Ütemezés feladat – Az objektum orientált megoldás főbb lépései:

El kell döntenünk, a feladat megoldása során alkalmazunk-e objektumokat, és ha igen milyeneket! Az objektumok, többek között, képesek hatékonyan modellezni a tömeges egyed előfordulásokat. Amikor egy feladatban sok azonosan viselkedő entitás jelenik meg, nagy valószínűséggel érdemes lesz bevezetni az entitásokat illetve viselkedésüket leíró új típus osztályt.

Az általunk bevezetett osztály neve *tabor* lesz. A *tabor* osztály egyes példányai a text állományban felsorolt nyári táborokat fogják képviselni.

Osztály bevezetése esetén meg kell terveznünk annak struktúráját, melyen keresztül az osztály képes lesz relevánsan megjeleníteni a „tömegjelenség" keretei között előforduló, azonosan viselkedő entitásokat! Ki kell tehát alakítani az osztálytagokat: az objektum mindenkori állapotát leíró állapotjelzőket (példányváltozókat), illetve az objektum cselekvőképességét leíró metódusokat.

A *tabor* osztály állapotjelzői: a tábor kezdetének és befejezésének dátuma, a tábort választó tanulók azonosítói és a tábor témája. Ezeket az állapotjelzőket a fejlesztés során továbbiakkal egészítjük ki.

A példányosítás során az állapotjelzőket inicializálnunk kell. Úgy döntünk, hogy az adatokat eredeti állapotban, string típusú rekordként adjuk át az objektumpéldányoknak. Emiatt az objektumoknak rendelkezni kell egy adat-kicsomagoló metódussal. A kicsomagolás elindítását a „születő" objektumpéldányok konstruktorára bízzuk. A konstruktor az objektum példányváltozóit is inicializálni fogja ezután.

A példányváltozókat inicializáló metódus neve *beallitAdatok()*. Az átadott adatrekord tárolását az *adatokStr* példány adattag valósítja meg.

Az adatok kicsomagolását követően inicializálunk egy származtatott állapotjelzőt is, ami a tábor kezdőnapjának a nyári szünet napjai alapján számolt napsorszáma lesz. A kezdőnap sorszámának kiszámítását is az objektumra bízzuk. Ennek érdekében ellátjuk az objektumot egy megfelelő, kezdőnap sorszámot számoló metódussal.

A kezdőnap sorszámát kiszámító *Sorszám(kezdőHónap, kezdőNap)* metódust a konstruktor hívja meg a példányosulás során, ezután inicializálja a *knapSorszam* változót is.

A *tabor* példányokat létrehozásuk pillanatában egy *taborok* nevű lista elemeiként láncolt listába szervezzük, azaz gyűjteményt készítünk a „tömegjelenség" példányaiból.

```
List<Tabor> taborok = new ArrayList<>();
Tabor tabor = new Tabor(olvasottSor); taborok.add(tabor);
```

A gyűjtemény elemeinek összehasonlíthatósága érdekében összehasonlító metódust definiálunk, amit szintén átadunk a *tabor* osztálynak. *Tabor* osztályunk végre működőképes, kialakítottuk az objektum orientált megoldás kellékeit.

```
Comparator<Tabor> SORSZ=new Comparator<Tabor>(){ ... }
```

A külső adatokat az adatszolgáltató karakterkódolást alkalmazva becsomagolta. A logikailag összetartozó adatok egymás mellé, azonos sorba kerültek, a sorokban szereplő adatokat szóköz karakter választja el:

6 26 7 ... foci¶
7 14 7 ... szinjatszo¶
7 27 8 ... hittan¶
...
8 16 8 ... filmes¶

A sorokba csomagolt adatrekordok nagyobb adategységbe, fájlba lettek csomagolva, ezzel alkalmassá váltak a „szállításra”.

taborok.txt:6_26_7_ ... _fociCRLF7_14_7_ ... _szinjatszoCRLF ... _filmesCRLF

Miután megkapjuk az adatokat, az adatok kicsomagolását a becsomagoláskor alkalmazott műveletek inverz műveleteinek alkalmazásával, két lépésben kell elvégezni. Az első lépésben a nagyobb csomagból, azaz a fájlból vissza kell nyerni az adatsorokat. Ezt a műveletet egy átmeneti tárolóba olvasni képes *BufferedReader* objektumra bízzuk, amely a *CRLF* sorhatároló jelek alapján képes visszabontani sorokra, azaz konzisztens adatrekordokra az adatállományt, „eldobva” a most már felesleges rekordhatároló jeleket. A második lépésben az egyes sorokat a szóköz határoló karakterek figyelembe vételével adategységekre bontjuk vissza. Ezt a műveletet egy *Scanner* objektumra bízzuk. A két művelet egymásutánjának mondatszerű elnevezése: *Kicsomagolás(külsőAdatok)*.

A kicsomagolást követően az adatokat burkoló „szállító struktúrák” „szétbomlanak”, gondoskodni kell tehát új releváns tároló struktúrákról. A strukturált illetve objektum orientált megoldást az különbözteti meg egymástól, hogy a kicsomagolás műveletét és az új releváns tároló struktúrák kialakítását különböző szemlélet alapján valósítjuk meg az egyes megoldásokban. A strukturált megoldásban a *Kicsomagolás* metódusa a „műveleti tér felett lebeg”, ami azt jelenti, hogy mi vagyunk felelősek a kicsomagolásért. Ezért a kicsomagolást közvetlenül a *main* metódusban mi magunk végezzük el explicit módon. Ebből következően a kicsomagolt adatok tároló struktúráinak létrehozásáról is nekünk kell gondoskodni. A *main* metódusban például a kezdő és befejező dátumok tárolására egy explicit kétdimenziós *sz [ ] [ ]* tömböt használunk fel, illetve a szövegsorok számának megfelelő méretű string tömbben tároljuk a jelentkezők nevének azonosítóit illetve a táborok témáit.

```
public static void main(String[] args)throws IOException {
...
int [][] sz = new int [sorDb][4];        //sz: számadatok
String [] b = new String[sorDb];         //b: betű azonosítók
String [] t = new String[sorDb];         //t: téma

for(int j=0;j<sorDb;j++){                //Kicsomagolás >>
  int i=0;
```

```
  Scanner scan = new Scanner(olvasottSor[j]);
  while(scan.hasNext()){
    if (scan.hasNextInt()){
      sz[j][i]=scan.nextInt();
      i++;
    } else {
        b[j]=scan.next();
        t[j]=scan.next();
      }
  }
}                                          //>> Kicsomagolás

...
}
```

A Java nyelvben valójában nem lehet képességeket „lebegtetni”. Képességek csak az objektumoknál lehetnek. Minden olyan képesség, ami látszólag egyetlen objektumnál sincs, ezért úgy tűnik, hogy „lebeg”, az valójában a *main* objektumnál van. A main objektum pedig a programozót testesíti meg!

Az objektum orientált megoldás irányába mozdulunk el, ha az adatstruktúrákat nem a main keretei között explicit definiáljuk és az adatok kicsomagolását, feldolgozását arra alkalmas objektumoknak adjuk át. A táborok kezdő és befejező dátumait nem egy publikus kétdimenziós tömbben tároljuk, hanem a szövegsorok számának megfelelő számú *tabor* objektumot példányosítunk és az egyes tábor objektumoknak példányosításuk során odaadjuk a kicsomagolás eredményét. Az adatok tárolása ettől kezdve a *tabor* objektumok „felelőssége”. Az objektumok védett módon, private módosító használatával tárolhatják a kapott adatokat, ezáltal megszűnik annak a lehetősége, hogy véletlenül felülírjuk az előző változatban még explicit módon nyilvánosan tárolt adatokat.

```
class Tabor{
  private int kezdHo; private int kezdNap;
  private int befHo; private int befNap;
  private String nevBetuk;private String taborTema;
  public Tabor(int kHo,int kNap,int bHo,int bNap,String betuk,String tema){
    kezdHo = kHo;kezdNap=kNap;
    befHo = bHo;befNap=bNap;
    nevBetuk=betuk;
    taborTema = tema;
  }

  ...
}

...
Tabor tabor = new Tabor(sz[i][0],sz[i][1],sz[i][2],sz[i][3],b[i],t[i]));
...
```

A vázolt megoldásban az adatokat duplikáltan tároljuk. Megvalósítottuk az explicit tárolást, de a biztonság kedvéért a kicsomagolt adatokat odaadtuk a *tabor* objektumoknak konstruktoraikon keresztül.

Megszüntethetjük ezt a kettősséget, ezáltal áttekinthetőbbé és logikusabbá tehetjük az objektum orientált megoldást, ha az adatok kicsomagolását végző metódust is odaadjuk a *tabor* objektumnak. Az adatokat a fájlburok eltávolítása után, eredeti csomagolásban kapják meg az objektumok, és saját maguk csomagolják ki azokat. Innentől kezdve az adatok teljes körű védelmet élveznek. Tulajdonképpen egyelőre nem is lehet hozzájuk férni! Írni kell tehát olyan nyilvános metódusokat, amelyek egy külső objektum érdeklődése esetén képesek tájékoztatást adni a most már belső adatok értékéről.

```
class Tabor{
  private int [] szam = new int[4];
  private String nevBetuk;private String taborTema;
  private int knapSorszam; private String adatokStr;
  public Tabor(String adatokStr) {
    this.adatokStr=adatokStr;
    this.beallitAdatok();
  }
  private void beallitAdatok(){
    Scanner scan = new Scanner(adatokStr);
    int i=0;
    while(scan.hasNext()){
      if (scan.hasNextInt()){
        szam[i]=scan.nextInt(); i++;
      } else {
          nevBetuk=scan.next();
          taborTema=scan.next();
        }
    }
  }
...
  public String lekertaborTema() {      //getter metódus
    return taborTema;
  }
...
}
public class Utemezes {
public static void main(String[] args)throws IOException {
  String olvasottSor;
  BufferedReader sorOlvaso = new BufferedReader(new FileReader("taborok.txt"));
  List<Tabor> taborok = new ArrayList<>();
  do {
    olvasottSor=sorOlvaso.readLine();
    if (olvasottSor!=null){
      Tabor tabor = new Tabor(olvasottSor); taborok.add(tabor);
    }
  } while (olvasottSor!=null);
...//listabejárás
}
```

Ebben a most bemutatott megoldásban nem 100 elemű String tömbbe olvassuk be a txt fájl sorait, ahogy azt a strukturált megoldás során tettük, hanem egy *ArrayList* sorrendi Lista elemei lesznek a sorok. Pontosabban a *List<Tabor>* interfészt megvalósító *ArrayList<>* elemei *Tabor* objektumok lesznek, melyek mindegyike megkap egy-egy egymást követő szövegsort paraméterként. Ebben a megoldásban láncolt listával dolgozunk, így nincs explicit *sorDb* változónk, amit a korábbi megoldásban a String tömb elemeinek indexeléséhez használtunk. A beolvasás során így nem értesülünk közvetlenül a beolvasott sorok számáról.

Ezt a megoldási variációt a korábbi strukturált megoldásban alkalmazott minimum kiválasztásos rendezés megkerülésének vágya motiválja. A gyűjteményeken ugyanis definiálva van egy *sort()* metódus, ami nagyfokú kényelmet biztosít adatok rendezéséhez, és ezt ki szeretnénk használni! A rendezést a nyári tábor kezdőnapjának sorszáma alapján kell elvégezni, az értéket a *Sorszam(int kezdHo, int kezdNap)* metódus szolgáltatja.

A *Sorszam(int kezdHo, int kezdNap)* metódust elhelyezhetjük a *main* metódust is tartalmazó, főprogramot megvalósító *Utemezes* osztályban. Ekkor a *Tabor* segédosztály belsejében *minősített név*vel kellene hivatkozni erre a metódusra: *Utemezes.Sorszam(kezdHo, kezdNap).*

Megoldásunk természetesebbé válik, ha a *Sorszam(kezdHo, kezdNap)* metódust is a *Tabor* osztályba helyezzük. A *taborok* lista *Tabor* elemeinek speciális, kezdősorszám szerinti rendezhetősége érdekében konstruálnunk kell még egy *Komparátor* objektumot is:

```
Comparator<Tabor> SORSZ = new Comparator<Tabor>().
```

A komparátor objektum belsejében definiálnunk kell a *compare* összehasonlítást végző metódust:

```
public static final Comparator<Tabor> SORSZ=new Comparator<Tabor>() {
  public int compare(Tabor o1, Tabor o2) {
    return (int)(o1.knapSorszam)- (int)(o2.knapSorszam);
  }
};
```

Az objektumokat aktuális pillanatnyi állapotuk alapján hasonlíthatjuk össze egymással. Az aktuális állapotot minden esetben az objektum állapotjelzői írják le az állapotjelzők értékein keresztül. Ezek az állapotjelzők, mint láttuk, az osztálypéldányok saját példányváltozói. A *Tábor* példányokat összehasonlíthatóságuk érdekében tehát elláttuk egy `knapSorszam` állapotjelzővel is, amit a `Sorszam(kezdHo, kezdNap)` metódus segítségével inicializáltunk. Természetesnek tűnik ezek után „odaadni” az objektum állapotjelzőjét inicializáló metódust magának az objektumnak. A *Tabor* objektumok rendezése során a `compare(Tabor o1,Tabor o2)` összehasonlító metódus `(o1.knapSorszam)-(o2.knapSorszam)` visszatérési értéke ezáltal kiszámítható lesz.

A *Sorszam(int kezdHo, int kezdNap)* metódus meghívásakor a helyes működés érdekében a Tabor objektumnak már rendelkeznie kell két szükséges állapotjelzővel, ugyanis a *knapSorszam* állapotjelzőt ezekből az állapotjelzőkből származtatjuk. Ezért a *Tabor* osztály konstruktorát, mint „cselekvőt” nemcsak arra kell utasítanunk, hogy vegye át a kicsomagolatlan adatsort, hiszen a

puszta adatátvétel, az adatok kicsomagolatlansága miatt „állapottalan állapotba” hagyná az éppen példányosult objektumot! Fel kell szólítanunk az adatok kicsomagolására, és a belső változók feltöltésére, az objektum „állapotba hozására”:

```
class Tabor{
  public static int SorszamInt(int ho, int nap){
    int napSorszam=0;
    int iskutolso=15;
    if (ho==6){napSorszam=nap-iskutolso;}
    if (ho==7){napSorszam=15+nap;}
    if (ho==8){napSorszam=46+nap;}
    return napSorszam;
  }
  private int [] szam = new int[4];
  private String nevBetuk;private String taborTema;
  private int knapSorszam; private String adatokStr;
  public Tabor(String adatokStr) {
    this.adatokStr=adatokStr;
    this.beallitAdatok();
  }
  private void beallitAdatok(){
    Scanner scan = new Scanner(adatokStr);
    int i=0;
    while(scan.hasNext()){
      if (scan.hasNextInt()){
        szam[i]=scan.nextInt();
        i++;
      } else {
          nevBetuk=scan.next();
          taborTema=scan.next();
        }
    }
    knapSorszam=SorszamInt(szam[0],szam[1]);
  }
  ...
  public String lekertaborTema() {
    return taborTema;
  }
  public int lekerknapSorszam() {
    return knapSorszam;
  }
  public static final Comparator<Tabor> SORSZ=new Comparator<Tabor>() {
    public int compare(Tabor o1, Tabor o2) {
      return (int)(o1.knapSorszam)- (int)(o2.knapSorszam);
    }
  };
}
```

Ezt követően rendezhetjük sorrendi Listánkat és kiírathatjuk az eredményeket. A lista bejárásához saját iterátort definiálunk az Iterátor interfész felhasználásával. Ehhez a List interfészt megvalósító sorrendi Listánk segítséget nyújt:

```
public class Utemezes {
public static void main(String[] args)throws IOException {
  ...
  } while (olvasottSor!=null);
//------------ nyári táborok listájának bejárása ---------------
  for (Iterator<Tabor> it = taborok.iterator(); it.hasNext();) {
    Tabor t = it.next();
    System.out.println(t.lekerknapSorszam()+"-"+t.lekertaborTema()+
                       " "+t.lekerkezdHo()+". "+t.lekerkezdNap());
  }
//- nyári táborok kezdőnap szerint rendezett listájának bejárása -
  System.out.println("Kezdőnap szerint rendezett: ");
  Collections.sort(taborok,Tabor.SORSZ);
  for (Iterator<Tabor> it = taborok.iterator(); it.hasNext();) {
    Tabor t = it.next();
    System.out.println(t.lekerknapSorszam()+"-"+t.lekertaborTema()+
                       " "+t.lekerkezdHo()+". "+t.lekerkezdNap());
  }
}
```

```
run:
11 - foci 6. 26
29 - szinjatszo 7. 14
42 - hittan 7. 27
43 - cserkesz 7. 28
24 - gombasz 7. 9
63 - sakk 8. 17
3 - evezos 6. 18
12 - sport 6. 27
61 - uszas 8. 15
55 - hegymaszo 8. 9
59 - torna 8. 13
50 - zenei 8. 4
10 - falmaszo 6. 25
3 - zenei 6. 18
70 - regesz 8. 24
7 - cserkesz 6. 22
73 - fotos 8. 27
42 - korus 7. 27
26 - fuvos 7. 11
23 - erdojaro 7. 8
65 - matek 8. 19
39 - fizika 7. 24
17 - csillagasz 7. 2
53 - cserkesz 8. 7
23 - cukrasz 7. 8
63 - futo 8. 17
30 - kornyezettudatos 7. 15
62 - filmes 8. 16

Kezdőnap szerint rendezett:
3 - evezos 6. 18
3 - zenei 6. 18
7 - cserkesz 6. 22
10 - falmaszo 6. 25
11 - foci 6. 26
12 - sport 6. 27
17 - csillagasz 7. 2
23 - erdojaro 7. 8
23 - cukrasz 7. 8
24 - gombasz 7. 9
26 - fuvos 7. 11
29 - szinjatszo 7. 14
30 - kornyezettudatos 7. 15
39 - fizika 7. 24
42 - hittan 7. 27
42 - korus 7. 27
43 - cserkesz 7. 28
50 - zenei 8. 4
53 - cserkesz 8. 7
55 - hegymaszo 8. 9
59 - torna 8. 13
61 - uszas 8. 15
62 - filmes 8. 16
63 - sakk 8. 17
63 - futo 8. 17
65 - matek 8. 19
70 - regesz 8. 24
73 - fotos 8. 27
```