

Angabe zur Klausur

Objektorientierte Programmierung

15. September 2025

Schwarzer Peter ist ein einfaches Kartenspiel für mindestens zwei Spieler, bei dem kein Gewinner, sondern ein Verlierer ermittelt wird. Ursprünglich war es ein Trinkspiel, bei dem der Verlierer die nächste Runde zahlen musste, inzwischen ist es aber besser bekannt als Kartenspiel für Kinder. Oft wird es mit einem speziell dafür gefertigten Blatt gespielt, in dem die Spielkarten Tiere abbilden, die mit einer Ausnahme, dem „Schwarzen Peter“ (siehe rechts), alle als Männchen (♂) und Weibchen (♀) vorkommen. Jeweils zwei Tiere derselben Art, aber unterschiedlichem Geschlecht bilden ein *Paar*.



Zu Anfang des Spieles werden alle Karten gemischt und möglichst gleichmäßig an alle Spieler verteilt. Bevor die Spieler reihum zum Zug kommen, darf jeder Spieler alle Paare ablegen, die er auf der Hand hat. Anschließend beginnt ein zuvor vereinbarter Spieler das Spiel. Dazu zieht er irgendeine Karte aus dem Blatt seines linken Nachbarn. Wenn die gezogene Karte mit einer aus seinem eigenen Blatt ein Paar bildet, darf er beide Karten ablegen.

Andernfalls muss er die gezogene Karte zu seinem Blatt hinzufügen. Das gilt insbesondere, wenn er den Schwarzen Peter gezogen hat, der zu keinem Paar gehört. Dann ist sein linker Nachbar an der Reihe; dieser zieht wiederum bei seinem linken Nachbarn irgendeine Karte usw. Sobald ein Spieler keine Karten mehr auf der Hand hat, scheidet er aus dem Spiel aus, wodurch seine beiden Nachbarn zu (neuen) Nachbarn werden. Wenn nur noch ein Spieler übrig ist, endet das Spiel. Dieser Spieler, der damit den Schwarzen Peter auf der Hand hat, ist der Verlierer.

Im Folgenden sollen Sie *Schwarzer Peter* als Kartenspiel in Java realisieren. Abb. 1 zeigt das grobe Klassendiagramm dieser Realisierung ohne Angabe von Attributen und Methoden. Der darin vorkommende Aufzählungstyp *Animal* ist wie folgt vorgegeben:

```
public enum Animal {  
    Dog, Hare, Hedgehog, Stork, Chicken, Goose, Duck, Pigeon,  
    Starling, Finch, Sparrow, Hamster, Mole, Mouse, Frog  
}
```

Für die Bearbeitung der nachstehenden Aufgaben gelten grundsätzlich die folgenden Hinweise:

- Schreiben Sie Code grundsätzlich in der Programmiersprache Java.
- Eventuell nötige Package-Vereinbarungen und Import-Anweisungen dürfen Sie weglassen; sie werden nicht bewertet. Gehen Sie aber davon aus, dass sich alle Klassen etc. von Abb. 1 im Package `game` befinden und man sie auch außerhalb von `game` benutzen können soll.
- Realisieren Sie alle Attribute und lokale Variablen als `final`, wenn das möglich ist.
- Vorgegebenen Code können Sie wie angegeben verwenden und müssen ihn dazu nicht erneut hinschreiben.

Aufgabe 1 (Aufzählungstyp *Sex*): 4 Punkte

Realisieren Sie den in Abb. 1 genutzten Aufzählungstyp *Sex*; er soll über die beiden Werte *Male* und *Female* verfügen. Sorgen Sie außerdem dafür, dass die Anweisung

```
System.out.println("männlich: " + Sex.Male + ", weiblich: " + Sex.Female);
```

die Ausgabe von

```
männlich: ♂, weiblich: ♀
```

auf der Konsole zur Folge hat. Die Zeichen ♂ und ♀ dürfen Sie einfach so hinschreiben, da sie in Unicode vorkommen.

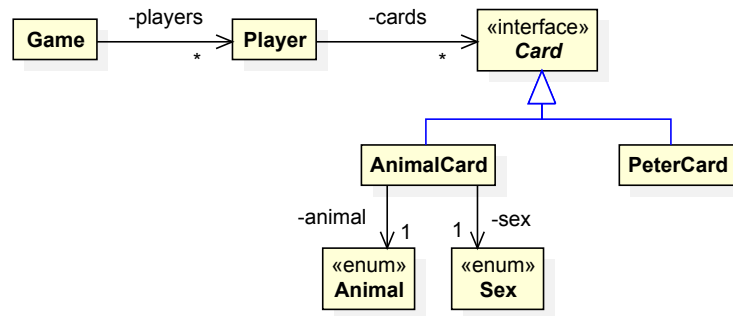


Abbildung 1: Grobes Klassendiagramm der Spielrealisierung.

Aufgabe 2 (Spielkarten): 3+19 = 22 Punkte

Die Spielkarten werden mit dem Interface `Card` und den beiden Klassen `AnimalCard` und `PeterCard` realisiert (siehe Abb. 1). `PeterCard` soll im Folgenden als *Singleton* realisiert werden. Seine einzige Instanz `PeterCard.INSTANCE` steht für die Karte des Schwarzen Peters. `AnimalCard` soll dagegen nicht als *Singleton* realisiert werden. Vielmehr soll jede Instanz von `AnimalCard` eine Spielkarte mit einem Tier des Aufzählungstyps `Animal` sein. Sein Geschlecht wird mit dem Aufzählungstyp `Sex` repräsentiert. So soll `new AnimalCard(Animal.Dog, Sex.Male)` eine Karte mit einem männlichen Hund darauf erzeugen.

Das Interface `Card` ist wie folgt definiert:

```

public interface Card {
    boolean formsPairWith(Card other);           /* Methode 1 */
    boolean formsPairWith(AnimalCard other);     /* Methode 2 */
}

```

Da `PeterCard` und `AnimalCard` dieses Interface implementieren, müssen beide Klassen diese Methoden bereitstellen. Sie sollen **true** oder **false** zurückgeben, je nachdem ob die beiden Karten `k1` und `k2` in einem Aufruf `k1.formsPairWith(k2)` ein Paar bilden, d. h. ob `k1` und `k2` zwei Tierkarten derselben Tierart, aber mit verschiedenem Geschlecht sind.

Beispiel:

```

final Card c1 = new AnimalCard(Animal.Dog, Sex.Male);
final AnimalCard c2 = new AnimalCard(Animal.Dog, Sex.Female);
final PeterCard c3 = PeterCard.INSTANCE;
final boolean b1 = c1.formsPairWith(c2); /* Fall 1, b1 = true */
final boolean b2 = c1.formsPairWith(c3); /* Fall 2, b2 = false */
final boolean b3 = c2.formsPairWith(c1); /* Fall 3, b3 = true */
final boolean b4 = c2.formsPairWith(c3); /* Fall 4, b4 = false */
final boolean b5 = c3.formsPairWith(c1); /* Fall 5, b5 = false */
final boolean b6 = c3.formsPairWith(c2); /* Fall 6, b6 = false */

```

Da `c1` und `c2` ein Hundepaar bilden, müssen die Werte von `b1` und `b3` beide **true** sein, in den anderen Fällen muss der jeweilige Aufruf von `formsPairWith` aber **false** liefern, weil der Schwarze Peter mit keiner Karte ein Paar bildet.

- Geben Sie für jeden der sechs Fälle (*Fall 1*, ..., *Fall 6*) an, welche der beiden Varianten von `formsPairWith` (*Methode 1* oder *Methode 2*) aufgerufen wird und in welcher Klasse die Methode definiert ist.
- Schreiben Sie die Klassen `PeterCard` und `AnimalCard` mit allen dafür notwendigen Attributen, Konstruktoren und Methoden, damit u. a. obiges Beispiel wie angegeben funktioniert. Außerdem soll der folgende Code auf der Konsole `[Peter, Dog ♀, Dog ♂]` ausgeben (oder in einer anderen Reihenfolge, weil Hashtabellen in dieser Beziehung nichtdeterministisch sind):

```

final Set<Card> set = new HashSet<>();
set.add(new AnimalCard(Animal.Dog, Sex.Male));
set.add(new AnimalCard(Animal.Dog, Sex.Male));
set.add(new AnimalCard(Animal.Dog, Sex.Female));
set.add(new AnimalCard(Animal.Dog, Sex.Female));
set.add(PeterCard.INSTANCE);
set.add(PeterCard.INSTANCE);
System.out.println(set);

```

Hinweise:

- Denken Sie daran, dass `PeterCard` als *Singleton* realisiert werden soll, dass es also nur exakt eine Instanz geben darf.
- `PeterCard` und `AnimalCard` sollen als Klassen, nicht als Records definiert werden.
- Stellen Sie sicher, dass Tierart und Geschlecht bei der Instanziierung von `AnimalCard` von **null** verschieden sind. Andernfalls soll eine geeignete Exception geworfen werden.
- Achten Sie bei der Realisierung der `formsPairWith`-Methode darauf, ausschließlich Konzepte der Objektorientierung zu verwenden. **Die Verwendung von `instanceof` oder Nutzung von Class-Objekten ist bei der Realisierung dieser Methoden nicht erlaubt und wird als falsch bewertet.**

Aufgabe 3 (Spieler): 11 Punkte

Realisieren Sie die Klasse `Player` zur Repräsentation der einzelnen Mitspieler. Es sollen die folgenden öffentlichen Konstruktoren und Methoden zur Verfügung stehen:

- `Player(String name)`: Erzeugt eine neue `Player`-Instanz mit dem angegebenen Namen des Mitspielers, aber mit einer leeren Kartenliste.
- `getCards()`: Liefert die Kartenliste dieser `Player`-Instanz zurück, d. h. die Liste aller Karten, die der Mitspieler auf der Hand hat. In welcher Reihenfolge Sie die Karten in der Liste verwalten, ist Ihnen überlassen.
- `addCard(Card card)`: Fügt die als Parameter `card` übergebene Karte zur Kartenliste dieser `Player`-Instanz hinzu. Wenn die Liste dadurch nun ein Paar bestehend aus `card` und einer anderen Karte enthält, werden beide Karten dieses Paares sofort von der Liste entfernt. Damit wird die Regel umgesetzt, dass man Kartenpaare ablegen darf.
- `removeCard(int index)`: Entfernt die Karte mit dem angegebenen Index aus der Kartenliste dieser `Player`-Instanz und gibt sie als Rückgabewert zurück. Der Index muss eine nichtnegative Zahl sein und kleiner als die Länge der Kartenliste. Wenn das nicht der Fall ist, soll eine geeignete Exception geworfen werden.

Diese Methode setzt damit die Funktionalität um, wenn der rechte Mitspieler eine Karte aus der Kartenliste dieser `Player`-Instanz zieht. Dazu wird der rechte Mitspieler einen erlaubten Index zufällig wählen.

- `toString()`: Gibt den Namen des Mitspielers zurück.

Aufgabe 4 (Testen): 7 Punkte

Schreiben Sie eine Test-Klasse unter Nutzung von **JUnit 4** mit genau einem Testfall, der die Funktionalität der `addCard`-Methode der Klasse `Player` testet. Ihr Testfall soll eine Instanz von `Player` erstellen und nacheinander den Schwarzen Peter, einen weiblichen Hund und zuletzt einen männlichen Hund als Karten hinzufügen. Nach jedem Hinzufügen soll Ihr Testfall prüfen, dass sich die richtigen Karten auf der Hand befinden.

Aufgabe 5 (Game): 16 Punkte

Realisieren Sie die Klasse `Game`. Jede Instanz soll ein Spiel *Schwarzer Peter* repräsentieren. Ein Konstruktor muss nicht geschrieben werden, jede Instanz von `Game` soll ein noch leeres Spiel ohne Mitspieler oder Spielkarten repräsentieren. Die Klasse soll die unten beschriebenen Methoden bereitstellen. Ein Spiel soll wie folgt ablaufen: Nachdem man ein `Game`-Objekt erstellt hat, fügt man mit `addPlayer` sukzessive die Mitspieler zum Spiel hinzu. Dann werden mit `dealCards` die Karten an die Mitspieler verteilt. Das eigentliche Spiel besteht dann aus einer Folge von `draw`-Aufrufen, bis der Verlierer des Spiels feststeht.

Realisieren Sie in `Game` die folgenden öffentlichen Methoden. Sie haben keinen Rückgabewert, sofern nichts anderes angegeben ist:

- `addPlayer(Player player)`: Fügt den angegebenen Mitspieler zum Spiel hinzu. Der als erstes hinzugefügte Spieler ist automatisch der Startspieler im Spiel, und die Mitspieler werden im Uhrzeigersinn hinzugefügt, d. h. der als zweites hinzugefügte Mitspieler sitzt links vom ersten, der als drittes hinzugefügte links vom zweiten und so weiter. Der Startspieler sitzt somit links vom zuletzt hinzugefügten Mitspieler.

Wurde zuvor bereits ein Spieler mit dem gleichen Namen zum Spiel hinzugefügt, soll eine geeignete Exception geworfen werden.

- `dealCards(List<Card> cards)`: Teilt die Karten der übergebenen Liste reihum im Uhrzeigersinn an die Mitspieler aus, beginnend mit dem Startspieler, d. h. der Startspieler erhält die erste Karte der Liste, der links vom ihm sitzende Spieler die zweite und so weiter. Das wird fortgesetzt, bis alle Karten in der Liste an die Mitspieler verteilt sind.

Die Methode soll eine geeignete Exception werfen, wenn sie aufgerufen wird, aber noch keine Mitspieler zum Spiel hinzugefügt wurden.

Beachten Sie, dass ein oder mehrere Spieler ausschließlich Paare ausgeteilt bekommen können. Diese Spieler scheiden sofort aus dem Spiel aus, da sie alle ihre Karten ablegen können.

- `getCurrentPlayer()`: Diese Methode gibt den Mitspieler zurück, der gerade an der Reihe ist. Anfangs ist das der Startspieler, durch Aufruf der Methode `draw` ändert sich das im Verlauf des Spiels aber laufend.
- `draw(int index)`: Lässt den Spieler, der gerade an der Reihe ist, bei seinem linken Nachbarn eine Karte ziehen. Der Parameter gibt an, welche der ihm verdeckt angebotenen Karten er dabei zieht (siehe Methode `removeCard` der Klasse `Player`) und zu seinen eigenen Karten hinzufügt. Wenn dabei ein Paar entsteht, darf er es natürlich ablegen (siehe Methode `addCard` der Klasse `Player`). Anschließend ist der nächste Spieler an der Reihe.

Beachte, dass es passieren kann, dass einer der beiden Spieler oder auch beide anschließend keine Karten mehr auf ihrer Hand haben, ab dann also nicht mehr am Spiel teilnehmen. Nach diesem Spielzug ist auf jeden Fall der erste Spieler links von dem, der gerade gezogen hat, als Nächstes an der Reihe, der noch Karten auf der Hand hat.

Diese Methode soll eine geeignete Exception werfen, wenn das Spiel zu Ende ist und daher keiner mehr ziehen muss oder darf, oder wenn der als Parameter übergebene Index außerhalb des erlaubten Bereichs liegt (siehe Methode `removeCard` der Klasse `Player`).

Der Rückgabewert dieser Methode soll die gezogene Karte sein.

- `getLoser()`: Die Methode gibt den Verlierer des Spiels zurück bzw. `null`, wenn er noch nicht feststeht,

Sie können weitere (auch nicht-öffentliche) Methoden schreiben, um sie in den oben beschriebenen Methoden aufzurufen. Denken Sie auch an Attribute, die Sie gegebenenfalls definieren müssen. Beachten Sie dazu insbesondere Abb. 1.