2.5 Kollekciok, Generics

Kollekciók

Az egyszer tömböknek már a korábban látott módokon lehetnek hiányosságai. Részben ezek kiküszöbölésére alkalmasak az ún. kollekciók (collections), amelyek a tömbökhöz hasonlóan egy bizonyos típus tárolására szolgálnak, ám további funkcionalitásokkal is bírnak. Használatuk kényelmes, és nagyban egyszersítheti munkánkat. Kett ilyet fogunk sorra venni a következkben, ezek a lista és a halmaz, valamint találkozni fogunk az asszociatív tömb (leképezés, dictionary, map) fogalmával is. Bvebben a kollekciókról.

Listák (List)

A tömböknél például továbbra is nagy problémát jelenthet, hogy feltöltés eltt meg kell adni számukra a maximális méretet. Ez azt is jelenti, hogy elvigyázatosságból olykor feleslegesen nagy tömböket tárolhatunk, amelyek nagy része kitöltetlen marad. Ez az információ sokszor csak futás közben derül ki, a program írásakor még nem (például tetszleges számú elem érkezik parancssori paraméterben), illetve ami még nagyobb gondot okozhat, az is megeshet, hogy a tömb létrehozásakor futás közben sem tudjuk még, hogy hány elemet szeretnénk benne tárolni (például a felhasználó tetszleges számú elemet ad meg konzolon, ezeket el kell tárolni). Ezért az egyszer tömb használatával komoly nehézségekbe ütközhetünk.

További problémát okozhat az is, hogy a tömbök már korábban látott `length` tulajdonsága a maximális számát tárolja, így ha elvigyázatlanul egy ciklust például ennyiszer ismétlünk, akkor könnyedén olyan elemre hivatkozhatunk, amely nem is létezik, és adott esetben akár `NullPointerException` típusú kivételt is kaphatunk. Így mondjuk egy változóban le kell tárolnunk, ténylegesen mennyi elem van a tömbben, és erre figyelni, de ez is hibalehetségeket rejthet.

Ezt a problémát illusztrálja a következ példa, amely tetszleges számú lebegpontos számot olvas be, és kiírja a szorzatukat (legtöbb esetben helytelenül):

```
import java.util.Scanner;

public class HelytelenOsszeadasKonzolrol {
    public static void main(String[] args) {

        double[] szamok = new double[100];
        Scanner sc = new Scanner(System.in);

        //addig olvassunk be számot, amíg 1-t nem kapunk
        int i=-1;
        do {
            i++;
            szamok[i]=sc.nextDouble(); //hibás, ha a felhasználó több számot akar 100-nál (kifut)
        } while(szamok[i]!=1);

        //számoljuk ki a kapott számok szorzatát
        int szorzat=0;
        for(i=0;i<szamok.length;i++) {
            szorzat*=szamok[i]; //hibás, ha a felhasználó nem pont 100 számot adott (nulláz)
        }

        System.out.println(szorzat);
    }
}
```

Illetve más esetekben is komplikált lehet a tömbök kezelése, még ha jól is kezeli ket az ember. Ez látható például egyszersített megoldáson az állatos példa `Csorda` osztályának `csordábaFogad` metódusára:

```
private int maximum = 100;
    private Allat[] tagok = new Allat[maximum];
    private int jelenlegi=0;

    public void csordabaFogad(Allat kit) {
        if (jelenlegi < maximum) {
            tagok[jelenlegi]=kit;
            jelenlegi++;
        }
    }
```

Ez a kód helyesen mködik, ám igencsak komplikált, illetve a maximumot túllép csordát nem tud kezelni.

Ezekre a problémákra megoldást nyújthat az ún. **lista (List)**, amely a tömbhöz nagyon hasonló mködés, ám sokkal rugalmasabban kezelhet. Ez a tömbhöz hasonlóan továbbra is egy típusból tud tárolni elemeket, ám ez tetszleges méretet felvehet, kevés tárolt adatnál kis méret, sok adatnál

nagy. Ez nem csak a memória-spórolás szempontjából fontos, ugyanis pontosan annyi eleme lesz, amennyit mi hozzáadunk. Tulajdonképpen megegyezik a *Programozás alapjain* már látott dinamikus tömb mködésével, ám itt nincs szükség mutatóval való foglalásra és felszabadításra, használata igen egyszer. Deklarációja a következképpen nézhet ki:

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class Listak {
    public static void main(String[] args) {
        List<Integer> lista = new ArrayList<>(); //tömbös megvalósítás
        List<Integer> lista2 = new LinkedList<>(); //láncolt listás megvalósítás
    }
}
```

A java.util csomag List osztálya egy interface, amely a már átvett ismereteink alapján azt jelenti, hogy önmagában nem végzi el a mveleteit, ez az t megvalósító osztályok dolga. Ennek megvalósításai viszont már használhatóak, ezek közül választhatnunk. Ezek lehetnek például az **ArrayList** és a **LinkedList**, de további megvalósítások is rendelkezésre állnak, ezekrl bvebben itt olvashatsz.

A két osztály pontosan ugyanazokat feladatokat látja el, csak a mögöttes mködésükben térnek el egymástól, de minden mveletük és ezek helyessége megegyezik. Az ArrayList egy tömbös megvalósításon alapul, a LinkedList pedig láncolt listákon, amit elz órán •mi is megvalósítottunk kézzel. A két listatípus tehát használati szempontból teljesen ugyanaz, **hiszen mindkett ugyanazt az interfészt implementálja**.

GENERIKUS TÍPUSMEGADÁS

A látott deklaráció elsre kicsit furcsának tnhet. A tárolt adatok típusának megadása itt **<>**(kacsacsrök) között történik. Egy `List<Double>` típus tehát egy lista, amely lebegpontos elemeket tárol. Amint látható, itt nem az egyszer primitív típusokat, hanem azok csomagoló (wrapper) osztályait kell megadni. Az egyenlségjel után pedig már egy konkrét megvalósítás konstruktorával kell példányosítanunk, Java 7.0 vagy afeletti verzióban már nem fontos a típus megadása újra, elegend az üres kacsacsröket kitenni (diamond operátor), ezzel is megkönnyítve a dolgunkat. Maximális méret megadására nincs szükség, az újonnan létrehozott lista mindig üres, és 0 elem.

Az imént látott szintaxis a generikus típusmegadást jelölik. Errl bvebben hallhatsz az eladáson. Gyakorlatilag statikus polimorfizmusról van szó, egy típusparamétert adunk meg, mivel az osztály maga úgy lett megírva, hogy a lehet legáltalánosabb legyen, és ne kelljen külön `IntegerList`, `StringList`, `AllatList`, stb. osztályokat megírnunk, hanem egy általános osztályt, mint sablont használunk, és a tényleges típust a kacsacsrök között mondjuk meg.

GENERIKUS OSZTÁLYOK

Ez természetesen nem csak a listák, leképezések (mapek) esetében használható, mi is csinálhatunk ilyen osztályokat minden további nélkül. A következekben egy nagyon egyszer osztályt mutatunk be:

```
public class ElrejtettErtek<GenerikusTipus> {
    private GenerikusTipus ertek;

    public ElrejtettErtek(GenerikusTipus ertek) {
        this.ertek = ertek;
    }

    public GenerikusTipus getErtek() {
        return ertek;
    }

    public void setErtek(GenerikusTipus ertek) {
        this.ertek = ertek;
    }

    @Override
    public String toString() {
        return "ElrejtettErtek [ertek=" + ertek + "]";
    }
}
```

Ez egy olyan osztály, ami egy valamilyen típusú értéket tud tárolni, erre van egy getter és egy szetter függvény, valamint egy `toString` metódus . Tehát, ha én példányosításkor azt mondom, hogy `ElrejtettErtek<String> ertek = new ElrejtettErtek<>("Szeretem az almát!");`, akkor a létrejöv objektumban egy szöveget tudok eltárolni, és így tovább.

```
public static void main(String[] args) {
        ElrejtettErtek<String> ertek = new ElrejtettErtek<>("Szeretem az almát!"); // Egy szöveget tudok
így tárolni
        ertek.setErtek("Sikerül vajon?"); // Igen
        //ertek.setErtek(103); //Nem fog sikerülni
        ElrejtettErtek<Integer> szamErtek = new ElrejtettErtek<>(120); // Egy egész számot tudok így
tárolni
        ElrejtettErtek<Allat> allatErtek = new ElrejtettErtek<>(new Medve("Jason")); // Egy állatot tudok
így eltárolni
    }
```

Vissza a listákhoz

Használatuk igencsak egyszer, új elem hozzáadása az `add` metódusával történik, ami olyan elemet vár, mint amilyen maga a lista. Ilyenkor a lista dinamikusan bvül, tehát amennyiben alapállapotában adjuk ki az utasítást, létrejön benne a 0. index elem, illetve már létez elemeknél a következ szabad indexen érhet el az új elem. Új elem tetszleges indexre is beszúrható, ilyenor az addig azon az indexen lév, és az összes nála nagyobb index elem egy indexszel feljebb lép, ezt az `add(index, elem);` metódussal vihetjük véghez, hasonló a sima `add`-hoz, csak az elem elé a kívánt indexet is meg kell adnunk. A lista rendelkezik egy `size` (vigyázat, ez nem length! és nem tulajdonság!) metódussal, amely az elemeinek számát tárolja, ez automatikusan változik a lista növekedésével/csökkenésével.

Egy bizonyos elemre itt nem `[]` jelek között kell hivatkoznunk, hanem a `get` viselkedést meghívva, például `lista.get(0)` a 0. index elem lekérése. A külsségektl eltekintve ez ugyanúgy indexelhet, mint a tömb.

Az elemek törlése is igencsak intuitív. Ezt a `remove` metódussal tudjuk meghívni. Ennek két típusa is létezik. Egyrészt megadhatjuk az indexet, másrészt konkrét elemet is adhatunk, amelynek els példányát töröljük.

> *Ha van egy `Integer` elemeket tároló listánk, akkor figyelni kell, mert ha nem index, hanem elem szerint szeretnénk törölni, akkor az alapesetben nem fog mködni, hiszen az int és csomagoló típusa nagyon összekeverhet lehet. Ilyen esetben, ha egy adott érték elemet szeretnénk törölni, akkor* ***mindig castolni kell****, például a `list.remove(3)` helyett írjunk `list.remove((Integer)3)`, vagy `list.remove(Integer.valueOf(3))`, esetleg `list.remove(new Integer(3))` parancsot.*

Ezen parancsokat a következ példakód illusztrálja:

```
List<Double> lista = new ArrayList<>(); //Polimorfizmus miatt kezelhetjük List-ként.

        //beszúrás a lista végére
        lista.add(1.2);
        lista.add(2.1);
        lista.add(3.10);
        lista.add(2.1);
        lista.add(3.05);

        //beszúrás a legels helyre -> 3.55, 1.2, 2.1, 3.1, 2.1, 3.05
        lista.add(0, 3.55);

        //a legels elem törlése -> 1.2, 2.1, 3.1, 2.1, 3.05
        lista.remove(0);

        //a legels 2.1 érték törlése -> 1.2, 3.1, 2.1, 3.05
        lista.remove(2.1);
```

Egy listához hozzáadhatunk egy másik listát is, a listák `addAll` metódusával, amely paraméterül a másik kollekciót várja.

Bejárás

A listák bejárására több módszer is lehetséges.

Egy már megszokott módszer lehet az index alapján történ bejárás, ahogy azt tömböknél is szokás:

```
for (int i=0; i < lista.size(); i++) {
            System.out.print(lista.get(i) + " ");
        }
```

Egy másik lehetség **iterátor** használata. Az iterátor egy olyan objektum (nem mellesleg az *Iterátor tervezési mintát valósítja meg, amelyrl bvebben az eladáson hallhatunk), amely képes egyenként bejárni a kollekciók összes elemét. Deklarációjakor szintén <> jelek között adhatjuk meg a típust, illetve ezután `new` kulcsszó helyett a lista `iterator()` metódusát hívjuk meg, amely elkészíti a megfelel iterátort. Az iterátorral használat közben lépkedni kell, amíg az utolsó elemet el nem érjük. Ezt általában célszer ciklussal tenni (legtöbbször while ciklussal). Azt lekérni, hogy az utolsó elemnél tartunk-e az iterátor `hasNext()` metódusával tudjuk, amely `boolean` értéket ad vissza. Ez nem állítja automatikusan a következ elemre az iterátort, azt a `next()` metódus teszi, amely a léptetésen túl visszatér a következ elemmel. Ennek hívásakor ügyelni kell arra,

hogy ez a ciklustól függetlenül is minden híváskor lépteti az iterátort, így ha nem tároljuk ideiglenes változóban az értéket (pl.: `double ideigl = it.next();`), akkor beleeshetünk abba a csapdába, hogy kétszer léptetjük két ellenrzés között, lekérve a következ következjét is, amelynek létezésére már nincsen garancia, st az utolsó elem léptetése után garantáltan hibás lesz.

```
Iterator<Double> it = lista.iterator();
        while(it.hasNext()) {
            Double elem = it.next();
            System.out.print(elem + " ");
        }
```

Egy harmadik lehetség, ha a for ciklus elemenkénti bejárását alkalmazzuk. Ez nagyon könny és értelemszer használatot biztosít. A megszokott for struktúrája helyett itt nem lesznek pontosvesszk, sem megállási feltétel. Egy elemet deklarálunk, amely a lista elemeinek típusával rendelkezik, utána kettsponttal elválasztva a lista nevét. A ciklus minden futásakor a következ elem fog a deklarált elembe kerülni. Ez a for ciklus a háttérben szintén iterátorral dolgozik, a különbség annyi az elz megoldáshoz képest, hogy ebben az esetben nem tudunk róla. :)

```
for (double elem : lista) {
            System.out.print(elem + " ");
        }
```

Amennyiben a lista összes elemét törölni szeretnénk, a `clear()` metódus alkalmazható (például `lista.clear()`).

A fent látott állatos példa `csordabaFogad` metódusa listákkal a következre egyszersíthet:

```
private List<Allat> tagok = new ArrayList<>();

    public void csordabaFogad(Allat kit) {
        tagok.add(kit);
    }
```

Törlés

Ahogy korábban is szó volt róla, a listából a `remove` nev metódussal törölhetünk egy elemet. Ez egyszernek hangzik, de egy esetben nem fog mködni: ha a listából mondjuk bejárás közben szeretnénk törölni. Nyugodtan próbáljuk meg törölni az összes elemet, amik mondjuk 1.5-nél kissebbek:

```
List<Double> lista = new ArrayList<>();

        lista.add(1.2);
        lista.add(2.1);
        lista.add(3.10);
        lista.add(2.1);
        lista.add(3.05);

        for(double elem : lista) {
            if(elem < 1.5){
                lista.remove(elem);
            }
        }
```

Ilyen esetekre jön jól a már ismertetett iterátor. Az iterátorral történ bejárás során egyszeren törölhetünk bármilyen nekünk nem tetsz elemet, az **it erátor** `remove` metódusát használva.

```
Iterator<Double> it = lista.iterator();
        while(it.hasNext()) {
            double elem = it.next();
            if(elem < 1.5){
                it.remove();
            }
        }
```

Halmazok (Set)

Az ún. halmaz (Set) a listához igencsak hasonló mind funkciójában, mind mködtetésében. Ez is interface, és ennek is két fajtáját érdemes ismernünk, a HashSet-et, amely hasítótáblás, illetve a TreeSet-et, amely piros-fekete fás megvalósítást jelöl. Ezen implementációk használata is teljesen megegyez egymással.

A listákhoz hasonlóan ugyanúgy egy típusból tárolhatnak tetszleges számú elemet, és ugyanúgy dinamikusan bvülnek. Ugyanúgy használható az `add` és `remove` metódus is. Két alapvet eltérés van a listáktól:

- **A halmazok minden elemet csak egyszer tartalmazhatnak.** Tehát mint amikor matematikai halmazokról beszélünk, azt nem tartjuk számon, hogy hány darab van egy elembl benne, csak hogy egy bizonyos elemet (például számot) tartalmaz-e. Ezzel kapcsolatban

felmerülhet azonban a kérdés, hogy mi van akkor, ha hozzáadunk egy számokat tároló halmazhoz egy 2-es elemet, aztán egy 3-ast, végül egy új 2-est. Ilyenkor érthet, hogy a 2-es is csak egyszer lesz benne, de mi történik az indexekkel? A választ a következ pontban találjuk:

- **A halmazok elemei index szerint nem rendezettek.** Tehát nem lehet ket index szerint lekérni, tehát az nem is tárolódik, hogy milyen sorrendben helyeztük bele az elemeket, csak az, hogy benne vannak-e.

Deklarációjuk a listákéhoz nagyon hasonló:

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class Halmazok {
    public static void main(String[] args) {
        Set<Integer> halmaz = new HashSet<>();  //hasítótáblás megvalósítás
        Set<Integer> halmaz2 = new TreeSet<>(); //piros-fekete fás megvalósítás
    }
}
```

Contains

A halmazok egyik legfontosabb tulajdonsága lehet, hogy tartalmaznak-e egy bizonyos elemet. Ezt könnyedén lekérdezhetjük a `contains` metód usának meghívásával. Ennek használata igen egyszer:

```
if(halmaz.contains(2)) System.out.println("A halmazban van 2");
        else System.out.println("A halmazban nincs 2");
```

Természetesen ez nem csak számokkal tud mködni, ha például az Állat osztályból származó példányokat teszünk a halmazba, akkor is használható.

Bejárás

Az elemek bejárására nagyjából ugyanúgy vannak lehetségeink, ahogy a listáknál. Természetesen mivel itt az index nem értelmezett a halmazra, index alapú bejárásra nincs lehetségünk. Az iterátoros, illetve az elemenkénti bejárást viszont komolyabb változtatások nélkül elérhetjük:

```
//bejárás - iterátorral
        Iterator<Integer> it = halmaz.iterator();
        while(it.hasNext()) {
            System.out.print(it.next() + " ");
        }
        System.out.println();

        //bejárás - elemenként
        for(int szam : halmaz) {
            System.out.print(szam + " ");
        }
        System.out.println();
```

Felmerülhet ilyenkor a kérdés, hogy milyen sorrendben fogjuk visszakapni a beírt elemeinket. Ez sok problémánál nem fontos, mert a sorrendtl teljesen független tevékenységet végez. A HashSet semmilyen rendezést nem garantál, a TreeSet viszont igen, ezért az elemeinek meg is kell valósítaniuk valamilyen rendezést (a primitív típusok csomagoló osztályai és a String ezt megteszik). Ha saját objektumokat tárolunk, definiálnunk kell kisebb-nagyobb-egyenl mveleteket comparator segítségével.

A halmazok használata tehát egyszer, és sok olyan eset elfordul, ahol könnyebben felhasználható a listáknál is. A korábban látott csorda például gond nélkül megvalósítható halmazokkal is, mivel minden állat legfeljebb egyszeresen lehet egy csordában. Az erre vonatkozó kód semmivel sem bonyolultabb, mint a listákkal való megvalósításé:

```
private Set<Allat> tagok = new HashSet<>();

    public void csordabaFogad(Allat kit) {
        tagok.add(kit);
    }
```

Leképezések (Map)

Az eddig látott tömbök és listák elemeire mind 0-tól kezdd, növekv indexekkel tudtunk hivatkozni. Viszont számos esetben hasznos lehetne, ha nem csak egész számokhoz rendelhetnénk elemeket, hanem más dolgokhoz is, például szavakhoz vagy objektumokhoz. Erre használhatóak az ún. leképezések, azaz Map-ek. Ebbl szintén két implementációt érdemes ismernünk, a Hash Map-et, amely hasítótáblán és a Tree Map-et, amely piros-fekete fán alapul.

Minden map kulcs-érték (key-value) párokból áll. Ebbl mindkett lehet bármely tetszleges referencia típusú. A kulcsokhoz értékeket rendelünk, amely azt jelenti, hogy egy bizonyos kulcshoz mindig egy érték tartozik. Egy érték viszont több kulcsnál is elfordulhat. Ebbl adódóan a kulcs a párt egyértelmen beazonosítja, míg az érték nem. Ezt felfoghatjuk úgy is, hogy számok helyett tetszleges típusú elemeket is megadhatunk indexként.

Deklarációjuk a következképpen nézhet ki:

```
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class Mapek {
    public static void main(String[] args) {
        Map<Integer,String> map1 = new HashMap<>();
        Map<Integer,String> map2 = new TreeMap<>();
    }
}
```

Észrevehetjük, hogy itt a <> jelek között már nem csak egy, hanem vesszvel elválasztva két típust kell megadnunk. Az els a kulcs, míg a második a hozzá rendelt érték típusa. A fenti példán tehát a kulcs egész szám, míg értéke szöveges.

Új elempár hozzáadása itt a 'put' metódussal történik.

```
map1.put(320,"Kék");
        map1.put(200,"Zöld");
```

A példán látható kód a map1 map-be helyez két kulcs-érték párt, a 320-as számhoz a "Kék" szót, míg 200-hoz a "Zöld" szót rendeli. Ezután már lekérhet a kulcshoz tartozó elem a listákhoz és halmazokhoz hasonlóan `get` metódussal:

```
String elem = map1.get(320);
```

Ilyenkor a "Kék" szöveget kapjuk. Fontos, hogy ez nem fordítható meg, itt nem mondhatnánk, hogy `map1.get("Kék")`. Ez azért van, mert akár a 200-hoz is rendelhettünk volna ugyanúgy "Kék"-et. Ha megpróbálnánk újabb 320-as kulcsú elemet tenni a map-be, akkor viszont felülírnánk az elzt, így ez mindig egyértelm.

Elemek törlése a listáknál már látott `remove` metódussal, a látottakkal megegyez módon alkalmazható, viszont itt is csak kulcs megadása lehetséges, az érték itt sem azonosít megfelelen. Törléskor természetesen mind a kulcs, mind az érték törlésre kerül. Kulcs-érték pár megadása is lehetséges viszont, ha csak bizonyos érték esetén szeretnénk törölni a párt.

Megfigyelhetjük, hogy a fenti mködés hasonló egy indexeléshez. Annyiban különbözik tle, hogy nem feltétlenül 0-tól indul, illetve nem csak sorban tartalmazhat indexeket. Ennél a map-ek azonban sokkal többre is képesek.

Az alábbiakban láthatunk egy példát, amely a konzolon kapott szavakat számolja meg, melyik szóból hány darab érkezett, ezeket egy map-ben tárolja.

```
//konzolon érkezett szavak számlálása mappel
        Map<String, Integer> map1 = HashMap<>();
        for(int i=0; i<args.length; i++) {
            if(map1.containsKey(args[i])) { //ha már láttuk a szót
                int darabszam = map1.get(args[i]);
                darabszam++;
                map1.put(args[i], darabszam); //felülírjuk az eddigi számát
            }
            else { //ha még nem láttuk a szót
                map1.put(args[i], 1);
            }
        }
```

Ilyen map-eket használhatunk a gyakorlatban például szövegfeldolgozás közben, ahol a szavak elfordulási számából tudunk következtetést levonni, sok algoritmusnak ez az alapja.

Bejárás

A map-eknél a halmazokhoz hasonlóan nincs egyértelm rendszer a kiírás sorrendjére. Indexenként ezek bejárására sincs lehetség (esetleg ha a listákhoz hasonló map-et készítünk, vagy fenntartunk egy index-halmazt, amelyet bejárva a kulcsokat kapjuk sorban). Itt is használható a kijárásra iterátor és az elemenkénti kiírás is mködik (kicsivel bonyolultabb fromákban):

```
//map bejárása - iterátorral
        Iterator elemek = map1.entrySet().iterator();
        while (elemek.hasNext()) {
          Entry elem = (Entry) elemek.next();
          System.out.println(elem.getKey() + "\t" + elem.getValue());
        }

        //map bejárása - elemenként
        for(Entry<String, Integer> elem : map1.entrySet()) {
            System.out.println(elem.getKey() + "\t" + elem.getValue());
        }
```

Ilyenkor nem kulcsokkal vagy elemekkel tudjuk bejárni a map-et, hanem a konkrét párokkal. Egy ilyen párt hívunk **Entry**nek. Ennek szintén két típust kell megadnunk, ez pontosan egy kulcs-érték párját jelöli a map-nek. A map tehát felfoghto úgy, mint ilyen entry-k halmaza. Ebbe pedig a map entry-jeit helyezhetjük úgy, hogy egyszeren ténylegesen entry-k halmazaként kezeljük az `entrySet` metódusával, amely egy halmazt ad vissza a map tartalmával.

Egy entry tehát egy kulcs-érték pár, amelynek kulcsát a `getKey()`, míg értékét a `getValue()`metódussal kaphatjuk meg.

A map-ekre is létezik a halmazoknál látott contains metódus, ám itt kett is van belle, a `containsKey` és a `containsValue`, amelyekkel a kulcsokat és az értékeket ellenrizhetjük. Ezek értelemszeren a nekik megfelel típust várják paraméterül.

A map-ek tehát szintén egyszer mködtetést biztosítanak, illetve szintén dinamikus méretet támogatnak. Alkalmazhatóak például objektumok számolására, két objektum egymáshoz rendelésére, vagy akár bármilyen érték ideiglenes objektumonkénti tárolására.

A már látott állatos példára visszatérve, ha létezik egy halmazban tárolt csorda a látott formában, akkor ahhoz megadható map, amely például számon tartja, hogy melyik fajból hány darabot tartalmaz. Egy ezt visszaadó metódus:

```
public Map<String, Integer> fajokatSzamol() {
        Map<String,Integer> fajSzamok = new HashMap<>();

        for(Allat allat : tagok) {  //bejárjuk a csordát
            if(!fajSzamok.containsKey(allat.getClass().getName())) { //ha még nincs ilyen faj a map-ben
                fajSzamok.put(allat.getClass().getName(), 1); //beleteszünk egyet
            }
            else {  //ha már láttunk ilyen fajt
                int eddigiSzam = fajSzamok.get(allat.getClass().getName()); //lekérjük az eddigi
hozzárendelt számot
                eddigiSzam++; //növeljük 1-el
                fajSzamok.put(allat.getClass().getName(), eddigiSzam); //újra beletesszük az új számmal,
felülírva a régit
            }
        }

        return fajSzamok;
    }
```

Lambda kifejezések

Egy grafikus felülettel ellátott alkalmazás esetében, amikor egy gombra eseménykezelt írunk (erre példa, a `08-Programozas-I.pdf` fájlban, egy másik kód action_listener[itt]), akkor egy névtelen interfész-implementációt készítünk, ami nagyban nehezíti a kód olvashatóságát, átláthatóságát. Mindemellet rengeteg felesleges kódrészlet is bekerül a kódunkba, amelyet Java 1.8-tól elkerülhetünk könnyedén, lambda kifejezések használatával.

A lambda függvények gyakorlatilag olyan névtelen metódusok, amelyet ott írunk meg, ahol használni szeretnénk. Gyakorlatilag akkor lehet haszna, ha például egy olyan interfészt szeretnénk helyben implementálni, aminek csak egy metódusa van, vagy például kollekciók hatékony, gyors, átlátható bejárásakor. Szóval egy interfész-implementációt tömörebben, gyorsabban, átláthatóbban írhatunk meg, mint eddig.

Mivel jelen gyakorlaton nem foglalkozunk Java GUI-val, így egy másik példán keresztül ismerjük meg ket, mégpedig a kollekciók segítségével. Elször egy **listát** (de halmazon is ugyanígy mködne) járunk be, majd pedig egy kulcs-érték párokból álló Map objektumot.

Egy lambda kifejezés szintaxisa: `(paraméter1, paraméter2) -> utasítás, vagy utasítás blokk`. A paraméterek típusát nem kell kiírnunk (de kiírhatjuk ket, ha szeretnénk). Egy paraméter esetén elhagyhatjuk a paraméterek körüli zárójelet.

```
public class Main {

    public static void main(String[] args) {
        List<String> szinek = new ArrayList<>();
        szinek.add("Kék");
        szinek.add("Zöld");
        szinek.add("Piros");
        szinek.add("Fekete");
        szinek.add("Sárga");
        szinek.add("Narancs");

        szinek.forEach(szin -> System.out.println(szin));
    }
}
```

Láthatjuk, hogy mennyivel egyszerbb használni, mint például egy hagyományos for ciklust. Amennyiben több utasítást használunk, akkor a megszokott módon kapcsos-zárójelek közé kell tenni az utasításokat a nyíl(->) után.

```
public class Main {

    public static void main(String[] args) {
        List<String> szinek = new ArrayList<>();
        szinek.add("Kék");
        szinek.add("Zöld");
        szinek.add("Piros");
        szinek.add("Fekete");
        szinek.add("Sárga");
        szinek.add("Narancs");

        szinek.forEach(szin -> {
            if (szin.charAt(0) > 'O') {
                System.out.println(szin);
            }
        });
    }
}
```

A fenti példában végigmegyünk a listán, és megnézzük, melyik szín kezddik egy 'O' után következ betvel, és azokat írjuk ki az alapértelmezett kimenetre. Jelen helyzetünkbe talán ez nem tnik nagy dolognak, mert sima iterátorral, vagy for ciklussal is bejárhattuk volna a listát, körülbelül ugyanennyi lenne kódban.

Azonban nézzük meg ezt a bejárást egy Map esetében, ahol már érezheten egyszersödik a helyzetünk. (Csak hogy az eladáson látott GUI elemek eseménykezeljérl ne is beszéljünk.)

```
public class Main {

    public static void main(String[] args) {
        Map<String, Integer> szinek = new HashMap<>();

        // Megkérdeztünk 1000 embert, kinek mi a kedvenc színe, ezt tároljuk le
        // ebben a mapben.
        szinek.put("Kék", 320);
        szinek.put("Zöld", 200);
        szinek.put("Sárga", 80);
        szinek.put("Barna", 95);
        szinek.put("Citrom", 105);
        szinek.put("Piros", 75);
        szinek.put("Lila", 125);

        szinek.forEach((szin, ertek) -> System.out.println(szin + " szín " + ertek + " ember kedvence."));
    }
}
```

Ahogy már láttuk, ha több utasítást szeretnénk végrehajtani, akkor kapcsos zárójelek közé kell tennünk az utasításokat.

```java
public class Main {

    public static void main(String[] args) {
        Map<String, Integer> szinek = new HashMap<>();

        // Megkérdeztünk 1000 embert, kinek mi a kedvenc színe, ezt tároljuk le
        // ebben a map-ben.
        szinek.put("Kék", 320);
        szinek.put("Zöld", 200);
        szinek.put("Sárga", 80);
        szinek.put("Barna", 95);
        szinek.put("Citrom", 105);
        szinek.put("Piros", 75);
        szinek.put("Lila", 125);

        szinek.forEach((szin, ertek) -> {
            if (ertek > 100) {
                System.out.println(szin + " szín " + ertek + " ember kedvence.");
            } else {
                System.out.println(szin + " szín nem túl sok ember kedvence.");
            }
        });
    }
}
```

Látszik, hogy a fent ismertetettekkel ellentétben lambda kifejezéssel nagyon egyszeren, átláthatóan járhatunk be egy map-et is. Remélhetleg mindenki kedvet kapott a lambdák további megismeréséhez, nekik ajánljuk a következ linkeket:

Oracle Lambda Expressions

Java 8 - Lambda Expressions

Lambda Expressions in Java 8

Feladatok

1. A korábbi saját láncolt lista implementációt módosítsuk úgy, hogy bármilyen típusú elemet tudjon tárolni, ne csak `Allat` típusút.
2. A korábbi veremes feladatot valósítsd meg úgy, hogy bármilyen típusú elemet tudjon tárolni, amit generikus típusparaméterként kelljen neki megadni.