treppenförmige Wege zu laufen (siehe unten). Implementieren Sie eine Klasse `TreppenRoboter`, die von `Roboter` abgeleitet ist und zusätzlich das Interface `TreppenTurtle` implementiert.
(Beachten Sie bei der Implementierung, dass die Treppe eine rechts-Bewegung mehr erfordert als nach-oben-Bewegungen.)
Beispiel:
Für den Wert $n = 5$ sollte die Treppe die unten gezeigte Gestalt haben.



Aufgabe 11.2
Spielen Sie Lotto!

Im Deutschen Fernsehen ist das Lotto „6 aus 49“ eine Institution.
Dabei markiert der Spieler auf seinem Tipp-Zettel sechs der Zahlen 1 bis 49. Die Lottofee zieht zufällig sechs Zahlen aus dem Bereich 1 bis 49. Bei hinreichend vielen Übereinstimmungen zwischen gezogenen und getippten Zahlen winkt ein Gewinn!

Definieren Sie eine Klasse LottoSpiel, mit der ein Lottospiel simuliert wird:

a) Die Klasse verfügt über (mindestens) die beiden Attribute ziehung und tipp. Beide sind vom Typ Menge! Wählen Sie eine geeignete Implementierung des Interfaces Menge.
Wenn Sie der gewählten Klasse weitere Methoden hinzufügen möchten, dann tun Sie das, indem Sie eine Unterklasse von der gewählten Implementierung ableiten und dort die zusätzlichen Methoden implementieren.

b) Sehen Sie in LottoSpiel (mindestens) folgende Methoden vor:

- ziehen(): Ziehung der Lottozahlen
  Rückgabe der Methode: eine **Menge** von 6 Zahlen zwischen 1 und 49, alle unterschiedlich
  Hierfür benötigen Sie einen Zufallszahlengenerator. Importieren Sie Java.util.Random und legen Sie ein Klassenattribut an
  ```
  static Random r = new Random();
  ```
  Mit dem Aufruf
  ```
  ... = r.nextInt(49) + 1;
  ```
  erhalten Sie eine zufällige int-Zahl zwischen 1 und 49 (einschließlich).
- Erzeugung des Tipps des Spielers
  Rückgabe der Methode: eine **Menge** von 6 Zahlen zwischen 1 und 49, alle unterschiedlich
  Sehen Sie hierfür mindestens die folgenden beiden Varianten vor:
  - tippFix(**int**[] arr): Erzeugung der Menge aus einem übergebenen Array, das in der Testklasse festgelegt werden kann
    (es gibt Leute, die jahrzehntelang jeden Samstag die gleichen Zahlen tippen)
  - tippZufall(): zufällige Erzeugung der Menge
    (es gibt Leute, die ihren Hund tippen lassen)
  - Wenn Sie möchten, auch noch eine dritte Variante
    tippEinlesen(): Erzeugung der Menge durch *Einlesen* der getippten Zahlen während des Programmablaufs. Hierzu benötigen Sie ein Scanner-Objekt (⇝ siehe Folien Kap13 „Exkurs“).
- bekanntGeben() der gezogenen Zahlen und des Tipps. Dabei werden die sechs gezogenen Lottozahlen bzw die sechs getippten Zahlen jeweils *in sortierter Reihenfolge* auf dem Monitor ausgegeben.
- auswerten() Ermitteln der Übereinstimmungen zwischen Ziehung und Tipp
  Rückgabe: die **Anzahl** der „Richtigen“

Aufgabe 11.3
Betrachten Sie noch einmal das Projekt „Mittelerde“ aus Übung 9.
(Eine sehr ausführliche Hilfestellung zur Lösung finden Sie in LEA im Ordner Kap09. Dank an unsere ehemalie Studentin und Kollegin Lena Riem!)

Ergänzen Sie das Projekt wie folgt:

- Wir wollen nun auch Gegenstände modellieren. Gegenstände haben eine Bezeichnung vom Typ String.
  Definieren Sie eine Klasse `Gegenstand` mit einem Konstruktor und einer get-Methode für die Bezeichnung.
- Manche Gegenstände (zB Waffen) sind tragbar. Tragbare Gegenstände haben ein Gewicht vom Typ double.
  Definieren Sie ein Interface `Tragbar`, in dem eine Methode `double gewicht()` definiert wird.
- Wesen können tragbare Dinge nehmen, wobei
  - Hobbits nur Dinge tragen können, deren Gewicht höchstens 20 (kg) beträgt.
  - Magier können beliebig schwere Dinge tragen.
  - Jedes Wesen kann nur ein Ding zur gleichen Zeit tragen.
- Fügen Sie eine neue Art von Wesen hinzu, nämlich Pferde.
- Manche Dinge sind „käuflich“ (vllt. sollten wir besser „kaufbar“ sagen). Kaufbare Dinge (zB Pferde, Waffen) haben einen Preis.
  Definieren Sie ein Interface `Kaufbar`, in dem eine Methode `double preis()` definiert wird.
- Fügen Sie Wesen einen „Geldbeutel“ hinzu.
- Fügen Sie Methoden hinzu, mit denen Wesen kaufbare Dinge kaufen können (sofern sie genügend Geld haben).
- Implementieren Sie eine Klasse `Waffe`, die von `Gegenstand` abgeleitet wird. Waffen sind tragbar und kaufbar.
- Zeichnen Sie das UML-Diagramm neu.
- Fügen Sie der Klasse `Wesen` ein Attribut `ding` vom Typ `Tragbar` und eine abstrakte Methode `void nehmen(Tragbar)` hinzu und eine (nicht-abstrakte) Methode `tragen()`, die eine Referenz auf das getragene Objekt liefert.
- Passen Sie die Klassen `Hobbit` und `Magier` entsprechend an, sodass die Methode `nehmen` dort implementiert wird.

Erweitern Sie die Aufgabenstellung nach eigener Lust und Laune, zB:

- Überlegen Sie, wie es implementiert werden kann, dass Wesen mehrere (wieviele? beliebig viele?) Gegenstände in ihrem Besitz haben können.
- Lassen Sie Wesen Gegenstände *anderen Wesen abkaufen* können.
- Pferde können Dinge oder Wesen tragen - sofern diese „tragbar“ sind. Hobbits und Magier sollten dann tragbar sein, andere Wesen sind nicht tragbar.
- . . . eigene weitere Ideen (?)