2.4 String, kivételkezelés

Stringek feldarabolása

Stringek split() metódusa

Szövegek feldarabolására két módot fogunk nézni, az els: a `String` objektumok rendelkeznek egy `split()` nev metódussal, ami egy reguláris kifejezést vár paraméterül, amely mentén tagolja a szöveget, és egy String tömbbel tér vissza, amely tartalmazza a szövegdarabokat. Bvebben a String#split() metódusáról.

```
String mondat = "Ez a mondat hat szóból áll.";
String[] szavak = mondat.split(" ");

for(int i=0; i < szavak.length; i++) {
    System.out.println("Az " + i + ". szo: " + szavak[i]);
}
```

Ennek kimenete:

```
Az 0. szo: Ez
Az 1. szo: a
Az 2. szo: mondat
Az 3. szo: hat
Az 4. szo: szóból
Az 5. szo: áll.
```

StringTokenizer

Egy másik lehetséges lehetség a `StringTokenizer` nev osztály használata, amely a `java.util`csomagban található, így elbb be kell importálnunk ezt az osztályt. Ezek után használhatjuk a kódunkban az osztályt, amelyet példányosítanunk kell, a konstruktorban egy szöveget vár, és opcionálisan egy szöveget, amelynek minden karaktere szóhatárt jelöl. Ezek után használhatjuk a `StringTokenizer` objektumunkat, ennek van egy `hasMoreTokens()` nev metódusa, ami azt mondja meg, hogy van-e még elem a darabolt szövegben, igazzal tér vissza ha van, hamissal különben. Egy szó darabkát lekérni a `nextToken()` metódussal lehet, amely az aktuális szövegtördelékkel tér vissza.

A StringTokenizer alapértelmezett szeparátorai a következ karakterek: `" \t\n\r\f"`

- szóköz
- tab karakter
- újsor karakter
- kocsi vissza (carriage return) karakter
- line feed

Ha ezektl eltér karakterekkel szeretnénk darabolni a szövegünket, akkor a konstruktorban második paraméterként megadhatjuk azokat a karaktereket, amelyek mentén szeretnénk darabolni (a karakterlánc bármely karakterére darabol).

Bvebben a StringTokenizer osztályról

```java
import java.util.StringTokenizer; //Fontos, hogy beimportáljuk használat eltt
public class Main {
  public static void main(String[] args) {
        String str = "abcd, szoveg,valami kiscica;kiskutya;medve hóember péklapát";

        // StringTokenizer létrehozása alapértelmezett szeparátorral
        StringTokenizer st = new StringTokenizer(str);

        System.out.println("StringTokenizer els futás az str-en: (alapértelmezett szeparátorral)");
        while (st.hasMoreTokens()) {
            String tmp = st.nextToken();
            System.out.println(tmp);
        }
        System.out.println();

        System.out.println("StringTokenizer második futás az str-en: (; , . szeparátorokkal)");
        st = new StringTokenizer(str, ";.,");

        while (st.hasMoreTokens()) {
            String tmp = st.nextToken();
            System.out.println(tmp);
        }
  }
}
```

Ennek kimenete:

```
StringTokenizer els futás az str-en: (alapértelmezett szeparátorral)
abcd,
szoveg,valami
kiscica;kiskutya;medve
hóember
péklapát

StringTokenizer második futás az str-en: (; , . szeparátorokkal)
abcd
 szoveg
valami      kiscica
kiskutya
medve hóember péklapát
```

Kivételek - hibakezelés

A kivételek valamiféle kivételes eseményt jelölnek, a Java nyelvben ezek segítségével van megoldva a hibakezelés.

> *Emlékezz vissza, mi történik, ha egy tömböt túl- vagy alulindexelsz! Ha nem emlékszel, próbáld ki!*

Ha valamit rosszul írunk a kódban, például lemarad egy pontosvessz, vagy kapcsol zárójel, akkor fordításidej hibát kapunk, ha megpróbáljuk fordítani a kódot. Azonban nem minden hiba derül ki fordításidben, ilyenek például az io mveletek, felhasználói interakció, hálózati kommunikáció. Ezeket a hibákat Javaban kivételekkel oldjuk meg, egészen pontosan kivétel objektumokkal, amelyeket hiba bekövetkezésekor "eldobunk", a hívás helyén pedig elkapjuk, és lekezeljük, vagy továbbdobjuk. Ennek segítségével gyakorlatilag "megpróbáljuk menteni a menthett", azaz, ha kivételes esemény történik, de azért a program mködése szempontjából nem végzetes, akkor valahogy lekezeljük a hibát, hogy a program zavartalanul mködhessen tovább (például egy számológép alkalmazásban megpróbálunk nullával osztani, kivételes esemény, de egy hibaüzenetet követen használhatjuk tovább a kalkulátort).

Amikor a program futása során egy metódusban hiba keletkezik (tehát egy kivételes esemény, ami a program normál mködése során nem fordul el), a metóduson belül egy kivétel objektum jön létre (vagy kapjuk egy, általunk meghívott metódustól) a memóriában, amit a futtatókörnyezetnek átadunk, a metódus végrehajtása megáll, és a hívó fél számára továbbküldjük a létrejött kivételobjektumot, aki lekezelheti, vagy továbbdobhatja azt.

Ha a hívási helyen le van kezelve a hiba (alapesetben ezt biztosítani kell), akkor lekezeljük. Ha több hibakezel kódunk is van, akkor az els megfelelt fogja használni, majd a program futása folytatódik normál módon.

Tehát eddig összegezve tudjuk, hogy a kivételek speciális objektumok, amelyek valamilyen hiba esetében jönnek létre, és le tudjuk ket kezelni. De hogy kell ezt elképzelni Javaban?

throw

Ha van egy metódusunk, ahol bizonyos feltételek fennállnak, és szeretnénk, hogy hiba dobódna (például ha egy metódusban egy 0 számot kapunk, és azzal osztanánk), akkor ott létre kell hoznunk egy kivétel objektumot, melyet eltárolhatunk referenciában, ha szeretnénk (általában

nem szeretnénk); ezt követen pedig ezt a kivételt el kell dobnunk, amire a `throw` kulcsszó fog szolgálni.

```
public int osztas(int a, int b) {
    if(b == 0) {
        // Ebben az esetben szeretnénk hibát dobni
        // throw-olni szeretnénk, de mit?
    }
    return a/b;
}
```

Egy másik példa a korábbról már megismert állatos példa, ahol mindenki házi feladata volt, hogy megvalósítsa, hogy csak szárazföldi, vagy csak vízi állatok lehessenek egy csordában, hiszen egy vegyes csorda nem igazán mködképes. A módosított Csorda osztály:

```
import allatok.*;

public class Csorda {
    private Allat[] tagok;
    private int maximum = 0;
    private int jelenlegi;

    private boolean szarazfoldiAllatok;

    public Csorda(int num) {
        this.tagok = new Allat[num];
        this.maximum = num;
        this.jelenlegi = 0;
    }

    public boolean csordabaFogad(Allat kit) {
        if (jelenlegi == 0) {
            if (kit instanceof SzarazfoldiAllat) {
                szarazfoldiAllatok = true;
            }
        }
        if (jelenlegi < maximum) {
            if ( (szarazfoldiAllatok && kit instanceof ViziAllat) ||
                 (!szarazfoldiAllatok && kit instanceof SzarazfoldiAllat) ) {
                    // Itt kellene hibát dobni
                }

                tagok[jelenlegi] = kit;
            jelenlegi++;
            return true;
        }
        return false;
    }

    public String toString() {
        String returnValue = "Allatok: ";
        for (int i = 0; i < jelenlegi; ++i) {
            returnValue += this.tagok[i].getNev();
            returnValue += ", ";
        }
        return returnValue;
    }
}
```

Exception

Már csak egy kivétel objektumra van szükségünk. Ez lehet saját kivétel osztály példánya, vagy egy a beépített Java kivételek közül. A kivételek sosztálya Javaban az `Exception`. Ez egy általános kivétel osztály, minden kivétel osztály ebbl származik. Néhány további kivételosztály:

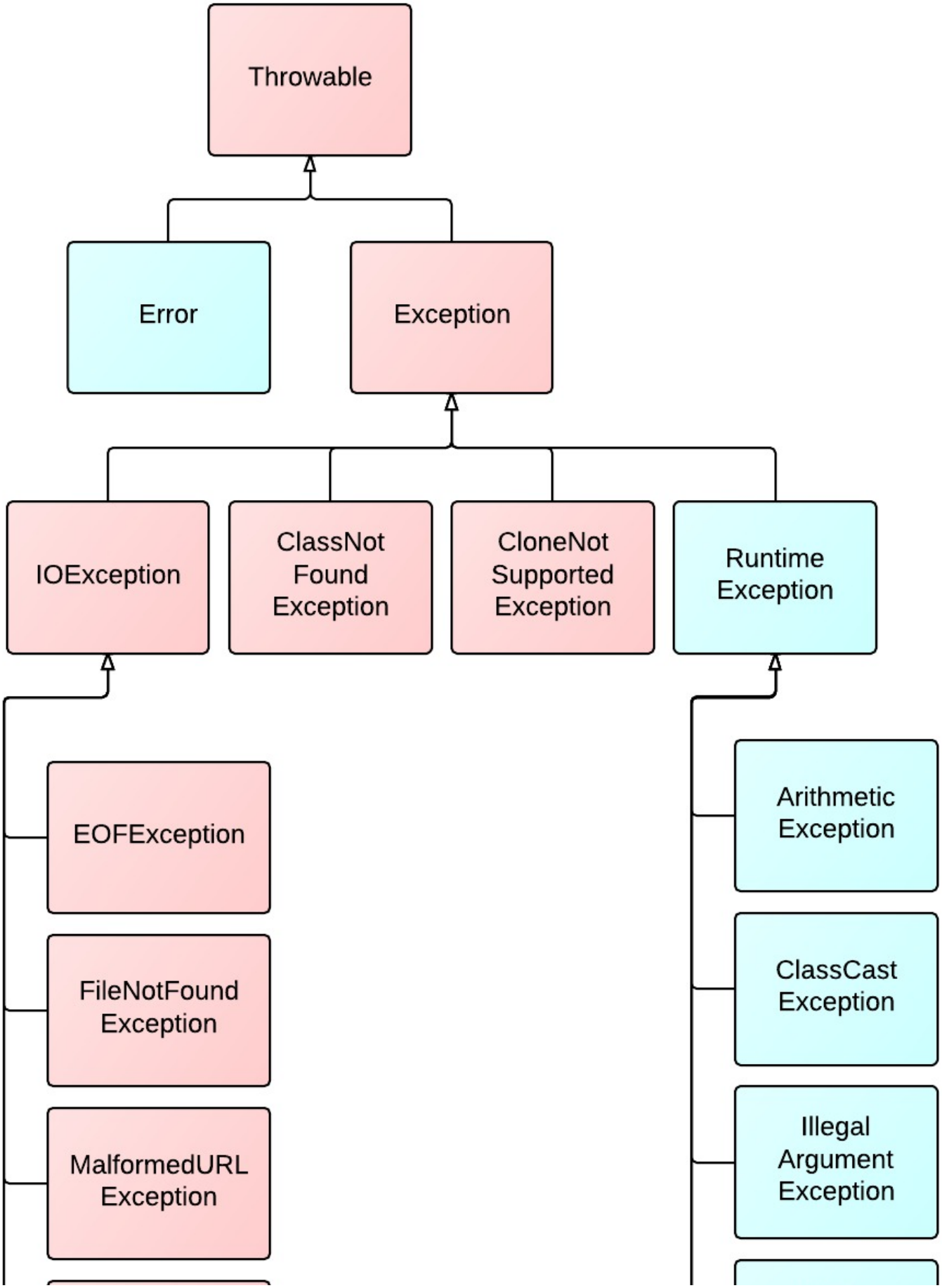
- ArithmeticException (pl.: nullával való osztás)
- ArrayIndexOutOfBoundsException (tömbindexelés)
- IllegalArgumentException
- IOException (IO mveletekkel kapcsolatos)
- SQLException
- NullPointerException
- ClassNotFoundException

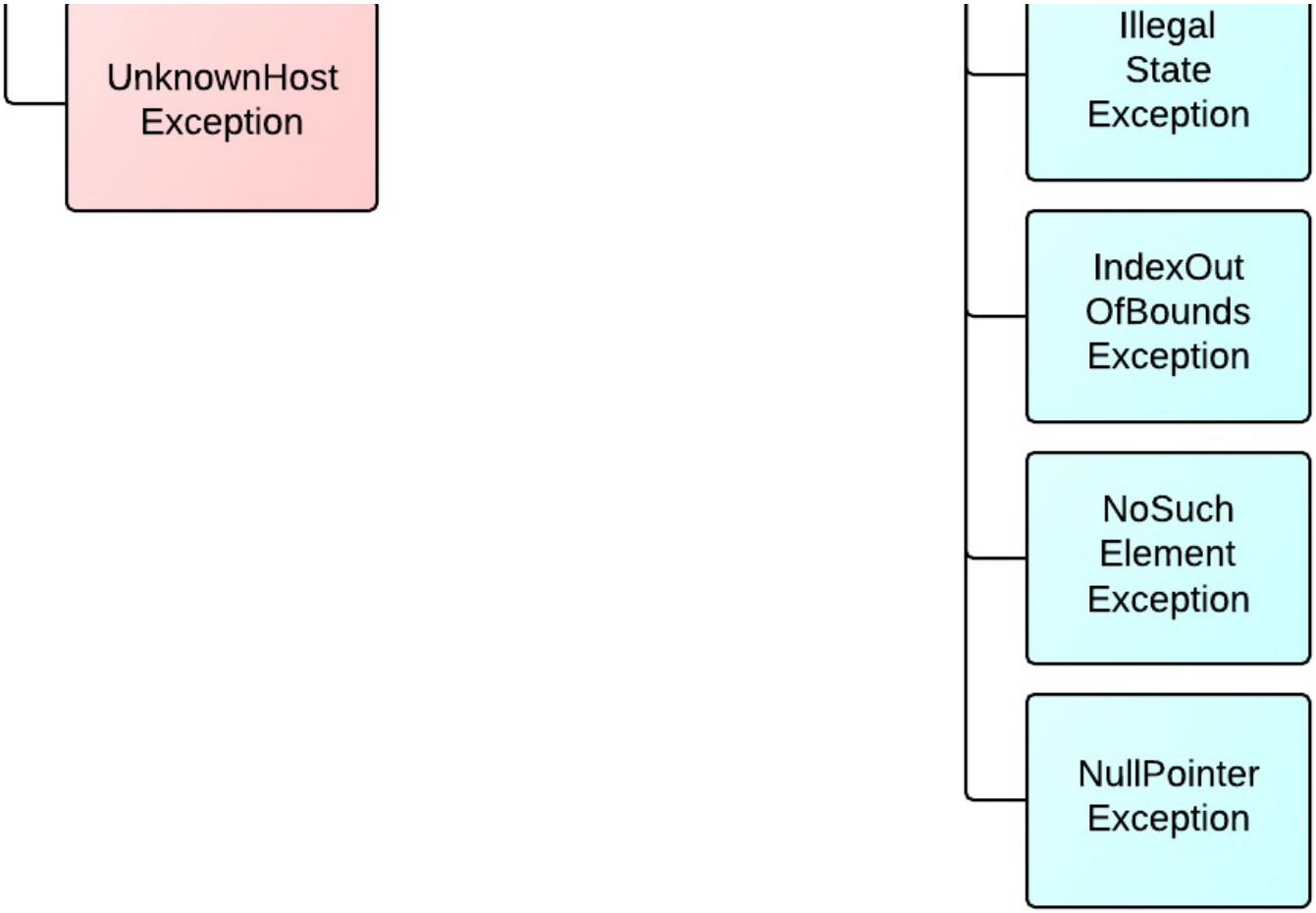
RUNTIMEEXCEPTION

Futásidej kivételek se. Ezek elkapása nem kötelez, mivel a program normál futása során nem dobódnak ilyen kivételek, ilyen például a `NullPointerException`. Ezeket is lekezelhetjük, ha szeretnénk, de nem biztos, hogy jó ötlet, mert ezen kivételek érkezése esetén általában jobban szeretnénk, ha a program futása megszakadjon és leálljon teljesen.

THROWABLE

A kivételek (`Exception` osztály és gyermekei) és `Error` osztályok se. Az `Error` típus komolyabb problémára utal, ezekkel általában nem foglalkozunk (ilyen például a virtuális gép meghibásodása, vagy ha kifogyunk a memóriából). Az osztályhierarchia látható az alábbi ábrán:


Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception



Ha szeretnénk egy saját kivétel osztályt csinálni, akkor annyi a dolgunk, hogy örököltetjük az `Exception` nev sosztályból. Erre egyszer példa az `InkompatibilisAllatok` osztály, amely az inkompatibilis állatok esetén fogunk dobni.

```
package allatok.kivetel;

public class InkompatibilisAllatok extends Exception {

    public InkompatibilisAllatok() {
        super();
    }

    public InkompatibilisAllatok(String message) {
        super(message);
    }

}
```

Ahol ezt el szeretnénk dobni, ott csak példányosítanunk kell, majd átadni a létrehozott objektumot a `throw`-nak. A kiegészített `csordabaFogad` metódus:

```
public boolean csordabaFogad(Allat kit) {
        if (jelenlegi == 0) {
            if (kit instanceof SzarazfoldiAllat) {
                szarazfoldiAllatok = true;
            }
        }
        if (jelenlegi < maximum) {
            if ( (szarazfoldiAllatok && kit instanceof ViziAllat) ||
                 (!szarazfoldiAllatok && kit instanceof SzarazfoldiAllat)) {
                    throw new InkompatibilisAllatok("Ez így nem fog menni!");
                }

                tagok[jelenlegi] = kit;
            jelenlegi++;
            return true;
        }
        return false;
    }
```

throws

Ez így még nem fog mködni. Az olyan metódusoknál, ahol kivétel objektumok jöhetnek létre és dobódhatnak, ott ezt jelezni kell a metódus fejlécében is, hogy bizony ebben a metódusban történhetnek kivételes dolgok is, amelyek lekezelése a hívó fél feladata lesz.

Ezt a jelzést úgy tudjuk megtenni, hogy a metódus paraméterezésében, a paramétereket bezáró zárójel után, de a metódus blokkjának nyitása eltt a `throws` kulcsszó után vesszvel felsoroljuk, hogy milyen kivételobjektumok dobódhatnak a metódusban. Ha nem szeretnénk sokat foglalkozni vele, egyszeren csak az sosztályt írjuk ki, vagyis annyit, hogy `Exception`. Ebben az esetben nem kell mást felsorolni, mert az összes kivételosztály az `Exception`-bl származik, így az bármely gyerekosztályt is jelenthet.

```
public boolean csordabaFogad(Allat kit) throws InkompatibilisAllatok {
        if (jelenlegi == 0) {
            if (kit instanceof SzarazfoldiAllat) {
                szarazfoldiAllatok = true;
            }
        }
        if (jelenlegi < maximum) {
            if ( (szarazfoldiAllatok && kit instanceof ViziAllat) ||
                 (!szarazfoldiAllatok && kit instanceof SzarazfoldiAllat)) {
                    throw new InkompatibilisAllatok("Ez így nem fog menni!");
                }

            tagok[jelenlegi] = kit;
            jelenlegi++;
            return true;
        }
        return false;
    }
```

Végre van kivétel objektumunk, amit el is tudunk dobni, és a kód is lefordulna. Már csak egy dolgunk maradt: a hívás helyén lekezelni az esetleges kivételt. Ehhez szükségünk lesz néhány új kulcsszóra:

try, catch, finally

`try` - ezzel a kulcsszóval kezddik a védett régió, az a kódrészlet, amely esetlegesen kivételt dobhat, a try blokkon belül vannak azok a metódusok, amelyek kivételt dobhatnak. `catch` - a try blokk után következik; a kivételkezel blokk(ok). `finally` - az a blokk, amely mindenképpen lefut, akár történt kivétel, akár nem. Általában itt zárjuk le a fájlokat, hálózati kapcsolatot.

Kezdjünk is bele: a `Csorda` osztály `csordabaFogad` metódusa már nem hívható csak úgy, hiszen kivételt dobhat, ha a körülmények éppen úgy állnak. Ezért ezt egy `try` blokkba kell tenni:

```
import allatok.*;

public class EgyszeruMain {

    public static void main(String[] args) {
        Allat elso = new Balna("Charlie");
        Allat masodik = new Csirke("Kotkoda");

        Csorda csorda = new Csorda(5);
        try {
            csorda.csordabaFogad(elso);
            csorda.csordabaFogad(masodik);
        } catch(InkompatibilisAllatok inkompatibilisAllatok){
            System.err.println("Inkompatibilis állatok! Szárazföldi és vízi állatok nem keveredhetnek!");
        }
    }
}
```

A lehetséges kivételt elkaphatjuk valamelyik `catch` ággal. Ebbl lehet egy vagy több is. A catch kulcsszó után megadhatjuk, hogy milyen kivételtípust szeretnénk elkapni, és azt, hogy ezt milyen néven fogjuk használni a kivétel lekezeléséért felel blokkban. A kivétel elkapásánál elkaphatunk speciális típust vagy egy általánosabb típust is. Ha nem szeretnénk sokat szöszölni vele, akkor elkaphatjuk a kivételek stípusát, az `Exception` osztály egy példányát, hiszen az elkapja a gyerek típusokat is. Több kivételkezel blokkot is írhatunk egymás után, ha az elkapott kivételeket típusoktól függen másképp szeretnénk lekezelni.

```
try {
    csorda.csordabaFogad(elso);
    csorda.csordabaFogad(masodik);
} catch(InkompatibilisAllatok inkompatibilisAllatok){
    System.err.println("Inkompatibilis állatok! Szárazföldi és vízi állatok nem keveredhetnek!");
} catch(Exception exc){
    System.err.println("Valami hiba van! :(");
}
```

Azonban, ha több, különböz kivételt is szeretnénk elkapni, de nem az si `Exception` nev osztályt használni, akkor Java 7-tl lehetségünk van az alábbi szintaxis használatára is:

```
try {
    csorda.csordabaFogad(elso);
    csorda.csordabaFogad(masodik);
} catch(InkompatibilisAllatok|ArrayIndexOutOfBoundsException exc){
    System.err.println("Baj van :(");
}
```

Ez körülbelül azt jelenti, hogy kapjuk el vagy az `InkompatibilisAllatok` egy példányát, vagy pedig az `ArrayIndexOutOfBoundsExcepti` on egy példányát és ezeket ugyanúgy kezeljük.

Ha van olyan kódrészlet, amelyrl biztosítani szeretnénk, hogy mindenképpen lefusson, akkor azt `finally` blokkba kell betennünk.

```
try {
    csorda.csordabaFogad(elso);
    csorda.csordabaFogad(masodik);
} catch(InkompatibilisAllatok|ArrayIndexOutOfBoundsException exc){
    System.err.println("Baj van :(");
} finally {
    System.out.println("Ez mindenképp lefut!");
}
```

Bels osztály

Mennyire jó lenne, ha egy csordának nem kellene megmondanunk az elemszámát, hanem attól függen, hogy hány elemet rakok bele, változna a mérete. A jó hír, hogy ezt megtehetjük, emlékezzünk vissza a *Programozás alapjai* kurzuson tanultakra, volt "valami" láncolt lista megvalósítás. Igaz, ott C-ben dolgoztunk, de azért az ott tanultakat jó eséllyel itt is el tudjuk sütni, de ehhez szükségünk lesz egy új osztályra, például `LáncEle m` néven.

Azonban érdemes kicsit gondolkozni rajta, hiszen ezt az osztályt nem szeretnénk a nyilvánosság elé tárni, st igaziból gyakorlati jelentsége nincs, hogy a `Csorda` milyen módon tárolja az csordában lév állatokat. Milyen jó lenne, ha megoldható lenne az, hogy van egy osztályunk, és azt csak a `Csorda` osztály számára tesszük láthatóvá. Osztályok láthatósága `public` vagy package-private (kulcsszó nélkül) lehet, úgyhogy erre nincs lehetségünk alapveten. Ami a jó hír, hogy nem ezen a módon, de mégis megtehetjük ezt, azaz készíthetünk egy osztályt, amelyet csak egy másik osztály lát. Ehhez egy **bels osztályt** kell készítenünk, a fenti `LancElem` néven.

Lehetségünk van osztályon belül, vagy akár metóduson belül deklarálni osztályokat. Ezek a "hagyományos "osztályokkal ellentétben lehetnek `pr`

`ivate`, `protected` láthatóságúak is (a "hagyományos" osztályok láthatósága csak `public` vagy package-private lehet). St, a bels osztályokat elláthatunk `static` módosítószóval is.

A bels osztályok hozzáférnek a küls osztály adattagjaihoz, metódusaihoz. Ez alól kivétel, ha a bels osztály statikus. Ez a kulcsszó jelen helyzetben gyakorlatilag annyit jelent, hogy a bels és küls osztálynak nincs köze egymáshoz.

A bels osztályok célja, hogy egy osztályon belül elrejtsünk egy máshol nem használt adatszerkezetet, algoritmust, ugyanakkor ezeket akár kívülrl is elérhetjük (ha úgy állítjuk be a láthatóságukat). Az osztályon belül egyszeren, a tanult módon példányosítjuk ket, azonban akár kívülrl is megtehetjük ezt.

Példányosítás, ha a bels osztály **nem** statikus láthatóságú:

```
class Kulso {
   private int num = 175;

   public class Belso {
      public int getNum() {
         System.out.println("Visszaterunk a szammal.");
         return num;
      }
   }
}

public class MainOsztaly {

   public static void main(String args[]) {
      Kulso kulsoPeldany = new Kulso();
      // Nem statikus belso osztaly
      Kulso.Belso belsoPeldany = kulsoPeldany.new Belso();
      System.out.println(belsoPeldany.getNum());
   }
}
```

Példányosítás, ha a bels osztály statikus láthatóságú:

```
class Kulso {
   static class Belso {
      public void print() {
         System.out.println("Belso osztaly kiiratasa");
      }
   }
}

public class MainOsztaly {

   public static void main(String args[]) {
      Kulso.Belso belsoPeldany = new Kulso.Belso();
      belsoPeldany.print();
   }
}
```

Ezek ismeretében valósítsuk meg a bels `LáncElem` osztályt, a `Csorda` osztályon belül. Mivel ezt máshol nem szeretnénk használni, nyugodtan tegyük privát láthatóságúvá. Ezt követen írjuk át a `Csorda` osztály, hogy az eddig használt statikus tömb helyett a saját láncolt lista megvalósításunkat használjuk!

```
public class Csorda {
    private class LancElem {
        public Allat tag;
        public LancElem kov = null;

        public LancElem(Allat tag, LancElem kov) {
            this.tag = tag;
            this.kov = kov;
        }
    }

    // Tömb helyett csak a lánc fejét tároljuk
    private LancElem fej;
    private int jelenlegi;

    private boolean szarazfoldiAllatok;

    public Csorda() {
        this.jelenlegi = 0;
        this.fej = null;
    }

    public boolean csordabaFogad(Allat kit) throws InkompatibilisAllatok {
        if (jelenlegi == 0) {
            if (kit instanceof SzarazfoldiAllat) {
                szarazfoldiAllatok = true;
            }
        }
        if ( (szarazfoldiAllatok && kit instanceof ViziAllat) ||
             (!szarazfoldiAllatok && kit instanceof SzarazfoldiAllat)) {
            throw new InkompatibilisAllatok();
        } else {
            LancElem tmp = new LancElem(kit, this.fej);
            this.fej = tmp;
            jelenlegi++;
            return true;
        }
    }

    public String toString() {
        String returnValue = "Allatok: ";
        LancElem jelenlegi = fej;
        while (jelenlegi != null) {
            returnValue += jelenlegi.tag.getNev();
            returnValue += ", ";
            jelenlegi = jelenlegi.kov;
        }
        return returnValue;
    }
}
```

Feladatok

- Hozz létre egy fix méret vermet egész számok tárolására (tömb segítségével) és valósítsd meg a push/pop mveleteket.
- Írj egy futtatható osztályt, mely a Main metódusban "push" vagy "pop" utasításokat vár a konzolról. Ha pop utasítást kap, hajtsa végre azt, és írja ki a konzolra a kivett elemet. Push utasítás esetén egy egész számnak kell következnie, ezt tegye be a verembe.
- Írj meg egy kivételosztályt, amit a fenti függvények (push/pop) akkor dobnak, ha a verem megtelt vagy üres.