Java kollekció keretrendszer

Készítette: Goldschmidt Balázs, BME IIT, 2019.

A feladat célja egy sörnyilvántartó alkalmazás készítése, amely a szabványos bemeneten fogad parancsokat, és a szabványos kimenetre írja ki az eredményt. A parancsokat soronként olvassa és dolgozza fel. Amikor az "exit" parancsot kapja, kilép.

A feladatok megoldásához felhasználandó osztályok leírásait az alábbi URL-en találjuk meg:

http://download.oracle.com/javase/8/docs/api/

1 Tárolandó adatszerkezet kialakítása

a) Készítsünk egy sör (*Beer*) osztályt, amelynek van név (*name, String*), jelleg (*style, String*), alkoholfok (*strength, double*) attribútuma!
b) Az attribútumokat lehessen konstruktorból inicializálni!
c) Az attribútumokhoz legyenek getter metódusok!
d) A toString() metódus írja ki az attribútumok értékét!
e) A főprogramban hozzunk létre 2 sört és írassuk ki őket a szabványos kimenetre!

2 Parancsok fogadása és feldolgozása

A nyilvántartás működéséhez az alkalmazásnak képesnek kell lennie arra, hogy menet közben további söröket lehessen felvenni. Ehhez az alkalmazás sorokat olvas, majd a sorokat a whitespace-ek mentén *String*-ekké töri. A töréshez használjuk a *String* osztály *split* metódusát! A *split* metódus paramétere legyen egy szóközt tartalmazó *String* ("␣"). Az így kapott tömb első *String*-je a parancs, a többi a parancs argumentuma.

Pl. ha a bemeneti sor:

```
"add Guinness stout 4.2"
```

akkor a *split* eredménye:

```
{"add", "Guinness", "stout", "4.2"}
```

a) Módosítsuk úgy a főprogramot, hogy az beolvas egy sort, majd a fenti módon széttöri, és kiírja a kapott tömb első elemét, majd azt, hogy hány elemű a tömb.
b) Módosítsuk tovább a főprogramot: az alkalmazás ciklusban olvassa a sorokat, és csak akkor lépjen ki, ha a beolvasott sor első szava "exit".

3 Kollekció alapok

Az alkalmazást bővítsük új sörök bevitelét és a meglevők listázását lehetővé tevő parancsokkal! Az első két parancsa az **add** és a **list**.

a) Az **add** parancs hatására új sör (*Beer*: név (*name*), jelleg (*style*), alkoholfok (*strength*)) jön létre, és adódik a meglevők listájához, pl*:*
   - add Guinness stout 4.2
   - add Leffe_bruin brown_ale 6.5
   - add Dr_Eher pilsner 5.2

   A sörök tárolására használjunk *ArrayListet*! Figyeljünk arra, hogy az *ArrayList* a parancsokat megvalósító metódusok számára elérhető legyen! Ha a metódusaink statikusak, akkor az *ArrayList* is legyen az.

b) A **list** paranccsal a nyilvántartás tartalma íratható ki (tetszőleges sorrendben).

Minden parancshoz egy saját függvényt kell implementálni. A függvények fejléce a következő mintát kövesse (ahol a "fun" helyett az adott parancs neve áll):

```
protected void fun(String[] cmd)
```

A parancsok feldolgozása a következő módon történjen. Készíteni kell egy *if-else if-...* sorozatot, ahol az egyes *if*-ek azt ellenőrzik, hogy a beolvasott parancs (az előző pontban előálló tömb első eleme) egy adott előre definiált paranccsal egyezik-e. Ha igen, akkor meghívja a parancsot megvalósító függvényt. Pl:

```
if ("add".equals(cmd[0])) {
      add(cmd);
}
```

Haladó szintű szorgalmi feladat:

A parancsokat megvalósító metódusok helyett a parancsokat implementáljuk úgy, hogy készítünk egy *Command* absztrakt osztályt, aminek két metódusa van: `String getName()` és `void action(ArrayList<Beer> beers, String[] cmd)`. A konstruktora beállítja a nevet, az *action* pedig absztrakt.
A parancsokhoz külön-külön osztályokat hozzunk létre, amelyek *Command*-ból öröklődnek és megvalósítják *action*-t. Pl. az *Add* osztály *getName* metódusa visszaadja, hogy "add", az *action* metódusa a *beers* listába teszi a *cmd* alapján létrehozott új sör objektumot.
A főprogramban egy listába (*ArrayList<Command>*) felvesszük az összes parancsot mint objektumot, és ebben keressük a bemeneten olvasott parancshoz tartozót (*getName()* segítségével).
(A nagyon haladók egy *HashMap<String,Command>*-ba is gyűjthetik a parancsokat, ekkor a parancs kiválasztása egyszerűbb. A *HashMap* használata csak későbbi előadáson fog előkerülni.)

4 Comparator

Bővítsük az alkalmazás listázó parancsát (**list**)! A listázás sorrendjét határozza meg a parancs opcionális argumentuma:

name - név
style - jelleg
strength - alkoholfok

A feladat megoldásához készítsünk minden rendezésfajtához egy-egy *java.util.Comparator* implementációt (*NameComparator*, *StyleComparator*, *StrengthComparator*). A lista rendezéséhez használjuk a *java.util.Collections* osztály megfelelő metódusát!

Szorgalmi feladat: ha több opcionális argumentum is van, akkor a rendezés kombinált legyen, pl: *"list style name"* parancs esetén az azonos jellegű söröket névsorrendben listázzuk.

5 Szűrés

a) Bővítsük az alkalmazást név szerinti kereséssel (**search** parancs). Ez paraméterként kapjon egy String-et, és csak azokat a söröket listázza ki, amelyek neve egyezik a paraméter értékével. Használjuk a for-each ciklust az elemeken való iteráláshoz!
b) Bővítsük az alkalmazást szabadszöveges kereséssel (**find** parancs). Ez paraméterként kapjon egy String-et, és csak azokat a söröket listázza ki, amelyek nevében szerepel a paraméter értéke! Pl. ha *alma*, *körte*, *barack* nevű söreink vannak, akkor a "**find a**" parancs kilistázza az *alma* és a *barack* nevű söröket.

6 Szűrés 2

A *search* és a *find* parancsok is kaphassanak extra paramétert, ami alapján keresnek:

name - név
style - jelleg
strength - alkoholfok (*search*: pontos egyezés, *find*: legalább ekkora érték kell)
weaker - alkoholfok (csak *find* esetén: maximum ekkora érték lehet)

Pl. *"find style pils"* ereménye minden olyan sör, aminek a jellegében szerepel a *"pils"* sztring.

7 Törlés

Az alkalmazáshoz készítsen törlő parancsot (**delete**), ami a neve alapján töröl egy sört. Használjunk iterátort a kereséshez és a törléshez!

Szorgalmi feladat: a kereséshez használja a *java.util.Collections.binarySearch* metódust az API leírásban megadott módon.