Java IO labor

Készítette: Goldschmidt Balázs, BME IIT, 2015.

A. A fájlrendszer elérése

A *java.io.File* osztály felhasználásával írjon a fájlrendszert bejárni képes Java alkalmazást, amely a standard bemenetről fogadja a felhasználói parancsokat!
A File osztály metódusai elérhetők az alábbi URL-en:

http://docs.oracle.com/javase/8/docs/api/java/io/File.html

1. Parancsok fogadása

A szabványos bemeneten érkező parancsok feldolgozása a következő elven valósuljon meg.
Az alkalmazás sorokat olvas, majd a sorokat a whitespace-ek mentén stringekké töri. A töréshez használjuk a *String* osztály *split* metódusát! A *split* metódus paramétere legyen egy szóközt tartalmazó *String* ("␣"). A *split* által visszaadott eredmény-tömb első *String*-je a parancs, a többi a parancs argumentuma.

Pl. ha a bemeneti sor:

"egyedem begyedem tengertánc"

akkor a *split* eredménye a következő tartalmú tömb lesz:

{"egyedem", "begyedem", "tengertánc"}

A parancsok értelmezése az alábbi módon történjen. Készíteni kell egy *if-else if-...* sorozatot, ahol az egyes *if*-ek azt ellenőrzik, hogy a parancs (az előző pontban létrehozott tömb első eleme) egy adott paranccsal egyeznek-e. Ha igen, akkor végrehajtja a parancsoz tartozó műveleteket. Pl:

```
if (cmd[0].equals("hello")) {
      System.out.println(“Hello world!”);
}
```

Készítsünk egy főprogramot (Main), amelyik végtelen ciklusban, sorokat olvasva parancsokat tud fogadni és feldolgozni. Az első parancs legyen az „exit”, aminek a hatására a program álljon le (a *System.exit* metódus segítségével)!

2. Parancsfeldolgozás

A parancsok végrehajtsához érdemes minden parancshoz egy-egy saját függvényt 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 ezután már egyszerű. Az előző pontban leírtaknak megfelelő *if-else-if…* sorozatot kell kibővíteni egy olyan új ággal, ami az újonnan megvalósított parancsra komparál. Ha ez sikeres, akkor meghívja a parancsot megvalósító függvényt. Pl:

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

3. Fájlrendszer alapjai

a) Készítsen egy parancsot (*pwd*), amely kiírja, hogy a programunk melyik mappából indult!

Az alkalmazás indulásakor a kezdő könyvtár nevét a "user.dir" rendszerbeállítás tartalmazza. Ennek lekéréséhez hívjuk meg a *System.getProperty()* metódust a fenti string paraméterrel! A visszatérési érték annak a mappának a neve, amelyikben éppen vagyunk.

b) A *pwd*-t bővítsük ki! A mappa neve után írjuk ki a mappában található fájlok és almappák darabszámát is!

Ehhez létre kell hozni egy File osztályú objektumot (*dir*). A konstruktornak paraméterként adjuk meg az előző pontban megkapott mappanevet.

A *dir* objektumon hívjuk meg a *listFiles* metódust, ami egy tömbben visszaadja a fájlokra és mappákra hivatkozó *File* objektumokat. Ennek a tömbnek a mérete a keresett érték.

4. Fájlrendszer bejárása

Ahhoz, hogy a programunk teljes értékűvé váljon, tudnia kell, hogy melyik mappa (könyvtár) az éppen aktuálisan kezelendő (ez megegyezik azzal a modellel, ahogy pl. a Windows Böngésző dolgozik: mindig egy konkrét mappa tartalmát mutatja, ebben a mappában tudunk almappákba kerülni vagy szülőmappába visszalépni.)

Fentiek miatt szükség lesz egy *File* típusú attribútumra (wd, *working directory*), amely tárolja, hogy éppen melyik az aktuális munkakönyvtár (directory, mappa, folder). Attól függően, hogy a függvényeink statikusak-e, ennek az attribútumnak is statikusnak kell lennie.

Fontos: innentől a *wd* ne lokális változó, hanem a *Main* osztály egy attribútuma (tagváltozója) legyen!

Ha az objektum-alapú megoldást alkalmazzuk a parancsok implementálására, akkor ezeknek az objektumoknak valahogyan meg kell kapniuk a wd *értékét.*

Az alkalmazás indulásakor a kezdő könyvtár nevét a "user.dir" rendszerbeállításból vegyük (*System.getProperty()* metódus adja vissza a nevet, ha paraméterként a fenti stringet adjuk át). Ennek segítségével példányosítsuk a *wd* által mutatott *File* típusú objektumot!

Írjunk egy parancsot („*ls*”), ami kilistázza a *wd* által mutatott mappa tartalmát!

5. Mappa-váltó parancsok

A továbbiakban mindig a *wd* attribútum lesz a kiindulási pontunk. Amikor könyvtárat váltunk, a *wd* értékét kell felülírnunk. Amikor egy adott nevű (pl. "szoveg.txt") fájlt el akarunk érni, akkor is ebből a könyvtárból indulunk ki. Pl.:

```
File f = new File(wd, "szoveg.txt");
```

A fentieknek megfelelően implementáljuk a következő, a mappákban való járkálást lehetővé tevő parancsokat!

- `pwd`: kiírja az aktuális könyvtár (*wd*) elérési útját (*getCanonicalPath()*)
- `cd <dir>`: az aktuális könyvtárból átlép a benne levő, <dir> nevű alkönyvtárba. Ha <dir> értéke "..", akkor egy szinttel feljebb lép (*getParentFile()*).
  Ha a <dir> nem létező könyvtár, akkor írjon ki hibaüzenetet! A parancs fogja-e módosítani a *wd* értékét?
- `ls`: javított listázás, amely kilistázza az aktuális könyvtárban levő fájlok és könyvtárak neveit, de van egy lehetséges paramétere:
  - `-l`: listáz, mint eddig, de a listában megjeleníti a fájlok méretét és típusát is (d - könyvtár, f - sima fájl)

A parancsok implementálásához a *System.setProperty()* metódus használata tilos.

6. Fájl-kezelő parancsok

A mappák bejárása után az alábbi, egyedi fájlokat kezelő műveleteket kell megvalósítani.
(Segítségként minden parancshoz megadjuk a *File* osztályban használandó metódus nevét.)
A *System.setProperty()* metódus használata tilos. Ne felejtse el bezárni a megnyitott fájlokat!

- `rm <file>`: törli a <file> nevű fájlt. (*delete()*)
  Ha probléma merül fel, akkor adjon hibajelzést.
- `mkdir <dir>`: létrehozza az aktuális könyvtárban a <dir> nevű könyvtárat (*mkdir()*)
  Ha <dir> már létezik, írjon ki hibaüzenetet!
- `mv <file1> <file2>`: <file1> fájlt átnevezi <file2>-re (*renameTo()*)
  Ha hiba történt, jelezze!
- `cp <file1> <file2>`: <file1>-et átmásolja <file2>-be. Használja a *FileInputStream* és *FileOutputStream* osztályokat, és bájtonként másolja át a tartalmat. Ha ideje engedi, próbáljon blokkos másolást[1].
  Ha a fájl nem létezik, adjon hibajelzést!
- `cat <file>`: kiírja a <file> nevű fájl tartalmát soronként a szabványos kimenetre. Használjon *FileReadert* és *BufferedReadert*!
  Ha a fájl nem létezik, adjon hibajelzést!

[1] Blokkos másolás esetén egyszerre nem csak egy-egy, hanem nagyobb mennyiségű bájtot másolunk (pl. 1024-et), egészen addig, amíg van beolvasható bájt.

Szorgalmi feladat:

- *BufferedReader* helyett próbálja ki a *Scanner*-t! (*hasNextLine* és *nextLine* metódusok)
- `length <file>`: kiírja a <file> nevű fájl hosszát.
  Ha a fájl nem létezik, adjon hibajelzést!
- `head -n <n> <file>`: kiírja a <file> nevű fájl első *<n>* sorát. Ha az opcionális *-n* paraméter hiányzik, *<n>* értéke legyen 10. Építsen a *cat* parancs megoldására.
  Ha a fájl nem létezik, adjon hibajelzést!
- `tail -n <n> <file>`: kiírja a <file> nevű fájl utolsó *<n>* sorát. Ha az opcionális *-n* paraméter hiányzik, *<n>* értéke legyen 10. Építsen a *head* parancs megoldására.
  Használja a *List* interfészt megvalósító *LinkedListet*!
  Ha a fájl nem létezik, adjon hibajelzést!
- `wc <file>`: kiírja a <file> nevű fájl statisztikai adatait: sorok száma, szavak száma, betűk száma. Használja a *String.split* metódust! Induljon ki abból, hogy a *cat* parancsot hogyan valósította meg!
  Ha a fájl nem létezik, adjon hibajelzést!
- `grep <pattern> <file>`: kiírja a <file> nevű fájl tartalmából a <pattern>-re illeszkedő sorokat. Használja a *String.matches* metódust! A minta elejére és végére helyezze el a „.*” karaktereket.
  Ha a fájl nem létezik, adjon hibajelzést!

B. Sorszűrő alkalmazás

7. Egyszerű sorszűrő

Készítsen sorszűrő Java alkalmazást!
Az alkalmazás a standard bementről olvas sorokat, és a standard kimenetre kiírja azokat, amelyek egy adott szövegmintának megfelelnek (*String.matches()*). A mintát az első parancssori opcióként vegye át!

8. Parancssori opciók

Írja át az 5. feladat alkalmazását úgy, hogy parancssori opcióként megadott fájlokra is működjön!
A programnak három opciója legyen: *-p <minta>*, *-i <file1>* és *-o <file2>*, amiket tetszőleges sorrendben meg lehet adni. Az opciók feldolgozáshoz alkalmazhatjuk a következő programrészletet:

```
String input = null;
String output = null;
String pattern = "";
for (int i = 0; i < args.length; i++) {
     if ((i+1 < args.length) && args[i].equals("-i")) {
          i++;
          input = args[i];
     } else if ((i+1 < args.length) && args[i].equals("-o")) {
          i++;
```

```
            output = args[i];
        } else if ((i+1 < args.length) && args[i].equals("-p")) {
            i++;
            pattern = args[i];
        }
    }
```

9. Tömörített fájlok

Bővítse ki a fenti alkalmazást úgy, hogy ha `-gi` vagy `-go` opciót is kap, akkor gzip tömörítéssel tömörített fájlból olvas illetve ír (*java.util.zip* csomagban *GZIPInputStream* és *GZIPOutputStream*).

Pl.: `-i hello.txt -p java -o bello.txt.gz -go` opciók esetén a tömörítetlen hello.txt-ből olvas, a "java" tartalmú sorokat írja gzip tömörítéssel a bello.txt.gz fájlba.