Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, R. Palenta, A. Reuss, S. Schulze Frielinghaus WS 2016/17

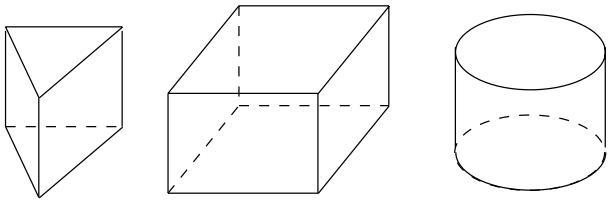
Übungsblatt 9 Abgabefrist: 16.01.2017 (vor 05:00 Uhr)

Σ Punkte: 40 | Σ Bonuspunkte: 0

Aufgabe 9.1 (P) Modellierung - Geometrische Figuren

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang und Aufgabentext).

Prismen oder *Zylinder* sind geometrische Körper, die durch Parallelverschiebung ihrer Grundfläche im Raum entstehen. Die folgende Abbildung zeigt als Beispiele für Prismen ein Dreiecksprisma, einen Quader und einen Zylinder, die durch Verschiebung eines Dreiecks, eines Rechtecks bzw. eines Kreises entstehen:



Prismen sind durch ihre Höhe und ihre jeweilige Grundfläche bestimmt. Als Grundflächen in Frage kommen (zum Beispiel):

- *regelmäßige n-Ecke*; bestimmt durch die Anzahl der Ecken und die Seitenlänge
- *Kreise*; bestimmt durch ihren Radius
- *Rechtecke*; bestimmt durch Breite und Länge

Eine Grundfläche muss die Berechnung Umfang und Flächeninhalt zur Verfügung stellen; ein Prisma die Berechnung von Oberfläche und Volumen.

In dieser Aufgabe soll eine Klassenhierarchie für diese geometrischen Körper in Java implementiert werden.

1. Erstellen Sie die Klasse `Grundflaeche.java`, als Oberklasse für alle Grundflächen. Legen sie Basisversionen der Methoden `umfang()` und `flaeche()`, sowie eine geeignete `toString()`-Methode an.
2. Erstellen Sie Klassen `Kreis.java`,`Rechteck.java` und `NEck.java` zur Repräsentation von Kreisen, Rechtecken und n-Ecken. Diese Klassen haben `Grundflaeche` als Oberklasse und erweitern die Oberklasse um die benötigten Objektvariablen und überschreiben die Methoden zur Flächen- und Umfang-berechnung. Ergänzen sie die `toString()`-Methode sinnvoll. Verwenden Sie nur Integer bei den Objektvariablen, um Ungenauigkeiten bei den arithmetischen Operationen zu vermeiden.
3. Erstellen sie die Klasse `Prisma.java` mit den benötigten Objektvariablen und den Methoden `volumen()` und `oberflaeche()`, sowie einer geeigneten `toString()`-Methode. Verwenden Sie wie zuvor bei Objektvariablen nur Integer für Größenangaben.
4. Ergänzen Sie alle Grundflächen um die Methode `istQuadrat()`, die zurückgibt, ob es sich bei der aktuellen Instanz um ein Quadrat handelt. Z.B. falls bei einem Rechteck Länge und Breite gleich sind.

5. Erstellen Sie die Klasse `Quadrat.java`, welche durch ihre Seitenlänge eindeutig bestimmt ist, als weitere Unterklasse von `Grundflaeche` und ergänzen sie alle Grundflächen um einem Methode `zuQuadrat()`, die, falls es sich um ein Quadrat (auch passende Rechtecke und n-Ecke) handelt, ein `Quadrat`-Object mit der entsprechenden Seitenlänge zurückgibt. Handelt es sich nicht um ein Quadrat, dann wird `null` zurückgegeben.
6. Ergänzen sie die Klasse `Prisma` um eine Methode `istWuerfel()`, die zurückgibt, ob es sich bei dem aktuellen Prisma um einen Würfel handelt.
7. Testen Sie Ihren Code.

Hinweise:

- Planen Sie vor dem Programmieren die Klassenhierarchie
- Fassen Sie gleichartige Attribute und Methoden in einer geeigneten Oberklasse zusammen.
- Verwenden sie Getter-Methoden.
- Greifen sie falls möglich auf bereits implementierte Methoden zurück.
- Die Fläche eines regelmäßigen n-Ecks mit Seitenlänge a ist $\frac{n*a^2}{4*\tan(\frac{\pi}{n})}$.
- Die Java-Klasse `Math` stellt in der Klassenvariablen `Math.PI` einen Wert für π sowie in der Klassenmethode `Math.tan()` die Berechnung des Tangens zur Verfügung. Diese Methode darf verwendet werden.
- Testen sie ihre Implementierung mit einer geeigneten Test-Klasse.

Aufgabe 9.2 (P) Polymorphie

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang und Aufgabentext).

Wiederholen Sie gemeinsam im Tutorium die in der Vorlesung besprochende Vorgehensweise zur Bestimmung der Methode, die bei einem Methodenaufruf ausgeführt wird. Besprechen Sie basierend auf dieser Vorgehensweise, welche Methoden bei den Aufrufen im Programm `ExampleOne.java` ausgeführt werden und überprüfen Sie Ihre Überlegungen *anschließend*.

Beachten Sie, dass es in Java im Allgemeinen schlechter Stil ist, mehrere Klassen in einer Datei zu haben (sogenannte nested classes). In `ExampleOne.java` wird das lediglich verwendet, um alle Klassen sofort überblicken und somit die Methodenaufrufe leichter bestimmen zu können.

Aufgabe 9.3 (P) LinkedList der Java-API

Machen Sie sich in der folgenden Aufgabe mit der Verwendung der einfach verketteten Liste der Java-Standardbibliothek vertraut.

Um die Methoden der Klasse `LinkedList` der Standard-API zu nutzen, müssen Sie nach der Paketdeklaration folgende Anweisungen einfügen:

```
import java.util.LinkedList;
```

Nun steht Ihnen eine *generische* Liste zur Verfügung. *Hinweis:* Das Thema *Generics* wird in der Vorlesung noch behandelt, hier geht es jedoch nicht darum, diese Thematik bereits zu verstehen, sondern lediglich um die Verwendung der `LinkedList` der Standard-API. Sie können via dem Ausdruck `new LinkedList<T>()` eine Liste erstellen, die nur Objekte vom Typ `T` enthält (sowie allen abgeleiteten Typen). Elemente `e` können Sie via der Methode `add(T e)` anhängen. Um eine Liste zu durchlaufen, also von dem ersten Element bis zum letzten Element, können Sie folgendes Grundgerüst verwenden:

```java
LinkedList<String> list = new LinkedList<String>();

// ... list.add("foo"); list.add("bar"); ...

for (String myString : list) {
  // ...
}
```

Der Variable `myString` wird in jedem Schleifendurchlauf ein Listenelement zugewiesen. Angefangen von dem ersten Element bis zum letzten. Innerhalb der Schleife können Sie dann via der Variable `myString` auf das jeweilige Listenelement zugreifen.

Erstellen Sie folgendes Programm:

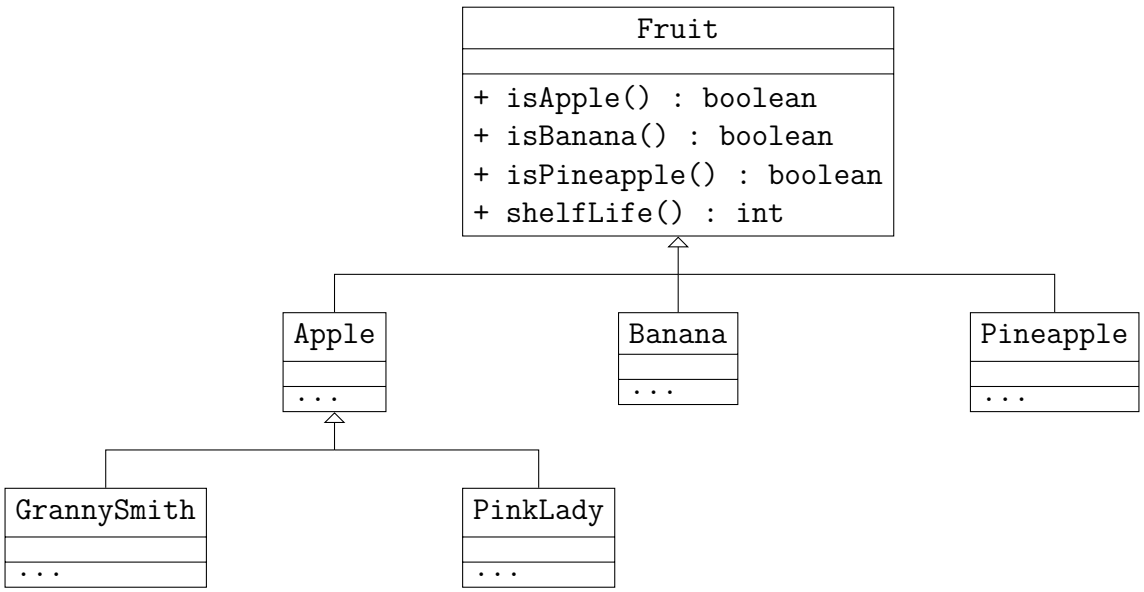
- Lesen Sie vom Benutzer eine ganze Zahl n ein.
- Erstellen Sie eine `LinkedList<String>` und speichern Sie in dieser alle ganzen Zahlen in aufsteigender Reihenfolge von 0 bis n (als `String`), falls $n \geq 0$. Ist n negativ bleibt die Liste leer.
- Geben Sie alle Elemente der Liste in der Reihenfolge, in der die Elemente in der Liste enthalten sind (beginnend mit dem ersten) aus, indem Sie mit der oben erläuterten `for`-Schleife durch die Liste iterieren.

Aufgabe 9.4 (H) Generische Früchte [10 Punkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang und Aufgabentext).

Alle Klassen müssen im Paket `pgdp` liegen.

Im Folgenden wollen wir ein Programm schreiben, welches einen Früchtekorb, bestehend aus Äpfeln, Bananen und Ananas, simuliert. Dafür benötigen wir eine Oberklasse `Fruit`, wovon die Klassen `Apple`, `Banana` und `Pineapple` erben sollen. Des Weiteren interessieren wir uns im Speziellen für die Äpfelsorten `Granny Smith` sowie `Pink Lady`, welche als Unterklassen der Klasse `Apple` implementiert werden sollen. D. h. Ihr Programm muss folgende Klassenhierarchie widerspiegeln:



In der Klasse `Fruit` sollen die Methoden `isApple`, `isBanana` sowie `isPineapple` jeweils `false` zurückliefern. Überschreiben Sie in den abgeleiteten Klassen die jeweilige Methode, sodass `true` zurückgeliefert wird, wenn es sich um das jeweilige Obst handelt. Achten Sie darauf, so wenig wie möglich aber so viel wie nötig an Methoden zu überschreiben.

Die Methode `shelfLife` gibt einen Wert zurück, der angibt, wie lange das jeweilige Obst haltbar ist (in Tagen). Dabei sind Früchte in Tagen wie folgt haltbar:

- Früchte ⇝ `-1`
- Äpfel ⇝ `30`
  - `Granny Smith` ⇝ `50`
  - `Pink Lady` ⇝ `20`
- Bananen ⇝ `7`
- Ananas ⇝ `20`

Beispiel: `new Apple().shelfLife()` liefert den Wert `30`.

Implementieren Sie nur die nötigsten Methoden und vermeiden Sie Redundanzen wann immer möglich.

Nun wollen wir den Früchtekorb wie folgt implementieren:

FruitBasket
- fruits : LinkedList<Fruit>
+ addFruit(f : Fruit) : void + getApples() : LinkedList<Apple> + getEqualOrLongerShelfLife(n : int) : LinkedList<Fruit>

Alle Früchte im Korb sollen in der privaten Liste `fruits` gespeichert werden. Da wir mittlerweile Profis im Listenprogrammieren sind, machen wir uns den Spaß und verwenden zum ersten Mal (nur in dieser Aufgabe) die einfach verkettete Liste aus der Java-Standardbibliothek (Sie dürfen alle verfügbaren `LinkedList`-Methoden der Standard-API verwenden). Fügen Sie an den Anfang, aber nach der Paketdeklaration in der Datei `FruitBasket.java` folgende Anweisungen ein:

```
import java.util.LinkedList;
```

Nun steht Ihnen eine *generische* Liste zur Verfügung. Sie können via dem Ausdruck `new LinkedList<T>()` eine Liste erstellen, die nur Objekte vom Typ `T` enthält (sowie allen abgeleiteten Typen). Elemente `e` können Sie via der Methode `add(T e)` anhängen. Um eine Liste zu durchlaufen, also von dem ersten Element bis zum letzten Element, können Sie folgendes Grundgerüst verwenden:

```
for (Fruit f : fruits) {
  // ...
}
```

Der Variable `f` wird in jedem Schleifendurchlauf ein Listenelement zugewiesen. Angefangen von dem ersten Element bis zum letzten. Innerhalb der Schleife können Sie dann via der Variable `f` auf das jeweilige Listenelement zugreifen.

Die Methode `addFruit(Fruit f)` soll das Obst `f` an die Liste `fruits` anhängen.

Die Methode `getApples()` liefert eine Liste aller Objekte vom Typen `Apple` zurück, die in der Liste `fruits` enthalten sind. Achten Sie auch auf Subtypen von `Apple`, diese sind schließlich auch vom Typen `Apple`!

Die Methode `getEqualOrLongerShelfLife(int n)` liefert eine Liste von Früchten zurück, die in der Liste `fruits` enthalten und mindestens `n` viele Tage haltbar sind.

Lösungsvorschlag 9.4

Korrekturbemerkung:

- Nur die nötigsten Methoden überschrieben. Zum Beispiel in der Klasse `Apple` wurde die Methode `isBanana` *nicht* überschrieben: 2 Punkte
- Die Methoden `isApple`, `isBanana` und `isPineapple` liefern in jeder Klasse den korrekten Wert: $6 \cdot 0.5 = 3$ Punkte
- Die Methode `shelfLife` ist in allen Klassen korrekt implementiert: 1 Punkt
- Methode `FruitBasket.getApples`: 2 Punkte
- Methode `FruitBasket.getEqualOrLongerShelfLife`: 2 Punkte

Hinweis: Die nächste Aufgabe beginnt auf der nächsten Seite.

Aufgabe 9.5 (H) Kampf der Geschlechter [30 Punkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang und Aufgabentext).

Achtung: Zu dieser Aufgabe wird ein Programmgerüst mitgeliefert. Dieses soll verwendet und inhaltlich ergänzt werden. Insbesondere dürfen die gegebenen Methodensignaturen und Rückgabetypen nicht geändert werden. Andernfalls gilt die Aufgabe als nicht bearbeitet.

Es soll ein Brettspiel für zwei Personen (im Folgenden `M` und `W` genannt) implementiert werden, bei dem es darum geht, Tiere auf dem Brett zu bewegen. Jeder Spieler hat die selbe Anzahl an Tieren einer bestimmten Art. Das Spiel wird auf einem Schachbrett gespielt (Felder `"a1"` bis `"h8"` vom Datentypen `String`). Um zu unterscheiden, zu welchem Spieler ein Tier jeweils gehört, treten Männchen (Spieler `M`) und Weibchen (Spielerin `W`) gegeneinander an.

Es folgt zunächst eine Beschreibung der Spielregeln. Anschließend folgen weitere Angaben zur Umsetzung des Spiels als Java-Programm.

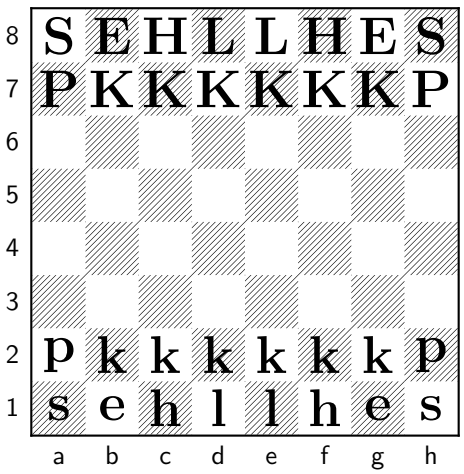
Jedes Tier ist entweder ein *Raubtier* oder ein *Vegetarier*. Die Raubtiere *fressen* die Vegetarier. Raubtiere *verhungern*, wenn sie eine bestimmte Zeit lang nicht gefressen haben. Es können nur Vegetarier gefressen werden. Die Art des Raubtiers oder Vegetariers ist dabei unerheblich. (D. h. ein Pinguin kann z. B. einen Elefanten verspeisen.) Die vorkommenden Tierarten sind:

- Vegetarier: Kaninchen, Elefant, Pferd
- Raubtiere: Leopard, Schlange, Pinguin

Die *Startaufstellung* ist ähnlich wie beim Schachspiel: Spielerin `W` hat

- sechs Kaninchen auf den Feldern `b2` bis `g2` (b2,c2,d2,e2,f2,g2)
- zwei Pinguine auf den Feldern `a2` und `h2`
- zwei Schlangen auf den Eckfeldern `a1` und `h1`
- zwei Elefanten auf den Feldern `b1` und `g1`
- zwei Pferde auf den Feldern `c1` und `f1`
- zwei Leoparden auf den Feldern `d1` und `e1`

Die Aufstellung für `M` entspricht der für `W`, aber gespiegelt an der Mittellinie des Bretts, d. h. die Tiere besetzen die Felder `a7` bis `h8`. Graphisch dargestellt (Kleinbuchstaben für Tiere von Spielerin W, Großbuchstaben für die von M) (L/l: Leopard(in), P/p: Pinguin(in), S/s: Schlange(rich), E/e: Elefant(in), K/k: Kaninchen(m/w), H/h: Pferd(engl. horse)):



Die Tiere bewegen sich nach einem weiter unten beschriebenen artspezifischen Muster fort, welches zunächst jeder Art für jedes Feld die Menge an prinzipiell möglichen Zielfeldern zuordnet, auf die gezogen werden kann. Um zu einem Zielfeld zu gelangen, müssen u. U. mehrere Zwischenfelder passiert werden. Beispiel: Eine Schlange kann vom Ausgangsfeld `a1` über die Zwischenfelder `b2` und `c1` zum Zielfeld `d2` gelangen.

Alle Zwischenfelder sind gleichzeitig auch mögliche Zielfelder. Die prinzipiell möglichen Züge der Tiere werden aufgrund von folgenden beiden Kriterien weiter eingeschränkt:

1. Auf dem Weg vom Ausgangsfeld zum Zielfeld (d. h. auf den Zwischenfeldern) dürfen keine anderen Tiere stehen.

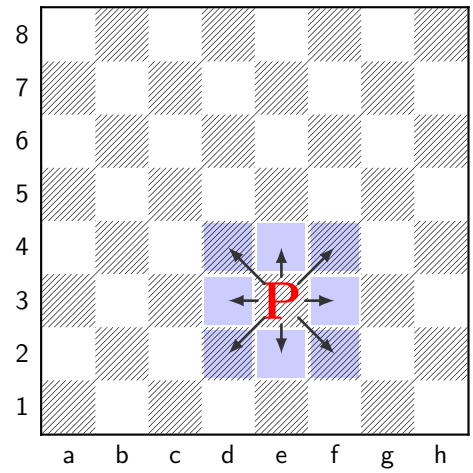
2. Auf ein besetztes Zielfeld kann gezogen werden, wenn es von einem gegnerischen Vegetarier besetzt ist und das ziehende Tier ein Raubtier ist. Das Raubtier frisst („schlägt“) dann das gegnerische Tier, welches vom Brett genommen wird und somit für den weiteren Spielverlauf nicht mehr zur Verfügung steht.

Im Folgenden werden die Zugmöglichkeiten der verschiedenen Arten beschrieben und für das Feld `e3` beispielhaft graphisch veranschaulicht.

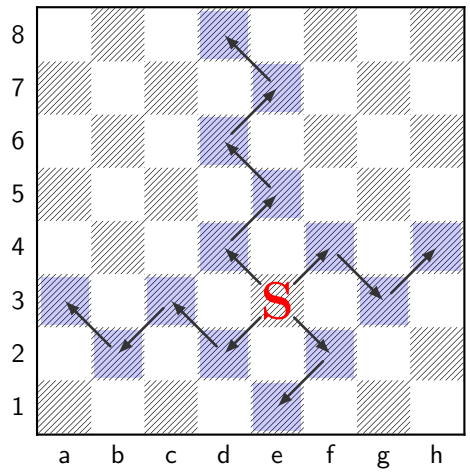
Pinguin: Bewegt sich genau ein Feld in gerader oder diagonaler Richtung.

Schlange: Bewegt sich im ersten Schritt (erstes Zwischenfeld) ein Feld diagonal. Danach bewegt sie sich diagonal weiter, wobei die Bewegungsrichtung zwischen links und rechts alterniert. Hierbei wird der Schritt auf das erste Zwischenfeld immer als Schritt nach links interpretiert. Der zweite Schritt geht also bzgl. der im ersten Schritt gemachten Bewegung nach rechts.

Mögliche Züge eines Pinguins (P) und einer Schlange (S) vom Feld e3 aus, sofern keine anderen Tiere im Weg stehen:



Zielfelder des Pinguins von e3 aus:
d4, e4, f4, d3, f3, d2, e2, f2

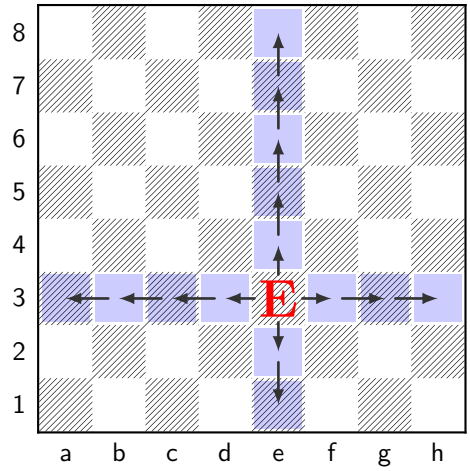


Wege der Schlange von e3 aus:
1. nach d8 via d4, e5, d6, e7
2. nach e1 via f2
3. nach h4 via f4, g3
4. nach a3 via d2, c3, b2

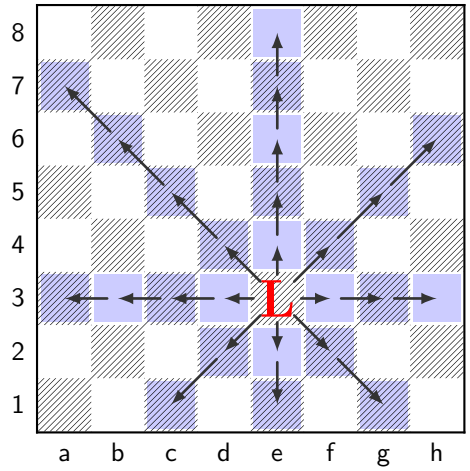
Elefant: Bewegt sich beliebig weit in gerader Richtung (entspricht einem Turm beim Schachspiel).

Leopard: Bewegt sich beliebig weit in gerader oder diagonaler Richtung (entspricht einer Dame beim Schachspiel).

Mögliche Züge eines Elefanten (E) und eines Leoparden (L) vom Feld e3 aus, sofern keine anderen Tiere im Weg stehen:



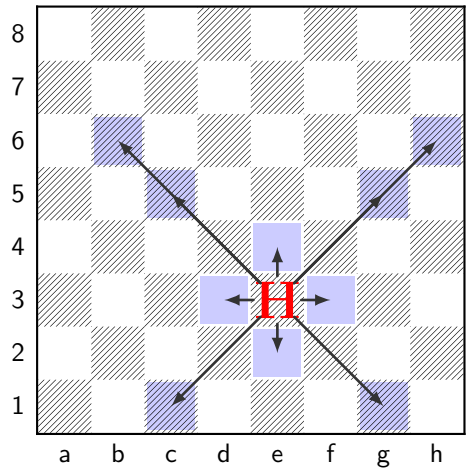
Wege des Elefanten von e3 aus:
1. nach e8 via e4, e5, e6, e7
2. nach e1 via e2
3. nach h3 via f3, g3
4. nach a3 via d3, c3, b3



Wege des Leoparden von e3 aus:
1. nach e8 via e4, e5, e6, e7
2. nach e1 via e2
3. nach h3 via f3, g3
4. nach a3 via d3, c3, b3
5. nach a7 via d4, c5, b6
6. nach h6 via f4, g5
7. nach c1 via d2
8. nach g1 via f2

Kaninchen: Das Kaninchen zieht genau gleich wie der Pinguin.

Pferd: Das Pferd darf geradeaus auf ein horizontal oder vertikal angrenzendes Feld ziehen. Außerdem darf es diagonal einmal **wahlweise zwei oder drei** Felder weit *springen*. Bei Zügen des Pferdes gibt es somit keine Zwischenfelder. Mögliche Züge eines Pferdes (H) vom Feld e3 aus:



Erreichbare Felder des Pferdes von e3 aus:
d3, f3, e2, e4, c1, g1, c5, b6, g5, h6
Keine Zwischenfelder!

Spielverlauf: Zu Beginn wird festgelegt, ob `W` oder `M` mit dem Ziehen beginnt. Der Spielverlauf gliedert sich in einzelne *Spielrunden*, in welchen zuerst die Person zieht, welche das Spiel begonnen hat, und danach die andere. Die Spielrunden werden so lange ausgeführt, bis das Spiel beendet ist. `W` und `M` ziehen in jeder Spielrunde jeweils bis zu vier eigene Tiere. Diese einzelnen Züge werden parallel ausgeführt, **nicht** nacheinander.

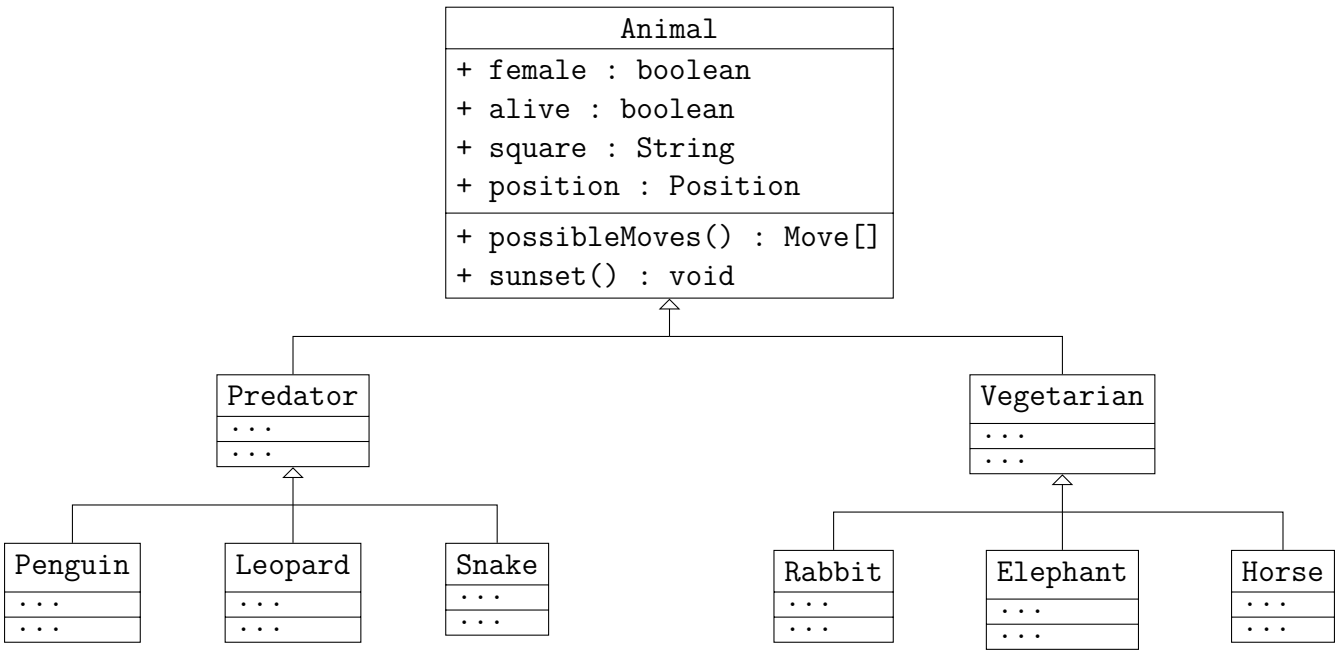
Es darf prinzipiell jeder Zug ausgewählt werden, der nach den oben beschriebenen Regeln in der Spielposition für das entsprechende Tier möglich ist. Jedoch müssen die Zielfelder der parallel ausgeführten Züge verschieden sein, d. h. zwei Tiere dürfen nicht auf das selbe Feld ziehen. Verschiedene Tiere dürfen aber beim parallelen Ausführen der Züge die selben Zwischenfelder benutzen, um zum Zielfeld zu gelangen. Jedes Zielfeld darf zugleich Zwischenfeld von parallel ausgeführten Zügen sein.

Es dürfen null bis drei Vegetarier und null bis ein Raubtier umgesetzt werden, d. h. man kann auch *passen*. Wird ein Tier vom Raubtier gefressen, wird es vom Spielbrett entfernt. Nachdem beide Spieler ihre Züge ausgeführt haben, werden die „Restlaufzeiten“ der Raubtiere aktualisiert und die verhungerten Tiere vom Spielbrett entfernt. Die Restlaufzeit bezeichnet die Anzahl Tage, welche das Raubtier noch ohne zu fressen auskommen kann. Sie wird als natürliche Zahl einschließlich der Null angegeben. Jede Spielrunde zählt als ein Tag und verringert die Restlaufzeit um 1. Ist der Wert 0 erreicht, muss das Raubtier in der folgenden Runde fressen, um im Spiel zu bleiben. Jede Tierart hat eine bestimmte Restlaufzeit. Frisst ein Raubtier in einer Spielrunde, so wird seine Restlaufzeit **sofort** auf den entsprechenden Wert zurückgesetzt. Alle Raubtiere sind zu Spielbeginn satt!

Spielende: Nach jeder Spielrunde wird ermittelt, ob das Spiel zu Ende ist. Sind alle Tiere vom Brett verschwunden, endet das Spiel unentschieden. Hat nur eine Seite keine Tiere mehr, verliert sie. Sind keine Raubtiere mehr übrig, so gewinnt, wer noch die meisten Vegetarier hat. Steht der Spielausgang während der letzten Spielrunde bereits fest, nachdem eine Seite gezogen hat, muss diese Spielrunde nicht mehr zu Ende gespielt werden.

Umsetzung:

Es ist folgende Klassenhierarchie mit englischen Bezeichnungen vorgegeben:



Programmgerüst: Verwenden und *ergänzen* Sie das beigefügte Programmgerüst. Beachten Sie dabei unbedingt die enthaltenen Kommentare! **Diese sind Teil der Aufgabenstellung** und damit bindend. Deklarationen und Signaturen dürfen *nicht geändert* werden. (D. h. z. B. darf eine Deklaration `private int x = 5;` nicht in `public int x;` geändert werden, sondern muss genau so stehenbleiben.) Es dürfen weitere Membervariablen, Methoden und Konstruktoren erstellt werden. Alle nötigen Klassen sind bereits vorhanden. Weitere Klassen dürfen Sie hinzufügen. Es sind u. a. folgende Elemente vorgegeben:

- Das Hauptprogramm befindet sich in der Klasse `Main`.
- Die Klasse `Game` ist für die Benutzerinteraktion gedacht. Sie enthält ein Attribut vom Typ `Position`, auf dem das Spiel gespielt werden kann.
- Objekte der Klasse `Position` repräsentieren eine Spielsituation.
- Die Klasse `IO` stellt Hilfsmethoden zum Einlesen von der Konsole bereit. Es ist somit nicht nötig, `MiniJava` zu verwenden. Stattdessen kann das Spielgeschehen auf der Konsole angezeigt und Eingaben von der Konsole eingelesen werden.
- Die Klasse `Globals` stellt String-Werte für die Ausgabe des Spielbretts auf der Konsole bereit. Ändern Sie den Wert von `Globals.ANSI` auf `false`, falls es zu Problemen mit der Darstellung der Unicode-Zeichen kommt. Abbildung 1 zeigt eine funktionierende Ausgabe der Unicode-Zeichen auf der Konsole.
- Die Klasse `Move` für Züge. Zur Eingabe soll die Form <Ausgangsfeld><Zielfeld> als String verwendet werden, also z. B. `"e2e4"` für einen Zug vom Feld `"e2"` zum Feld `"e4"`. Listen (z. B. von Zügen) werden generell als Arrays **ohne** leere Einträge (`null`) dargestellt. Für diese Aufgabe brauchen Sie also keine `Generics`.

  Hinweis: Sie dürfen die Methode `java.util.Arrays.copyOfRange` aus der Java-API verwenden, um ein Teilarray zu kopieren und so leere Einträge zu vermeiden.

Weitere Hinweise: Vermeiden Sie unnützen Code und Redundanzen nach Möglichkeit. Fügen Sie also beispielsweise nicht allen Vegetarier-Klassen das selbe Attribut hinzu, sondern deklarieren Sie es weiter oben in der Klassenhierarchie (z. B. in der Klasse `Vegetarian`). Die konkrete Art der Umsetzung fließt auch in die Bewertung ein. (Es wird also nicht nur Funktionalität bewertet.)

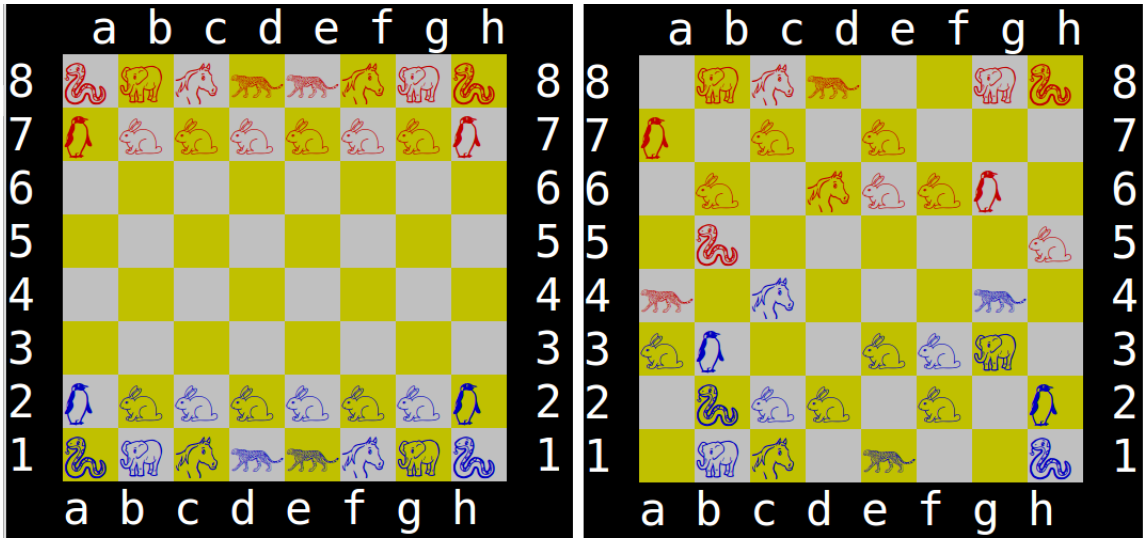


Abbildung 1: Startaufstellung (links) und Spielsituation in einer Linux-Konsole

Lösungsvorschlag 9.5

siehe Code-Ergänzungen im Programmgerüst

Korrekturbemerkungen:

f getestete Funktionalität (reset(), Züge, Ausgang)
e erwartetes Design, Abzug bei Fehlen
z Zusatz-Funktionalität (insbesondere Benutzerinteraktionen)

1. e super-Aufrufe (-1.0)
2. e Vegetarian.sunset() (-1.0)
3. Predator:
   (a) e daysToLiveWithoutPrey (-0.5)
   (b) e eatPrey() (-0.5)
   (c) e daysBeforeDeath() bzw. daysWithoutPreyLeft (-0.5)
   (d) e Konstruktor (-1.0)
   (e) e sunset() (-2.0)
4. e AnimalXY.Konstruktor (-0.5 jeweils)
5. Move (gesamt 2):
   (a) e beide Konstruktoren (-1.0)
   (b) 0.5 z equals(Object other)
   (c) 1.5 z / f toString()
6. Position (gesamt 12):
   (a) 3.0 f reset(...)
   (b) 6.0 f applyMoves(...)
   (c) 3.0 f theWinner()
7. Game (gesamt 10):
   (a) 1.0 z Anzeige: Brett und wer am Zug ist
   (b) 1.0 z Anzeige: mögliche Züge
   (c) 1.0 z Anzeige: Restlaufzeiten
   (d) 1.0 z Anzeige: Gewinner/Unentschieden
   (e) 0.5 z Check: ≤ 1 Raubtier
   (f) 0.5 z Check: ≤ 3 Vegetarier
   (g) 1.0 z Check: Zielfelderkonflikt
   (h) 1.0 z Check: immer nur ein Zug pro Tier
   (i) 1.0 z Einlesen von Zügen
   (j) 1.0 z Ungültige Eingabe wiederholen
   (k) 1.0 z Züge ausführen (lassen)
8. f AnimalXY.possibleMoves():
   (a) 6.0 f korrekte Züge – Punktabzug (-0.5) pro *Fehlerkategorie*: Richtung des Zuges (nach unten, nach oben, nach links oben usw.), Tierart, ggf. Schlagzug. Runter bis auf 0 Punkte.
   (b) e Bei leeren (null) Einträgen -3 Punkte
9. Zusätzlicher Abzug bei:
   (a) Veränderung der Spielfeldausgabe (außer Wert von Globals.ANSI) -3 Punkte
   (b) Falls sich das Spiel nicht allein mit reset, applyMoves, theWinner spielen lässt: -4 Punkte (mindestens aber 0 Punkte für alle drei Methoden zusammen, d. h. ggf. 0 Punkte, falls reset, applyMoves, theWinner ≤ 4 Punkte ergeben würden)
   (c) Code-Redundanz: (bis zu) -2 Punkte

Erlaubte Java-Methoden

MiniJava:

```
public static int readInt()
public static int readInt(String s)
public static int read()
public static int read(String s)
public static String readString()
public static String readString(String s)
public static void write(String output)
public static void write(int output)
public static int drawCard()
public static int dice()
```

returns an integer from the interval $[2, 11]$

returns an integer from the interval $[1, 6]$

Object bzw. allen anderen Klassen/Interfaces usw.:

```
public boolean equals(Object obj)
public final Class getClass()
public final void notify()
public final void notifyAll()
public final void wait() throws InterruptedException
```

String:

```
public char charAt(int index)
public boolean isEmpty()
public int length()
```

System:

`System.out.print(x)` prints the object `x` to the console

`System.out.println(x)` prints the object `x` to the console and terminates the line

Thread:

```
public void interrupt()
public final void join() throws InterruptedException
public void start()
public void run()
```

Selbstgeschriebene Methoden, die selber nur erlaubte Methoden verwenden, sind erlaubt, sofern dies nicht explizit in der Aufgabenstellung verboten wurde. Methoden, die im gegebenen Programmgerüst definiert wurden, dürfen verwendet werden.