A feladat összefoglaló leírása

Ebben a feladatban a közismert Snake játék működését fogjuk szimulálni. A játékban egy rögzített méretű pályán kell egy kígyót irányítani, mely almákat szed össze, és eszik meg, egyre növelve a méretét, miközben nem ütközhet sem falba, sem saját magába.

A feladat megoldása során be kell tartani a [kódolási konvenciókat](http://people.inf.elte.hu/bonnie/java/kodminoseg.html). Ennek megfelelően ügyeljünk arra, hogy a megadottakon kívül egyetlen osztály se tartalmazzon más publikus metódust vagy adattagot, illetve egyik csomag se tartalmazzon más osztályokat! Ha az implementáció megköveteli, akkor az osztályok rejtett adattagokkal és metódusokkal szabadon bővíthetők.

A fordításhoz legalább a Java Standard Edition 8 használata kötelező.

A feladathoz tartozik egy [letölthető segédlet](http://people.inf.elte.hu/koguaai/java/snake.zip), ahol találunk több minta bemeneti fájlt és kimeneti fájlt, valamint egy futtató Main osztályt, egy magyarázó leírással kiegészítve.

Beadás módja

* A megoldást egyetlen .zip állományként kell feltölteni, amely tartalmazza a csomagnak megfelelő könyvtárszerkezetben az összes forráskódot.
* A .zip állomány *NE* tartalmazzon egy (se több) legkülső összefogó mappát, közvetlen az elvárt csomagszerkezetnek megfelelő könyvtárak tagozódjanak alá. (Ne legyen egy bead, bead\_NEPTUN vagy bármi hasonló, legkülső mappa a zip-ben.)
* A fordítás során keletkező .class állományokat és a JUnit futtatásához használatos két .jar fájlt sem szabad mellékelni!

A feladat részletes ismertetése

1. Segédtípusok

snake.exception.CollisionException

Készítsük el a snake.exception.CollisionException ellenőrzött kivételosztályt (származzon az Exception osztályból). Ez a kivétel fogja jelezni, ha a kígyó nekiütközik valaminek.

snake.exception.InvalidIndexException

Készítsük el a snake.exception.InvalidIndexException *nem* ellenőrzött kivételosztályt (származzon a RuntimeException osztályból). Ez a kivétel fogja jelezni, ha a játéktér határain kívülre próbálnánk menni.

snake.util.Position

Hozzuk létre a snake.util.Position osztályt, mely egy oszlop- és sorkoordinátáival azonosított pozíciót fog reprezentálni a játéktéren.

Adattagjai:

* SIZE\_OF\_BOARD: publikus, osztályszintű, módosíthatatlan, egész szám típusú. A játéktér sorainak és oszlopainak maximális darabszáma. Értéke 10.
* row: módosíthatatlan, egész szám típusú. A pozíció sorindexe.
* column: módosíthatatlan, egész szám típusú. A pozíció oszlopindexe.

*Megjegyzés:* mivel minden adattagja módosíthatatlan, a Position egy *immutable* típus.

Metódusai:

* Legyen egy publikus, két egész szám paramétert fogadó konstruktora. A két paraméter a sor- és oszlopindex. Ezek akkor megfelelőek, ha értékük nemnegatív és kisebb, mint a SIZE\_OF\_BOARD értéke. Ha valamelyik nem megfelelő, dobjunk snake.exception.InvalidIndexException kivételt. Különben mentsük le az értéküket a megfelelő adattagokba.
* Legyen egy publikus getRow és egy publikus getColumn metódusa a megfelelő adattagok lekérdezésére.
* Legyen egy publikus, logikai visszatérési értékű isSame metódusa, mely egy Position objektumot vár paraméterül. Ha a paraméter nem null, és sor-, illetve oszlopindexe megegyezik az aktuális objektuméval, akkor térjünk vissza igazzal, máskülönben hamissal.

*Megjegyzés:* a feladat egy későbbi fázisában szükség lesz a Position osztály hashCode és equals metódusainak megírására is. Ha ezt megtesszük előre, akkor bárhol, ahol két Position objektumot kell összehasonlítani, tetszőlegesen használható az isSame helyett az előre megírt equals is.

snake.util.Direction

Készítsük el a snake.util.Direction felsorolási típust, melynek lehetséges értékei a következők legyenek: UP, DOWN, RIGHT és LEFT.

2. Alapszerkezet

Hozzuk létre a feladat alapszerkezetét adó típusokat!

snake.Game

A program vezérlését a snake.Game osztály fogja végezni. Ezt egyelőre még csak hozzuk létre, később fogjuk metódusokkal kiegészíteni.

snake.Tile

Készítsük el a snake.Tile interfészt, mely a játéktér egy mezőjét fogja ábrázolni. Legyen egyetlen metódusa, a Position visszatérési értékű, paraméter nélküli getPosition, melynek implementációi a játékmező aktuális pozícióját fogják visszaadni.

snake.Snake

A kígyót a snake.Snake interfészen keresztül szeretnénk majd vezérelni. Ez az interfész származzon a Tile interfészből.

Metódusai:

* Legyen egy publikus, visszatérési érték nélküli, Direction paramétert fogadó move metódusa, mely dobhat CollisionException-t. Ennek a metódusnak az implementációi majd az adott irányba fogják mozdítani a kígyót 1 mezőnyit, ha ez nem okoz ütközést.
* Legyen egy publikus, visszatérési érték nélküli, kétparaméteres move metódusa, mely dobhat CollisionException-t. Az első paramétere Direction típusú, a második egy egész szám (times). Ennek a metódusnak az implementációi majd az adott irányba fogják mozdítani a kígyót times mezőnyit, ha ez nem okoz ütközést.

snake.Apple

Készítsük el a Tile interfészt megvalósító snake.Apple osztályt, mely az almákat fogja ábrázolni.

Adattagjai:

* position: módosíthatatlan, Position típusú. Az alma pozícióját tárolja.

Metódusai:

* Legyen egy publikus konstruktora, melynek egyetlen Position típusú paramétere van. Tárolja le a paramétert az adattagba.
* Valósítsa meg a Tile interfészből örökölt getPosition metódust: térjen vissza az adattag értékével.

3. A kígyó

Készítsük el a kígyó részeit ábrázoló alábbi két típust. Ezek még csak a kígyó alapvető mozgatásáért felelnek, azzal, hogy a kígyó megegye az almákat, és azoktól megnőjön, később foglalkozunk majd.

snake.parts.SnakeTail

Hozzuk létre a Tile interfészt megvalósító snake.parts.SnakeTail osztályt, mely a kígyó farkát fogja ábrázolni.

Adattagjai:

* position: Position típusú. A kígyó farkának aktuális pozícióját tárolja.

Metódusai:

* Legyen egy publikus konstruktora, melynek egyetlen Position típusú paramétere van. Tárolja le a paramétert az adattagba.
* Valósítsa meg a Tile interfészből örökölt getPosition metódust: térjen vissza az adattag értékével.
* Legyen egy védett, visszatérési érték nélküli, Position paraméterű moveTo metódusa. Mozgassa a kígyó farkát a megadott pozícióra, azaz módosítsa az adattagot a paraméter értékére.
* Legyen egy védett, logikai visszatérési értékű, Position paraméterű isAt metódusa. Ellenőrizze, hogy a kígyó az adott pozíción van-e, azaz, hogy az aktuális pozíciója egyenlő-e a paraméterül kapottal.

snake.parts.SnakeHead

Hozzuk létre a snake.parts.SnakeHead osztályt, mely a kígyó fejét ábrázolja. Valósítsa meg a Snake interfészt, hiszen a kígyónak a fejét akarjuk irányítani. Származzon a SnakeTailosztályból, hogy bizonyos örökölt metódusokat ne kelljen újradefiniálnunk.

Adattagjai:

* game: módosíthatatlan, Game típusú. A játék vezérléséért felelős objektumot fogja tárolni.
* tail: SnakeTail típusú. A kígyó farkát tárolja.

Metódusai:

* Legyen egy publikus, SnakeHead(Position position, Position positionOfTail, Game game) konstruktora. Az első paraméter a fej pozíciója: adja át az ősosztály konstruktorának paraméterül. A game paramétert mentse le a megfelelő adattagba. Végül hozzon létre egy új SnakeTail objektumot, a positionOfTail értékével felparaméterezve, melyet tároljon el a tail adattagjában.
* Valósítsa meg a Snake interfészből örökölt move(Direction dir, int times) metódust. Hívja meg times-szor az egyparaméteres move metódust. Ha a times értéke 0vagy negatív, ne hívja meg egyszer se.
* Valósítsa meg a Snake interfészből örökölt move(Direction dir) metódust. Ez a metódus ágazzon szét aszerint, hogy a dir paraméter értéke milyen. Minden esetben hívjuk meg az alábbiakban meghatározott, két egész paramétert váró move segédmetódust a következő paraméterezéssel:
  + UP esetén: row - 1, column
  + DOWN esetén: row + 1, column
  + RIGHT esetén: row, column + 1
  + LEFT esetén: row, column - 1
* Egy privát, két egész paramétert váró move segédmetódusban hozzunk létre a kapott paraméterekkel egy új Position objektumot, majd végezzük el az alábbiakat.
  + Ha InvalidIndexException-t kapunk, dobjunk helyette CollisionException-t (a kígyó nekiment a pálya szélének).
  + Ellenőrizzük az isAt metódussal, hogy a kígyó farka a célul kitűzött pozíción van-e, ha igen, szintén dobjunk CollisionException-t (a kígyó nekiütközött saját magának).
  + Különben mozgassuk a kígyót az új pozícióra, a farkát (tail adattag) pedig arra a pozícióra, ahol eddig a feje volt.

*Megjegyzés:* Gondoljuk végig előre, hogy milyen paraméterekkel és milyen sorrendben érdemes a kígyó fejét és farkát mozgató metódusokat meghívni.

4. Game osztály

Egészítsük ki a Game osztályt.

Adattagjai:

* apples: módosíthatatlan, Apple objektumokat tartalmazó lista. A játékban elérhető almákat tartalmazza.
* snake: módosíthatatlan, Snake típusú. A játékban a kígyót fogja tárolni.

Metódusai:

* Legyen egy osztályszintű, Apple típusú objektumok listájával visszatérő toApples(List<String> lines) metódusa. Ez a listában megkapott szövegeket almákká alakítja. Ha a kapott lista null, a metódus dobjon IllegalArgumentExceptiont. Különben dolgozza fel a lista elemeit a következőképp:
  + Mindegyik szöveg pontosan két darab, szóközzel elválasztott egész számot tartalmaz (pl. 5 2). Ezek az alma sor-, illetve oszlopindexe.
  + A metódus hozzon létre ezek segítségével egy megfelelő Position objektumot, majd azzal felparaméterezve egy új Apple objektumot, és tegye ezt az eredménylistába.
  + Ha valamelyik szöveg mégsem a megadott alakú (pl. több vagy kevesebb szóközzel elválasztott részből áll, nem számok a részei stb.), vagy bármelyik index nem megfelelő (a Position konstruktora kivételt dob), akkor azt a szöveget hagyjuk figyelmen kívül (ilyenkor ne tegyünk semmit az eredménylistába)!
* Legyen egy publikus Game(List<String> apples) konstruktora. A paraméterül kapott listát alakítsa almák listájává a toApples metódusnál leírt konverziós szabályok szerint, majd mentse le a megfelelő adattagba. Ezenkívül hozzon létre egy SnakeHead objektumot, és a létrejött objektum konstruktorának adja át három paraméterként rendre:
  + a 0, 1 pozíciót (mindig innen indul majd a kígyó feje);
  + a 0, 0 pozíciót (mindig innen indul majd a kígyó farka);
  + az aktuális Game objektumot.
* Legyen egy Apple visszatérési értékű, paraméter nélküli getApple metódusa. Ez mindig térjen vissza az apples adattagban tárolt lista első elemével, amennyiben a lista nem üres. Ha a lista üres, adjon vissza null-t.

*Megjegyzés:* a lista első eleme a nulladik indexű elem.

* Legyen egy visszatérési érték nélküli, paraméter nélküli ateApple metódusa. Ennek meghívása jelenti azt, hogy a kígyó megette az aktuális almát (mely mindig az apples lista első eleme). A metódus törölje ki az apples lista első elemét. (A lista eggyel rövidebb lesz.)

*Megjegyzés:* a lista első eleme a nulladik indexű elem.

* Legyen egy szöveggel visszatérő play(List<String> moves) metódusa, mely azért felel, hogy a paraméterül kapott mozgató utasításokat kiadja a kígyónak, és egy szöveges kimenetet állítson elő. A szöveges kimenetet majd a főprogram fogja egy fájlba írni.
  + A moves lista minden egyes sora két alakot vehet fel. Vagy egyetlen irányt jelző szóból áll (a Direction felsorolási típus egyik konstansa), vagy egy irányt jelző szóból és egy szóközzel elválasztott egész számból (ismétlésszám).

Pl.

UP

RIGHT 5

* + Ha valamelyik sor nem ilyen alakú, hagyjuk figyelmen kívül! A helyes sorokat dolgozzuk fel, és hívjuk meg a snake változóban tárolt kígyó megfelelő move metódusát értelemszerűen felparaméterezve.

*Megjegyzés:* a megfelelő szövegrészletek Direction típusú objektummá alakításához használhatjuk a Direction.valueOf metódust.

*Megjegyzés:* nem kell ellenőriznünk, hogy az ismétlésszám pozitív-e. Negatív vagy 0 ismétlésszám esetén egyszerűen nem fog elmozdulni a kígyó.

* + Ha CollisionException-t kapunk, a metódus fűzze hozzá a kimenethez a "GAME OVER" szöveget, és térjen vissza az eddig elkészített szöveggel.
  + Minden egyes helyes sor feldolgozása után írjuk ki a játéktábla aktuális állását (ha nem történt ütközés). Ehhez hozzuk létre a privát, visszatérési érték nélküli, StringBuilder paramétert váró printState segédmetódust. Ez a metódus a megkapott StringBuilder végére fogja fűzni a játéktábla aktuális állapotát (ez is része a play által visszaadott kimenetnek).

A printState metódus törzsének elkészítéséhez egy további típusra és néhány metódusra lesz szükség, melyeket a következő, 5. részben készítünk el.

* + Ha már az összes elvárt mozgatást (a lista összes tagját) feldolgozta a play metódus, és eddig nem történt ütközés, szintén térjen vissza az eddig elkészített szöveggel. (Ilyenkor *ne* fűzzük hozzá a "GAME OVER" szöveget.)

**FONTOS!** A feladat logikai felépítése szerint az 5. és 6. rész következik, viszont az utolsó 7. rész JUnit tesztek elkészítése a már megírt toApples metódushoz. Javasoljuk rögtön a mostani 4. rész után a tesztek megírását. (Egy valós szoftverfejlesztési projektben sem szeparálódhat nagyon a kód megírása és tesztelése.)

5. Kiíratás

snake.util.Position

Készítsük el a Position osztály hashCode és equals metódusát! Két Position objektum akkor egyenlő, ha sor- és oszlopindexük is megegyezik.

snake.util.PositionMap

Hozzuk létre a snake.util.PositionMap osztályt, melynek egyetlen T típusparamétere van.

Adattagjai:

* map: módosíthatatlan, Position kulcsokhoz T típusú elemeket társító Map.
* defaultValue: módosíthatatlan, T típusú.

Metódusai:

* Legyen egy publikus PositionMap(T defaultValue) konstruktora, mely letárolja a kapott paramétert a megfelelő adattagba. A map adattagot inicializáljuk egy üres HashMap-pel.
* Legyen egy visszatérési érték nélküli put(Position pos, T element) metódusa. Ez a metódus helyezze el a map adattagban tárolt Map-ben a paraméterül kapott kulcs-érték párt.
* Legyen egy T visszatérési értékű get(Position pos) metódusa. Ez a metódus kérje le a pos kulcshoz tartozó értéket a map-ből. Ha van ilyen, térjen vissza vele. Ha nincs, térjen vissza a defaultValue-val.

snake.Tile

Egészítsük ki a Tile interfészt a visszatérési érték nélküli print(PositionMap<Character> map) metódussal, melynek implementációi azért felelnek majd, hogy a paraméterül kapott map-be elhelyezzenek egy, a megfelelő mezőt ábrázoló karaktert.

Ezt a metódust a Tile interfészt megvalósító osztályainkban a következőképpen definiáljuk felül:

* Apple: helyezzünk be a map-be az alma pozíciójára egy 'o' karaktert.
* SnakeTail: helyezzünk be a map-be a kígyó farkának pozíciójára egy '~' karaktert.
* SnakeHead: helyezzünk be a map-be a kígyó fejének pozíciójára egy '@' karaktert, valamint hívjuk meg a kígyó farkának (tail adattag) print metódusát is, a map-et átadva paraméterül.

snake.Game.printState

Az előző részfeladatban létrehozott printState segédmetódust a Game osztályban a következőképpen definiálhatjuk:

* Hozzon létre egy új PositionMap<Character> objektumot, egy szóköz karaktert adva neki paraméterül.
* Hívja meg a snake adattagnak és az apples lista első elemének (amennyiben a lista nem üres) print metódusát a létrehozott PositionMap-pel felparaméterezve. Ezzel az aktuális alma és a kígyó kirajzolódik a térképre.
* Két egymásba ágyazott ciklussal járja végig a pálya elérhető indexeit (a sor- és az oszlopindex legalább 0, és SIZE\_OF\_BOARD-nál kisebb), készítsen el mindegyiknek megfelelően egy Position objektumot, és kérje le az adott pozíción lévő karaktert a map-ből. A karaktert fűzze hozzá a paraméterül kapott StringBuilder-hez.
* Soronként fűzzön hozzá egy sortörés karaktert.
* Miután a ciklus véget ért, a StringBuilderhez még fűzze hozzá a metódus a "==========" karakterláncot (10 darab egyenlőségjel) és ezt követően egy sortörés karaktert, ezekkel jelezve az aktuális kirajzolás szélét.

*Megjegyzés:* Különböző operációs rendszerek különböző sortörés karaktereket használnak (\n,\r,\r\n). A platform-semleges sortörések érdekében használjuk a System.lineSeparator() metódust.

*Megjegyzés:* ezen a ponton a feladat már futtatható a mellékelt Main osztály és az első mintabemenet segítségével. Az ugyanis nem tartalmazza egyetlen alma elfogyasztását sem.

6. Almaevés és növekedés

snake.parts.SnakeTailPart

Készítsük el a snake.parts.SnakeTailPart osztályt, mely származik a SnakeTail osztályból. Ez az osztály fog felelni azért, hogy a kígyó farka (teste) növelhető legyen, ahogy újabb és újabb almákat fogyaszt el. Az egyes példányok úgy fognak viselkedni, mintha SnakeTail objektumok lennének, leírják nem csak magukat, de a kígyó teljes (mögöttük lévő) farkát. Azaz minden metódusra úgy adnak megoldást, hogy a mögöttük lévő elemeket is megfelelően mozgatják vagy figyelembe veszik (rekurzív metódushívásokkal).

Adattagjai:

* next: módosíthatatlan, SnakeTail típusú. A következő részt tartalmazza a kígyó farkából. Feltehetjük, hogy sosem null.

Metódusai:

* Legyen egy publikus SnakeTailPart(Position position, SnakeTail next) metódusa, mely átadja az ősosztály konstruktorának a kapott pozíciót, a next paraméteret pedig elmenti a megfelelő adattagba. Ha a next értéke null, dobjon IllegalArgumentException-t.
* Írja felül a SnakeTail osztályból örökölt, védett moveTo metódust. Először mozgassa a next adattagban lévő következő elemet az aktuális pozíciójára (a moveTo metódus rekurzív hívásával), majd hívja meg az ősben definiált változatát, hogy saját magát a paraméterül kapott pozícióra mozgathassa. (Így a kígyó farkának következő eleme oda kerül, ahol eddig ez az elem volt, míg ez az elem oda, ahová a metódus paramétere mutat.)
* Írja felül a SnakeTail osztályból örökölt, védett isAt metódust. Ha az aktuális objektum pozíciója egyenlő a paraméterül kapott pozícióval, a metódus térjen vissza igazzal. Különben ellenőrizze, hogy a kígyó farkának hátralévő része nincs-e az adott pozíción a next objektum isAt metódusával.
* Írja felül a Tile interfészben deklarált és a SnakeTail osztályban már implementált print metódust. Helyezzen a paraméterül kapott map-be egy '#' karaktert, illetve hívja meg a next objektum print metódusát is.

snake.parts.SnakeHead.move(int, int)

Egészítsük ki a SnakeHead osztály két egész paramétert váró move metódusát, hogy ha a fej az aktuális almára ér, megegye.

* Miután a metódusban létrehoztuk a következő lépésnek megfelelő pozíciót, valamint ellenőriztük, hogy ez egy érvényes pozíció-e, ahol a kígyó farka nincs ott (nem történt ütközés), vizsgáljuk meg, hogy az új pozíció, ahová lépünk, nem egyenlő-e az aktuális alma pozíciójával. Az almát a game adattag getApple metódusával kérhetjük le. (Figyeljünk, hogy az eredmény null is lehet, ha már nincs több alma! Ilyenkor értelemszerűen nem történt evés.)
* Ha az alma *nincs* a következő pozíción, akkor mozgassuk a kígyó farkát az eddigi pozícióra, a fejét pedig a következőre, ahogy eddig.
* Ha az alma épp a következő pozíción van, hozzunk létre egy új SnakeTailPart objektumot, az aktuális pozícióval és a kígyó eddigi farkával felparaméterezve (tail adattag), majd ezt mentsük a tail adattagba. Emellett hívjuk meg a gameadattag ateApple metódusát, hogy jelezzük az alma elfogyasztását, és mozgassuk a fejet a következő pozícióra. (Ilyenkor a kígyó farkát nem mozgatjuk, hiszen épp oda helyezzük az új testrészt, ahová amúgy azt húzná...)

7. Tesztelés

Készítsük el a snake.tests.ToApplesTest teszt osztályt, melyben leteszteljük a Game osztály toApples metódusát JUnit 4 egységtesztekkel. Más metódus és osztály tesztelése nem szükséges a beadandó teljesítéséhez.

A JUnit futtatásához szükséges két .jar állomány megtalálható a [letölthető segédlet](http://people.inf.elte.hu/koguaai/java/snake.zip)ben.

A tesztelésnek alaposnak kell lennie, legalább 10 különböző teszteset implementálásával.

A tesztek megalkotásakor gondoljunk az alábbi esetekre:

* null paraméter,
* üres lista,
* egy szöveg szóközzel kezdődik/végződik, vagy több szóközt tartalmaz,
* a sor-/oszlopindex nem szám,
* a sor-/oszlopindex negatív vagy 10-nél nem kisebb,
* határesetek (pl. 0, 9, 10),
* helyes bemenet: a ciklus egyszer/kétszer/többször lefut.

A kód futtatása

A feladathoz tartozó [letölthető segédlet](http://people.inf.elte.hu/koguaai/java/snake.zip)ben található Main.java program, valamint mintabemenetek és mintakimenetek segítségével futtatható és ellenőrizhető a program. A szintén a zip-hez csatolt README.txt fájl tartalmaz részletes leírást mindezek használatáról.

Figyelem: a helyes kimeneti fájl még nem feltétlenül jelenti azt, hogy a megoldás helyes.

Jó munkát!