Kártyajáték

Óbudai Egyetem, Java Standard Edition
Mérnök Informatikus szak, BSc
Labor 4

Bedők Dávid
2016.09.28.
v0.1

Szabályok (követelmények)

A kártyajátékot egy pakli **magyar kártyá**val játsza **N személy** (N <= 4). Minden egyes kártyalap rendelkezik egy **érték**kel, melyet a kártya **szín**éből és **figurá**jából számolhatunk ki. Új játék során egy **megkevert** pakliból minden egyes játékos sorban **K lap**ot kap (K <=8). Az a játékos lesz a nyerő, akinek a kezében tartott lapok értékeinek **összege a legnagyobb**!

Entitások (osztályok) felderítése

Első lépésben meg kell határoznunk, hogy mely elemek lesznek az objektum-orientált világunkban osztályok, és melyek nem!

Hogy mi lesz osztály egy alkalmazásban (minek lesz önálló modellje), az mindig az adott alkalmazástól függ! Pl. egy autó szerelő műhelyben önálló entitás lesz az autó ablaktörlője, de egy autó verseny szimulátorban ez már lehet hogy csak egy mezőként (vagy még úgy se) realizálódik.

Mi az osztállyá válás határa?

Ez bizony egy igen jó kérdés...

Leegyszerűsítve, ha valamely vélt entitáshoz találunk olyan műveletet, felelősséget, mely egyértelműen (és közmegállapodás szerint is) az adott entitáshoz tartozik, "ő" tudja végrehajtani, vagy az "ő" felelőssége, akkor biztosak lehetünk abban, hogy "ő" osztályként realizálódhat a modellünkben!

Keressük meg melyek lehetnek osztályok a kártyajátékban!

Entitások

- Kártya figurája (CardRank)
  - 7, 8, 9, 10, Alsó, Felső, Király, Ász
- Kártya színe (CardSuit)
  - Zöld, Tök, Makk, Szív
- Kártya (Card)
  - Pl. Zöld 10, Szív Felső, Tök Ász
- Pakli (Deck)
  - 32 darab különböző kártyát tartalmaz
- Játékos (Player)
  - A játékos kezében lévő kártyákat kezeli
- Játék (Game)
  - Összefogja a fenti osztályokat

Kártya figurája (CardRank)

Felelőssége, hogy szolgáltasson egy szám értéket a kártya különböző figuráihoz annak érdekében, hogy ezt felhasználva egy kártyának (melyhez még egy szín is tartozik) lehessen értéket számolni.

A kártya figurájának "értéke" pl. a következő legyen (zárójelben az érték):

7 (7), 8 (8), 9 (9), 10 (10), Alsó (15), Felső (20), Király (30), Ász (50)

Java Enum

Mivel tudjuk, hogy pontosan 8 fix példányunk lesz a **CardRank** osztályból, ezért nagyon kényelmes megoldás, ha a Java Enum típusát használjuk.

Viszont hol fogjuk tárolni az értékeket?

A Java Enum egy **referencia típusú osztály**, hasonló lehetőségekkel mint egy "hagyományos" osztály, azonban némi megszorításokkal tudjuk csak kezelni.

CardRank enum

```
public enum CardRank {
    r7,
    r8,
    r9,
    r10,
    Under,
    Over,
    King,
    Ace;
}
```

Megjegyzés: számmal nem kezdődhet elnevezés (r*)

Java Enum háttere

A Java Enum típusban felsorolt értékek valójában `static final` konstansok, melyek típusa megegyezik* a bennfoglaló enum osztállyal.

```
public enum CardRank {
    r7,
    r8;
}
```

```
public enum CardRank {
    r7(),
    r8();
}
```

```
public class CardRank {
    private static final CardRank r7 = new CardRank();
    private static final CardRank r8 = new CardRank();
}
```

*: van kivétel (*abstract enum*)

Java Enum lehetőségei / korlátozásai

- Csak **private ctor**-ral rendelhezhet (nem lehet új példányt létrehozni hozzá)
- Tartalmazhatnak **mezők**et (akár mutable mezőket is, de javasolt csak **final** mezőkkel dolgozni)
- Nem lehet normál módon örököltetni, nem lehet indikálni az ős osztályát (Enum), viszont implementálhat interface-eket (ezen fogalmakról később tanulunk)

Megjegyzés: a fenti lista nem teljes...

CardRank - módosított verzió

```
public enum CardRank {
    r7(7),
    r8(8),
    r9(9),
    r10(10),
    Under(15),
    Over(20),
    King(30),
    Ace(50);

    private final int value;

    private CardRank(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }
}
```

Kártya színe, kártyalap értéke

A kártya színe (`CardSuit`) teljesen hasonló módon képezhető, Java Enum segítségével. A színek értéke a következő legyen:

makk (1), tök (2), zöld (3), szív (4)

Egy kártyalap értéke pedig a figurájának és a színe "értékének" szorzata lesz. Pl.:

tök 8 (16 = 2 * 8)

zöld ász (150 = 3 * 50)

CardSuit enum

```
public enum CardSuit {
    Acorns(1),
    Bells(2),
    Leaves(3),
    Hearts(4);

    private final int value;

    private CardSuit(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }
}
```

CardRank tesztelése

```
private static void testEnum(Random rand) {
    System.out.println("# Test Enum");
    CardRank cardRank = CardRank.King;
    System.out.println("Value of " + cardRank + ": " +
cardRank.getValue());
    System.out.println(CardRank.randomRank(rand));
}
```

Kártya osztály

A kártya osztály valójában nem más, mint a kártya színének és figurájának **egysége zárás**a (encapsulation).

Mivel ezen értékek sosem változnak meg egy kártya élete során (nem csalunk!), ezért mindkét mező **final** legyen!

A kártya felelőssége, hogy szolgáltatni tudja értékét (value() metódus)!

Card class

```
public class Card {

    private final CardRank rank;
    private final CardSuit suit;

    public Card(CardRank rank, CardSuit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public int value() {
        return this.rank.getValue() * this.suit.getValue();
    }

}
```

Megjegyzés: figyeljük meg a nevezéktant. A kártya osztályon belül a CardRank típusú mező nevében szükségtelen a card szót megismételni (pl. cardRank helyett egyszerűen csak rank).

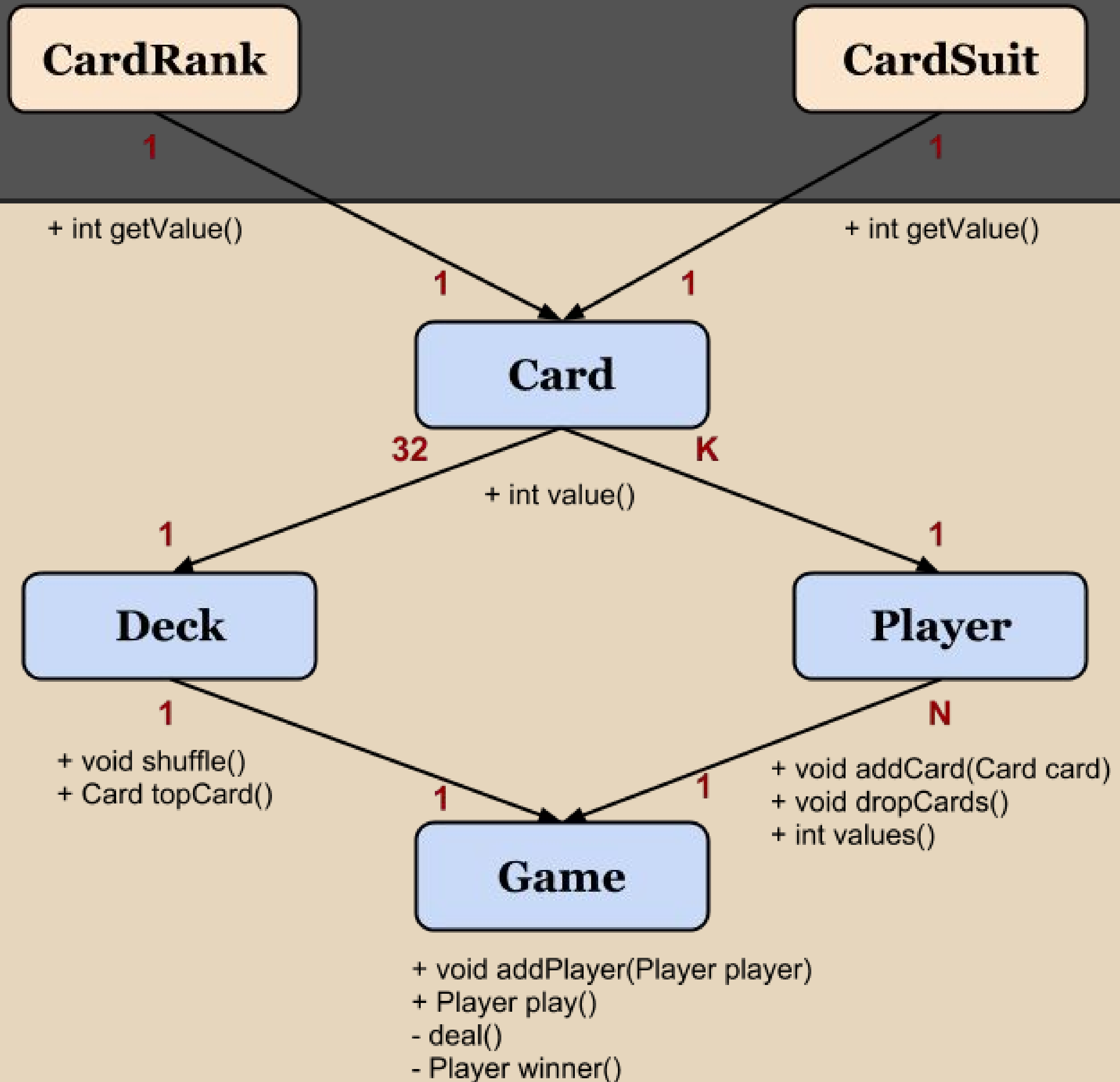
Card tesztelése

```
private static void testCard(Random rand) {
    System.out.println("# Test Card");
    Card card = new Card(CardRank.Ace,
CardSuit.Leaves);
    System.out.println(card);
    System.out.println("Value of card: " +
card.value());
}
```

Osztályok és felelősségek

Megjegyzés: A + jel public, a - jel private metódust jelöl (a private metódusok csak a fontosabb helyeken vannak jelölve).

Az ábra nem öröklési függőséget, hanem tartalmazási kapcsolatokat ír le!



Pakli osztály

A pakli osztály egyrészt létrehoz 32 különböző kártyalapot (ezt lehetőleg redundancia nélkül), majd a továbbiakban ezt kezeli.

A játék során a pakli két szituációban szerepel:

- új játék indulásakor meg kell kevernünk (shuffle() metódus)
- játék közben sorban minden játékos kap 1-1 lapot a pakli tetejéről (a pakli felelőssége csupán az, hogy mindig vissza tudja adni a legfelső lapot (topCard())

Deck class I

```
public class Deck {

    private final Random rand;
    private final Card[] cards;
    private int topCardIndex;

    public Deck(Random rand) {
        this.rand = rand;
        CardRank[] cardRanks = CardRank.values();
        CardSuit[] cardSuits = CardSuit.values();
        this.cards = new Card[cardRanks.length * cardSuits.length];
        int index = 0;
        for (CardSuit suit : cardSuits) {
            for (CardRank rank : cardRanks) {
                this.cards[index++] = new Card(rank, suit);
            }
        }
        this.topCardIndex = 0;
    }

}
```

Leszármazott Enum példányai

Figyeljük meg, hogy hogyan nyerjük ki a forráskódban a leszármazott Enum osztályunk saját példányait/értékeit!

```
CardRank[] cardRanks = CardRank.values();
```

Ez a **values()** metódus egy speciális, történelmi metódus a Java-ban. Nem az Enum ősosztálytól öröklődik, hanem a fordító automatikusan csatolja hozzá a mi leszármazott Enum osztályunkhoz (template/generic előtt is létezett, az elmélet részletei nem scope számunkra)

Deck class II - keverés

```
public class Deck {

    private static final int ROTATE_NUM = 100;

    public void shuffle() {
        this.topCardIndex = 0;
        int size = this.cards.length;
        for (int i = 0; i < Deck.ROTATE_NUM; i++) {
            this.changeCards(this.rand.nextInt(size),
this.rand.nextInt(size));
        }
    }

    private void changeCards(int indexA, int indexB) {
        Card card = this.cards[indexA];
        this.cards[indexA] = this.cards[indexB];
        this.cards[indexB] = card;
    }
}
```

Megjegyzés: van egyszerűbb módja is a keverésnek, de mi maradunk az elemi algoritmusoknál, mintsem a Java API-t ismerjük meg részleteiben (Collections.shuffle()).

Deck class III - legfelső lap

```
public class Deck {

    public Card topCard() {
        Card card = null;
        if (this.topCardIndex < this.cards.length) {
            card = this.cards[this.topCardIndex++];
        }
        return card;
    }

    @Override
    public String toString() {
        StringBuilder info = new StringBuilder(100);
        info.append("---[ Deck ]---\n");
        for (int i = 0; i < this.cards.length; i++) {
            info.append(this.cards[i]);
            if (i == this.topCardIndex) {
                info.append(" <-- top card");
            }
            info.append("\n");
        }
        return info.toString();
    }

}
```

Deck tesztelése

```
private static void testDeck(Random rand) {
    System.out.println("# Test Deck");
    Deck deck = new Deck(rand);
    System.out.println(deck);
    System.out.println(deck.topCard());
    System.out.println(deck.topCard());
    System.out.println(deck.topCard());
    System.out.println(deck);
    deck.shuffle();
    System.out.println(deck);
}
```

Játékos osztály

A játékosok valamilyen azonosításra alkalmas nevük mellett a kezükben tartott kártyalapokat kezelik. Ezeket meg tudják kapni egyesével (addCard(Card card) metódus), illetve vissza tudják adni (dropCards() metódus).

Ezeken kívül felelőssége még a kezében tartott kártyalapok összeértékének szolgáltatása is.

Player class I

```
public class Player {

    public static final int NUM_OF_CARDS = 3;

    private final String name;
    private final Card[] cards;
    private int currentCardIndex;

    public Player(String name) {
        this.name = name;
        this.cards = new Card[Player.NUM_OF_CARDS];
        this.currentCardIndex = 0;
    }

    public void addCard(Card card) {
        if (this.currentCardIndex < this.cards.length) {
            this.cards[this.currentCardIndex++] = card;
        }
    }

    public void dropCards() {
        this.currentCardIndex = 0;
    }

}
```

Player class II

összegzés

```
public class Player {

    public int cardValues() {
        int values = 0;
        for (int i = 0; i < this.currentCardIndex; i++) {
            values += this.cards[i].value();
        }
        return values;
    }

    @Override
    public String toString() {
        StringBuilder info = new StringBuilder(40);
        info.append(this.name).append(" card values: 
").append(this.cardValues()).append("\n");
        for (int i = 0; i < this.currentCardIndex; i++) {
            info.append("[").append((i + 1)).append("] ");
            info.append(this.cards[i]).append("\n");
        }
        return info.toString();
    }

}
```

Player tesztelése

```
private static void testPlayer() {
    System.out.println("# Test Player");
    Player player = new Player("Nemecsek Erno");
    player.addCard(new Card(CardRank.King, CardSuit.Hearts));
    player.addCard(new Card(CardRank.r7, CardSuit.Leaves));
    player.addCard(new Card(CardRank.Over, CardSuit.Bells));
    System.out.println(player);
    player.dropCards();
    System.out.println(player);
}
```

Játék osztály

A Játék osztály felelőssége minden eddigi osztály összefogása, valamint a kártyajátékba beülő játékosok kezelése.

A **void addPlayer(Player player)** metódus segítségével új játékos csatlakozik a partihoz (overloading miatt két ilyen nevű metódus létezik!), míg a **Player play()** metódus segítségével egy új játékot tudunk szimulálni, mely visszatér a nyertes játékossal!

Game class I

```
public class Game {

    private static final int MAX_PLAYER = 5;
    private final Deck deck;
    private final Player[] players;
    private int numberOfPlayers;

    public Game(Random rand) {
        this.deck = new Deck(rand);
        this.players = new Player[Game.MAX_PLAYER];
        this.numberOfPlayers = 0;
    }

    private void addPlayer(Player player) {
        if (this.numberOfPlayers < this.players.length) {
            this.players[this.numberOfPlayers++] = player;
        }
    }

    public void addPlayer(String name) {
        this.addPlayer(new Player(name));
    }

}
```

Game class II

```
public class Game {

    public Player play() {
        this.deck.shuffle();
        this.deal();
        return this.winner();
    }

    private void deal() {
        for (int i = 0; i < Player.NUM_OF_CARDS; i++) {
            for (int j = 0; j < this.numberOfPlayers; j++) {
                this.players[j].addCard(this.deck.topCard());
            }
        }
    }

}
```

Game class III

maximumkiválasztás

```
public class Game {

    private Player winner() {
        Player winner = null;
        if (this.numberOfPlayers > 0) {
            winner = this.players[0];
            int maxCardValues =
this.players[0].cardValues();
            for (int i = 1; i < this.numberOfPlayers; i++) {
                int cardValues =
this.players[i].cardValues();
                if (maxCardValues < cardValues) {
                    maxCardValues = cardValues;
                    winner = this.players[i];
                }
            }
        }
        return winner;
    }
```

Game class IV

```
public class Game {

    @Override
    public String toString() {
        StringBuilder info = new StringBuilder(100);
        info.append("<< Game >>").append("\n");
        for (int i = 0; i < this.numberOfPlayers; i++) {
            info.append("[").append((i + 1)).append("] ");
            info.append(this.players[i]).append("\n");
        }
        info.append(this.deck).append("\n");
        return info.toString();
    }

}
```

Game tesztelése

```
private static void testGame(Random rand) {
    System.out.println("# Test Game");
    Game game = new Game(rand);
    game.addPlayer("Nemecsek Erno");
    game.addPlayer("Darth Vader");
    game.addPlayer("Anakin Skywalker");
    System.out.println(game.play());
    System.out.println(game);
}
```

Tesztelés

Sose felejtsük el az elkészült osztályok tesztelését! Ha jól építettük fel az objektum-orientál modellünkket, akkor a tesztekből azt kell látnunk, hogy egyre komplexebb műveleteket hajtunk végre, mégis az új osztályok tesztelő kódjának bonyolultsága nagyrészt megegyezik azt őt megelőző osztály/probléma szint tesztelésével!

Az objektum-orientált programozás egyik talán legnagyobb előnye, hogy jó tervezés esetén a funkciók bővülése nem eredményezi a forráskód bonyolultsági fokának növelését, ezáltal hosszú távon karbantartható, felügyelhető kódot eredményez!

Végszó

Figyeljük meg hogy az eredeti követelmények szerint hogyan építettük fel az alkalmazást.

Érdemes kicsit elgondolkodni a személy kezében lévő kártyalapok értékének kalkulációján. A `player.cardValues()` metódus bejárja a kézben tartott lapokat, majd mindegyiken meghívja a `card.value()` metódust, ami pedig a `CardRank` és `CardSuit` `value()` metódusaival dolgozik. Mindenki a saját felelősségi szintjén dolgozik!