Feladattár

Informatika ismeretek érettségire felkészítő feladatok és Java nyelvű megoldásaik

Készült a GINOP 6.2.3 projekt keretében a
Nyíregyházi Szakképzési Centrum Széchenyi István
Közgazdasági, Informatikai Szakgimnáziumában

Készítette:
Komoróczy Tamás
Huri Gábor

2019

Bevezető

Ez a feladattár nem a hagyományos módon készült feladatgyűjtemény. Célunk az volt a létrehozásával, hogy az Informatika ismeretek érettségi vizsgához mintákat adjon, így a feladatok nehézsége, bonyolultsága, vagy a megoldáshoz szükséges idő mennyisége nem összevethető az érettségi programozás feladatával. Arra törekedtünk, hogy a feladatok megoldása során használandó eszközök, algoritmusok lefedjék a vizsgakövetelmények legfontosabb részeit. Ezért a feladatok sikeres megoldása szükséges, de nem elégséges feltétele az érettségin a programozás feladat sikeres megoldásának!

A feladatok a középszintű és emelt szintű vizsgákra való felkészülés során is ajánlottak.

A feladatgyűjtemény két részből áll. Az elsőben a kitűzött feladatok esetében, lépésenként segítséget nyújtunk rávezető kérdésekkel, kódrészletekkel, illetve teljes kóddal is. A teljes megoldást is közreadjuk letölthető formában.

A lépésekhez magyarázatok tartoznak, így különböző utakat bejárva bővíthetik a tudásukat a diákok, vagy csak egyszerűen ellenőrizhetik az önálló munkájukat.

A második részben olyan feladatok találhatóak, amelyek már közelebb vannak az érettségi feladatokhoz, azonban ezekre nem adunk megoldást, azt az olvasóra bízzuk.

Célunk volt, hogy az érettségin mindkét nyelvet tanulók kapjanak támogatást, de elsősorban a Java nyelvet preferáljuk. A C# nyelvű megoldásokat a feladattár honlapján, lehetőségeinket figyelembe véve tervezzük közreadni.

A megoldások során támaszkodunk a következő főbb ismeretkörökre programozásból:

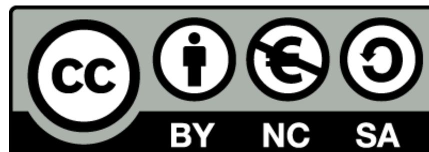
- Primitív típusú változók és a velük végezhető műveletek
- String típus és alapvető metódusai
- Adatbevitel konzolról
- Alapvető vezérlési szerkezetek
- Állomány műveletek. Java esetében a RandomAccessFile osztály használatát tételezzük fel. Kiegészítő ismeretként más megoldás is megadhatunk.
- Tömbök használata, beleértve a tömbök tömbjét is
- Elemi algoritmusok
- Saját metódusok készítése

- Osztályok használata. (OOP szemlélet figyelembe vétele)
- Kivételkezelés
- Gyűjtemények, kiemelve a következőket:
    - ArrayList használata
    - TreeSet használata

A források letölthetők feladatonként a következő helyről:

A feladattár a GINOP 6.2.3 keretében jött létre és a Nyíregyházi Szakképzési Centrum a jogtulajdonos.

This work is licensed under the Creative Commons Nevezd meg! - Ne add el! - Így add tovább! 2.5 Magyarország License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/hu/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



A megoldásokban használt jelölésekről

Az ilyen dobozokban fogjuk elhelyezni a kóddal kapcsolatos megjegyzéseinket

Az ilyen dobozokban fogjuk elhelyezni az algoritmusokkal kapcsolatos megjegyzéseinket

Az ilyen dobozokban a rávezető kérdéseket fogjuk megadni

Az ilyen dobozokban pedig a veszélyhelyzetekre hívjuk fel a figyelmet.

Hatoslottó

A hatoslottó 205 és 2018 között kihúzott számait a **`hatos.csv`** állomány tartalmazza. Az állomány néhány sora így néz ki:

```
2018.12.30;21;26;28;30;38;41
2018.12.23;12;26;28;34;37;42
```

Látható, hogy az adatokat pontosvessző választja el. Elsőként a húzás dátuma szerepel. A dátumban a szeparátorkarakter a pont. Ezt követi növekvő sorrendben a kihúzott 6 szám. Hatos lottóban a legalább 3 számot eltaláló tippek nyernek.

A megoldásnál vegyük figyelembe, hogy a felhasználóval való kommunikáció nyelve magyar, ezért a nyelvtanilag helyesen történjen a kommunikáció, bár az ékezetmentes kiírás is elfogadott! A mintán található feliratok, ha a feladat szövege mást nem mond, csak iránymutatásként szerepelnek.

A megoldás során a „Tiszta kód" elveket tartsuk be, de nem feltétlenül kell megjegyzéseket fűzni a kódhoz!

A megoldás során törekedjünk megfelelő metódusokat használni! A megoldás során az osztály tulajdonságait az adott osztályon belül valósítsd meg!

1. Hozd létre a **`hatoslotto`** nevű grafikus alkalmazást a következő feladatok megoldására. A felületen helyezz el egy gombot, amellyel az alkalmazásból normál módon lehet kilépni!
2. Készíts egy nyomógombot, amelyet aktiválva a forrásállomány adatait egy arra megfelelő adatszerkezetbe beolvassa.
3. Az adatokat jelenítsd meg tetszőleges formátumban. A mintán láthatsz egy lehetőséget!
4. Készíts egy szövegbeviteli mezőt, amelyben a saját számaid tudod megadni szóközzel elválasztva!
5. A számok ellenőrzésre hozz létre kivételkezelő osztályokat! Kivétel esetén, a hiba jellegének megfelelő párbeszéd ablakban értesítsd a felhasználót!
6. A mintán látható elrendezésben helyezz el egy nyomógombot, amely megnyomására a helyesen bevitt lottószámok sorba rendezve jelennek meg, ez alatt pedig add meg, hányszor és milyen találatot értél volna el! Kövesd a mintát.

7. A mintán látható helyen jelenítsd meg a **`szerencsemalac.png`** képet. Ha fölé viszed a kurzort, jelenjen meg a sok szerencsét felirat piros színnel a Hatos lottó címfelirat alatt!

Minta:

Az alábbi mintán látszik, hogy az alkalmazás ikonját lecseréltük. Ez nem tananyag, azonban a megoldás során megmutatjuk, hogyan kell az alapértelmezett ikont lecserélni!

Hatos lottó

Hatos lottó

Heti nyerőszámok 2005-2018

Összes húzás

sorok száma: 731

6 szám szóközzel elválasztva

Saját tipp

húzás hete	nyerőszámok
2018.12.30	21 26 28 30 38 41
2018.12.23	12 26 28 34 37 42
2018.12.16	4 7 15 17 28 40
2018.12.09	19 22 26 27 32 45
2018.12.02	4 24 26 30 40 45
2018.11.25	8 20 21 28 38 39
2018.11.18	8 21 26 30 39 40
2018.11.11	3 16 20 28 35 36
2018.11.04	1 4 10 34 42 45
2018.10.28	1 10 11 31 34 38
2018.10.21	1 16 30 33 41 44
2018.10.14	1 3 11 20 26 33
2018.10.07	9 13 19 24 36 39
2018.09.30	13 16 17 24 33 43

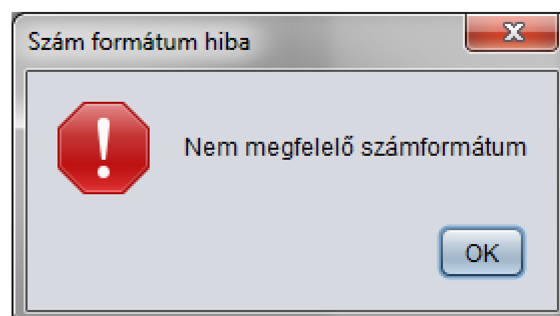
4 7 15 17 18 19

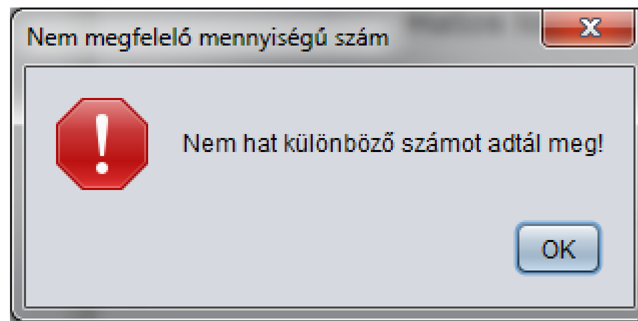
Kiértékelés

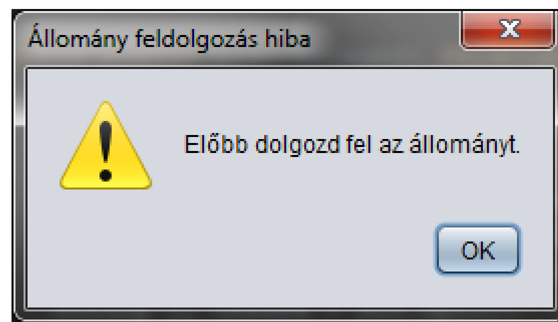
4 7 15 17 18 19

6 találat: 0 db
5 találat: 0 db
4 találat: 2 db
3 találat: 16 db

Kilépés




Nem megfelelő mennyiségű szám
Nem hat különböző számot adtál meg!
OK


Állomány feldolgozás hiba
Előbb dolgozd fel az állományt.
OK

Hatoslottó megoldás

Az objektum-orientált programozás lépései:

- adatszerkezetek kialakítása
- algoritmizálás
- kódolás
- tesztelés

E feladat megoldásának bemutatása során ezen lépéseket követjük. A felhasználói technikára helyezzük a hangsúlyt, nem pedig a megoldás logikai menetére.

A tesztelés az elkészült részfeladatok működésének tesztelését, hibák keresését, javítását jelenti, nem pedig egységtesztek készítését és tesztesetek vizsgálatát, mert az nem része az érettségi követelménynek.

A feladat megoldásának menete eltérő is lehet az általunk ajánlottól, azonban vannak olyan lépések, amelyeket az egymásra épülés miatt mindenképp korábban kell megtenni, mint másokat.

1. lépés

Olvassuk el és értsük meg a feladatot. Amennyiben nem sikerült a feladat megértése elsőre, érdemes újra, akár többször is átolvasni.

2. lépés

Készítsük elő a szükséges projektet! Készítsünk `JFrame`-et tetszőleges néven és állítsuk be, hogy itt lesz a fő `main()` metódusunk!

3. lépés

Érdemes létrehozni egy `kepek` vagy `forras` nevű csomagot és bemásolni a képet ebbe a mappába.

Az ablak ikonjának beállításához hozzuk létre a következő metódust a forrásunkban:

```
private void setIcon(String url){
    this.setIconImage(Toolkit.getDefaultToolkit().getImage(url));
}
```

Keressük meg a `JFrame` osztályt kiterjesztő osztályunk konstruktorát és a következő sorban hívjuk meg az előbb létrehozott metódust!

```
setIcon("src/forras/clover.png");
```

Kiegészítő anyag.
Érdekességként mutattuk be!

4. lépés

Tervezzük meg a felületet. Helyezzük fel a szükséges grafikus komponenseket: `JButton`, `jLabel`, `JTextField`, `JTable`! Állítsuk be a kívánt feliratokat és a megfelelő `JLabel`-be helyezzük el a malac képét! A megoldáshoz nem szükséges, de célszerű az

alapértelmezett objektum neveket átírni a Tiszta kód elveinek megfelelően. Saját munkánkat megkönnyíti, az esetleges hibakeresést jelentősen meggyorsítja, ha azoknak a grafikus vezérlőknek, amelyek tulajdonságaikat (jellemzően a tartalmukat) a program futása során dinamikusan változtatják, állítsunk be olyan változónevet (Code fül / Variable Name), ami alapján a lehető legkönnyebben beazonosíthatjuk őket.

A pig.png képhez kattintsunk a megfelelő `JLabel`-re jobb gombbal és válasszuk a tulajdonságok (Properties) lehetőséget
Keressük meg és állítsuk be az icon tulajdonságnak a pig.png képet

Állítsuk be a grafikus elemek további tulajdonságait! Figyeljük meg, hogy a `JFrame`-nek is adtunk címkét!

5. lépés

A Kilépés gombra az actionPerformed (Events fül) eseményre állítsuk be a megfelelő, kilépéshez szükséges kódot! (Design nézetben a gombra történő dupla kattintás automatikusan létrehozza ennek a metódusnak a fejrészét)

```
System.exit(0);
```

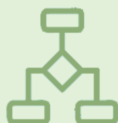
6. lépés

Tervezzük meg az adatszerkezeteket! Ki kell alakítani a megfelelő adatok tárolásához szükséges változókat.

A feladat megoldásához használjuk ki az OOP lehetőségeit. Az objektum-orientált szemléletmód szerint a forrásadatok alapján osztály(oka)t készítünk, mely(ek)nek tulajdonságai és viselkedései vannak.

A tulajdonságokból (más megközelítésből állapotok) tagadatok lesznek, a viselkedésekből (szolgáltatás) metódusok.

Általában – ebben a feladatban is – több, azonos típusú adathalmazt kell feldolgoznunk. Ez a gyakorlatban egy adattábla vagy (mint jelen esetben) egy külső állomány több sorát jelenti, melyekből több példányt kell létrehozni. Ezeket el kell tárolnunk valamilyen tömbben, vagy listában.



Az osztály(ok) neve; a tagadatok típusa, neve; a példányok tárolását megvalósító változó(k) típusa és neve lesznek azok a stratégiailag fontos alapváltozók, melyeket legelőször meg kell határoznunk.

A feladat megoldásához ezeken kívül több, úgynevezett segédváltozóra is szükség lesz, de ezeket az adott funkció programozása közben fogjuk definiálni.

Mi legyen az osztály neve? Milyen tagadatok kerüljenek bele.
Ezek hogyan kapnak értéket?

Az osztály neve a példánkban legyen `HetiNyerőszám`, mivel ilyen adatokat szerepelnek a külső állomány soraiban.

A tagadatok típusa és neve legyen: `String dátum` és `int[] nyerőszámok`

Összetartozó, azonos típusú adatok esetén mindig használjunk tömböt, vagy gyűjteményt! Jelentős mértékben megnehezítené a munkánkat, ha ezen a ponton 6 db `int` típusú tagadatot deklarálnánk az osztályban.

A tagadatok védelme érdekében `private` láthatósági szinttel deklaráljuk azokat. Osztályon kívüli elérésüket hozzáférő metódusok segítségével oldjuk meg. A megoldás természetesen nem változik meg, ha az alapértelmezett `public` láthatóságot használjuk, azonban fordítsunk figyelmet a Tiszta kód elvű programozásra is!

A feladat típusából adódóan a tagadatok értékei már a forrásban rendelkezésre állnak, , ezért a konstruktorban inicializáljuk azokat. Mivel menet közben nem változtatják értékeiket, akár `final`-ként is deklarálhatnánk, és ez nem befolyásolná a megoldást.

Egy külső állomány beolvasott soraiból képzett `String` típusú adat alapján hogyan tudja a konstruktor inicializálni a tagadatokat?

A sorokat fel kell darabolni egy szeparátor string mentén.

```
public HetiNyerőszám(String sor {
    String [] szeletek = sor.split(";");
    this.dátum=szeletek[0];
    for(int i=1; i< szeletek.length; i++)
        this.nyerőszámok[i-1]=Integer.parseInt(szeletek[i]);
}
```

Nem zárhatjuk ki azt a lehetőséget, hogy az adatok nem úgy érkeznek, ahogy szeretnénk. A karakterláncból számmá alakítás futás közbeni hibát (kivételt) okozhat, ha nem numerikus karaktereket kellene átalakítani.

Tesztelés vagy hibakeresés során hasznos lehet, ha a túlterhelés lehetőségeit kihasználva több konstruktort is létrehozunk más-más paraméterekkel.

7. lépés

A saját osztályunk példányainak sorozatát milyen adatszerkezetben tároljuk el?

Lehetőségként felmerül a tömb és a dinamikus lista. Mi az utóbbit preferáljuk számos előnye miatt.
A választás tehát, listában: `List<HetiNyerőszám> nyeroszamok`.

8. lépés

A beolvasás algoritmizálása mindig a szokásos, tanult módon történik. Mi most a `RandomAccessFile` osztályt használjuk, de emellett számos megoldás létezik még.

```
public static List<HetiNyerőszám> beolvasas(String file){
    Ellenőrző.számol = 1;
    List<HetiNyerőszám> lista = new ArrayList<>();
    try {
        RandomAccessFile beolvas = new RandomAccessFile(file, "r");
        while( beolvas.getFilePointer() < beolvas.length() ){
            lista.add(new HetiNyerőszám( beolvas.readLine() ) );
        }
        beolvas.close(); // A program és a fájl közötti kapcsolat
bezárása
    } catch (IOException ex) {
        JOptionPane.showMessageDialog(null,
exc.getMessage(),"Állomány hiba",JOptionPane.ERROR_MESSAGE);
        System.err.println("Hiba: "+ex.getMessage());
    } finally{
        return lista;
    }
}
```

9. lépés

Milyen komponensek alkalmasak adatok listájának megjelenítésére?

Mi a `JTable` osztály egy példányát választottuk, de a `JList`, a `JComboBox`, vagy a `JTextArea` is alkalmas a sorozatok adatainak megjelenítésére a grafikus felületen.

```
    private void tablafeltolto(List<HetiNyerőszám> list){
        DefaultTableModel dtm = new DefaultTableModel();
        String[] oszlopnevek = {"húzás hete", "nyerőszámok"};
        dtm.setColumnIdentifiers(oszlopnevek);
        String hatszam;
        for (HetiNyerőszám hetiNyeroszam : list) {
            hatszam="";
```

```
            for (int i = 0; i <
hetiNyeroszam.getNyerőszámok().length; i++){
                if(hetiNyeroszam.getNyerőszámok()[i]<10)
                    hatszam = hatszam.concat("  "); // plusz szóköz
az egyjegyűek elé
                hatszam =
hatszam.concat(hetiNyeroszam.getNyerőszámok()[i]+" ");
             }
            dtm.addRow(new String[]{hetiNyeroszam.getDátum(),
hatszam});
        }
        this.eredmenyTabla.setModel(dtm);
        int[] oszlopSzelessegek = {120, 180};
        for (int i = 0; i < oszlopSzelessegek.length; i++)
            this.eredmenyTabla.getColumnModel().getColumn(i).
setPreferredWidth(oszlopSzelessegek[i]);
    }
```

A `JTable` példánynak nincs olyan metódusa, amivel a táblázat sorait fel lehetne tölteni. Minden táblának van azonban egy alapértelmezett modellje, ami meghatározza az oszlopok számát és az oszlopcímeket. Az eredeti modellt le kell cserélnünk egy sajátra, s ennek segítségével tudjuk a táblázat tulajdonságait megadni, és adatokkal feltölteni.

10. lépés

A felhasználó tippjeinek az elvárttól eltérő módon történő esetleges megadása milyen futás közbeni hibákat okozhat?

Lehet, hogy tévedésből, vagy szándékosan a felhasználó túl kevés számot ad meg. Az is előfordulhat, hogy nem számformátumú karakterek kerülnek feldolgozásra. Az ilyen események hatására a Java program működése leállna.

A szintaktikai hibákra még a futtatás előtt felhívja figyelmünket a fejlesztő környezet. A futás közbeni logikai hibákat, viszont nekünk kell lekezelnünk. Erre a legmegfelelőbb megoldás a saját kivételosztály(ok) készítése.

Az egyik kivételkezelő osztály kódja:

```
public class KevésSzámException extends RuntimeException{
    public KevésSzámException() {
    }
    public KevésSzámException(String message) {
        super(message);
    }
}
```

11. lépés

A kiértékelés végrehajtása előtt fel kell dolgoznunk a felhasználó által megadott adatokból eredő esetleges hibákat

A feldolgozást végző metódus első része:

```
    public static List<Integer> feldolgoz(String sor) throws
KevésSzámException, HibásSzámException{
        if(Ellenőrző.számol == 0)
            JOptionPane.showMessageDialog(null, "Előbb dolgozd fel
az állományt.","Állomány feldolgozás
hiba",JOptionPane.ERROR_MESSAGE);
        if(sor.length() == 0)
            throw new KevésSzámException("Nem adtál meg semmit!");
        if(sor.length() != sor.replaceAll("[^0-9 ]", "").length())
            throw new HibásSzámException("Nem megfelelő
számformátum");
        ...
```

A kivételek lekezelését elhalasztottuk a `feldolgoz()` metódust meghívó metódusba.

Melyik az az utolsó metódus, amelyikben már nem halasztható tovább a kivételek elkapása?

A kivételek elkapásának legkésőbb a `main()` metódusban meg kell történnie.

A `feldolgoz()` metódus további része:

```
        ...
        String [] részek = sor.split(" ");
        TreeSet<Integer> temp = new TreeSet<>();
        for(String elem : részek){
            temp.add(Integer.parseInt(elem));
        }
        if(temp.size() != 6)
            throw new KevésSzámException("Nem hat különböző számot
adtál meg!");
        List<Integer> result = new ArrayList<>();
        result.addAll(temp); //És még rendezve is van!!!
        return result;
    }
```

A `TreeSet<Integer>` ideális adatszerkezetet a felhasználó tippjeinek emelkedő sorrendben történő kiírásához és a tippek számának ellenőrzéséhez

Kihasználjuk a `TreeSet<>` két hasznos tulajdonságát. Egyrészt emelkedő sorrendbe teszi a sorozat elemeit, másrészt nem fordulhat elő benne kétszer ugyanaz az érték.

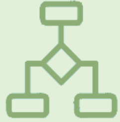
12. lépés

A felhasználó tippjeinek kiértékelése előtt a kivételek elkapása és a várható hibák lekezelése:

```
    private void
kiertekelesGombActionPerformed(java.awt.event.ActionEvent evt) {
        this.valasz.setText("");
        this.valasz.setBackground(Color.red);
        String szamok = this.tippTextField.getText();

        //Hibafeldolgozás!!!!
        List<Integer> szamokKT = null;
        try{
             szamokKT = feldolgoz( this.tippTextField.getText() );
        }catch(HibásSzámException kivétel){
            JOptionPane.showMessageDialog(null,
kivétel.getMessage(),"Szám formátum
hiba",JOptionPane.ERROR_MESSAGE);
        }
        catch(KevésSzámException kivétel){
            JOptionPane.showMessageDialog(null,
kivétel.getMessage(),"Nem megfelelő mennyiségű
szám",JOptionPane.ERROR_MESSAGE);
        }

        if(szamok.length()<1)
            this.valasz.setText("Nincs saját tipp");
        else {
            int szokozokSzama=0;
            for (int i = 0; i < szamok.length(); i++) {
                if(szamok.charAt(i)==' '){
                    szokozokSzama++;
                }
            }
            if(szokozokSzama!=5){
                this.valasz.setText("6 számra van szükség");
            }
            else {
                ...
```

13. lépés

A kiértékelés gombra kattintás – mint felhasználói interakció – metódusának folytatása, ha a kapott tippek minden kritériumnak megfelelnek:



A kapott szöveges adatokat számmá kell konvertálni, hogy sorba tudjuk rendezni azokat a feladat előírása szerint.

A sorozat sorba rendezésére használható a buborékrendező eljárás is, amikor az egymás melletti értékeket hasonlítjuk össze, majd felcseréljük azokat, ha kel. Így lépkedünk végig a sorozat elemein. Ezt az vizsgálatot és cserét n-1 alkalommal kell megtennünk, ahol n a sorozat elmeinek száma.

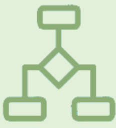
Példánkban mi a kiválasztásos rendezést valósítjuk meg, amelyben kiválasztunk egy pozíciót (pl a sorozat első indexét), és beletesszük a legkisebb elemet. A műveletet addig ismételjük, amíg végül minden pozícióba a megfelelő elem kerül.

Az előbbi metódus folytatása:

```
                ...
                String[] szovegtippek = new String[6];
                szovegtippek = szamok.split(" ");
                int[] szamtippek = new int[6];
                for (int i = 0; i < szovegtippek.length; i++) {
                    try {

szamtippek[i]=Integer.parseInt(szovegtippek[i]);
                    } catch(NumberFormatException e){
                        System.out.println("Nem számformátum");
                        szamtippek[i]=0;
                    }
                }

                int csere;
                for (int i = 0; i < szamtippek.length-1; i++) {
                    for (int j = i+1; j < szamtippek.length; j++) {
                        if(szamtippek[i]>szamtippek[j]) {
                            csere = szamtippek[i];
                            szamtippek[i]=szamtippek[j];
                            szamtippek[j]=csere;
                        }
                    }
                }

                String kiiratottSzamok="";
                for (Integer elem : szamokKT) {
                    kiiratottSzamok += elem + " ";
                }
                kiiratottSzamok = kiiratottSzamok.trim();

                this.valasz.setBackground(new java.awt.Color(51,
102, 0));
                this.valasz.setText(kiiratottSzamok);
                ...
```

14. lépés

Hogyan használhatjuk ki a `TreeSet<>` adatszerkezet használatának lehetőségeit a nyertes példányok kiválogatására?



Ha összefésüljük a felhasználó tippjeit a listában tárolt heti nyerőszámokkal, akkor a megkapott sorozat számai alapján eldönthető, hogy hány találatot ért volna el az adott héten a felhasználó tippje.

A három találattól többet elérő eseteket meg is kell számolnunk és kiíratnunk a feladatban előírt minta szerint egy `JList` komponensben.

A kiértékelést végző metódus befejezése:

```
                    ...
                    TreeSet<Integer> merged = new TreeSet<>();
                    int [] nyertesDarab = new int[7];
                    for(int i=0; i<nyertesDarab.length; i++)
                        nyertesDarab[i] = 0;
                    for(HetiNyerőszám elem: nyeroszamok)
                        nyertesDarab[elem.találatokSzáma(szamokKT)]++;

                    int talalat;
                    int nyert3=0, nyert4=0, nyert5=0, nyert6=0;
                    for (HetiNyerőszám hetiNyeroszam : this.nyeroszamok)
{
                        for (int i = 0; i <
hetiNyeroszam.getNyerőszámok().length; i++) {

merged.add(hetiNyeroszam.getNyerőszámok()[i]);
                        }
                        for (int i = 0; i < szamtippek.length; i++) {
                            merged.add(szamtippek[i]);
                        }
                        talalat = 12-merged.size();
                        if(talalat==6){
                            nyert6++;
                        }
                        else if(talalat==5){
                            nyert5++;
                        }
                        else if(talalat==4){
                            nyert4++;
                        }
                        else if(talalat==3){
                            nyert3++;
                        }
                        merged.clear();
                    }
                    System.out.println("6: "+nyert6);
```

```
                    System.out.println("5: "+nyert5);
                    System.out.println("4: "+nyert4);
                    System.out.println("3: "+nyert3);
                    DefaultListModel dlm = new DefaultListModel();
                    dlm.clear();
"+nyert3+" db");
                    for(int i=6; i>=3; i--)
                         dlm.addElement(i+" találat: "+nyertesDarab[i]+"
db");
                    this.nyeremenyekLista.setModel(dlm);
               }
          }
     }
```

A `JList` példánynak is új modellt kell beállítanunk, melynek `addEelement()` metódusával tudunk új sorokat hozzáadni.

Korongok

Egy cég műanyag korongokat gyárt. A termékpalettán szereplő korongok típusai szerepelnek a **korongok.txt** UTF-8 kódolású állományban. A sorok adatait pontosvessző választja el. Először a méret, majd a szín, végül egy szorzó található. Az állomány néhány sora így néz ki:

```
6.5/2;orange;1
4.4/3;yellow;-2
```

A méreteket / jel választja el. Az első érték a korong sugara centiméterben kifejezve, a másik a magassága milliméterben mérve. A színeket a gyár angolul adja meg.

A szorzó jelentése: Minden korongnál van egy alapár. Ez mindenkor 32Ft. A korong tényleges ára a szorzótól függ, amely megmutatja, hogy hányszor 10%-ot kell még az alapárra rászámolni. A szorzó negatív is lehet! Tehát a minta első korongjának az ára az alapár 10%-kal emelt értéke, a második korongnál, pedig 20%-kal kerül kevesebbe ez a korong, mint az alapár.

Az ár csak egész forint lehet, ezt az általánosan elfogadott kerekítési szabályok betartásával érhetjük el. Az első korong így 35Ft-ba, míg a második 26Ft-ba kerül.

Az alapárat konstansnak deklaráljuk!

A megoldásnál vegyük figyelembe, hogy a felhasználóval való kommunikáció nyelve magyar, ezért a nyelvtanilag helyesen történjen a kommunikáció, bár az ékezetmentes kiírás is elfogadott! A megoldás során a „Tiszta kód" elveket tartsuk be, de nem feltétlenül kell megjegyzéseket fűzni a kódhoz!

A megoldás során törekedjünk megfelelő metódusokat használni! A megoldás során az osztály tulajdonságait az adott osztályon belül valósítsuk meg!

Minden kiírást igénylő feladat előtt írjuk ki, melyik feladatról van szó!

1. Hozzuk létre a korongok nevű projektet a következő feladatok megoldására. Olvassuk be egy megfelelő adatszerkezetbe az állományban található adatokat!
2. Hány különböző korongot gyártunk?
3. Kérjünk be egy szint a felhasználótól! Gyárt ilyen színű korongot a gyár?
4. Melyik a legnagyobb térfogatú korong? Elég egy ilyet megadni. A kiírásnál az adott korong minden alapadatát írjuk ki. (Amelyek benne voltak a forrásban)

5. Készítsünk kimutatást, hogy az egyes színekből hány különböző korongot gyártunk? Az állományban minden sor egyedi!
6. Kérjünk be a felhasználótól egy térfogatot mm3-ben! A szorzó szerint növekvő sorrendben írjuk ki az ettől a térfogattól nagyobb térfogatú korongokat abban a formában, ahogy a forrásban van, de sorszámozzuk meg a sorokat a minta szerint!

```
13: 6.5/2;orange;1
```

7. A cég a csomagolás során egy dobozba minden általa gyártott korongból pontosan 1000 darabot helyez be. Mennyi ennek a doboznak az értéke, ha a csomagolás értéke 1000 forint, és minden megkezdett 700 korong után 85Ft méretdíjat kell felszámolni még?
8. A feladat megoldásánál törekedj arra, hogy a main() metódusban a pontosvesszők száma 8-nál több nem lehet! (Maximum 8 sort tartalmazhat. Ha nem sikerül így elsőre, old meg még egyszer, az eredeti megoldást felhasználva, hogy megfeleljen a megoldásod ennek a feltételnek!)

Korongok megoldás

1. lépés

Figyeljünk oda, a projektet a feladatban meghatározott néven hozzuk létre.

2. lépés

Megfelelő adatszerkezet kialakítása.

A megfelelő adatszerkezet egy saját osztály példányait tartalmazó gyűjtemény

Készítsünk egy saját osztályt Korong néven. Az osztály tagadatai a külső állomány oszlopai alapján kerüljenek kialakításra. Használjunk megfelelő típust és válasszunk beszédes változónevet! A könnyebb megérthetőség miatt magyar azonosítók használata javasolt. A konstruktor egy `String`-et várjon, amit feldarabolunk egy szeparátor mentén és a megfelelő típuskonverziók után inicializáljuk a tagadatokat. A szeparátor általában ';' szokott lenni, de ebben a feladatban egy '/' jel is szerepel, ezért első lépésben ezt alakítsuk át.

```
package koronggyar;

public class Korong {
    private int alapar;
    private double sugarCm;
    private int magassagMm;
    private String szin;
    private int szorzo;

    public Korong(String sor) {
        this.alapar=32;
        sor = sor.replace("/", ";");
        String[] szeletek=sor.split(";");
        this.szin=szeletek[2];
        try {
            this.sugarCm=Double.parseDouble(szeletek[0]);
            this.magassagMm=Integer.parseInt(szeletek[1]);
            this.szorzo=Integer.parseInt(szeletek[3]);
        } catch (NumberFormatException e) {
            System.out.println("Hiba: "+e.getMessage());
        }
    }

    public int getAlapar() {
        return alapar;
    }

    public double getSugarCm() {      return sugarCm;     }
    public int getMagassagMm() {      return magassagMm;     }
    public String getSzin() {         return szin;     }
    public int getSzorzo() {          return szorzo;     }
```

A tagadatok (osztályváltozók) védelme érdekében `private` láthatósági szinttel deklaráljuk azokat. Osztályon kívüli elérésüket hozzáférő metódusok segítségével oldjuk meg.

Javasolt az átalakításból eredő esetleges kivételek kezelése.

```
    @Override
    public String toString() {
        return "Korong{" + " sugarCm=" + sugarCm + ",
magassagMm=" + magassagMm + ", szin=" + szin + ", szorzo=" +
szorzo + '}';
    }
}
```

3. lépés

Legjobb, ha a külső állományt bemásoljuk a létrehozott projekt mappájába, hogy ne kelljen elérési utat megadnunk a beolvasás során.

Az állomány sorainak beolvasására - és egy esetleges későbbi kiírásra - készítsünk egy külön osztályt, melyben osztálymetódus(oka)t hozzunk létre. A beolvasás metódusa egy `ArrayList<>` adatszerkezettel térjen vissza, melyben a fájl soraiból képezett saját osztály egy-egy példánya kerül.

```
public class AllomanyMuvelet {
    static ArrayList<Korong> beolvaso(String file){
        ArrayList<Korong> list = new ArrayList<>();
        RandomAccessFile be;
        try {
            be = new RandomAccessFile(file, "r");
            while(be.getFilePointer() < be.length()){
                list.add(new Korong(be.readLine()));
            }
            be.close();
        } catch (IOException e) {
            System.out.println("Hiba: "+e.getMessage());
        }
        return list;
    }
}
```

4. lépés

A main metódusban hívjuk meg az állománybeolvasó metódust, és egy kibővített `for` ciklussal járjuk végig a gyűjteményt, kiíratva az egyes példányokat, hogy meggyőződjünk a sikeres beolvasásról. Ha megnyugodtunk, akkor ez törölhető.

```
package koronggyar;

import java.util.ArrayList;

public class Koronggyar {
    public static void main(String[] args){
        ArrayList<Korong> korongok =
AllomanyMuvelet.beolvaso("korongok.txt");
        for (Korong korong : korongok) {
            System.out.println(korong);
        }
    }
```

A Korong osztályunk melyik metódusa teszi lehetővé, hogy ilyen egyszerű legyen egy-egy példány adatainak kiíratása?

```
}
```

5. lépés

A gyűjteménybe beolvasott sorok száma, annak `size()` metódusával kérdezhető le

Az `ArrayList<>` adatszerkezet méretének kiírása

```
System.out.println("2. feladat: "+korongok.size()+" különböző
korongot gyártunk");
```

6. lépés

<u>Adatbekérés és eldöntés tétele</u>

A `Scanner` osztály segítségével kérjünk be egy karakterláncot, majd ezt adjuk át egy metódusnak, mely logikai értékkel tér vissza attól függően, hogy gyártunk olyan színű korongot vagy sem.

Eldöntés tételének megvalósítása egy metódusban: végigjárjuk a sorozat elemeit, és ha az aktuális elem megfelel egy feltételnek, akkor a metódus `true` értékkel visszatér. Ha egyszer sem teljesül a feltétel, akkor `false` értéket adunk vissza.

a feladat metódusa:

```
static boolean gyartunkOlyanSzinuKorongot(ArrayList<Korong>
korongok, String szin){
```

Adhatnánk más nevet is a korongokat tároló paraméternek?
Mi a különbség a globális és lokális változók között?

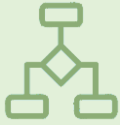
```
    for (Korong korong : korongok) {
        if(korong.getSzin().equals(szin))
            return true;
    }
    return false;
}
```

A referencia típusú változók - köztük a karakterláncok - dinamikus tárkezelést valósítanak meg (egy memóriacímre mutatnak), ezért két `String` - vagy tetszőleges két objektum - összehasonlítására nem az == operátor, hanem az `equals()` metódus szolgál.

A metódus meghívása a main metódusból:

```
System.out.print("3. feladat: Kérek egy színt: ");
Scanner be = new Scanner(System.in);
String szin = be.nextLine();
if(gyartunkOlyanSzinuKorongot(korongok, szin))
```

```
      System.out.println("Gyártunk ilyen színű korongot:
"+szin);
else
      System.out.println("Nem gyártunk ilyen színű korongot:
"+szin);
be.close();
```



A kommunikáció a felhasználóval mindig legyen teljes. Közöljük, hogy milyen adatot várunk tőle, s pontosan tájékoztassuk az eredményről.

7. lépés

Térfogatszámítás és szélsőérték keresés

A szélsőérték keresés elemi algoritmusának megvalósítása metódusban: Vezessünk be egy `maxIndex` nevű egész típusú változót és inicializáljuk 0-ra. Járjuk végig a sorozat elemeit, és ha az aktuális elemről kiderül, hogy a vizsgált értéke nagyobb, mint a `maxIndex`-edik elemé, akkor a `maxIndex` változó megkapja az aktuális elem indexének értékét. Ennek segítségével a metódus visszatérhet a megfelelő elemmel.

A térfogatot tároló tagadat és ennek értékét kiszámoló metódus az osztályban:

```
private double terfogatMm3;
...
public double getTerfogatMm3(){
    return Math.pow(this.sugarCm*10,
2)*Math.PI*this.magassagMm;
}
```

A térfogat kiszámítása egyetlen logikai egység, ezért erre a célra hozzunk létre egy külön metódust. Mivel a térfogat a korong tulajdonsága, ezért a Korong osztályban kell egy tagadatot és egy ennek beállítására szolgáló metódust létrehozni.

Az elemi algoritmus metódusa a main metódus alatt:

```
static Korong legnagyobbKorong(ArrayList<Korong> korongok){
    int maxIndex=0;
    for (int i = 1; i < korongok.size(); i++) {
        if(korongok.get(i).getTerfogatMm3() >
korongok.get(maxIndex).getTerfogatMm3())
             maxIndex = i;
    }
    return korongok.get(maxIndex);
}
```

Szélsőérték keresés esetén ne használjunk kibővített `for` ciklust, mert szükségünk van a ciklusváltozóra.

A metódus meghívása:

```
System.out.println("4. feladat: A legnagyobb térfogatú
korong:");
```

```
System.out.println(legnagyobbKorong(korongok));
```

8. lépés

Egyedi elemek kiválogatása plusz megszámlálás elemi algoritmusa

Egy sorozatban a halmaz kiválogatás nagyon hosszú megoldással jár, ezért sok buktatót és hibalehetőséget rejt magában. Használjunk helyette a `TreeSet<>` adatszerkezetet!

A `TreeSet<>` adatszerkezetben kizárólag egyedi értékek szerepelhetnek. Járjuk végig a sorozat elemeit és az aktuális elem – a Korong osztály 1 példánya – színét, mely `String` típusú adjuk hozzá egy `TreeSet<>`-hez.

Külső ciklus: járjuk végig a `TreeSet<String>` elemeit

Egy `db` nevű egész típusú változó kapjon 0 kezdőértéket.

Belső ciklus: járjuk végig az `ArrayList<Korong>` elemeit. Ha az aktuális elem színe megegyezik a külső ciklus által kiválasztott színnel, akkor a `db` nevű változó értékét inkrementáljuk (növeljük eggyel).

A belső ciklus végén írassuk ki a színt és a hozzá tartozó `db` értéket.

```
static void szinenkentiKorongokSzama(ArrayList<Korong>
korongok){
    TreeSet<String> szinek = new TreeSet<>();
    for (Korong korong : korongok) {
        szinek.add(korong.getSzin());
    }
    int db;
    for (String szin : szinek) {
        db=0;
        for (Korong korong : korongok) {
            if(szin.equals(korong.getSzin()))
                db++;
        }
        System.out.println(szin+"\t"+db+" db");
    }
}
```

A metódus meghívása a main metódusból:

```
System.out.println("5. feladat: Kimutatás a színenként gyártott
korongok számáról:");
szinenkentiKorongokSzama(korongok);
```

9. lépés

Kiválogatás és sorba rendezés

Válogassuk ki a bekért értéknél nagyobb térfogatú példányokat, de most nem íratjuk ki őket, hanem egy új `ArrayList<>`-ben eltároljuk őket és ez a gyűjtemény lesz a metódus visszatérési értéke.

Egy másik metódus ezt a gyűjteményt várja paraméterként. Sorba rendezi azt a `szorzo` tagadat értéke szerint növekvő sorrendbe és a kiíratás során sorszámokkal látja el a megadott minta szerint.

Bontsuk két külön metódusra a feladatot,
mert jobb, ha egy metódus csak egyetlen feladatot végez el

A `Collections` osztály `sort()` osztálymetódusával egyszerűen sorba rendezhetjük a gyűjteményünk példányait, de ehhez segítségül kell hívnunk a `Comparable<>` interfészt.

Ahhoz, hogy a kiírás a minta szerint történjen, átalakíthatjuk az Korong osztályunk `toString()` metódusát. Ezt ugyan a 4. feladatban is használtuk, de ott is megfelelő lesz a kiírásnak ez az új módja.

A Korong osztályban végzett módosítások:

```
public class Korong implements Comparable<Korong> {
```

A saját osztályunk példányait egy kiválasztott tagadat szerint legkönnyebben úgy tudjuk sorba rendezni, hogy az osztályunk implementálja a `Comparable<>` interfészt.

```
...
```

Ekkor meg kell valósítani a `compareTo()` metódust és meghatározni, hogy melyik tagadat legyen a sorba rendezés alapja.

```
    @Override
    public int compareTo(Korong masik) {
        return this.szorzo - masik.szorzo;
    }

    @Override
    public String toString() {
        return this.sugarCm + "/" + this.magassagMm + ";" +
this.szin + ";" + this.szorzo;
    }
}
```

A feladat megoldásának két metódusa:

```
static void sorszamozvaRendezveKiir(ArrayList<Korong>
korongok){
```

Hogy változhat meg a gyűjtemény tartalma csupán attól, hogy egy metódus paraméterében szerepel?

```
    Collections.sort(korongok);
    for (int i = 0; i < korongok.size(); i++) {
        System.out.println((i+1)+": "+korongok.get(i));
    }
}
```

Ne kibővített `for` ciklussal járjuk végig a sorozatot, mert szükségünk van egy sorszámra.

```
static ArrayList<Korong> nagyobbak(ArrayList<Korong> korongok,
double terfogat){
    ArrayList<Korong> nagyobbKorongok = new ArrayList<>();
    for (Korong korong : korongok) {
        if(korong.getTerfogatMm3()>terfogat)
            nagyobbKorongok.add(korong);
    }
    return nagyobbKorongok;
}
```

A rendezés nem előzheti meg a kiválogatást.
Amíg egy n elemű sorozatban a kiválogatásra fordított idő n-nel, addig a sorba rendezésre fordított idő $\log_2 n$-nel arányos.

A metódusok meghívása a main-ben:

```
System.out.print("6. feladat: Kérek egy térfogatot mm3-ben: ");
try {
    sorszamozvaRendezveKiir(nagyobbak(korongok,
be.nextDouble()));
} catch (NumberFormatException e) {
    System.out.println("Hiba: "+e.getMessage());
}
```

10. lépés

Összegzés tétele

A darabonkénti ár kiszámítása az `alapar` és a `szorzo` tagadatok segítségével történik. Az alapárhoz adjuk hozzá az alapárnak a szorzó/10.0-val osztott értékét.

Az ár csak egész szám lehet, mielőtt 1000-rel megszorozzuk, kerekítenünk kell azt.

A 85 Ft méretdíj beszorzása előtt is a teljes darabszámot ne 700-zal, hanem 700.0-val osszuk, valamint kerekítsük felfelé.

A feladat megoldásának metódusa:

```
static int osszErtek(ArrayList<Korong> korongok){
    int osszeg = 0;
    for (Korong korong : korongok) {
        osszeg+=1000*(Math.round(korong.getAlapar() +
korong.getAlapar()*korong.getSzorzo()/10.0));
    }
```

Ne feledjük el, hogy a / operátor egész számok között egészosztást végez, ezért a két operandus közül az egyik legyen valós típusú. E miatt kell 10.0-val osztanunk

```
    osszeg+=1000; // csomagolás
    osszeg+=85*Math.ceil(korongok.size()*1000/700.0); //
méretdíj
    return osszeg;
}
```

A metódus meghívása a `main`-ben:

```
System.out.println("7. feladat: A csomag összértéke: " +
osszErtek(korongok));
```

11. lépés

<u>A main metódus lehető legátláthatóbbá tétele</u>

Az egyes feladatokat egy-egy sorral úgy tudunk megoldani, hogy a több sorból álló megoldásokból külön `void` metódusokat készítünk, s a `main` metódusba kizárólag ezek meghívása kerül.

A többsoros megoldásokból készített metódusok:

```
static void feladat3(ArrayList<Korong> korongok){
    System.out.print("3. feladat: Kérek egy színt: ");
    Scanner be = new Scanner(System.in);
    String szin = be.nextLine();
    if(gyartunkOlyanSzinuKorongot(korongok, szin))
        System.out.println("Gyártunk ilyen színű korongot:
"+szin);
    else
        System.out.println("Nem gyártunk ilyen színű korongot:
"+szin);
}
```

Ne zárjuk be a kapcsolatot a `Scanner` osztály példányának `close()` metódusával, mert az bezárja a kapcsolatot a billentyűzet és a program között ami miatt a második adatbekérés már nem fog sikerülni.

```
static void feladat4(ArrayList<Korong> korongok){
```

```
    System.out.println("4. feladat: A legnagyobb térfogatú
korong:");
    System.out.println(legnagyobbKorong(korongok));
}

static void feladat5(ArrayList<Korong> korongok){
    System.out.println("5. feladat: Kimutatás a színenként
gyárott korongok számáról:");
    szinenkentiKorongokSzama(korongok);
}

static void feladat6(ArrayList<Korong> korongok){
    System.out.print("6. feladat: Kérek egy térfogatot mm3-ben:
");
    try {
        sorszamozvaRendezveKiir(nagyobbak(korongok,
be.nextDouble()));
    } catch (NumberFormatException e) {
        System.out.println("Hiba: "+e.getMessage());
    }
    be.close();
}
```

A végleges `main` metódus tartalma:

```
ArrayList<Korong> korongok =
AllomanyMuvelet.beolvaso("korongok.txt");
System.out.println("2. feladat: "+korongok.size()+" különböző
korongot gyártunk");
feladat3(korongok);
feladat4(korongok);
feladat5(korongok);
feladat6(korongok);
System.out.println("7. feladat: A csomag összértéke:
"+osszErtek(korongok));
```

Szőlő termés

A Fenékmelléki hegyközség éves bortermelési adatai a **termes.csv** szöveges állományban találhatóak meg. A hegyközség 5 főbb területre tagozódik és az itt termelt szőlőkből előállított borokat is külön kell a tagoknak lejelenteni. A sorok adatait pontosvessző választja el. Az első sor az oszlopok fejlécét tartalmazza. Az állomány néhány sora így néz ki:

```
1020;Csabagyongye;8,624;9,541;;9,549;6,747
1002;Cserszegi fuszeres;;9,081;4,298;7,598;8,179
```

Az első adat a gazda helyközségi azonosítója. A másodikban az adott szőlő neve szerepel. Ezt követi az öt területen az adott gazda, adott szőlőből előállított borának a mennyisége 1000 literben kifejezve. A területeket csak a sorszámukkal azonosítják római számmal. Tehát Csabagyongye szőlőből a 1020-as kódú gazda 8624 litert állított elő az I-es területen és nincs szőlője a III-as területen.

Nem minden gazda termel minden területen egy adott fajtájú szőlőt, ott a mennyiségek hiányoznak.

A **termelok.csv** állomány a termelők nevét tartalmazza szóközzel elválasztva. Minden gazdának csak egy vezetékneve van, de keresztneve több is lehet! Az állomány néhány sora így néz ki:

```
1007 Kaparos Zsolt
1003 Szunyogh Balazs Hugo
1020 Kalandor Andor
1002 Rugka Pal
```

Az állományokban csak az angol abc betűi szerepelnek az egyszerűbb megoldhatóság miatt. A megoldásnál ezt vedd figyelembe!

A megoldásnál vegyük figyelembe, hogy a felhasználóval való kommunikáció nyelve magyar, ezért a nyelvtanilag helyesen történjen a kommunikáció, bár az ékezetmentes kiírás is elfogadott! A megoldás során a „Tiszta kód” elveket tartsuk be, de nem feltétlenül kell megjegyzéseket fűzni a kódhoz!

A megoldás során törekedjünk megfelelő metódusokat használni! A megoldás során az osztály tulajdonságait az adott osztályon belül valósítsuk meg!

Minden kiírást igénylő feladat előtt írjuk ki, melyik feladatról van szó!

1. Hozzuk létre a **szoleszet** nevű projektet a következő feladatok megoldására. Olvassuk be egy megfelelő adatszerkezetbe az állományokban található adatokat! Az adatok tárolására hozzunk létre osztályt, majd az objektumokat megfelelő dinamikus adatszerkezetben tároljuk! Ha a megoldás során szükségesnek tartja, az adatokat több különböző osztály létrehozásával is megoldhatja!
2. Hány gazda adatai szerepelnek a nyilvántartásban?
3. Hány különböző szőlőfajtát termelnek a hegyközség borászai?
4. Melyik fajtából termett a legtöbb?
5. A **mennyiseg.txt** állományba írd ki a termelőket a megtermelt összmennyiség szerint csökkenő sorrendben. A bor mennyiségét hektoliterben (`1hl = 100l`), egészre kerekítve add meg az állományban. Használd a megoldásnál a következő mintát:

```
...
Kaparós Zsolt 440
Bencze Emese 289
...
```

6. Kérj be a felhasználótól egy vezetéknevet! Szerepel ilyen vezetéknév a nyilvántartásban? Ha nem tájékoztasd a felhasználót, ha igen írd ki a képernyőre, hogy hányadik a bortermelők listájában a megtermelt bor mennyisége szerint. Figyelj oda, hogy előfordulhat több azonos vezetéknév a hegyközségben!
7. Írjuk ki a képernyőre hogy az egyes gazdák mely szőlőből és mely területen nem termelnek szőlőt! Egy sorban csak egy terület szerepeljen! A következő mintát használd a megoldáshoz:

```
Tóth Amarilló Bianca V
Kaparós Zsolt Budai zöld IV
```

Szőlő termés megoldás

1. lépés

Hozzuk létre a szoleszet nevű projektet.

2. Lépés

<u>Megfelelő adatszerkezet kialakítása.</u>

A két külső állományból készítsünk egy-egy saját osztályt. Az ezekből készített példányokat fogjuk egy-egy gyűjteményben eltárolni.

Készítsünk két saját osztályt Termelo és Termes néven. Az osztályok tagadatai a külső állomány oszlopai alapján kerüljenek kialakításra. Használjunk megfelelő típust és válasszunk beszédes változónevet! A könnyebb megérthetőség miatt magyar azonosítók használata javasolt. A konstruktor egy `String`-et várjon, amit feldarabolunk a ";", illetve szóköz szeparátor mentén és a megfelelő típuskonverziók után inicializáljuk a tagadatokat.

```
package szoleszet;

public class Termelo{
    private int azonosito;
    private String nev;
```

A tagadatok láthatósága `private`.
Ezért hozzáférő metódusokat is kell készíteni hozzájuk.

```
   public Termelo(String sor) {
        try {
            this.azonosito = Integer.parseInt(sor.substring(0,
sor.indexOf(" ")));
        } catch(NumberFormatException e){
            System.out.println("Nem szám formátum:
"+e.getMessage());
        }
```

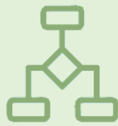
A szövegből számmá alakítás futás közbeni hibát eredményezhet. Jobb, ha kivételkezeléssel ezt megelőzzük.

```
        this.nev = sor.substring(sor.indexOf(" ")+1);
    }

    public int getAzonosito() {        return azonosito;    }

    public String getNev() {        return nev;    }

    @Override
    public String toString() {
```

```
        return "Termelo{" + "azonosito=" + azonosito + ", nev="
+ nev + '}';
    }
}
```

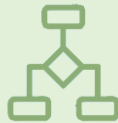
A másik osztály:

```
package szoleszet;

public class Termes {
    private int gazda;
    private String szolo;
    private int[] teruletenkentiLiter;
```



Logikailag összetartozó, azonos típusú adatok esetén sokkal hatékonyabb feladatmegoldást tesz lehetővé, ha listában, vagy gyűjteményen tároljuk el azokat.

```
    public Termes(String sor) {
        this.teruletenkentiLiter = new int[5];
        String[] szeletek = sor.split(";");
        this.szolo = szeletek[1];
        try {
            this.gazda = Integer.parseInt(szeletek[0]);
            for (int i = 2; i < szeletek.length; i++) {
```



A hiányzó mennyiségek adatait le kell cserélni 0 értékre.

```
                if(szeletek[i].isEmpty())
                    this.teruletenkentiLiter[i-2]=0;
                else
                    this.teruletenkentiLiter[i-
2]=(int)(Double.parseDouble(szeletek[i].replace(",",
"."))*1000);
            }
        } catch (NumberFormatException ex) {
            System.out.println("Nem szám formátum:
"+ex.getMessage());
        }
    }

    public int getGazda() {        return gazda;     }

    public String getSzolo() {        return szolo;     }

    public int[] getTeruletenkentiLiter() {
        return teruletenkentiLiter;
    }

    @Override
    public String toString() {
        String literek = "";
```

```
        for (int i : this.teruletenkentiLiter) {
            literek = literek.concat(" "+i);
        }
        return "Termes{"
                + "gazda=" + gazda
                + ", szolo=" + szolo
                + ", teruletenkentiLiter=" + literek + '}';
    }
}
```

A tömb típusú tagadat miatt kissé bonyolultabb a `toString()` metódus

3. lépés

Másoljuk be a két külső állományt a létrehozott projekt mappájába.

Az állományok sorainak beolvasására és a kiírásra készítsünk egy külön osztályt, melyben osztálymetódusokat hozzunk létre. A beolvasás metódusai egy `ArrayList<>` adatszerkezettel térjenek vissza, melyben a fájl soraiból képezett saját osztály egy-egy példánya kerül.

```
package szoleszet;

import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.ArrayList;

public class AllomanyMuveletek {

static ArrayList<Termes> termeseketBeolvas(String file){
    ArrayList<Termes> list = new ArrayList<>();
    RandomAccessFile be;
    try {
        be = new RandomAccessFile(file, "r");
        be.readLine();
```

Ha az állomány első sora nem tárolandó adatokat tartalmaz, akkor a példányosítást csak a következő sortól kedve szabad elkezdeni.

```
        while(be.getFilePointer()<be.length()){
            list.add(new Termes(be.readLine()));
        }
        be.close();
    } catch (IOException e) {
        System.out.println("Hiba: "+e.getMessage());
    }
    return list;
}

static ArrayList<Termelo> termeloketBeolvas(String file){
    ArrayList<Termelo> list = new ArrayList<>();
```

```
    RandomAccessFile be;
    try {
        be = new RandomAccessFile(file, "r");
        while(be.getFilePointer()<be.length())
            list.add(new Termelo(be.readLine()));
        be.close();
    } catch (IOException e) {
        System.out.println("Hiba: "+e.getMessage());
    }
    return list;
}
}
```

4. lépés

A `main()` metódusban hívjuk meg a két beolvasó metódust és tároljuk el egy-egy gyűjteményben.

```
ArrayList<Termes> termesek =
AllomanyMuveletek.termeseketBeolvas("termes.csv");

ArrayList<Termelo> termelok =
AllomanyMuveletek.termeloketBeolvas("termelok.csv");
```

5. lépés

A termelőket tartalmazó gyűjtemény hossza adja meg a termelők számát.

```
System.out.println("2. feladat: "+termelok.size()+" gazdáról
vannak adataink.");
```

6. lépés

A különböző szőlőfajták száma nem egyenlő a termeléseket tartalmazó gyűjtemény hosszával, hiszen egy-egy szőlőfajta több sorban is szerepel. Egy sorozat egyedi elemeinek kiválogatásához mindig célszerű `TreeSet<>` adatszerkezetet használni.

A eladat megoldására szolgáló metódus:

```
    static TreeSet<String> egyediSzolofajtak(ArrayList<Termes>
termesek){
        TreeSet<String> szolofajtak = new TreeSet<>();
        for (Termes termes : termesek) {
            szolofajtak.add(termes.getSzolo());
        }
        return szolofajtak;
    }
```

A visszatérési érték lehetett volna egyből a darabszám, viszont az egyedi szőlőfajtákra a későbbi feladatok során is szükség lehet. Ez a metódus újra meghívható, így elkerülhető a kódismétlés.

A metódus meghívása a `main[]`-ből

```
System.out.println("3. feladat:
"+egyediSzolofajtak(termesek).size()+" különböző szőlőfajtát
termelnek.");
```

7. lépés

Egy példányon belül termelt liter adatok összegének kiszámítására és tárolására vezessünk be a `Termes` osztályban egy új metódust, ami kiszámolja azt.

```
public int getOsszesTermeltLiter(){
    int osszeg = 0;
    for (int liter : teruletenkentiLiter) {
        osszeg+=liter;
    }
    return osszeg;
}
```

8. lépés

Összegzés és szélsőérték keresés

Az egyes szőlőfajtából készített bor mennyiségének kiszámítása és ezek közül a legnagyobb értékű kiválasztása két külön logikai egység, ezért két metódust kell készítenünk.

Az összegzést végző metódusban bevezetünk egy `liter` nevű változót és 0-ra inicializáljuk. Végigjárjuk a sorozat elemeit. Ha az aktuális elem szőlőfajtája egyezik a paraméterként kapott `String`-gel, akkor a `Termes` osztály új liter értékeket összeadó metódusának segítségével a `liter` nevű változót növeljük.

A szélsőérték kereső metódusban bevezetünk egy `maxindex` nevű változót és 0 kezdőértékkel látjuk el. Végigjárjuk a sorozat elemeit annak második elemétől kezdve. Ha az aktuális elem az előbbi összegző metódus által kiszámolt értéke nagyobb lenne, mint a `maxindex`-edik elemé, akkor a `maxindex` megkapja az aktuális elem indexét.

Összegző metódus

```
static int egyfajtaSzolobolBorLiter(ArrayList<Termes> termesek,
String fajta){
    int liter=0;
    for (Termes termes : termesek) {
        if(termes.getSzolo().equals(fajta))
            liter+=termes.getOsszesTermeltLiter();
    }
    return liter;
}
```

Szélsőérték kereső metódus

```
static String legtobbetTermoSzolofajta(ArrayList<Termes>
termesek){
    int maxindex=0;
```

Szélsőérték kereséskor szükségünk van ciklusváltozóra. Ezért ilyenkor ne alkalmazzunk kibővített `for` ciklust.

```
 for (int i = 1; i < termesek.size(); i++) {
        if(egyfajtaSzolobolBorLiter(termesek,
termesek.get(i).getSzolo())>egyfajtaSzolobolBorLiter(termesek,
termesek.get(maxindex).getSzolo()))
            maxindex=i;
    }
    return termesek.get(maxindex).getSzolo();
}
```

Ha egy `TreeSet<>` adatszerkezetbe az előző feladatban elkészített metódussal betöltöttük volna az egyeid szőlőfajtákat, akkor elég lett volna ezt végigjárni a szélsőérték megtalálásához?

9. lépés

Összegzés tétele

Ki kell tudnunk számolni az egyes termelők által készített bor összmennyiségét.

```
static int egyGazdaOsszHektoLiter(ArrayList<Termes> termesek,
int gazdaAzonosito){
    int osszeg=0;
    for (Termes termes : termesek) {
        if(termes.getGazda()==gazdaAzonosito)
            osszeg+=termes.getOsszesTermeltLiter();
    }
    return (int)(Math.round(osszeg/100.0));
}
```

A / operátor egész számok között egészosztást végez, ezért valós számmal kell osztani az összes litert a kerekített hektoliter kiszámításához

10. lépés

Van egy `Termelo` osztályunk, amiben a gazdák nevei már szerepelnek, vegyünk fel egy új tagadatot, a megtermelt összmennyiség tárolására.

Az új tagadat és hozzáférő metódusai

```
private int keszitettBorHektoLiter;

public int getKeszitettBorHektoLiter() {
```

```
    return keszitettBorHektoLiter;
}

public void setKeszitettBorHektoLiter(int
keszitettBorHektoLiter) {
    this.keszitettBorHektoLiter = keszitettBorHektoLiter;
}
```

A sorbarendezést lehetővé tevő interrész megvalósítása az osztály fejrészében

```
public class Termelo implements Comparable<Termelo>
```

Két objektum összehasonlítását végző metódus. A `Collections.sort()` metódus ennek alapján végzi majd a sorba rendezést.

```
@Override
public int compareTo(Termelo o) {
    return this.keszitettBorHektoLiter -
o.keszitettBorHektoLiter;
}
```

Amikor összetartozó értékpárokat kell sorba rendeznünk az egyik érték szerint, akkor használjuk ki az objektum-orientált szemléletmód előnyeit.

11. lépés

Szükségünk van egy olyan gyűjteményre, amiben már az új, kibővített `Termelo` osztálybeli objektumok szerepelnek.

```
static ArrayList<Termelo> adatbovito(ArrayList<Termes>
termesek, ArrayList<Termelo> termelok){
    for (Termelo termelo : termelok) {

termelo.setKeszitettBorHektoLiter(egyGazdaOsszHektoLiter(termes
ek, termelo.getAzonosito()));
    }
    Collections.sort(termelok, Collections.reverseOrder());
    return termelok;
}
```

A sorba rendezés alapértelmezetten növekvő sorrendbe teszi a példányokat. Nekünk csökkenő sorrendre van szükségünk.

12. lépés

A fenti metódussal előállított gyűjteményt már át lehet adni egy külső fájlba író metódusnak az `AllomanyMuveletek` osztályban

```
static void mennyisegeketKiir(ArrayList<Termelo>
bovitettTermelok, String file){
    RandomAccessFile kiir;
    try {
        kiir = new RandomAccessFile(file, "rw");
        for (Termelo termelo : bovitettTermelok) {
            kiir.writeBytes(termelo.getNev()+"
"+termelo.getKeszitettBorHektoLiter()+"\r\n");
        }
        kiir.close();
    } catch (IOException e) {
        System.out.println("Hiba a kiírás során:
"+e.getMessage());
    }
}
```

13. lépés

A fenti lépéseknek köszönhetően a `main[]` metódusba egyetlen elegáns sor kerül

```
AllomanyMuveletek.mennyisegeketKiir(adatbovito(termesek,
termelok), "mennyiseg.txt");
```

14. lépés

<u>Kiválogatás tétele</u>

Az új, kibővített Termelo osztálybeli példányokat tartalmazó gyűjteményben kell keressük a termelőt vezetéknév alapján.

Végigjárjuk a sorozat elemeit, és ha az aktuális elem megfelel a feladatban írt feltételnek, akkor a kért adatot (rangsor szerinti sorszám) megjelenítjük.

Ebben a metódusban le kell kezelnünk azt is, ha nincs találat a paraméterként kapott vezetéknévre.

```
static void rangsorolo(ArrayList<Termes> termesek,
ArrayList<Termelo> termelok, String vezeteknev){
    ArrayList<Termelo> bovitettTermelok = adatbovito(termesek,
termelok);
    String nev;
    int db=0;
```

Hagyományos `for` ciklussal dolgozunk, mert felhasználjuk az index értékét.

```
    for (int i=0; i<bovitettTermelok.size(); i++) {
```

```
        nev = bovitettTermelok.get(i).getNev();
        if(nev.substring(0, nev.indexOf("
")).equals(vezeteknev)){
            db++;
            System.out.println(nev+" a bortermelők listájában
a(z) "+(i+1)+ ". helyen áll");
        }
    }
    if(db==0)
        System.out.println("Nincs ilyen vezetéknevű
bortermelő");
}
```

A metódus meghívása a `main[]`-ből

```
System.out.print("6. feladat: Kérek egy vezetéknevet: ");

rangsorolo(termesek, termelok, new Scanner(System.in, "ISO-
8859-2").nextLine());
```

A külső állomány ANSI kódolású. Az ékezetes karakterek összehasonlítása csak akkor fog helyesen működni, ha a felhasználó által bekért karaktereket is erre a kódolásra alakítjuk,

15. lépés

A kiíratás a Termes osztály példányait tartalmazó gyűjtemény alapján történik majd, de szükségünk van a gazdák neveire is.

```
static String gazdaNeve(ArrayList<Termelo> termelok, int
gazdaAzonosito){
    for (Termelo termelo : termelok) {
        if(termelo.getAzonosito()==gazdaAzonosito)
            return termelo.getNev();
    }
    return "-- --";
}
```

Ha a feltétel egyszer sem teljesülne, akkor is kell valamilyen visszatérési érték. Persze helyes paraméterek megadásával ez nem történhet meg.

16. lépés

Kiválogatás

A kiválogatás metódusa

```
static void nemTermeltSzolofajtak(ArrayList<Termes> termesek,
ArrayList<Termelo> termelok){
```

```
    String[] teruletek = {"I", "II", "III", "IV", "V"};
    for (Termes termes : termesek) {
        for (int i = 0; i <
termes.getTeruletenkentiLiter().length; i++) {
            if(termes.getTeruletenkentiLiter()[i]==0)
                System.out.println(gazdaNeve(termelok,
termes.getGazda())+" "+termes.getSzolo()+" "+teruletek[i]);
        }
    }
}
```

A metódus meghívása a `main[]` metódusból

```
System.out.println("7. feladat: Nem termelt szőlőfajták:");
nemTermeltSzolofajtak(termesek, termelok);
```

Borospince

A Fenékmelléki hegyközség egyik kiemelkedő borászata a Jura Borászat Manufaktúra. A borászat pincéjében található hordókat űrmérték szerint a **hordok.txt** szöveges állományban tárolták el a következők szerint. A sorban az első érték az űrmérték literben megadva, majd kettőspont után jönnek az egyes hordókban található bormennyiségek, szintén literben. Az állomány néhány sora így néz ki:

```
1200:460•23•1320•x43
600:d•340•580•23••345•21x•567
```

A szóközöket nagyméretű ponttal jelöltük, hogy azonosítható legyen, hol szerepel szóköz!

Sajnos a főborász unokája megrongálta az állományt, így helytelen adatok is kerültek a hordók adataiba. A kettőspont előtti űrmértéket már kijavították, azok helyesek.

A borászatban ha egy hordóban az űrmérték 10%-a alá esik a tárolt mennyiség egy hordóban, akkor azt kimossák, hogy felhasználhassák a következő feladathoz. Biztonsági szempontok miatt a hordóban nem lehet több bor, mint az űrmérték 98%, így azokat a hordókat amelyekben ennél több bor van feltüntetve, azok adatait is hibásnak kell tekinteni.

A megoldásnál vegyük figyelembe, hogy a felhasználóval való kommunikáció nyelve magyar, ezért a nyelvtanilag helyesen történjen a kommunikáció, bár az ékezetmentes kiírás is elfogadott! A megoldás során a „Tiszta kód" elveket tartsuk be, de nem feltétlenül kell megjegyzéseket fűzni a kódhoz!

A megoldás során törekedjünk megfelelő metódusokat használni! A megoldás során az osztály tulajdonságait az adott osztályon belül valósítsuk meg!

Minden kiírást igénylő feladat előtt írjuk ki, melyik feladatról van szó!

1. Hozd létre a **borospince** nevű grafikus alkalmazást a következő feladatok megoldására.
2. A minták között egy ajánlott elrendezés és kinézeti képek találhatóak. Ennek megfelelően alakítsa ki a grafikus felületet!
3. Olvassuk be egy megfelelő adatszerkezetbe az állományban található adatokat!
4. Készíts kivételkezelő osztályokat az egyes kivételekhez és az osztályban valósítsd meg az adatok ellenőrzését. A hibás adatokat tartalmazó hordókról tájékoztasd a

felhasználót a minta szerint, hogy mely hordók esetén és milyen problémát tartalmaz az állomány.

5. Legördülő listával jelenítsük meg az elérhető űrmértékeket!
6. Szövegbeviteli mező segítségével kérjünk be egy hordó sorszámot! Írjuk ki hogy mennyi bor található az adott hordóban! Nem kell ellenőrizni, hogy a bevitt érték egész számon kívül más is lehet-e, de ha helytelen egész került megadásra, arról tájékoztassuk a felhasználót egy üzenetdobozban!
7. Az előzőleg megjelenített érték mellé írjuk ki, hogy ez az adott űrmérték esetén hány százalékot jelent! A kiírást két tizedesre kerekítve adjuk meg!
8. A főborász szeretné, ha kapacitás szerint, azon belül a tárolt bor mennyisége szerint a jó adatokat tartalmazó hordók esetében az adatokat a **`rendezve.txt`** állományba írnánk ki a következő mintának megfelelően:

```
460/1200
340/600
345/600
567/600
580/600
```

Látható, hogy kapacitás szerint csökkenően, azon belül a bor mennyisége szerint növekvően rendezettek az adatok.

Minta:

Borospince

Helyes adatok

Űrmértékek: 600 liter

Sorszám 1 Választható: 1-4 OK

A hordó tartalma: 340 liter 56,67 %

Hibás adatok

űrmérték	adat	hiba
1200	23	<10%
1200	1320	>97%
1200	x43	nem szám
600	d	nem szám
600	23	<10%
600		nincs adat
600	21x	nem szám
340	340	>97%
340	d256	nem szám
200	xcdfdsed	nem szám

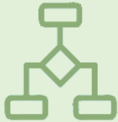
Borospince megoldás

Mi legyen az adatszerkezet?
Találunk-e hasonlóságot, valamilyen szabályszerűséget az egyes sorok között? Mit határoz meg számunkra a feladat az adatok tárolására vonatkozóan?

A külső állomány sorai ugyanolyan sorrendben egyforma típusú elemi adatokat tartalmaznak.

Minden sor első adata egy hordó űrmértéke (egész szám), másik adata pedig egy egész számok listája. A lista elemeinek száma változó.

A feladat előírja, hogy hibatípusonként külön kell kezelni a hibás adatokat.



Mindig törekedjünk az OOP szemléletű megvalósításra. Az állomány sorai egy relációs adattáblához hasonlítanak, ahol mindkét oszlop beazonosítható, elnevezhető. Az ilyen adathalmazok számára legelőnyösebb egy saját osztályt létrehozni, melynek tagadatai az oszlopok, egy-egy sora pedig az osztály egy-egy példánya.

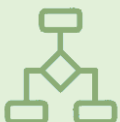
Mik azok a tulajdonságok, amik az egyedek összességére vonatkoznak?

Az űrmérték, az egyes hordók töltöttsége, a hibás adatok, valamint a hibák fajtái.

```
public final class Hordo {
   private int urmertek;
   private ArrayList<Integer> hordok;
   private ArrayList<String> hibasAdatok;
   private ArrayList<String> hibaTipusok;

   …
}
```

Érdemes lehet kipróbálni az összetartozó hibás típusok és hibás adatok tárolását egyetlen `Hashtable<String, String>` adatszerkezetben

Azonos típusú adatok listájának tárolására, mindig törekedjünk tömb helyett gyűjteményt választani. Nem kell előre méreteznünk és egyszerűbb a kezelése (pl kiíratás, sorbarendezés).



Mivel a hibás adatokat hibafajtánként külön kell kezelni, az állomány soraiban lévő hibás adatok mellett, azok hiba típusát is be kell azonosítani. Ezeket is gyűjteménybe.

A tagadatok láthatósága mindig legyen `private`, mert hozzáférésükre vonatkozó szabályokat külön metódusokkal megoldva korlátozhatjuk a más osztályból való elérésüket.

Az adott tagadatról el kell döntenünk, hogy az osztályé, vagy a példányé lesz. Az osztályváltozók példányosítás nélkül is elérhető egy másik osztályból.

Jelen feladatban nincs szükség ilyenre, az űrmérték s a töltöttségi adatok kifejezetten egy-egy példányra vonatkoznak.

Az osztály melyik eleme szolgál a tagadatok kezdőértékeinek beállítására? Mi alapján tudja ezt megtenni?

A tagadatok inicializálására a konstruktor szolgál. Paraméterében az állomány egy sorát kapja meg, az ebből kinyert adatok segítségével adhat értéket az egyes tagadatoknak. Az egyes sorokat egy szeparátor mentén feldaraboljuk, majd a kapott elemeket használjuk a tagadatok kezdeti értékeinek megadására.

A sor kétfajta szeparátort tartalmaz! Szóköz és kettőspont.

Mely tagadatokat célszerű a konstruktorban beállítani, melyeket külön hozzáférő metódusban?

Ha a példányosítás pillanatában már rendelkezésre áll a szükséges adat, akkor már a konstruktorban érdemes inicializálni a tagadat. Ha a program futásának egy későbbi szakaszában kapjuk azt meg (pl. felhasználó általi megadás), akkor külön beállító (setter) metódust kell készítenünk az osztályban.

Jelen példában minden adat azonnal rendelkezésre áll, de a hibás adatok/hibák típusai értékpárok beállítására mégis érdemes egy metódust létrehozni a kódismétlés elkerülése érdekében. Hiszen egy metódusban egyszerre be tudjuk állítani a két összetartozó tagadat értékét. A beállító metódust viszont már a konstruktorban meghívjuk.

Konstrukor a `Hordo` osztályban

```
public Hordo(String sor) {
   String[] szeletek;
   this.hordok = new ArrayList<>();
   this.hibasAdatok = new ArrayList<>();
   this.hibaTipusok = new ArrayList<>();
   int liter;
   try {
```

```
      this.urmertek = Integer.parseInt(sor.split(":")[0]);
      szeletek = sor.split(":")[1].split(" ");
     for (String szelet : szeletek) {
        try {
           if (szelet.length() == 0) {
              throw new NincsAdatException(szelet+";nincs adat");
           }
           else if (szelet.length() != szelet.replaceAll("[^0-9 ]",
           "").length()) {
              throw new NemSzamFormatumException(szelet+";nem
szám");
           } else {
              liter = Integer.parseInt(szelet);
              if (liter < this.urmertek / 10.0) {
                  throw new KimosasraVarException(liter + ";<10%");
              } else if (liter > this.urmertek / 100.0 * 97) {
                  throw new VeszelyesenTeleException(liter +
";>97%");
              } else {
                  this.hordok.add(liter);
              }
           }
        } catch (NincsAdatException ex) {
                     this.setHibasAdatok(ex.getSor());
        } catch (NemSzamFormatumException ex) {
                     this.setHibasAdatok(ex.getSor());
        } catch (KimosasraVarException ex) {
                     this.setHibasAdatok(ex.getSor());
        } catch (VeszelyesenTeleException ex) {
                     this.setHibasAdatok(ex.getSor());
        }
      }
   } catch (NumberFormatException ex) {
      System.out.println("Hiba: " + ex.getMessage());
   }
}
```

Hozzáférő metódusok a `Hordo` osztályban

```
       public void setHibasAdatok(String adatpar) {
           this.hibasAdatok.add(adatpar.split(";")[0]);
           this.hibaTipusok.add(adatpar.split(";")[1]);
       }

       public int getUrmertek() {
           return urmertek;
       }

       public ArrayList<Integer> getHordok() {
           return hordok;
       }


       public ArrayList<String> getHibasAdatok() {
           return hibasAdatok;
       }
```

```
    public ArrayList<String> getHibaTipusok() {
        return hibaTipusok;
    }
```

Az egyik saját kivétel osztály kódja: (a többi is megtalálható a teljes kódot tartalmazó tömörített fájlban)

```
public class NincsAdatException extends Exception{
    private String sor;

    public NincsAdatException(String message){
        super(message);
        this.sor = message;
    }

    public String getSor() {
        return sor;
    }
}
```

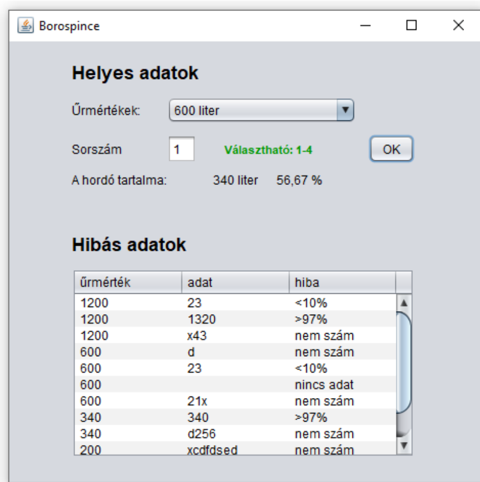
A létrehozandó példányokat el is kell tárolni. Mi az erre a célra megfelelő adatszerkezet?

A gyűjtemény. Pl. `ArrayList<Hordo> hordok = new ArrayList<>();`

A külső állomány sorainak beolvasására példánkban a `RandomAccessFile` osztályt választottuk, de használhatók erre a célra más Java osztályok is. Az állománybeolvasás külön metódusba került, melynek visszatérési értéke egy `ArrayList<Hordo>`

```
    public ArrayList<Hordo> beolvaso(String file){
        ArrayList<Hordo> list = new ArrayList<>();
        RandomAccessFile be;
        try {
            be = new RandomAccessFile(file, "r");
            while(be.getFilePointer()<be.length()){
                list.add(new Hordo(be.readLine()));
            }
            be.close();
        } catch (IOException e) {
            System.out.println("Hiba: "+e.getMessage());
        }
        return list;
    }
```

A grafikus interfész kialakításához a következő `Swing` komponensekre van szükségünk:



Az egyes elemek a Swing Controlls csoportból kerülnek fel a keretre (JFrame).

Szövegek megjelenítésre a JLabel osztály példányait használjuk.

Az űrmértékek kiválasztához: JComboBox

A sorszám beírásához: JTextField

A kiértékeléshez: JButton

A hibák megjelenítéséhez: JTable

Azokon a cimkéken kívül, amelyek tartalma változatlan marad a program futása során, nagyon hasznos, ha minden grafikus komponens beszédes változónevet kap. A Variable Name tulajdonság értékét a code fülön lehet megváltoztatni.

Milyen események válthatnak ki változásokat a grafikus programban?

Eseményfigyelők:

1. A feladatok megoldásához bizonyos műveleteknek az ablak megjelenésekor már végre kell hajtódniuk.
   - állománybeolvasás
   - űrmértékeket tartalmazó legördülő lista feltöltése
   - hibás adatok táblázat feltöltése

   Ehhez a `JFrame` példány `windowOpened` eseményfigyelőjét kell használnunk

2. Az űrmérték választásakor meg kell jelennie egy cimkében, hogy az adott űrmértékből milyen sorszámú hordók választhatóak.
   Ehhez a `JComboBox` példány `actionPerformed` eseményfigyelőjét használjuk

3. A kiválasztott hordó tartalma és a töltöttségi százalék megjelenítése a gombra kattintással történik
   Ehhez a `JButton` elem `actionPerformed` eseményfigyelőjét válasszuk,

Az űrmértékeket feltöltő és a hibás adatokat megjelenítő metódusnak is szüksége van az állományból beolvasott sorok alapján készült – saját `Hordo` osztályaink példányait tartalmazó – gyűjteményre. Hogyan célszerű a kódismétlést elkerülni?

A gyűjteményt a `JFrame` osztályunk tagadataként kell deklarálnunk. Az ablak megnyitásakor fel is töltjük az állományt beolvasó metódussal. Így átadhatjuk azt paraméterként a két szükséges metódusnak.

```
private void formWindowOpened(java.awt.event.WindowEvent evt) {
    this.joHordok = beolvaso("src/model/hordok.txt");
    urmertekFeltolto(this.joHordok);
    hibaTablaFeltolto(this.joHordok);
}
```

A legördülő lista feltöltése egy sorozat elemeivel elég egyszerű

```
public void urmertekFeltolto(ArrayList<Hordo> hordok){
    this.elerhetoUrmertekek.removeAllItems();
    this.elerhetoUrmertekek.addItem("Válasszon!");
    for (Hordo hordo : hordok) {
        if(hordo.getHordok().size()>0)
           this.elerhetoUrmertekek.addItem(hordo.getUrmertek()+"
liter");
    }
}
```

A táblázat feltöltése már nehezebb.

A táblázatok rendelkeznek egy saját modellel, ami meghatározza a táblázat sorainak, oszlopainak számát, az oszlopazonosítókat. Nekünk azonban saját modellt kell létrehozni és ezt beállítani a táblázatunkhoz.

```
public void hibaTablaFeltolto(ArrayList<Hordo> hordok){
    DefaultTableModel model = new DefaultTableModel();
    model.setColumnIdentifiers(new String[]{"űrmérték", "adat",
"hiba"});
    for (Hordo hordo : hordok) {
        for (int i = 0; i < hordo.getHibasAdatok().size(); i++) {
            model.addRow(new String[]{hordo.getUrmertek()+"",
hordo.getHibasAdatok().get(i), hordo.getHibaTipusok().get(i)});
        }
    }
    this.hibaTabla.setModel(model);
}
```

Ha a felhasználó választott űrmértéket, akkor ez történik:

```
private void
elerhetoUrmertekekActionPerformed(java.awt.event.ActionEvent evt) {
   hatarertekBeallitas(0, 0);
   this.valaszLiter.setText("");
   this.valaszSzazalek.setText("");
   int index = this.elerhetoUrmertekek.getSelectedIndex();
   if(index>0) {
       this.hordo = this.joHordok.get(index-1);
       if(hordo.getHordok().size()>0){
           hatarertekBeallitas(1, hordo.getHordok().size());
       }
       this.sorszamBeiro.setText("1");
   }
}
```

Miért kell 1-et levonnunk a kiválasztott listaelem indexéből, és az miért nem lehet 0?

Mert a nulladik indexű, azaz első eleme a „Válasszon” felirat, amire a felhasználó tájékoztatása miatt van szükség.

A határértékek megjelenítésének metódusa:

```
public void hatarertekBeallitas(int min, int max){
   this.minIndex=min;
   this.maxIndex=max;
   if(min==0)
        this.intervallumLabel.setText("");
   else
        this.intervallumLabel.setText("Választható: 
"+this.minIndex+"-"+this.maxIndex);
}
```

Ha a felhasználó megnyomta a gombot, akkor ez történik:

```
private void okButtonActionPerformed(java.awt.event.ActionEvent evt)
{
   int sorszam=0;
   try {
       sorszam = Integer.parseInt(this.sorszamBeiro.getText());
       if(sorszam<this.minIndex || sorszam>this.maxIndex)
          throw new IntervallumonKivuliErtekException(sorszam +": 
Intervallumon kívüli érték");
       else {
          this.valaszLiter.setText(hordo.getHordok().get(sorszam-1)
+ " liter");
          this.valaszSzazalek.setText(String.format("%.2f", 
hordo.getToltottsegSzazalek().get(sorszam-1))+" %");
       }
   } catch(NumberFormatException ex){
       JOptionPane.showMessageDialog(rootPane, "Nem szám formátum", 
"Hiba", JOptionPane.WARNING_MESSAGE);
   } catch(IntervallumonKivuliErtekException ex){
       JOptionPane.showMessageDialog(rootPane, ex.getSor(), "Hiba", 
JOptionPane.WARNING_MESSAGE);
   }
}
```

A töltöttségi százalék kiszámítását az OOP szemléletnek megfelelően az osztályban valósítjuk meg, mert ez az adott példányok tulajdonsága.

Utólag ki kell tehát egészítenünk az osztályunk tagadatait. Ezt azért tehetjük meg egyszerűen, mert az alap adatstruktúra a feladatnak megfelelően lett kialakítva

```
private ArrayList<Double> toltottsegSzazalek;
```

A konstruktorban inicializálni kell azt.

```
this.toltottsegSzazalek.add((double) liter / this.urmertek * 100);
```

Lekérdező metódust is kell generálni számára

```
public ArrayList<Double> getToltottsegSzazalek() {
     return toltottsegSzazalek;
}
```

A `JOptionPane` osztály üzenet ablakainak milyen típusai ismered?

A külső állományba kiírás metódusában hogyan rendezzük sorba az egész számok sorozatát tartalmazó `hordo` gyűjtemény elemeit?

A `Collections` osztály `sort()` osztálymetódusa sorba tudja rendezni a gyűjteményeket, így nem szükséges az elemi algoritmust megvalósítanunk.

```
public void kiiro(ArrayList<Hordo> hordok, String file){
   RandomAccessFile kiir;
   ArrayList<Integer> literek = new ArrayList<>();
   try {
      kiir = new RandomAccessFile(file, "rw");
      for (Hordo hordo : hordok) {
         literek = hordo.getHordok();
         Collections.sort(literek);
         for (Integer liter : literek) {
             kiir.writeBytes(liter+"/"+hordo.getUrmertek()+"\r\n");
         }
      }
      kiir.close();
   } catch (IOException ex) {
      System.out.println("Hiba: "+ex.getMessage());
   }
}
```