Programmieren II (Java)

5. Praktikum: Collections und Iteratoren

Sommersemester 2022
Christopher Auer, Tobias Lehner

Abgabetermine

Lernziele

- `Iterator` und `Iterable`
- *Java-Collections*: erstellen, befüllen, bearbeiten und abfragen
- `Comparator` und `Comparable`

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*
  - Beachten Sie dazu die Hinweise auf der Moodle-Kursseite

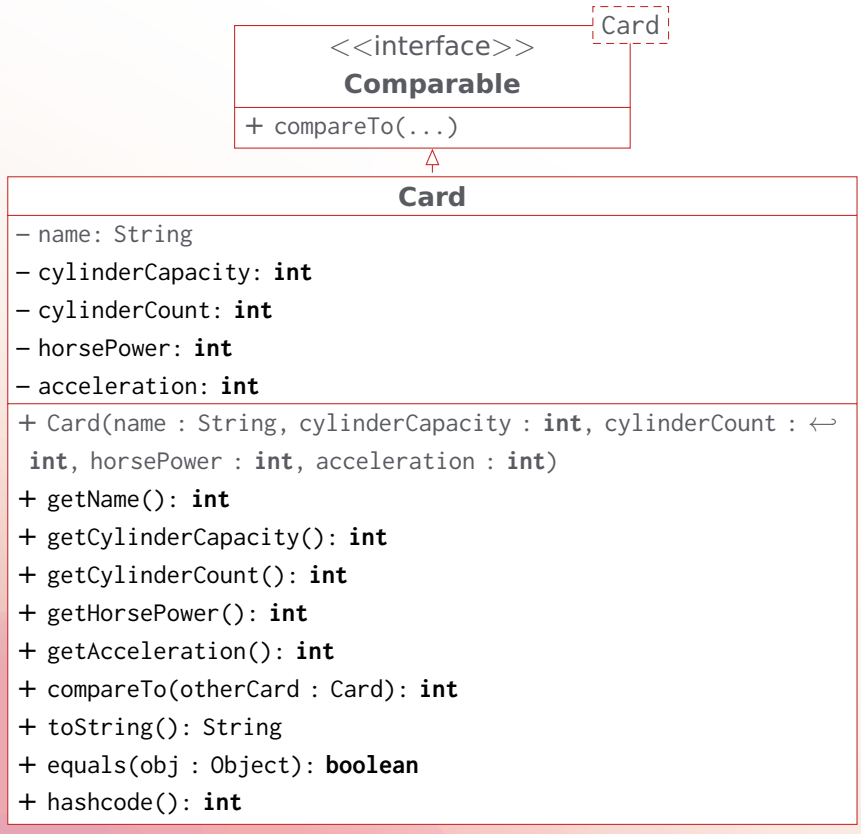
Aufgabe 1: Quartett ★☆ bis ★★

In dieser Aufgabe soll ein Quartett-Spiel programmiert werden, das so ähnlich funktioniert, wie früher auf dem Pausenhof. Der größte Unterschied ist, dass Sie als einziger menschlicher Spieler gegen Computer-Spieler antreten. Als solcher haben Sie auch einen kleinen Vorteil: Solange Sie im Spiel sind, dürfen Sie bestimmen, welche Eigenschaft des Quartetts in der aktuellen Runde verglichen wird.

- Für diese Aufgabe werden die JUnit-Tests sowie eine Klasse `QuartetMain` mitgeliefert.
- `QuartetMain` erzeugt einen Kartenstapel sowie auch Spieler und startet das Spiel. Wenn Sie mit Ihren Aufgaben fertig sind, sollte `QuartetMain` funktionieren.
- Mit den JUnit-Tests können Sie Ihre Klassen überprüfen. Bitte führen Sie keine Änderungen an den JUnit-Test durch.
- In der Aufgabe finden Sie UML-Diagramme und die zugehörigen Beschreibungen. Die Signaturen der beschriebenen Methoden stellen die für die JUnit-Tests verwendeten Schnittstellen dar. *Selbstverständlich steht es Ihnen frei weitere ergänzende Methoden oder Hilfsmethoden zu programmieren.*
- Erzeugen Sie, wie immer, bei einem *ungültigen Parameter* eine `IllegalArgumentException`.
- Die Aufgabe sollte innerhalb des **package** `quartet` programmiert werden.

Karte `Card`

Ein Objekt der Klasse Card repräsentiert eine Karte des Quartettspiels. Zur Vereinfachung wird von einem Auto-Quartettspiel ausgegangen, weswegen die Attribute, anhand derer die Karten verglichen werden, fest implementiert werden. Folgendes Klassendiagramm ist zur Klasse `Card` gegeben:



- `Card` wird durch die Objektattribute `name` (der Name des Autos), `cylinderCapacity` (der Hubraum des Autos), `cylinderCount` (die Anzahl der Zylinder des Autos), `horsePower` (die Leistung des Autos in Pferdestärken) sowie `acceleration` (Beschleunigung von 0 - 100 km/h in ganzen Sekunden) bestimmt. Definieren Sie die Attribute als Konstanten und implementieren Sie die entsprechenden Getter-Methoden.
- Die Attribute werde vom Konstruktor initialisiert. Reagieren Sie mit Exceptions, wenn
  - `name` `NULL` oder leer ist.
  - `cylinderCapacity`, `cylinderCount`, `horsePower` oder `acceleration` kleiner oder gleich 0 sind.
- Definieren Sie eine *natürliche Ordnung* für die Klasse `Card` indem Sie das Interface `Comparable` implementieren. Vergleichen Sie zwei Instanzen von `Card` anhand
  1. der *Beschleunigung* (umso kleiner umso besser)
  2. der *Pferdestärken* (umso größer umso besser)
  3. des *Hubraums* (umso größer umso besser)
  4. der *Zylinderanzahl* (umso größer umso besser)
  5. des *Namens* (in alphabetischer Reihenfolge)
- Implementieren Sie eine `toString`-Methode, welche einen String im folgenden Format liefert:

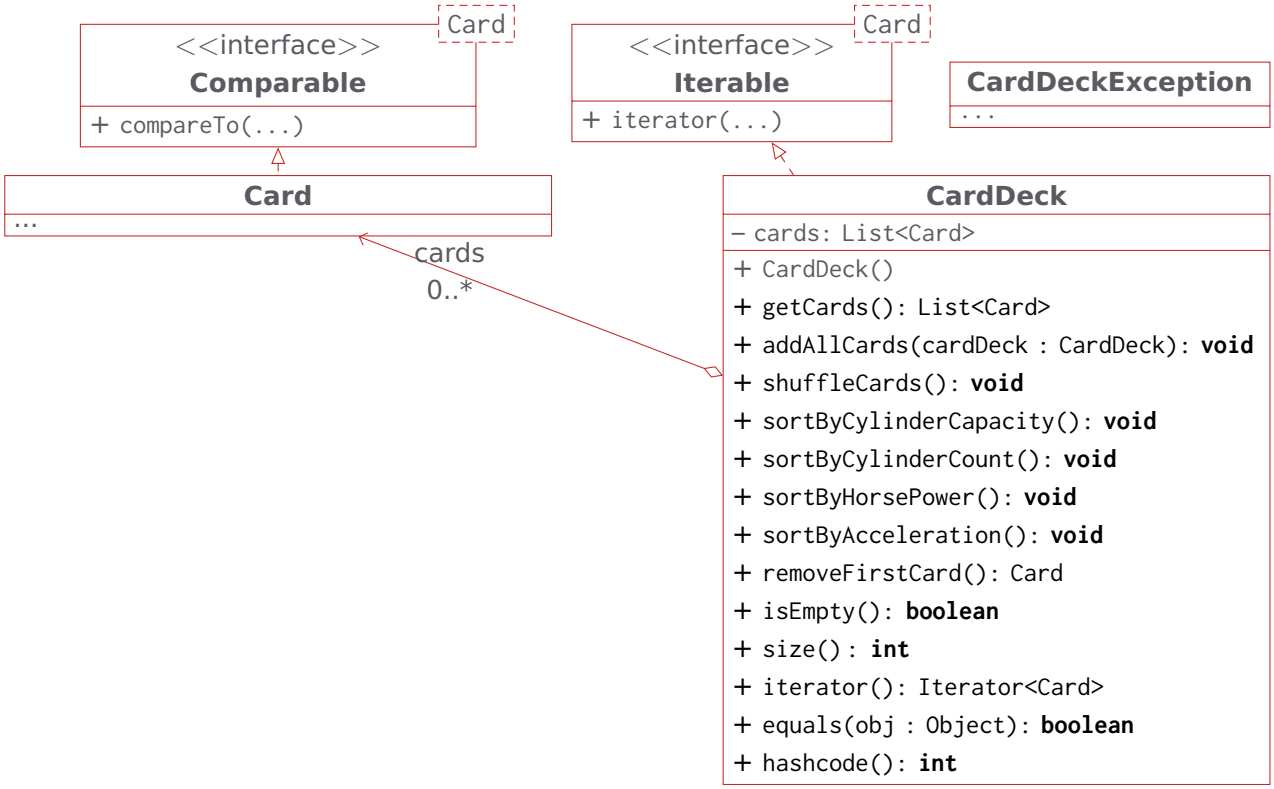
```
+++ NAME_AUTO +++
Hubraum:       HUBRAUM_AUTO
Zylinder:      ZYLINDERANZAHL_AUTO
PS:            PFERDESTAERKEN_AUTO
0 - 100 km/h:  BESCHLEUNIGUNG_AUTO
```

- Implementieren Sie die `equals`- und `hashCode`-Methoden der Klasse. Vergleichen Sie anhand aller konstanten Attribute.

Kartenstapel `CardDeck`

Die Klasse `CardDeck` repräsentiert einen Kartenstapel. Kartenstapel werden an mehreren Stellen gebraucht. Anfangs liegen die Spielkarten in Form eines einzigen Kartenstapels vor. Dieser Kartenstapel wird an die Spieler verteilt, die folglich jeweils einen eigenen Kartenstapel besitzen. Zur Vereinfachung werden später auch alle Karten, die in der aktuellen Runde gespielt werden, in einem Kartenstapel zusammengefasst. Das Klassendiagramm zur Klasse `CardDeck` finden Sie auf der nächsten Seite.

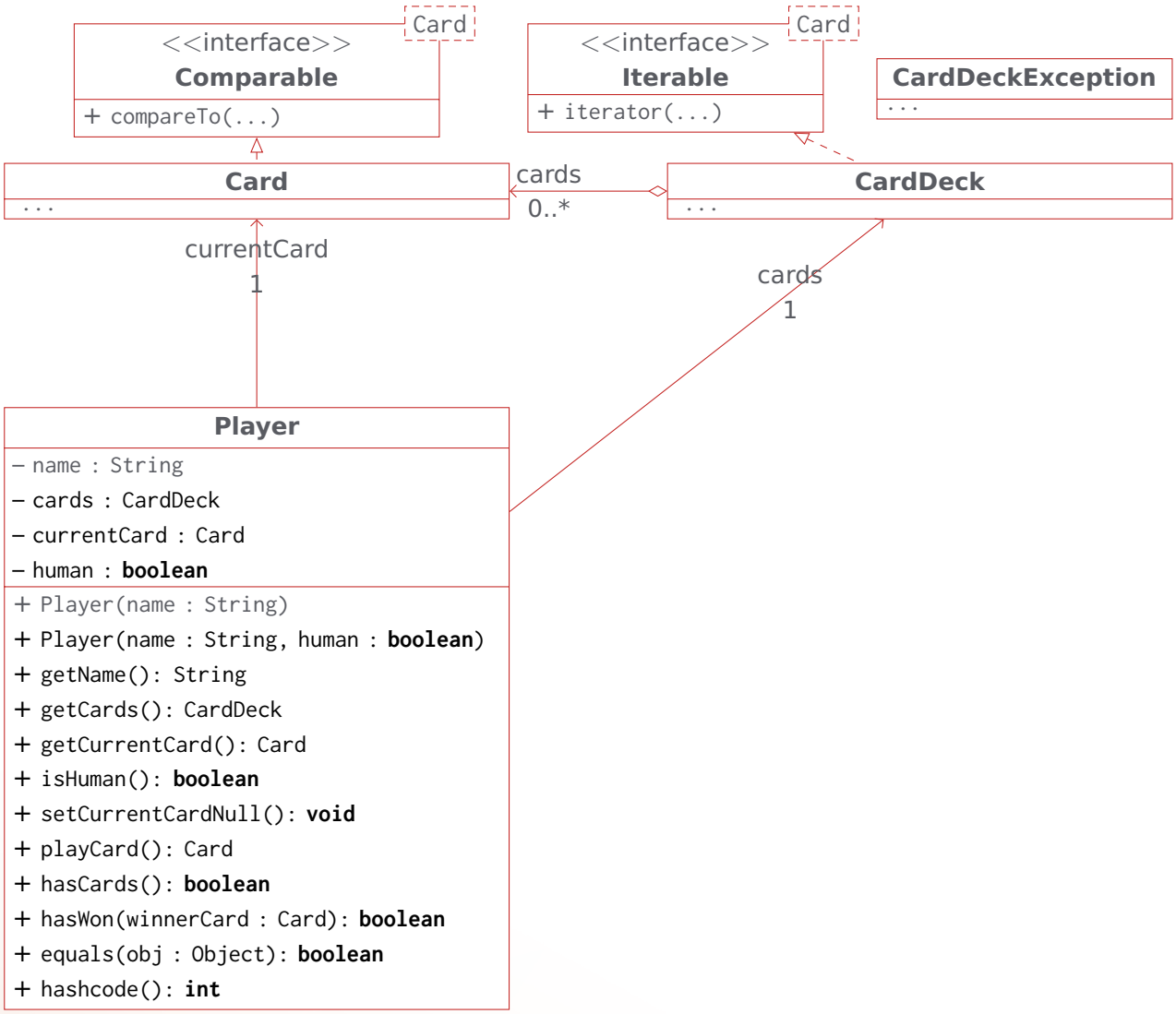
- Die Klasse `CardDeck` besitzt nur ein *einziges konstantes Objektattribut*: `cards` repräsentiert die Karten, die im konkreten Kartenstapel abgelegt sind.
- Initialisiert wird `cards` vom Konstruktor `CardDeck()`: Dabei wird für das Attribut `cards` eine neue `ArrayList` vom Typ `Card` erzeugt.
- Da nur ein Objektattribut existiert, ist auch nur eine einzige Getter-Methode zu implementieren.
- Die Methode `addCard(Card card)` fügt dem Attribut `cards` die übergebene Karte hinzu. Wenn **`null`** statt einem `Card`-Objekt übergeben wird, muss eine `IllegalArgumentException` geworfen werden. Weiterhin muss überprüft werden, ob das übergebene `Card`-Objekt bereits im `cards`-Attribut enthalten ist. Wen dem so ist, soll eine `CardDeckException` geworfen werden. Bei `CardDeckException` handelt es sich um eine *eigene ungeprüfte Ausnahme-Klasse*. Implementieren Sie diese mit den Konstruktoren, wie in der *Vorlesung gezeigt*!
- Karten sollen auch aus dem Kartenstapel entfernt werden können: `Card removeFirstCard()` entfernt *die erste* Karte des `cards`-Attributs und *verwendet sie als Rückgabewert*. Wenn `cards` keine Karten enthält, wird null zurückgegeben.



- `addAllCards(cardDeck : CardDeck)` fügt jede einzelne Karte aus dem übergebenen `CardDeck`-Objekt dem eigenen cards-Attribut hinzu. Gleichzeitig wird das hinzugefügte `Card`-Objekt *aus dem übergebenen `CardDeck`-Objekt entfernt*. Nutzen Sie hierfür die Methode `removeFirstCard()` des übergebenen `CardDeck`-Objekts.
- `shuffleCards()` mischt die Karten des `cards`-Attributs.
- **`int`** `size()` ermittelt die Anzahl der Karten, die sich im aktuellen Kartenstapel befinden.
- **`boolean`** `isEmpty()` liefert eine Aussage darüber, ob der aktuelle Kartenstapel leer ist.
- `sortByCylinderCapacity()` sortiert die Karten des Attributs `cards` *absteigend* anhand des Hubraums. Die Karten mit viel Hubraum sind nach der Sortierung also vorne im Stapel. Haben zwei Karten den gleichen Hubraum, dann wird anhand der *natürlichen Ordnung sortiert*. Implementieren Sie dazu einen `Comparator` als *private innere Klasse*.
  - `sortByCylinderCount()` sortiert die Karten des Attributs `cards` *absteigend* anhand der Anzahl der Zylinder.
  - `sortByHorsePower()` sortiert die Karten des Attributs `cards` *absteigend* anhand der Pferdestärken.
  - `sortByAcceleration()` sortiert die Karten des Attributs `cards` *aufsteigend* anhand der Beschleunigung.
  - Gehen Sie bei den drei vorangegangenen Methoden analog zur Methode `sortByCylinderCapacity()` vor.
- `CardDeck` implementiert das Interface `Iterable` für die Klasse `Card`.
  - Schreiben Sie hierfür eine Methode `Iterator<Card> iterator()`, die einen selbst definierten `Iterator` zurückgibt. Schreiben Sie den `Iterator` als *anonyme Klasse*.
  - Die Methode `hasNext` liefert true, solange im Attribut `cards` Karten enthalten sind.
  - `Card next()` hat ein ungewöhnliches Verhalten: `Card next()` entfernt die erste Karte aus dem Kartenstapel und gibt sie zurück. Wenn keine Karte mehr im Stapel ist, wird eine `NoSuchElementException` geworfen.
- Implementieren Sie die `equals`- und `hashCode`-Methoden der Klasse. Vergleichen Sie anhand aller konstanten Attribute.

Spieler `Player`

Die Klasse `Player` repräsentiert einen Spieler. Ein Spieler kann menschlich oder nicht menschlich sein. Folgendes Klassendiagramm ist zur Klasse `Player` gegeben:



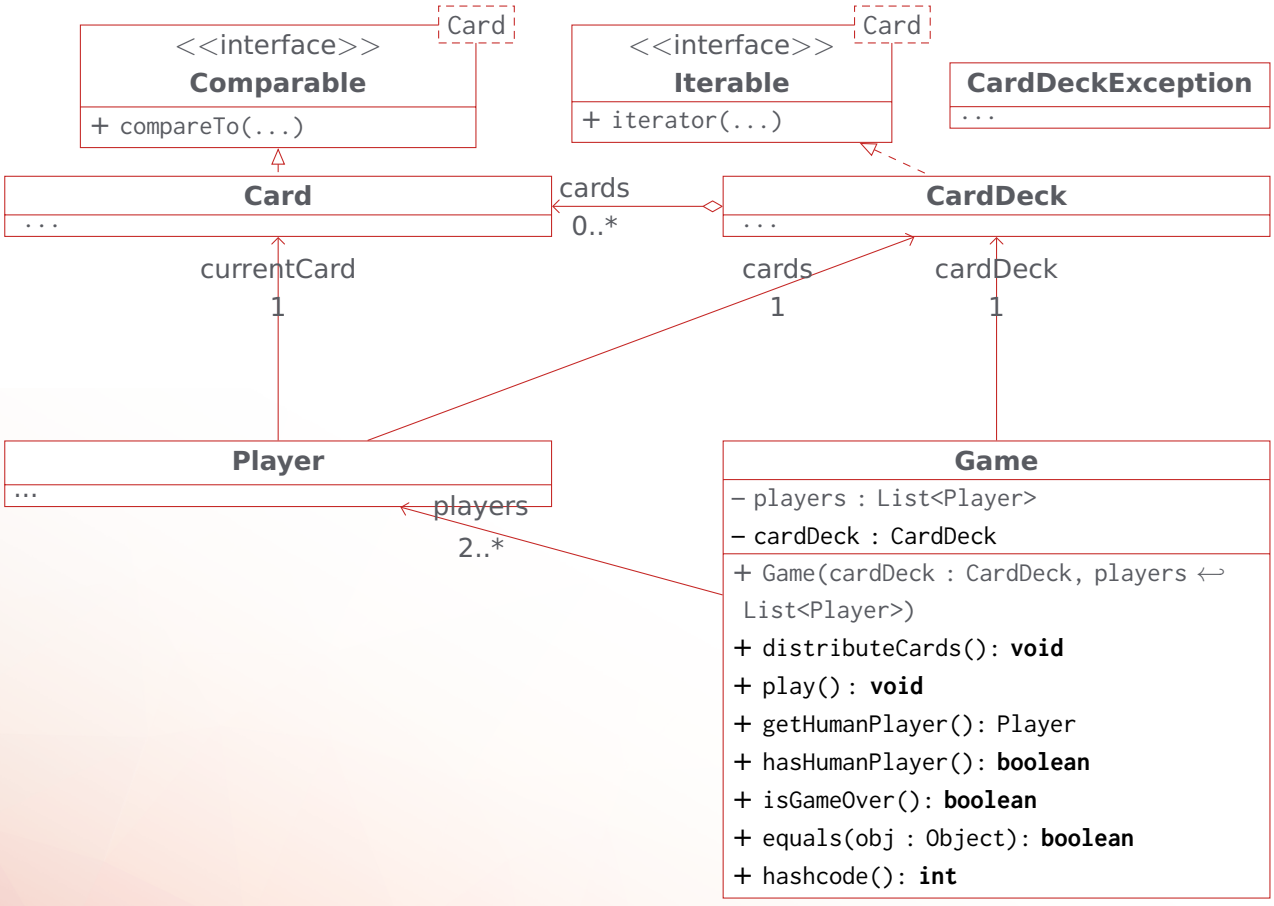
- Die Klasse `Player` hat vier Objektattribute:
  - `name` enthält den Namen des Spielers.
  - `cards` enthält den Kartenstapel des Spielers.
  - `currentCard` enthält die Karte, die der Spieler in der aktuellen Runde spielt.
  - `human` definiert, ob der Spieler menschlich ist.
- `Player` wird von zwei Konstruktoren initialisiert:
  - `Player(String name, boolean human)` nimmt die Werte für die gleichlautenden Objektattribute entgegen. Wenn `name` **`null`** oder leer ist, wird eine Exception geworfen. Weiterhin wird im Konstruktor cards als neues `CardDeck`-Objekt initialisiert.
  - `Player(String name)` ruft den Konstruktor `Player(String name, boolean human)` auf und verwendet für `human` den Wert **`false`**.
- Für die Objektattribute müssen nur die Getter-Methoden implementiert werden.
- Die Methode `setCurrentCardNull()` setzt die in der aktuellen Runde gespielte Karte auf **`null`**.
- Die Methode `Card playCard()` prüft zunächst, ob der Spieler noch Karten in seinem Kartenstapel hat (vgl. **`boolean`** `hasCards()`. Wenn dem so ist, entnimmt die Methode die erste Karte,

schreibt die entnommene Karte in `currentCard` und gibt `currentCard` als Ergebnis der Methode zurück. Wenn der Spieler keine Karte mehr hat, ist das Ergebnis der Methode **`null`**. *Verwenden Sie für das Entnehmen der Karte die Methode* `removeFirstCard()` *des* `CardDeck`*-Objekts.*

- **`boolean`** `hasCards()` gibt Auskunft darüber, ob der Spieler noch Karten in seinem Kartenstapel hat.
- **`boolean`** `hasWon(Card card)` gibt Auskunft darüber, ob er Spieler die aktuelle Runde gewonnen hat. Hierfür wird die übergebene Gewinnerkarte mit `currentCard` verglichen.
- Implementieren Sie die `equals`- und `hashCode`-Methoden der Klasse. Vergleichen Sie anhand der Attribute `name` und `human`.

Spiel `Game`

Die Klasse `Game` enthält die gesamte Logik für den Ablauf des Spiels. Die Klasse kümmert sich um die Kommunikation mit dem Spieler, organisiert den Ablauf des Spiels, ermittelt den Gewinner und übergibt dem Gewinner die Karten der aktuellen Runde.



- Die Klasse `Game` verwendet mindestens zwei Objektattribute:
  - `players` enthält eine Liste von Spielern, die am Spiel teilnehmen.
  - `cardDeck` enthält anfangs die Karten, die im Spiel verwendet werden.
- Mit dem Konstruktor `Game(CardDeck cardDeck, List<Player> players)` werden die Objektattribute initialisiert. Für ein Spiel werden mindestens zwei Spieler gebraucht. Höchstens einer er Spieler darf menschlich ein. Die Anzahl der Karten muss mindestens der Anzahl der Spieler entsprechen. Weiterhin mischt der Kostruktor auch die Karten im übergebenen Kartenstapel.
- Mit `distributeCards()` werden die Karten an die Spieler der Reihe nach verteilt: Als erstes erhält der erste Spieler in `players` eine Karte, danach geht es weiter bis zum letzten Spieler.

Befinden sich noch Karten im Stapel, erhält der erste Spieler in `players` seine zweite Karte. Das Austeilen setzt fort, so lange nicht alle Karten verteilt sind. Dadurch kann es vorkommen, dass Spieler eine Karte weniger haben als Mitspieler. Nutzen Sie für das Verteilen den selbst erstellten Iterator der Klasse `CardDeck` Nachdem alle Karten verteilt sind, sollte das `cardDeck`-Attribut der Klasse folglich leer sein.

- `Player getHumanPlayer()` liefert den menschlichen Spieler des `players`-Attributs. Wenn kein menschlicher Spieler im Attribut enthalten ist, dann liefert die Methode **`null`**.
- **`boolean`** `hasHumanPlayer()` liefert eine Aussage darüber, ob ein menschlicher Spieler im `players`-Attribut enthalten ist.
- `play()` koordiniert den Ablauf des Spiels. Die Methode kann sehr umfangreich werden. Hilfsmethoden und -attribute helfen Ihnen den Überblick zu bewahren.
  - Als erstes werden mit Hilfe von `distributeCards()` die Karten an die Spieler verteilt.
  - Danach werden so viele Runden gespielt, bis ein Spieler alle Karten hat. In jeder Runde
    - Erfolgt zunächst eine Konsolenausgabe mit einer Information der aktuellen Rundenzahl.
    - Sofern ein menschlicher Spieler beteiligt ist, erhält der Spieler eine Information, wie viele Karten noch im Stapel des Spielers sind. Auch wenn der menschliche Spieler keine Karten mehr in seinem Kartenstapel hat, erhält er die entsprechende Information.

```
----------------------------------------
Runde 1
Sie haben noch 4 Karten.
```

    - Von jedem Spieler wird die erste Karte seines Kartenstapels eingesammelt. Nutzen Sie hierfür die `playCard()` Methode der `Player`-Klasse. Die Karten sammeln Sie am besten in einem neuen `CardDeck`-Objekt. Selbstverständlich können nur von Spieler, die noch Karten in Ihrem Kartenstapel haben, Karten eingesammelt werden.
    - Wenn ein menschlicher Spieler im Spiel ist und dieser auch in der aktuellen Runde eine Karte beitragen konnte,
      - dann wird zunächst die aktuelle Karte des Spielers in der Konsole ausgegeben
      - und dann gefragt, mit welchem Attribut der Spieler spielen will.

```
Ihre aktuelle Karte:
 +++ Alfa Romeo 2000 Spider +++
Hubraum:       1962
Zylinder:      4
PS:            131
0 - 100 km/h:  11

Welche Eigenschaft wollen Sie spielen?
0: Hubraum
1: Zylinder
2: PS
3: 0 - 100 km/h
```

    - Wenn kein menschlicher Spieler (mehr) im Spiel ist, wird die gespielte Eigenschaft anhand einer Zufallszahl zwischen 0 und 3 bestimmt.
    - Die gespielte Eigenschaft wird auf der Konsole ausgegeben.
    - Entsprechend der Auswahl wird der Kartenstapel mit den eingesetzten Karten sortiert. Die Methoden hierfür haben Sie bereits implementiert.
    - Nach er Sortierung befindet sich die Gewinnerkarte an der ersten Position des Kartenstapels.
    - Anhand der Gewinnerkarte und der Methode **`boolean`** `hasWon(Card winnerCard)` der Klasse `Player` kann der Gewinner der Runde ermittelt werden. Ihm werden alle eingesetzten

Karten zugeschlagen.

- Wenn der menschliche Spieler gewonnenhat, wird auf der Konsole „Sie haben die Runde gewonnen “ ausgegeben.
- Wenn der Gewinner *nicht* der menschliche Spieler ist, dann wird der Name des Gewinners ausgegeben und die Gewinnerkarte:

```
Egon hat die Runde gewonnen. Seine Karte war::
 +++ Oldsmobile 98 Regency +++
Hubraum:       7325
Zylinder:      8
PS:            190
0 - 100 km/h:  11
```

- Vor dem Start der nächsten Runde, wird von allen Spielern das Attribut `currentCard` auf **`null`** gesetzt.
- Das Spiel endet erst, wenn ein Spieler alle Karten hat. Die Methode `isGameOver()` überprüft ob das Spiel vorbei ist und gibt die entsprechende Antwort zurück.