Objektum orientált programozás
11. gyakorlat
Kivételkezelés, fájlkezelés, dinamikus tömb

1. Saját kivétel: bankszámla fedezetének ellenőrzése.
Forrás: https://www.tutorialspoint.com/java/java_exceptions.htm

Definiáljon saját csomagban egy bankszámla osztályt.
Adattagjai: bankszámla száma, bankszámla egyenlege.
Konstruktora: egy paraméteres, a bankszámla számát kell megadni paraméterként. Új számla nyitásakor az egyenleg 0.
Metódusai:
- Getter metódusok a számlaszám és az egyenleg lekérdezéséhez.
- Pénzbetét: a paraméterként megadott összeggel növeli a számla egyenlegét.
- Pénzkivét: a paraméterként megadott össszeggel megpróbálja csökkenteni a számla egyenlegét. Ha nincs elég fedezet, kivételt dob (InsufficientFundsException – saját kivétel). A létrehozandó kivételobjektumnak át kell adni a tranzakció végrehajtásához hiányzó összeget.

Definiálja saját csomagban az InsufficientFundsException kivétel osztályt. Kötelezően lekezelendő kivétel legyen.
Adattagjai: a tranzakció végrehajtásához hiányzó összeg.
Konstruktora egy paraméteres, a hiányzó összeget inicializálja. Legyen getter metódusa a hiányzó összeg lekérdezéséhez.

Definiáljon egy futtatható osztályt, amelyben használja a bankszámla és a definiált kivétel osztályokat.
1. Hozzon létre egy bankszámla objektumot és helyezzen el rajta pénzt. Írassa ki a számla-egyenleget.
2. Próbáljon meg kétszer pénzt levenni a számláról. Először akkora összeggel, hogy sikerüljön. Másodszor úgy, hogy megdobódjon az InsufficientFundsException kivétel. Kezelje le a kivételt: írja ki, hogy nem sikerült a pénzlevétel és mekkora összeg hiányzik a számláról.

2. Fájlkezelés

Javaban szöveges és bináris adatállományokat lehet kezelni. A fájlkezelés lépései:
1. fájl objektum létrehozása,
2. fájl adatfolyam létrehozása a fájl objektumból,
3. fájlkezelő műveletek megadása,
4. fájl lezárása.

a) Szöveges fájlkezeléshez használható osztályok: FileReader és FileWriter. Ezek konstruktora kivételt dobhat.

```
public FileReader(String fileName) throws FileNotFoundException
public FileWriter(String fileName) throws IOException
```

Feladat: Egy futtatható osztályban tároljuk el dinamikus tömbben a csoport névsorát. Írjuk ki fájlba a listát, majd olvassuk vissza és módosítsuk a listát: vegyünk fel új hallgatót, ill. töröljünk hallgatót. Rendezzük a listát, majd írjuk vissza fájlba.

A feladat váza:

1. Dinamikus tömb létrehozása, feltöltése adatokkal

```
ArrayList<String> names = new ArrayList<String>();
names.add("Kiss Katalin");
```

2. File objektum létrehozása

```
File file = new File("names.txt");
```

3. Lista fájlba írása, majd visszaolvasása

```
writeInFile(file, names);
names = readFromFile(file);
```

4. Lista módosítása, rendezése, majd visszaírása fájlba.

ArrayList rendezése, ha a lista nem objektumokat tartalmaz: `Collections.sort(names);`

```
private static void writeInFile(File file, ArrayList<String> names) {
      //Ha a fájlt hozzáírásra (append) akarjuk megnyitni
      //a FileWriter 2 paraméteres konstruktorát kell használni úgy,
      //hogy a második hívási argumentum 'true'
       try (FileWriter fileWriter = new FileWriter(file))
       {
            for (String name : names) {
                fileWriter.write(name+"\n");
            }
            fileWriter.flush();           //output buffer kiürítése

        } catch (IOException e) {
            System.out.println("Error in writing file...");
        }
}
```

A try paraméterében megnyitott erőforrások a blokk végén automatikusan lezárásra kerülnek (try-with-resources szintaktika Java 7-től). Egyébként a programozónak kellene erről gondoskodni a close() metódushívással.

A FileReader osztály read metódusával karakterenként lehet fájlból olvasni. Most soronként szeretnénk és minden beolvasott sort hozzáfűzzük a nevek listájához. Ehhez a BufferedReader osztály readLine metódusát kell használjuk. Ez a metódus IOException kivételt dobhat.

```
private static ArrayList<String> readFromFile(File file) {
      ArrayList<String> names = new ArrayList<String>();
      String line;
      try ( FileReader fileReader = new FileReader(file);
            BufferedReader reader = new BufferedReader(fileReader))
      {
            while ((line = reader.readLine()) != null) {
                  names.add(line);
            }
      } catch (IOException e) {
            System.out.println("Error in reading file...");
      }
      return names;
}
```

Fájlból sorokat olvasni a Scanner osztály használatával is lehet:

```
ArrayList<String> names = new ArrayList<String>();
Scanner sc = new Scanner(file);
while(sc.hasNextLine()) {
      names.add(sc.nextLine());
}
```

b) Hozzunk létre saját csomagban (*myclasses*) egy Személy és egy Hallgató osztályt (a Hallgató a Személy leszármazottja). A Személy osztály adattagja egy név (sztring), a Hallgató osztály saját adattagja a neptunkód (sztring, vagy 6 elemű karaktertömb). Mindkét osztályhoz definiáljuk a generál metódusokat (konstruktor, toString, getter/setter metódusok). Készítsünk egy másik csomagban futtatható osztályt (myclasses.test), amelyben létrehozunk és feltöltünk egy dinamikus tömböt Hallgató objektum referenciákkal. Ezt a listát írjuk ki fájlba, majd olvassuk vissza a fájlból.

Hallgató lista létrehozása, feltöltése:

```
ArrayList<Student> list = new ArrayList<Student>();
list.add(new Student("Kiss Tamás","ab1234"));
```

Objektumok tárolása szöveges állományban:

```
for (String item : list) {
        fileWriter.write(item.getName()+";"+item.getNeptuncode()+"\n");
}
```

Objektumok olvasása szöveges állományból:

```
ArrayList<Student> list = new ArrayList<Student>();
…
while ((line = reader.readLine()) != null) {
        String[] instr = line.split(";");
        list.add(new Student(instr[0],instr[1]));
}
```

Objektumok tárolása bináris állományban.

Objektumok bináris fájlba írása: a tárolandó objektumoknak szerializálhatónak kell lenniük (az osztálytípus implementálja a *Serializable* interfészt)

```
public static void writeInFile(File file, ArrayList<Student> list) {
    try (FileOutputStream fout = new FileOutputStream(file);
         ObjectOutputStream objout = new ObjectOutputStream(fout))
    {
        for(Student item : list)
            objout.writeObject(item);

    } catch(Exception e) {
        System.out.println(e.getStackTrace());
    }
}
```

Objektumok bináris fájlból olvasása: a fájl végén `EOFException` dobódik, ami a normál működéshez tartozik, ezért ezt a hibakezelő részt (catch ágat) üresen szokták hagyni.

```
public static ArrayList<Student> readFromFile(File file) {
        ArrayList<Student> list = new ArrayList<Student>();
        try (FileInputStream fin = new FileInputStream(file);
             ObjectInputStream objin = new ObjectInputStream(fin))
        {
            Object obj;
            while((obj = objin.readObject()) != null) {
                if (obj instanceof Student)
                    list.add((Student)obj);       }
        } catch(EOFException e) {
            System.out.println("End of file reading ...");
        } catch(Exception e) {
            System.out.println(e.getStackTrace());
        }
        return list;
}
```

A `(Student)obj` konverzió akkor működik, ha a Hallgató osztályban van paraméter nélküli (default) konstruktor (ez hívódik meg). Mivel a Hallgató leszármazott osztály, a konstruktor hívási lánc működéséhez az ősben is kell legyen paraméter nélküli konstruktor.

Megjegyzés: a bináris állományban tárolt objektumok szerializálhatósága miatt ezen állományok append-módú (hozzáírásra történő) megnyitása nem tartozik az Objektum Orientált Programozás alapjait oktató tantárgy témakörébe (lásd később a Java programozás tantárgyban).

Dinamikus objektumtömb rendezése:
Az ArrayList a JCF (Java Collection Framework) keretrendszerben definiált osztály, ezért használhatók a Collections interfész metódusai.

a) A Comparable interfész implementálása esetén:

```
Collections.sort(list);
Collections.sort(list, Collections.reverseOrder());
```

A Hallgató osztályban implementálva az interfészt:

```
public int compareTo(Student o) {
      return this.getName().compareTo(o.getName());
}
```

b) A Comparator interfész implementálása esetén:

```
Collections.sort(list, new Student.IDSorter());
Collections.sort(list, new Student.IDSorter().reversed());
```

A Hallgató osztályban beágyazott osztályként, vagy külön osztályként definiálva:

```
public static class IDSorter implements Comparator<Student> {
    public int compare(Student o1, Student o2) {
        return o1.getNeptunID().compareTo(o2.getNeptunID());
    }
}
```

c) Listák rendezéséhez is használhatunk Java 8 Comparator-t:

```
Comparator<Student> compareById = (Student o1, Student o2) ->
                    o1.getNeptunID().compareTo( o2.getNeptunID() );
Collections.sort(list, compareById);
System.out.println(list);
```

Házi feladat

Hozzunk létre saját csomagban egy olyan osztályt, ami iskolai ellenőrzőben (egyetemi indexben) egy bejegyzést reprezentál.
Adattagjai: tantárgy neve, dátum, érdemjegy.
Konstruktora: a megadott három paraméterrel inicializálja az adattagokat. Ellenőrzést végez: ha az érdemjegy nem 1 és 5 közötti érték, InvalidMarkException kivételt dob (saját kivétel).
Metódusai: getter metódusok az adattagok lekérdezéséhez, toString.

Hozzuk létre saját csomagban az InvalidMarkException kivételosztályt.
Adattagja: érdemjegy.
Konstruktora: a hibás érdemjegy. Kiírja, hogy a megadott érdemjegy nem érvényes.
Metódusa: érdemjegy lekérdező metódus.

Hozzunk létre egy futtatható teszt osztályt.
1. Olvassunk be egy 5 elemű tömbbe index bejegyzéseket (tantárgy, dátum, érdemjegy). Az érdemjegyet szövegként olvassuk be és próbáljuk int-re konvertálni. Ha nem sikerül, automatikusan NumberFormatException kivétel dobódik. Ha sikerül, akkor próbáljunk létrehozni egy index bejegyzés objektumot. Ha az érdemjegy nem 1 és 5 közé esik, akkor InvalidMarkException kivétel fog dobódni.
2. Az index bejegyzéseket tároló tömböt rendezzük dátum szerint.
3. Írjuk ki fájlba a tömb tartalmát.
4. Listázzuk a fájl tartalmát.
5. Írjuk át úgy a feladatot, hogy a beolvasott index bejegyzéseket dinamikus tömbben tároljuk; és végezzük el így a fájl műveleteket és a rendezést.