Általános tudnivalók

Ebben az ismertetésben az osztályok, valamint a minimálisan szükséges metódusok leírásai fognak szerepelni. A feladatmegoldás során fontos betartani az elnevezésekre és típusokra vonatkozó megszorításokat, illetve a szövegek formázási szabályait.

Segédfüggvények létrehozhatóak, a feladatban nem megkötött adattagok és elnevezéseik is a feladat megoldójára vannak bízva. Törekedjünk arra, hogy az osztályok belső reprezentációját a *lehető legjobban védjük*, tehát csak akkor engedjünk, és csak olyan hozzáférést, amelyre a feladat felszólít, vagy amit azt osztályt használó kódrészlet megkíván!

A beadott megoldásodnak működnie kell a mellékelt tesztprogramokkal, de ez nem elégséges feltétele az elfogadásnak. A megírt forráskód legyen kellően általános és újrafelhasználható!

Használható segédanyagok: Java dokumentáció, legfeljebb egy üres lap és toll. Ha bármilyen kérdés, észrevétel felmerül, azt a felügyelőknek kell jelezni, *NEM* a diáktársaknak!

A feladat összefoglaló leírása

A feladatban az UNO játékot valósítjuk meg.

A programhoz tartozik egységtesztelő amely az egyes osztályok funkcionalitását teszteli, illetve a várható pontszámot mutatja, és egy segédosztály, amit az egységtesztelő használ (ez utóbbit csak akkor lehet lefordítani, ha a `Game` osztály már meg van valósítva).

A feladat részletes ismertetése

uno.Color (1 pont)

- Vegyünk fel az UNO színeit (`GREEN`, `BLUE`, `YELLOW`, `RED`) tartalmazó felsorolási típust.

uno.Game

- Hozzuk létre a `Game` osztályt, ami egy UNO játékot reprezentál. Az osztályt később töltjük föl adattagokkal és metódusokkal.

uno.card.Card

- Készítsünk egy interface-t, amit minden UNO kártyalap implementál.
- Legyen négy függvénye:
    - A `canPlaceOn` azt dönti el, hogy a lap ráhelyezhető-e a paraméterben megadott lapra.
    - Az `effect` a lap játékban kifejtett hatását végzi el. Legyen egy `Game` típusú paramétere, amit majd később hozunk létre.
    - A `orderNum` egy számértékkel tér vissza, ami alapján a lapokat rendezzük.

    - A `getColor`, ami megmondja a lap színét (joker esetén a választottat).
- Terjessze ki a `Comparable<Card>` interfészt.

uno.card.ColorCard

- A `Card` osztálynak legyen egy absztrakt `ColorCard` megvalósítása, amivel a négy szín lapjait jelöljük.
- A konstruktor tárolja el a lap színét egy adattagban, és ezt a `getColor` függvény megvalósítása adja vissza.
- A `compareTo` metódusa más `ColorCard` kártyákkal először szín alapján hasonlítsa össze a lapokat, aztán az `orderNum` függvény értéke alapján. Egy `ColorCard` kártya azonban mindig kisebbnek számít a nem `ColorCard` (hanem Joker) kártyákhoz képest.

uno.card.NumberCard (5 pont)

A `ColorCard` osztálynak legyen egy leszármazottja, amiben a számozott kártyák lesznek.

- A számozott kártyák a konstruktorukban kapnak egy számértéket amit eltárolnak egy `number` mezőben. Ha a konstruktorban kapott érték nem 0 és 9 közötti, akkor kivált egy`IllegalArgumentException`-t.
- A `canPlaceOn` függvény igazat ad, ha a kártya színe ugyanaz mint a másiké vagy a másik is számozott kártya ugyanazzal a számmal.
- Az `orderNum` függvény az esetükben ezt az eltárolt számot adja vissza.
- Az `effect`-jük üres.
- Az örökölt `toString` függvényüket definiáljuk felül úgy, hogy a színt és a számot írja ki, például `GREEN 8`

uno.card.WildCard (3 pont)

Hozzuk létre a joker laptípust, ami bármely lapra ráhelyezhető. Származzon a `Card` típusból.

- A `canPlaceOn` függvény mindig igazat ad.
- Az `orderNum` függvény 13-at ad vissza.
- Az `effect`-je üres.
- Legyen egy `chooseColor` függvénye, ami eltárolja a lap színét egy adattagban, és ezt a `getColor` függvény megvalósítása adja vissza.
- Az örökölt `toString` függvényüket definiáljuk felül úgy, hogy a `Wild` szöveget írja ki, majd zárójelben a választott színt, például `Wild (GREEN)`.

uno.Player (7 pont)

Készítsük el a játékost reprezentáló osztályt.

- Tároljuk el a játékos nevét egy `String` adattagban. Ez legyen lekérdezhető egy `getName()` függvénnyel. A nevet a konstruktorban kapja meg. *(1 pont)*
- A játékoshoz tároljuk el a kezében tartott lapokat. Ehhez használjunk `TreeSet` konténert (az elemtípusa `Card`), hogy a kézben a lapok rendezetten jelenjenek meg.

- Legyen egy `draw` függvénye, ami egy kártyát kap. Ha a kártya nem null, akkor hozzáadja a kézben tartott lapokhoz. *(1 pont)*
- Legyen egy `hasWon` függvénye, ami akkor ad igazat, ha a kéz üres. *(1 pont)*
- Legyen egy `terminalMessage` függvénye, ami egy stringben visszaadja a játékos nevét és a kézben található lapokat (a pontos formátum nem számít). *(2 pont)*
- Legyen egy `chooseCard` függvénye, ami egy string alapján visszaad egy kártyát. Ha a szöveg üres, akkor null-t ad vissza. Ha nem üres a szöveg, akkor egy pozitív számnak kell benne lennie. A kézben levő kártyákat 1-től számozva kiválasztja a megadott kártyát, kiveszi a kézből és visszaadja. *(2 pont)*

uno.Game (16 pont)

Folytassuk a `Game` osztály megvalósítását.

- Legyen egy osztályszintű `getAllCards` metódusa, ami kártyák láncolt listáját adja vissza.
  - Egyelőre minden színből 2db-ot az 1 és 9 közötti lapokból, 1db-ot a 0-ás számozott lapból *(2 pont)*
- Az osztálynak öt adattagja van:
  - A `players` a játékban résztvevő játékosok listája.
  - Az `actualPlayer` az éppen aktuális játékos indexe (0-tól kezdve).
  - A `pile` a húzópakli, lapok láncolt listája.
  - A `played` a kijátszott lapok kupaca, szintén lapok láncolt listája.
  - A `forwardOrder` logikai érték, azt jelzi, hogy előrefelé megy-e a kör. Alapértelmezetten igaz.
- Négy egyszerű függvénnyel lehet a játék állapotát lekérdezni, vezérelni:
  A `getActualPlayer` visszaadja az aktuális játékost (`Player` objektum). A `nextPlayer` a következő játékosra lép (figyelembe véve, hogy merre megy a kör).
  A `reverseOrder` megfordítja a kör irányát (a `forwardOrder` negálódik).
  A `putOnTop` metódus egy kártyát vár, és azt a kijátszott kártyák tetejére teszi. *(4 pont)*
- A konstruktora kapja meg a játékosok neveit, majd hozzon létre belőlük `Player` objektumokat és tárolja el a `players` adattagban.
- A `startGame` metódus a `pile`-t állítsa be a `getAllCards` metódus által létrehozott paklira, majd keverje meg a `java.util.Collections.shuffle` függvénnyel. Minden játékos`draw` metódusát hívja meg hatszor a paklából elvett kártyával (`nextCard`). Majd addig pakoljon a paklából a kijátszott lapok kupacára, amíg nem jön egy számozott lap.
- A `nextCard` elveszi a pakli legfelső lapját és visszaadja, ha a pakli nem üres. Ha az, akkor a kijátszott lapok legfelső lapját kivéve, az összes lapot a pakliba rakja összekeverve és ezután adja vissza a pakli legfelső lapját. *(5 pont)*
- Legyen egy `play()` metódusa, ami elindítja a játékot. A `play` metódus visszatérési értéke a győztes játékos neve. A játéknak akkor van vége, ha a soron következő játékos kezében nincsen lap (`hasWon`). A játék minden lépésben kiírja a legfelső kijátszott lapot, majd az aktuális játékosnak szóló üzenetet (`terminalMessage`). Ezután bekéri a játékos akcióját a konzolablakban (az adatbekérést a `Game` osztály egy védett `readLine()` függvényen keresztül végezzük a `System.in` stream-ből). Ezt az aktuális játékos`execute` függvényének adja át, ami választ egy lapot. Ha a lap null, vagy nem helyezhető el a legfelső kijátszott lapra, akkor az aktuális játékos a draw függvénnyel fölhúzza a pakli legfelső lapját és visszakapja azt is, amit lerakott. Különben a lap kijátszásra kerül, a joker lapokhoz bekérjük a választott színt, majd beállítjuk a lapon. Ezután a lap`effect`-je végrehajtódik, a lap a kijátszott lapok tetejére lesz helyezve. Akár húzás, akár kijátszás történt, ezután a következő játékos jön. *(5 pont)*

uno.card.SkipCard, uno.card.ReverseCard, uno.card.TakeTwoCard (6 pont)

Hozzuk létre a ColorCard három további leszármazottját.

- A `canPlaceOn` függvény igazat ad, ha a kártya színe ugyanaz mint a másiké vagy a másik kártya is azonos típusú (például `SkipCard` esetén a másik is `SkipCard`)
- Az `orderNum` függvény adjon vissza 10, 11 és 12 értékeket a `SkipCard`, `ReverseCard` és `TakeTwoCard` esetében.
- Az `effect`-jük: `SkipCard` esetén egy `nextPlayer()` hívás a játékon, `reverseCard` esetén `reverseOrder()` hívás a játékon, `takeTwo` esetén `nextPlayer()` hívás majd kétszer az aktuális játékos (már a következő) felhúzza a `nextCard()` által adott lapot.
- Az örökölt `toString` függvényüket definiáljuk felül úgy, hogy a `<szín> skip`, `<szín> reverse` és `szín +2` szövegeket adja vissza.

uno.card.TakeFourCard (3 pont)

Hozzuk létre az értékes négy lapot húzató kártyát. Ez legyen a `WildCard` leszármazottja.

- Az `effect`-je legyen: `nextPlayer()` hívás majd négyszer az aktuális játékos (már a következő) felhúzza a `nextCard()` által adott lapot.
- Az `orderNum` függvény 14-et adjon vissza.
- A toString függvény a `+4` szöveg után zárójelbe írja a választott színt, például `+4 (RED)`.

uno.Game.getAllCards(2 pont)

A `Game` osztályban most már kiegészíthetjük `getAllCards` függvény törzsét: a láncolt listába rakjunk be további:

- 2db letiltó, körfordító és +2-es lapot minden szinből.
- 4db joker és +4-es lapot.

Próba

Írjunk egy main függvényt, ami létrehoz egy új `Game` objektumot és a `startGame()` és `play()` függvények segítségével játsszunk a játékkal (de a mellettünk ülőket ne invitáljuk meg, ha még nem fejezték be a feladatot).

Pontozás

A tesztelő által adott pontszám csak becslésnek tekinthető, a gyakorlatvezető levonhat pontokat, vagy adhat részpontokat.

Ponthatárok:

- 0 - 13 : elégtelen (1)

- 14 - 21 : elégséges (2)
- 22 - 28 : közepes (3)
- 29 - 35 : jó (4)
- 36 - 43 : jeles (5)

Jó munkát, jó játékot!