

A második osztály neve *Cat*, ez az *Animal*-nak egy leszármazott osztálya:

```
public class Cat extends Animal {
    public static void hide() {
        System.out.println("The hide method in Cat.");
    }
    public void override() {
        System.out.println("The override method in Cat.");
    }
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = (Animal)myCat;
        myAnimal.hide();
        myAnimal.override();
    }
}
```

A *Cat* osztály felülírja az *override* metódust az *Animal*-ban, és elrejti a *hide* osztálymetódust az *Animal*-ban. Ebben az osztályban a *main* metódus létrehoz egy *Cat* példányt, be-teszi az *Animal* típusú hivatkozás alá is, majd előhívja mind az elrejtett, mind a felülírt metódust. A program eredménye a következő:

```
The hide method in Animal.
The override method in Cat.
```

A szülőosztályból hívjuk meg a rejtett metódust, a leszármazott osztályból pedig a felülírtat. Osztálymetódushoz a futtatórendszer azt a metódust hívja meg, mely a hivatkozás szerkesztési idejű típusában van definiálva, amellyel a metódust elnevezték. A példánkban az *myAnimal* szerkesztési idejű típusa az *Animal*. Elképpén a futtatórendszer az *Animal*-ban definiált rejtett metódust hívja meg. A példánymetódusnál a futtatórendszer a hivatkozás futásidejű típusában meghatározott metódust hívja meg. A példában az *myAnimal* futásidejű típusa a *Cat*. Elképpén a futtatórendszer a *Cat*-ban definiált felülírt metódust hívja meg.

Egy példánymetódus nem tud felülírni egy osztálymetódust, és egy osztálymetódus nem tud elrejteni egy példánymetódust. Mindkét esetben fordítási hibát kapunk.

12.2. Tagváltozók elrejtése

Egy osztály változója, ha ugyanazt a nevet viseli, mint a felsőbb osztály egy változója, akkor elrejt a felsőbb osztály változóját, még akkor is, ha különböző a típusuk. Az leszármazott osztályokon belül a felsőbb osztálybeli változóra nem utalhatunk egyszerűen a nevével. Ehelyett a tagváltozót el tudjuk érni az *osztályon keresztül*, amiről majd a következő fejezet fog szólni. Általánosságban véve nem célszerű a tagváltozók elrejtése.

12.3. A *super* használata

Ha egy metódus felülírja az *osztálya* metódusainak egyikét, akkor a *super* használata-val segítségül hívható a felülírt metódus. A *super* arra is használható, hogy egy rejtett tag variánsra utaljunk. Ez a szülőosztály:

```
public class Superclass {
    public boolean avariable;
```

```

public void amethod() {
    avariable = true;
}
}

```

Most következzen a *Subclass* nevű leszármazott osztály, mely felülírja *amethod*-ot és *aVariable*-t:

```

public class Subclass Superclass {
    public boolean avariable; //hides avariable in Superclass
    public void amethod() { //overrides amethod in Superclass
        avariable = false;
        super.amethod();
        System.out.println(avariable);
        System.out.println(super.avariable);
    }
}

```

A leszármazott osztályon belül az *aVariable* név a *SubClass*-ban deklaráltra utal, amely a szülőosztályban deklarálta elrejt. Hasonlóképpen, az *aMethod* név a *SubClass*-ban deklaráltra utal, amely felsőbb osztályban deklarálta felülírja. Tehát ha egy a szülőosztályból örökölt *aVariable*-ra és *aMethod*-ra szeretnénk utalni, a leszármazott osztálynak egy minősített nevet kell használnia, használva a *super*-t, mint azt láttuk. A *Subclass* *aMethod* metódusa a következőket írja ki:

```

false
true

```

Használhatjuk a *super*-t a konstruktoron belül is az őszülő konstruktor meghívására. A következő kód példa bemutatja a *Thread* osztály egy részét – az osztály lenyegében többszörös programfutást tesz lehetővé –, amely végrehajt egy animációt. Az *AnimationThread* osztály konstruktorba beállít néhány kezdeti értékeket, ilyenek például a képernyő és a képek száma, majd a végén letölti a képeket:

```

class AnimationThread extends Thread {
    int framesPerSecond;
    int numImages;
    Image[] images;

    AnimationThread(int fps, int num) {
        super("AnimationThread");
        this.framesPerSecond = fps;
        this.numImages = num;
        this.images = new Image[numImages];
    }

    for (int i = 0; i <= numImages; i++) {
        // Load all the images.
        ...
    }
    ...
}

```

A felkövetéssel szedett sor a közvetlen szülőosztály konstruktorának explicit meghívása, melyet a *Thread* nyújt. Ez a *Thread* konstruktor átvessz egy *String*-et, és így nevezi a *Thread*-et. Ha a leszármazott osztály konstruktorában van explicit *super* konstruktorhívás, akkor annak az elsőnek kell lennie. Ha egy konstruktor nem hív meg explicit módon egy szülőosztálybeli konstruktor, akkor a Java futtatórendszer automatikusan (implicit)

a szülőosztály paraméter nélkül ki hívja meg, még mielőtt a konstruktoron belül bármi utasítást végrehajtana.

Megjegyzés: ha a szülő osztályban nem áll rendelkezésre paraméter nélküli konstruktor, akkor fordítási hibát kapunk. Ilyen esetben kötelesek vagyunk explicit paraméteres konstruktorhívást alkalmazni.

12.4. Az *Object* osztály módszerei

Az *Object* osztály minden osztály közös őse, az osztályhierarchia tetején áll. Minden osztály közvetlen vagy közvetett módon utódja az *Object* osztálynak, így minden osztály rendelkezik az *Object* osztály módszereivel. Ez az osztály definiálja azt az alapvető működést, mely minden objektummal rendelkezésre áll.

Az *Object* osztály által nyújtott legfontosabb módszerek a következők:

- *clone*
- *equals* és *hashCode*
- *finalize*
- *toString*
- *getClass*

Ezeket sorra tárgyaljuk a következőkben.

A *clone* módszer

A *clone* módszert akkor használjuk, ha létre szeretnénk hozni egy objektumot, egy már meglévő objektumból (másolatot készíteni róla). Az adott osztállyal megegyező típusú új példányt hoz létre:

```
Object cloneObjectClone();
```

A módszer a *CloneNotSupportedException* kivételt dobja, ha a klónozás nem támogatott az osztály számára. A klónozás akkor támogatott, ha az osztály implementálja a *Cloneable* interfészt. Habár az *Object* tartalmazza a *Clone* módszert, nincsen megvalósítva az interfész. Ha az objektum, ahol a *clone*-ra hivatkozunk, nem implementálja a *cloneable* interfészt, egy eredetivel azonos típusú és értékű objektum jön létre. Leggyeszebb azonban, ha az osztály deklarációjában létrehozunk egy *implements Cloneable* sort.

Bizonyos osztályoknál a helyes működés feltétele a *clone* felüldefiníciója. Tekintsünk egy *Stack* osztályt, mely tartalmaz egy tagváltozót, mely az *Object*-ek tömbjére hivatkozik. Ha a *Stack* az *Object* osztály *clone* módszerére épül, akkor az eredeti és a másolt *Stack* ugyanazokat az elemeket fogja tartalmazni, mivel az adattag tömb, és másolásakor csak referencia másolás fog történni.

A *Stack* osztálynak olyan *clone* implementációra van szüksége, amely lemásolja a *Stack* objektum adatait, ezzel biztosítva a megfelelő tartalom szétválasztást:

```
public class Stack implements Cloneable {
    private Object[] items;
    private int top;
    ...
}
```

```
protected Stack clone() {
    try {
        Stack s = (Stack)super.clone(); //clone the stack
        s.items = (Object[])s.clone(); //clone the array
        return s; // return the clone
    } catch (CloneNotSupportedException e) {
        //This shouldn't happen because Stack is Cloneable.
        throw new InternalError();
    }
}
```

Az implementáció viszonylag egyszerű. Először a *clone* módszer *Object* implementációját hívódik meg a *super.clone* segítségével, mely létrehoz és inicializál egy *Stack* objektumot. Ilyenkor mindkét objektum ugyanazokat az objektumokat tartalmazza. Ezután a módszer lemásolja az objektumokat, és a módszer *Stack*-el tér vissza.

Megjegyzés: A *clone* módszer nem a *new*-t használja a másolat létrehozásánál és nem hív konstruktorokat, helyette a *super.clone*-t használja, mely létrehozza az objektumot a megfelelő típussal, és engedélyezi a másolást, minnek eredményeképpen a kívánt másolatot kapjuk.

Erdemes még azt is megfigyelni, hogy az adattag másolását sem „kézzel” végezte a módszer, hanem a tömb objektum *clone* módszerével. Ez a módszer egy másik azonos méretű tömböt hoz létre, aminek elemeiről is másolat készül. (A tömbben tárolt tagokról már nem fog másolat készülni, de ez nem is célja egy verem másolásának.)

Az *equals* és *hashCode* módszerek

Az *equals* módszer két objektumot hasonlít össze és dönti el, hogy egyenlő-e vagy sem (ha egyenlő, *true*-val tér vissza). Ha önmagával hasonlítottunk egy objektumot, *true*-t ad vissza.

A következő programrészlet összehasonlít két *Integer*-t:

```
Integer one = new Integer(1);
Integer anotherOne = new Integer(1);
if (one.equals(anotherOne)) {
    System.out.println("objects are equal");
}
```

A program kimenete:

```
| objects are equal
```

Egyenlők, mivel az értékük megegyezik. Ha két objektum egyenlő az *equals* módszer szerint, akkor a *hashCode* módszer által szolgáltatott értékeknek is meg kell egyezniük. (Figyelem, fordítva ez nem feltétlenül igaz!)

Ha az *equals* működése nem megfelelő az osztályunk számára, akkor felül kell írunk az osztályunkban.

A *hashCode* módszer állítja elő az objektumok hash kódját, ami például akkor lehet szükséges, ha az objektumot hashtáblában tároljuk. Hash kódként (a módszer visszatérési értékeként) mindig egy *int* típusú számot kapunk.

Heyles hash függvény írása egyszerű, azonban hatékony függvény írása nehéz lehet, komolyabb munkát igényel. Ez a téma azonban már nem fér bele a jegyzetünkbe.

A finalize módszer

Az *Object* osztály ugyancsak tartalmazza a *finalize* módszert. A személynévtároló meghívja, ha már nincs egyetlen hivatkozás sem az objektumra. A *finalize* módszer automatikusan meghívódik, melyet a legtöbb osztály használ, ezért nem is kell külön meghívni. A *finalize* módszerrel leggyakrabban nem kell törödnünk, az östől örökölt módszer többnyire megfelelően működik.

A toString módszer

Az objektumot *String*-ként ábrázolja. Hasznos minden új osztály definíciója során felülírni, hogy a megfelelő értékeket reprezentálhasson. Használhatjuk a *toString*-et a *System.out.println*-nel együtt az objektumok szöveges megjelenítésére, pl.:

```
System.out.println(new Double(Math.PI).toString());
```

A futás eredménye:

```
3.14159
```

Nagyon hasznos ez a módszer akkor, ha a program tesztelési fázisában bizonyos objektumok tartalmát ellenőrizni szeretnénk. Ilyenkor csak ki kell írni a kérdéses objektumot, például a konzolra:

```
System.out.println(anObject);
```

A getClass módszer

Visszaadja a futásidejű osztályát az objektumnak. Az *Object* osztály nem engedi meg a *getClass* módszer felüldefinícióját (*final*).

A következő módszer az objektum osztálynevet jeleníti meg:

```
void PrintClassName(Object obj) {
    System.out.println("The Object's class is "
        + obj.getClass().getName());
}
```

A következő példa létrehoz az *obj* típusával megegyező másik objektum példányt:

```
Object createNewInstanceOf(Object obj) {
    return obj.getClass().newInstance();
}
```

Ha tudjuk az osztály nevét, kaphatunk egy *Class* objektumot az osztálynevből. A következő két sor egyaránt ugyanazon végeredményt produkálja (a második változat hatékonyabb):

```
String.class
Class.forName("String")
```

12.5. Végleges osztályok és módszerek**Végleges osztályok**

A *final* kulcsszó segítségével deklarált változók értékét az inicializálás után nem lehet megváltoztatni, a leszármazott osztály nem módosíthatja, befolyásolhatja az eredeti műkódést. Fontos szempont a rendszer biztonságának növelése és az objektum orientált tervezés szempontjából.

Biztonság: Az egyik módszer, amit a hackerek használnak rendszerek feltörésénél, egy származtatott osztály létrehozása egy osztályból, majd helyettesítése az eredetivel. A származtatott osztály a metódushívás szempontjából úgy néz ki, mint az eredeti, de a vi-selkedésé teljesen más is lehet, ami hibás működést eredményezhet. Ennek elkerülése érdekében deklarálhatjuk osztályunkat véglegessé, mely megakadályozza a származtatott osztályok létrehozását. A *String* osztály is végleges. Ez az osztály nélkülözhetetlen a Java Platform működéséhez. Ez biztosítja, hogy minden *String* a megfelelő módon működjön.

Ha megpróbáljuk lefordítani egy *final* osztály leszármazott osztályát, hibahüzenetet fogunk kapni.

Tervezés: Az objektumorientált tervezésnél érdemes megállapítani, hogy mely osztályokat szeretnénk véglegessé tenni, és tegyük is az adott osztályokat véglegessé a *final* módosító segítségével:

```
final class ChessAlgorithm {
    ...
}
```

Minden leszármaztatási próbálkozás hibás lesz.

Végleges metódusok

A *final* kulcsszót használjuk a deklarációban, ha azt akarjuk elérni, hogy ne lehessen a metódust származtatott osztályban felüldefiniálni. Az *Object* metódusai közül van, amelyek *final* típusú, és van, amelyek nem.

A következő példában a *ChessAlgorithm* osztályban a *nextMove* metódus tesszük véglegessé:

```
class ChessAlgorithm {
    ...
    final void nextMove(ChessPiece pieceMoved,
        BoardLocation newLocation) {
        ...
    }
}
```

12.6. Ellenőrző kérdések

- Mit jelent az, hogy egyik osztály leszármazottja a másiknak?
- Lehet-e egy osztályreferenciát a szülőosztály felé konvertálni?
- Lehet-e egy osztályreferenciát a leszármazott osztály felé konvertálni?
- Lehet-e Java-ban különböző típusú értékek közöttértékkadás? Ha igen, mikor?
- Ha létrehozunk egy példányt, és egy szülőosztály típusa szerinti referenciával hívunk ki a *nextMove* metódust, a szülőosztály vagy a leszármazott osztály szerinti metódus hívódik-e meg?
- Mit jelent az osztályok újrafelhasználhatósága? Hogyan valósul meg Java-ban?
- Az osztály mely tagjait öröklí a leszármazott osztály?
- Mikor lehet egy metódust a leszármazottban elfedni (elrejtteni)?

- Hogyan lehet hivatkozni a leszármazott osztályban az `ős` elrejtett adattagjára?
- Mire használható a *super* kulcsszó?
- Milyen esetben szükséges az `ős` osztály konstruktorát explicit meghívni?

Igaz vagy hamis? Indokolja!

- Egy Java fordítási egységben pontosan egy osztály szerepel.
- Bármiely *.class* kiterjesztésű állományt lehet közvetlenül futtatni.
- Lehet-e eltérés az `ős`ben definiált metódus és a leszármazottban felülírt változat láthatóságában?
- Lehet olyan metódus, amelyet egy leszármazottban nem lehet felülírni?
- Végleges osztálynak kell-e végleges metódust tartalmazni?
- A végleges metódust tartalmazó osztály maga is végleges?

Melyik egy publikus, absztrakt metódus helyes deklarációja?

- `public abstract void add();`
- `public abstract void add() {}`
- `public virtual add();`

A leszármazott osztály konstruktorában hova kell írni a szülőosztály konstruktorának hívását?

- akárhova
- a konstruktor első sorába
- a konstruktor utolsó sorába
- nem kell meghívni

12.7. Gyakorló feladatok

Készítsen absztrakt *Jármű* osztályt, amely más speciálisabb *Jármű* osztályok (pl. *Autó*) közös `ős`e lehet!

A konstruktornak lehessen megadni, hogy alkalmass-e szárazföldi, vízi és légi közlekedésre (külön-külön, tehát 3 logikai paraméterrel). Tárolja el ezeket az adatokat, hogy később lekérdezhetőek legyenek.

Készítsen *szárazföldi*, *vízi* és *légi* metódusokat a konstruktorban beállított értékek lekérdezésére.

Készítsen *Autó* osztályt, amely az `ős`éhez képest tudja tárolni az utasok számát is, és a konstruktorán keresztül ezt be lehessen állítani. Készítsen ehhez is lekérdező metódust.

13. Beágyazott osztályok

Megadhatunk egy osztályt egy másik osztály tagjaként. Egy ilyen osztályt beágyazott osztálynak hívunk, és a következőképpen néz ki:

```
class EnclosingClass {
    ...
    class ANestedClass {
        ...
    }
}
```

A beágyazott osztályokat arra használjuk, hogy kifejezzük és érvényesítsük két osztály között a kapcsolatot. Megadhatunk egy osztályt egy másik osztályon belül, hogyha a beágyazott osztálynak a magába foglalt osztályon belül környezetben van értelme. Pl. a szöveggkurzornak csak a szöveggkurzornak komponensen belül környezetben van értelme.

A beágyazó osztály tagjaként a beágyazott osztály kiváltságos helyzetben van. Korlátlan hozzáféréssel rendelkezik a beágyazó osztály tagjaihoz még akkor is, hogy ha azok privátként vannak deklarálva. Azonban ez a speciális kiváltság nem mindig speciális. A hozzáférést biztosító tagok korlátozzák a hozzáféréseket az olyan osztálytagokhoz, amelyek a beágyazó osztályon kívül esnek. A beágyazott osztály a beágyazó osztályon belül található, ebből kifolyólag hozzáférhet a beágyazó osztály tagjaihoz.

Mint ahogyan más tagokat is, a beágyazott osztályokat is statikusként, avagy nem statikusként lehet deklarálni, ezért ezeket pontosan így is hívják: **statikus beágyazott osztály**. A nem statikus beágyazott osztályokat **belső osztály**oknak hívjuk.

```
class EnclosingClass {
    ...
    static class StaticNestedClass {
        ...
    }
    ...
    class InnerClass {
        ...
    }
}
```

Ahogy a statikus metódusok és változók esetén, amelyeket mi osztálymetódusoknak és változóknak hívunk, a statikus beágyazott osztályt a beágyazó osztályával kapcsoljuk össze. Ahogy az osztálymetódusok, a statikus beágyazott osztályok sem hivatkozhatnak közvetlenül olyan példányváltozókra vagy metódusokra, amely az ő beágyazó osztályában van megadva. A példánymetódusok és változók esetén egy belső osztály az ő beágyazó osztályának a példányával kapcsolódik össze, és közvetlen hozzáférése van annak az objektumnak a példányváltozóhoz és metódusaihoz. Mivel egy belső osztályt egy példányával társítanak, ezért ön maga nem definiálhat akármilyen statikus tagot.

Hogy a továbbiakban könnyebb legyen megkülönböztetni a beágyazott osztály és a belső osztály fogalmát, érdemes ezekre a következőképpen tekinteni: a beágyazott osztály két osztály közötti szintaktikus kapcsolatot fejez ki, azaz szintaktikailag az egyik osztályban lévő kód megtalálható a másikon belül. Ezzel ellentétben a belső osztály olyan objektumok közötti kapcsolatot fejez ki, amelyek két osztálynak a példányai. Tekintsük a következő osztályokat:


```

class EnclosingClass {
    ...
    class InnerClass {
        ...
    }
}

```

Az e két osztály közötti kapcsolatalnál nem az az érdekes, hogy az *InnerClass* szintaktikusan definiálva van az *EnclosingClass*-on belül, hanem az, hogy az *InnerClass*-nak a példány csak az *EnclosingClass*-nak a példájában belül létezhet, és közvetlen hozzáférése van a példányváltozók és a beágyazó osztály példánymetódusaihoz.

A beágyazott osztályok mindkét fajtájával találkozhatunk a Java API-n belül, és kell is használni őket. Azonban a legtöbb beágyazott osztály, amelyeket használunk, valószínűleg belső osztály lesz.

13.1. Belső osztályok

Ahhoz, hogy megértsük, mi a belső osztály és mire jó, tekintsük át újra a *Stack* osztályt. Tegyük fel, hogy ehhez az osztályhoz hozzá szeretnénk adni egy olyan tulajdonságot, amely lehetővé teszi egy másik osztály számára, hogy kilistázza a veremben lévő elemeket vagy tagokat *java.util.Iterator* interfész használatával. Ez az interfész három metódus deklarációt tartalmaz:

```

public boolean hasNext();
public Object next();
public void remove();

```

Az *Iterator* definiálja azt az interfészt, ami végigmegy egy rendszer az elemeken egy megadott sorrend szerint, amelyet a következőképpen adunk meg:

```

while (hasNext()) {
    next();
}

```

A *Stack* osztály önmaga nem tudja végrehajtani az *Iterator* interfészt, mert az *Iterator* interfész API-ja bizonyos korlátokat szab: két különböző objektum nem listázhatja kis egyeztetve a veremben lévő elemeket. Ugyanis nincs lehetőség arra, hogy megtudjuk, ki hívja meg a következő metódust, és a listázást nem lehet újraindítani, mert az *Iterator* interfésznek nincsen olyan metódusa, amely támogatná ezt. Így a kilistázást csak egyszer lehet végrehajtani, mert az *Iterator* interfésznek nincs arra metódusa, hogy az elejéhez visszamenjen a bejárás. Ehelyett egy belső osztály van, amely elvégzi a verem számára ezt a munkát. A belső osztály hozzáférhet a verem elemeihez, és már csak azért is hozzá kell, hogy tudjon férni, mert a veremnek a publikus interfésze csak LIFO hozzáférést támogat. Itt lép be a képbe a belső osztály. Itt látható egy verem implementáció, ami definiál egy belső osztályt, amelyet *StackIterator*-nak hívunk, és kilistázza a verem elemeit.

```

public class Stack {
    private Object[] items;

    public Iterator iterator() {
        return new StackIterator();
    }
}

```

```

class StackIterator implements Iterator {
    int currentItem = items.size() - 1;
    public boolean hasNext() {
        ...
    }
    public Object next() {
        ...
    }
    public void remove() {
        ...
    }
}

```

Figyeljünk meg, hogy a *StackIterator* osztály közvetlenül hivatkozik a veremben levő elemek példányainak változóira. A belső osztályokat elsősorban arra használjuk, hogy implementáljuk a segítő osztályokat, úgy, mint itt, ebben a példában is láthatjuk. Ha GUI eseményeket akarunk használni, akkor ismernünk kell a belső osztályok használatának szabályait, mert az eseménykezelés mechanizmusa elég erőteljesen használja ezeket. Deklarálhatunk egy belső osztályt anélkül, hogy nevet adnánk neki. Az ilyen osztályokat névtelen osztályoknak hívjuk. Itt látható még egy változata a *Stack* osztálynak, ebben az esetben névtelen osztályt használunk az *Iterator* számára.

```

public class Stack {
    private Object[] items;
    public Iterator iterator() {
        return new Iterator() {
            int currentItem = items.size() - 1;
            public boolean hasNext() {
                ...
            }
            public Object next() {
                ...
            }
            public void remove() {
                ...
            }
        }
    }
}

```

A névtelen osztályok nehezessé tehetik a kód olvasását, ezért csak olyan osztályokhoz érdemes használnunk, amelyek nagyon kicsik (nem több mint egy-két metódus), és amelyek használata eléggé egyértelmű, mint például az eseménykezelő osztály.

13.2. Néhány további érdekesség

Más osztályokhoz hasonlóan a beágyazott osztályokat is lehet absztrakttnak vagy véglegesnek deklarálni. Ennek a két módosítónak a jelentése a beágyazott osztály esetén is ugyanaz, mint más osztályoknál.

Ugyanígy használhatjuk a hozzáférés módosítókat – mint például a *private*, *public* és *protected* –, hogy korlátozzuk a beágyazott osztályokhoz való hozzáférést, mint ahogy minden más osztálytag esetén tehetjük azt. Nem csak a névtelen, hanem bármilyen be-

ágyazott osztály deklarálható bármilyen kódblokkon belül. A beágyazott osztály, amely egy metóduson vagy egy másik kódblokkon belül van deklarálva, hozzáférhet bármilyen hatókörön belüli végleges vagy lokális változóhoz.

13.3. Ellenőrző kérdések

A következő kódban melyik nem kerülhet az XX helyére?

```
public class MyClassl {  
    public static void main(String argv[]) { }  
    XX class MyInner { }  
}
```

- *public*
- *private*
- *static*
- *friend*

14.Felsorolás típus

A felsorolás típus egy olyan típus, melynek megengedett értékei fix konstansokból állnak. Javában a felsorolás típust *enum* szóval definiáljuk. Pl. a hét napjait így definiálhatjuk:

```
enum Days {
    VASARNAP, HETFO, KEDD, SZERDA, CSUTORTOK, PENTEK, SZOMBAT};
}
```

Vegyük észre, hogy konvencionálisan a nevek a felsorolás típusnál nagy betűkkel írandók, így a kód olvasásánál könnyen felismerhetők a konstansok. Bármikor használhatunk felsorolás típust, ha szükségünk van kötött konstans értékekre. Ez magában foglalja a természetes felsorolás típusokat, mint a Naprendszer bolygói, a hét napjai, kártyapakli lapjainak neve/értéke és minden olyan esetet, ahol az összes lehetséges étéket tudjuk a fordításkor.

A Java programozási nyelv felsorolás típusa sokkal hatékonyabb, mint más nyelvekben szereplő megfelelői, melyek csak nevesített egész számok. Az *enum* deklaráció egy osztályt definiál, úgynevezett *enum* típust, melynek legfontosabb jellemzői a következők:

- Beszédesebbek az egyszerű literáloknál
- Típusbiztosak
- Saját névterük van
- Erdemes *switch-case* szerkezetben felsorolási típus alapján szervezni
- Van egy statikus *values* metódusuk, mely egy tömböt ad vissza, melyben a típus értékei szerepelnek deklarálási sorrendben. Ez a módszer pl. *for-each* ciklussal nagyon hasznos.

- Tartalmazhat metódusokat, adattagokat, implementálhat interfészeket stb.
- Minden *Object* metódust implementálnak. Összehasonlíthatók és szerializálhatók.

A következő példában a *Planet* egy felsorolás típus, mely a Naprendszer bolygóit jeleníti meg. A bolygónak van egy konstans tömeg és sugár paramétere. Minden konstans tömeggel és sugárral deklaráltnak, melyet átadnak a konstruktornak a létrehozáskor. Vegyük észre, hogy a felsorolás típus konstruktorát értelemszerűen privát. Ha megpróbálunk egy publikus konstruktor létrehozni a felsorolás típusnak, akkor a fordító hibáüzenetet fog visszaadni.

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURN (5.688e+26, 6.0268e7),
    URANUS (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO (1.27e+22, 1.137e6);
}
```

```

private final double mass; //in kilograms
private final double radius; //in meters
Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
}

public double mass() { return mass; }
public double radius() { return radius; }

public static final double G = 6.67300E-11;

public double surfaceGravity() {
    return G * mass / (radius * radius);
}

public double surfaceWeight(double otherMass) {
    return otherMass * surfaceGravity();
}
}

```

A bolygónak az értékei mellett metódusai is vannak, melyeken keresztül kinyerhető a felszíni gravitáció és egy tárgy súlya minden bolygón. A következő példaprogram egy ember Földön mért súlya alapján kiszámolja, majd kiírja ugyanezenek az embernek a súlyát minden bolygón.

```

public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight/EARTH.surfaceGravity();
    for (Planet p : Planet.values()) {
        System.out.printf("Your weight on %s is %f\n",
            p, p.surfaceWeight(mass));
    }
}

```

A kimenet:

```

$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031

```

Csak egyetlen megkötése van a felsorolás típusnak: habár a felsorolás típusok osztályok, nem definiálható hierarchia számukra. Más szavakkal: nem lehet leszármazottja a felsorolási típusnak.

Végezetül a *java.util* csomag tartalmaz két speciális *Set* és *Map* implementációt, mely támogatja a felsorolás típusokat, az *EnumSet*-et és az *EnumMap*-et.

14.1. Ellenőrző kérdések

- Mikor érdemes felsorolás típust alkalmazni?
- Milyen korlátozások vannak a felsorolás típusra más osztályokhoz képest?

15. Általános programozás

Az általános (generikus) programozás típusparaméterek segítségével teszi lehetővé, hogy osztályokat, interfészeket hozzassunk létre úgy, hogy bizonyos paraméterek típusait a példányosításakor dönthessük el.

15.1. Általános típus definiálása és használata

Ahhoz, hogy általános típust definiáljunk, a típusdefiniciónak tartalmaznia kell egy vagy több típus paramétert a típus neve után. A típus paraméterek vesszővel elválasztott listái '<' és '>' között szerepelnek. Konvencionálisan a típus paraméterek nagybetűsek. A típus paraméterek aztán megjelennek a típus metódusaiiban, vagy a metódus paraméterlistájában, vagy visszatérési érték típusként. A gyakorlatban a legtöbb fejlesztőnek nincs szüksége új általános típusok definiálására, de szükségesség megfigyezni a szintaxisát és a használatát az általános típusnak.

Nézünk újra át a *Stack* osztályt, melyet az osztályok létrehozásánál mutattunk be. A generikus verzió (*Stack*<*T*>), egy gyűjteményt használ (*ArrayList*<*T*>), hogy a veremben tárolja az elemeket.

```
public class Stack2<T> {
    private ArrayList<T> items;
    ...
    public void push(T item) { ... }
    public T pop() { ... }
    public boolean isEmpty() { ... }
}
```

Vegyük észre, hogy a *T* típus paraméter bevezetésre került az osztálynev után, és ezek után feltűnik, mint a *push* metódus paraméter típusa, és a *pop* metódus visszatérési típusa.

A gyűjtemények gyakran használatosak arra, hogy bemutassuk az általános típus használatát, mivel nagyon jellemzőek az interfészekben és osztályokban. A valóságban a gyűjtemények voltak a fő motivációs erő az általános típusok bevezetésénél a Java nyelvben, mivel elérhetővé teszik a fordítás idejű ellenőrzését a gyűjteményeken végzett műveletek típusbiztonságának. Amikor specifikáljuk egy gyűjteményben tárolt objektum típusát:

- a fordító tud ellenőrizni bármilyen műveletet, ami egy objektumot ad a gyűjteményhez

- ismert az objektum típusa, mely a gyűjteményből lett kinyerve, ezért nincs szükség arra, hogy *cast*-oljuk (átalakítsuk) típusa. Ugyanakkor nincs lehetőség arra sem, hogy átalakítsunk egy rossz típusa, és ekkor megtapasztaljunk egy futás idejű *ClassCastException* kivételt.

Ahogy az előbbieken írtuk, ha egy általános típust használunk, helyettesítjük a paraméter egy aktuális típus paramétert, nagyjából ugyanezen módszerrel helyettesítünk egy metódushoz a paramétereihez tartozó aktuális értékeket. Egy aktuális típus paraméternek referencia típusnak kell lenni, nem lehet primitív. Például, itt látható az, hogy hogyan lehet létrehozni *Stack2*-t *String* típus paraméterrel; és ezek után *push*-olni és *pop*-olni a „hi” *String*-et.

Egy nem generikus verem kód így nézne ki:

```
Stack2<String> s = new Stack2<String>();
s.push("hi");
String greeting = s.pop(); //no cast required here

Stack s = new Stack();
s.push("hi");
String greeting = (String)s.pop(); //cast required here
```

Vegyünk észre, hogy amikor az általános típust használjuk, akkor a fordító egy olyan technikával fordítja le az általános típust, amit típus törlésnek hívnak.

Gyakorlatilag a fordító törli az összes olyan információt, mely a típus paraméterrel, vagy a típus paraméterekkel kapcsolatos. Például a `Stack2<string>` típust lefordítja `Stack2`-re, melyet nyers típusnak neveznek. Abból következtethetünk a típus törlésre, hogy a típus paraméter nem érhető el futásidőben ahhoz, hogy használjuk típuskénszerítésben, vagy mint az *instanceof* eljárás paramétere.

15.2. Kapcsolatok az általános típusok között

Valószínűleg azt várjuk, hogy a `Stack2<Object>` a szülő típusa a `Stack2<String>`-nek, mert az *Object* szülője a *String*-nek. A valóságban nem létezik ilyen kapcsolat az általános típusok példányai között. A szülő-gyermek kapcsolat hiánya az általános típus példányai között nehézkessé teheti a többalakú (polimorf) eljárások megírását.

Tegyük fel, hogy szeretnénk megírni egy eljárást, ami kiírja egy gyűjteményben tárolt objektumokat az objektumok típusától függetlenül a konzolra.

```
public void printAll(Collection<Object> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

Választhatnánk, hogy létrehozunk egy *String* listát és ezt a metódust használjuk az összes *String* kiírására:

```
List<String> list = new ArrayList<String>();
...
printAll(list); //error
```

Ha ezt használjuk, akkor észre fogjuk venni, hogy az utolsó sor fordítási hibát ad. Mivel az *ArrayList<String>* nem leszármazott típusa a *Collection<Object>*-nek, ezért nem adható át, mint paraméter a kiírásnak annak ellenére, hogy a két típus ugyanannak az általános típusnak a leszármazottai, rokon öröklött típus paraméterekkel. Más részből az öröklés miatt kapcsolatban álló általános típusok ugyanazzal a típus paraméterrel kompatibilisek:

```
public void printAll(Collection<Object> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}

List<Object> list = new ArrayList<Object>();
...
printAll(list); //this works
```

A *List<Object>* kompatibilis a *Collection<Object>*-el, mert a két típus példánya egy általános szülőtypusnak és annak leszármazott típusának, és a példányok ugyanahhoz a típus paraméterhez tartoznak, konkrétan az *Object*-hez.

15.3. Helyettesítő típus

Ahogy körbejárjuk a *printAll* metódus első változata által felvetett kérdést, leszögezhetjük, hogy a *printAll* paramétere egy gyűjtemény, melynek elemi típusa lehet bármi, amit *Collection<?>* formában leírhatunk:

```
public void printAll(Collection<? c>) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

A *?*-es típus határozatlan típusként ismert. A gyűjteményből bármikor kiolvasható az objektum, mert a visszatérési érték mindig garantáltan *Object*. Azonban nem adható objektum a gyűjteményhez, mert a *?* ismeretlen típust jelöl, és nem lehet egyértelműen tudni, hogy a hozzá adni kívánt objektum leszármazott típusa-e az ismeretlen típusnak. Az egyetlen kivétel a *null*, amely eleme minden típusnak.

Korlátozhatjuk (vagy kényszeríthetjük) is a helyettesítő típust valamely típus segítségével. A korlátozott helyettesítő típus akkor hasznos, mikor csak részben ismerjük a paramétereket. Például tegyük fel, hogy van egy osztály hierarchiánk geometriai alakzatokból (*Shape*), és ennek leszármazott típusaiból (*Circle*, *Rectangle*, és így tovább). A rajzoló program, ami ezekre az objektumokra hivatkozik, meghív egy *drawAll* nevű metódust, hogy egy gyűjteményt rajzoljon ezekből az alakzatokból:

```
public void drawAll(Collection<Shape> shapes) {
    for (Shape s : shapes) {
        s.draw();
    }
}
```

Mivel láthatjuk, hogy nem megengedett a *Shape* típus valamely leszármazott típusának (például a *Circle*) alkalmazása, ezért ez a metódus csak korlátozottan használható: például nem hívható meg a *Collection<Circle>* esetén. Hogy lehetővé tegyük a *Shape* típus valamely leszármazott típusának típus paraméterként való alkalmazását, kiterjeszthetjük az alakzatok gyűjtemény típus paraméterét helyettesítő típusként. De mivel tudjuk, hogy a típus paraméter valamilyen alakzat lesz, a helyettesítést korlátozhatjuk a *Shape* típusra a következőképpen:

```
| void drawAll(Collection<? extends Shape> shapes) { ... }
```

Ez lehetővé teszi a *drawAll* számára, hogy elfogadjon bármilyen, a *Shape* típus leszármazott típusából álló gyűjteményt.

Osszefoglalva, a helyettesítő típus felső korlátozással specializálható a *<? extends Type>* módon, ezáltal megfélelővé téve az adott *Type* minden leszármazott típusához. A helyettesítő típust alulról is korlátozhatjuk. Egy alulról korlátozott helyettesítő a *<? super Type>* módon írható le, és a *Type* minden osztályára használható. Megfigyelhetjük, hogy továbbra sem lehetséges ismeretlen típusú gyűjteményhez objektumot hozzáadni, és ez nem lehetséges korlátozott, ismeretlen típusú gyűjtemény esetén sem.

15.4. Általános metódusok definiálása és használata

Nem csak a típusokat, a metódusokat is paraméterezhetjük. A statikus és nem statikus metódusoknak épp úgy, ahogy a konstruktoroknak, lehet típus paraméterük.

A metódusok típus paraméter deklarálásának szintaxisa megegyezik az osztályoknál használt szintaxissal. A típusparaméter listát a kerek zárójelek közé kell helyezni, még a metódus visszatérési értékének típusa elé. Például a következő gyűjtemény osztály metódus feltölt egy `<? super T>` típusú listát `T` típusú objektumokkal:

```
static <T> void fill(List<? super T> list, T obj)
```

Az általános metódusok lehetővé teszik a típus paraméterek használatát, hogy egyértelművé tegyék a típusok kapcsolatát egy vagy több paraméter esetén, egy metódus, vagy annak visszatérési értéke számára (vagy mindkettőnek). Az általános metódusok típus paraméterei általában osztálytól függetlenek vagy interéjsz-szintű típus paraméterek. A *Collections* osztály algoritmusainak definíciója sok lehetőséget kínál az általános metódusok használatára.

Az egyetlen különbség az általános típusok és általános metódusok között az, hogy az általános metódusokat hagyományosan metódusként hívjuk meg. A típus paramétereket a hívás függvényében állapítjuk meg, ahogy a *fill* metódus alábbi hívásánál is:

```
public static void main(String[] args) {
    List<String> list = new ArrayList<String>(10);
    for (int i = 0; i < 10; i++) {
        list.add("");
    }
    String filler = args[0];
    Collections.fill(list, filler);
    ...
}
```

15.5. Általános típusok használata az öröklésben

A fejezetben már korábban tárgyalt *Cat* osztálynak tekintjük a *getLitter* nevű metódusát, melynek visszatérési értéke egy *Cat* objektumokból álló gyűjtemény:

```
public Collection getLitter(int size) {
    ArrayList litter = new ArrayList(size);
    for (int i = 0; i < size; i++) {
        litter.add(i, new Cat());
    }
    return litter;
}
```

Megfigyelhető, hogy a *Collection* objektum hatarozatlan típusú: a gyűjteményben lévő objektumok típusa nem meghatározott. Azonos helyzet állt fenn minden további metódusnál is, melyek objektumot adnak vissza, még mielőtt az általános típusok elérhetővé váltak volna a Java 5.0-as verzióban.

Tegyük fel, hogy írunk egy programot *Cats* néven, mely ezt a visszatérési értéket adja át egy gyűjteménynek, melynek típusa kifejezetten *Cat* kell, hogy legyen:

```

public static void main(String[] args) {
    Collection<Cat> litter = myCat.getLitter(3);
    for (Cat c : litter) {
        System.out.println(c.getColor());
    }
}

```

Mikor lefordítja a *Cats.java* állományt, a következő figyelmeztetést kapja:

A *Cats.java* ellenőrzetlen vagy nem biztonságos műveleteket használ. Fordítsa le újra a -Xlint:unchecked kapcsolóval a részletek megtekintéséhez.

Az *Xlint:checked* kapcsoló használata információgyűjtéshez:

```

% javac -Xlint:unchecked Cats.java
Cats.java:5: warning: [unchecked] unchecked conversion
    found   : java.util.Collection
    required: java.util.Collection<Cat>
    Collection<Cat> litter = myCat.getLitter(3);

```

Összegezve, ha a *Cat*-et olyan fordítóval fordítjuk újra, mely támogatja a típus paramétereket, a következő figyelmeztetést kapjuk:

```

% javac -Xlint:unchecked Cat.java
Cat.java:19: warning: [unchecked] unchecked call to
add(int,E) as a member of the raw type java.util.ArrayList
    litter.add(1, new Cat());

```

Habár a kód hibátlan, ez a figyelmeztetés mutatja, hogy a fordító nem tudja biztosítani a művelet pontosságát, mikor speciális típusú gyűjteményeket használunk. Amikor „ellenőrzetlen” figyelmeztetést kap, ellenőriznie kell, hogy a művelet, mely a figyelmeztetést generálta, valóban megfelelő-e.

Végül tekintsünk át egy listát a teljes *Stack2* osztályról:

```

public class Stack2<T> implements Cloneable {
    private ArrayList<T> items;
    private int top=0;

    public Stack2() {
        items = new ArrayList<T>();
    }

    public void push(T item) {
        items.add(item);
        top++;
    }

    public T pop() {
        if (items.size() == 0)
            throw new EmptyStackException();
        T obj = items.get(--top);
        return obj;
    }

    public boolean isEmpty() {
        if (items.size() == 0)
            return true;
        else
            return false;
    }
}

```

```

protected Stack2<T> clone() {
    try {
        Stack2<T> s = (Stack2<T>)super.clone();
        s.items = (ArrayList<T>)items.clone();
        return s; // Return the clone
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}

```

Megfigyelhetjük, hogy a *clone* módszer a *clone* metódusokat az osztályából és a tartalmazott listájából hívja meg. A *clone* metódusok öröklődő metódusok, mert még az általános típusok elérhetősége előtt definiálva lettek.

Mikor lefordítja a *Stack2.java* állományt, a következő figyelmeztetést kapja:

```

% javac -Xlint:unchecked Stack2.java
Stack2.java:32: warning: [unchecked] unchecked cast
    found   : java.lang.Object
    required: Stack2<T>
    Stack2<T> s = (Stack2<T>)super.clone();
Stack2.java:33: warning: [unchecked] unchecked cast
    found   : java.lang.Object
    required: java.util.ArrayList<T>
    s.items = (ArrayList<T>)items.clone();
2 warnings

```

Ez a figyelmeztetés mutatja, hogy a fordító nem képes biztosítani az eljárás kifogástalan működését. Más szóval, mivel a *clone* metódust úgy definiálták, hogy egy *Object* osztálybeli objektumot adjon visszatérési értéknek, ezért a fordító nem tudja biztosítani, hogy a gyűjtemény visszatérési értéke *Stack2<T>* legyen. Azonban a *clone* metódus feletti megállapodás szerint a művelet engedélyezett, habár „ellenőrizetlen” figyelmeztetést okoz.

Rendteleg apró dologra kell figyelni az általános típusok használatánál az öröklésben.

15.6. Ellenőrző kérdések

- Mikor érdemes típus paramétert alkalmazni?
- Mi az előnye a típus paraméter alkalmazásának?

16. Interfészek

Az interfész olyan viselkedéseket definiál, amelyet az osztályhierarchia tetszőleges osztályával megvalósíthatunk. Egy interfész metódusok halmazát definiálja, de nem valósítja meg azokat. Egy konkrét osztály megvalósítja az interfészt, ha az összes metódusát megvalósítja.

Definíció: Az interfész implementáció nélküli metódusok nével ellátott halmaza.

Mivel az interfész a megvalósítás nélküli, vagyis absztrakt metódusok listája, alig különbözik az absztrakt osztálytól. A különbségek:

- Az interfész egyetlen metódust sem implementálhat, az absztrakt osztály igen.
- Az osztály megvalósíthat több interfészt, de csak egy ösztályja lehet.
- Az interfész nem része az osztályhierarchiának. Egy mástól "független" osztályok is megvalósíthatják ugyanazt az interfészt.

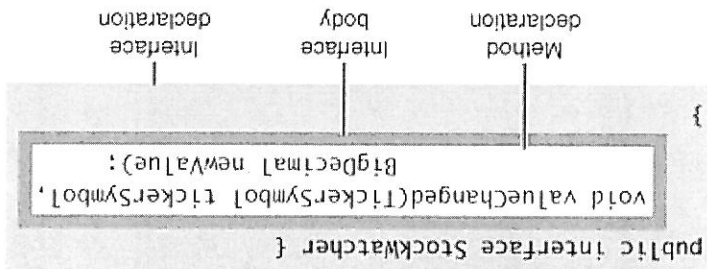
Ebben a fejezetben egy olyan példaprogramot fogjuk használni, ami a Tervezési minták (*Design Patterns*) között szokás emlegetni megfigyelő (*Observer*) néven. Ez az osztály megengedi más osztályoknak, hogy regisztrálják magukat bizonyos adatai változásának figyelésére. A *StockApplet* osztály fog megvalósítani egy olyan metódust, amivel regisztrálni tud a változás figyeléséhez:

```
public class StockMonitor {
    public void watchStock(StockWatcher watcher,
        TickerSymbol tickerSymbol,
        BigDecimal delta) {
        ...
    }
}
```

Az első paraméter egy *StockWatcher* objektum. A *StockWatcher* annak az interfésznek a neve, amelyet hamarosan látni fogunk. Az interfész egyetlen *valueChanged* nevű metódust definiál. Egy objektum akkor tudja magát megfigyelőként regisztrálni, ha az osztály megvalósítja ezt az interfészt. Amikor a *StockMonitor* osztály érzékeli a változást, meghívja a figyelő *valueChanged* metódusát.

16.1. Interfész definiálása

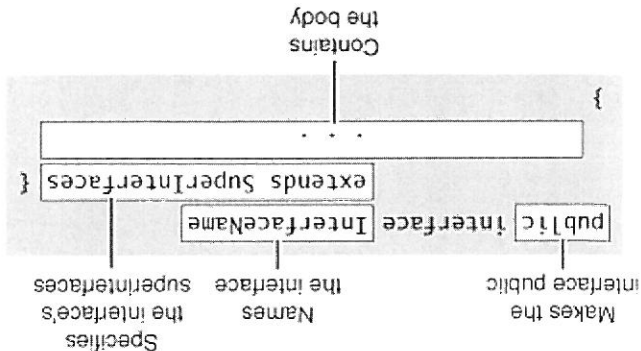
A következő ábra bemutatja az interfész definíció két összetevőjét: az interfész deklarációt és a törzset.



A *StockWatcher* interfész deklarálja, de nem implementálja a *valueChanged* metódust. Az interfészt megvalósító osztályok fogják a metódust implementálni.

Interfész deklaráció

A következő ábra az interfész deklaráció minden részét bemutatja:



Az interfész deklarációban két elem kötelező: az *interface* kulcsszó és az interfész neve. Ez után szerepelhetnek a szülőinterfészek.

Az interfész törzs

Az interfész törzs metódus deklarációkat tartalmaz, '-'el lezárva. Minden deklarált metódus értelmezésűen publikus és absztrakt (bár maguk a kulcsszavak nem írhatók ki). Az interfész tartalmazhat konstans deklarációt is. Minden konstans értelmezésűen publikus, statikus és végleges. (A kulcsszavak itt sem írhatók ki.) Nem használhatók a *transient*, *volatile*, *synchronized*, *private* és *protected* módosítók sem.

16.2. Interfészek implementálása

Egy interfész viselkedési formákat definiál. Az interfészt megvalósító osztály deklaráció-jában szerepel az *implements* záradék. Az osztály akár egy vagy több interfészt is megva-lósíthat. (Az interfészek között megengedett a többszörös öröklődés.)

Konvenció: az *implements* záradék az *extends* záradékot követi, ha mindkettő van. A következő példa applet implementálja a *StockWatcher* interfészt.

- Az interfész csak metódust tartalmazhat, változót nem.
- Minden osztály legfeljebb egy interfészt implementálhat.
- Az interfész összes változója *implicit static*.
- Az interfésznek nincs konstruktora.

Igaz vagy hamis? Indokolja!

16.4. Ellenőrző kérdések

```

public class StockMonitor {
    public void watchStock(Stock stock,
        StockWatcher watcher,
        TickerSymbol tickerSymbol,
        BigDecimal delta) {
        ...
    }
}

```

Amikor új interfészt definiálunk, referenciát definiálunk egy új adattípushoz. Az interfész nevét akárhol lehet használni, ahol egyéb adattípus is előfordulhat.

16.3. Az *interface* használata típusként

Amikor egy osztály megvalósít egy interfészt, akkor alapvetően aláír egy szerződést. Az osztálynak implementálni kell az interfészben és szülőinterfészekben deklarált összes metódust, vagy az osztályt absztraktként kell deklarálni. (Az ilyen absztraktki osztályoknak csak akkor lehet nem absztraktként egy leszármazottja, ha az ezen az öröklési szinten meg nem valósított metódusokat már maradátként megvalósítja.)

A *StockApplet* megvalósítja a *StockWatcher* interfészt, azaz szolgáltatásként nyújtja a *valueChanged* metódust.

```

public class StockApplet extends Applet
    implements StockWatcher {
    public void valueChanged(TickerSymbol tickerSymbol,
        BigDecimal newValue) {
        switch (tickerSymbol) {
            case SUNW:
                ...
                break;
            case ORCL:
                ...
                break;
            case CSCO:
                ...
                break;
            default:
                // handle unknown stocks
                ...
                break;
        }
    }
}

```

- Az interfész összes metódusa implicit *public*.
- Ha egy osztály megvalósít egy interfészt, azt a *uses* kulcsszóval kell jelezni.
- Az interfész összes metódusa implicit *abstract*.
- Az interfész összes változója implicit *public*.
- Az interfészt megvalósító osztálynak minden, az interfészben deklarált metódust implementálni kell.
- Létre lehet hozni interfész-referenciát.
- Minden interfésznek van őse.
- Az interfész összes változója implicit *final*.
- Minden interfészt legalább egy osztályban meg kell valósítani.
- Az interfész metódusainak nincs törzse.
- Az interfész és az absztrakciósztály között nincs lényeges különbség.

17. Csomagok

A típusok könnyebb megtalálásához és használatához, névutközések elkerüléséhez és az elérés szabályozásához a programozók egybe csomagolhatják az összetartozó típusaikat csomagokká.

Definíció: A csomag összetartozó típusok gyűjteménye.

A Java platform típusai funkciók szerint különböző csomagokba vannak szervezve: az alapvető osztályok a *java.lang* csomagban, az I/O osztályok a *java.io*-ban, és így tovább. Ezen kívül a saját típusainkat is tehetjük csomagokba.

A következő osztályokat megvizsgálva látszik, hogy közös csomagba érdemes őket sorolni, mivel grafikus objektumok csoportjába tartoznak a körök, téglalapok, vonalak és pontok. Ha írunk egy *Draggable* interfészt, az azt megvalósító osztályok lehetővé teszik a vonzolást is.

```
//in the Graphic.java file
public abstract class Graphic {
    . . .
}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {
    . . .
}

//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable {
    . . .
}

//in the Draggable.java file
public interface Draggable {
    . . .
}
```

A következő okok miatt érdemes az osztályokat és interfészeket közös csomagba helyezni:

- Más programozók számára is látszik, hogy kapcsolódó típusokról van szó.
- Más programozók is láthatják, hol kell keresni a grafikához kapcsolódó osztályokat.
- A típusaink nevei nem kerülnek összeütközésbe más csomagok neveivel, mert a csomagok önálló névtérreket hoznak létre.
- A típusaink korlátlanul láthatják egymást, de egyéb típusok csak korlátozottan férhetnek a típusokhoz.

17.1. Csomag létrehozása

A csomag létrehozásához mindössze bele kell tenni egy típust (osztály, interfész). A csomag megnevezést a forrásállomány elején, a típusdefiniciók előtt kell megtenni.

Ez után a *graphics* csomag összes forráskódja elején ugyanezt a csomagmegjelölést kell alkalmaznunk, hogy közös csomagba kerüljenek:

```
package graphics;

public class Circle extends Graphic implements Dragable {
    . . .
}

package graphics;

public class Rectangle extends Graphic implements Dragable {
    . . .
}
```

Ha nem alkalmazunk csomagmegjelölést, akkor az osztály(ok) egy úgynevezett alapértelmezett név nélküli csomagba kerülnek.

17.2. Egy csomag elnevezése

Az egész világon írnak a programozók a Java programnyelvhez osztályokat, interfészeket, kulcsszavakat és megjegyzéseket, és valószínűleg két programozó ugyanazt a nevet két különböző osztállyal fogja használni. Valójában, az előző példa esetén, amikor definiálunk egy *Rectangle* osztályt, akkor a *Rectangle* osztály már benne van a *java.awt* csomagban. A fordító mégis engedélyezi két osztálynak ugyanazt a nevet. Miért? Mert azok különböző csomagokban vannak, és mindegyik osztálynak a teljes neve magába foglalja a csomag nevét. Tehát a *graphics* csomagban lévő *Rectangle* osztály teljes neve *graphics.Rectangle*, és a *java.awt* csomagban lévő *Rectangle* osztály teljes neve *java.awt.Rectangle*. Ez rendszerint csak akkor működik jól, hogyha két egymástól független programozó nem ugyanazt a nevet adja a csomagoknak. Mivel hárítható el ez a probléma? Megállapodással.

Megállapodás: Fordított *domain* (tartomány) nevet használunk a csomagok nevének ilyen módon: *com.company.package*. Nevütközés előfordulhat egyetlen cégen belül is, amit a cégnek le kell kezelni egy belső megállapodással. Lehet, hogy emiatt tartalmazni fog tartomány vagy projekt neveket a társaság neve után, mint például *com.company-region.package*.

17.3. Csomag tagok használata

Csak a publikus csomag tagok érhetők el a csomagon kívül. Ahhoz hogy használjunk egy publikus csomag tagot a csomagján kívülről, a következők valamelyikét kell tennünk (vagy akár többet):

- A teljes (vagy más néven minősített) néven keresztül kell hivatkoznunk rá
- Importáljuk a csomag tagot
- Importáljuk a tag teljes csomagját

Mindegyiket különböző szituációkban alkalmazhatjuk, amelyeket a következő részekben tisztázunk.

Név szerinti hivatkozás egy csomag tagra

Eddig a példákban a típusokra az egyszerű nevelükön keresztül hivatkoztunk, mint például *Rectangle*. Akkor használhatjuk egy csomag tagjának az egyszerű nevét, ha az osztály ugyanabban a csomagban van, mint a tag, vagy ha a tag importálva van.

Ha megpróbálunk használni egy tagot egy másik csomagtól, és a csomagot nem importáltuk, akkor a tag teljes nevét használunk kell. A *Rectangle* osztály teljes neve:

```
| graphics.Rectangle
```

A következőképpen használhatjuk a minősített nevet, hogy létrehozzuk a *graphics.Rectangle* osztály egy példányát:

```
| graphics.Rectangle myRect = new graphics.Rectangle();
```

Ha minősített neveket használunk, valószínűleg bosszantó lesz, hogy újra és újra be kell gépelni a *graphics.Rectangle*-t. Ezen kívül rendezetlen és nehezen olvasható programot kapunk. Ilyen esetekben inkább importáljuk a tagot.

Egy csomag tag importálása

Ahhoz hogy importálhassuk a megadott tagot az aktuális fájlban, a fájl elején ki kell adni az *import* utasítást az osztály vagy interfész definiálása előtt, de a *package* utasítás után, ha van olyan. Úgy tudjuk importálni a *Circle* osztályt a *graphics* csomagtól, hogy:

```
| import graphics.Circle;
```

Most már tudunk hivatkozni a *Circle* osztályra az egyszerű nevével:

```
| Circle myCircle = new Circle();
```

Ez a szemlélet akkor működik jól, ha csak néhány tagot használunk a *graphics* csomagtól. De ha sok típusát használjuk egy csomagnak, akkor inkább importáljuk az egész csomagot.

Egy teljes csomag importálása

Ahhoz hogy egy csomagnak az összes típusát importáljuk, az *import* utasítást a csillag (*) helyettesítő karakterrel kell használnunk:

```
| import graphics.*;
```

Most már hivatkozhatunk bármelyik osztályra vagy interfészre a *graphics* csomagtól az egyszerű rövid nevével:

```
| Circle myCircle = new Circle();  
| Rectangle myRectangle = new Rectangle();
```

Az *import* utasításban a csillag karakterrel az összes osztályt megadjuk a csomagnak. Nem használhatunk olyan megjelölést, amely egy csomagban egy osztály részhalma-zára utal. Például helytelen megjelölés az, hogy a *graphics* csomagtól az 'A' betűvel kezdődő összes osztály importáljuk:

```
| import graphics.A*;
```

Ez az utasítás fordítási hibát generál. Az *import* utasítással, egy csomagnak egyetlen tag-ját vagy egész csomagot importálhatunk.

Ugyanígy nem megengedett, hogy egyszerre több csomagot importáljunk, pl.:

```
| import graphics.*.*;
```

Megjegyzés: Ha kevesebb tagot akarunk importálni, akkor megengedett, hogy csak egy osztályt, és annak a belső osztályait importáljuk. Például, ha a `graphics.Rectangle` osztálynak a belső osztályait akarjuk használni, mint `Rectangle.DoubleWide` és `Rectangle.Square`, a következőképpen importálhatjuk:

```
import graphics.Rectangle.*;
```

Könnyítésül a Java fordító automatikusan importál három teljes csomagot:

- A névtelen csomagot (ha nem hozunk létre csomagot, vagyis nem használjuk a `package` utasítást)
- Az alapértelmezés szerinti aktuális csomagot (ha létrehozunk csomagot)
- A `java.lang` csomagot

Megjegyzés: A csomagok nem hierarchikusak. Ha például importáljuk a `java.util.*`-ot, nem hivatkozhatunk a `Pattern` osztályra. Minden esetben úgy kell hivatkoznunk, hogy `java.util.regex.Pattern`, vagy ha importáljuk a `java.util.regex.*`-ot, akkor csak egyszerűen `Pattern`.

Névütközés megszüntetése

Ha egy tag ugyanazon a néven megtalálható egy másik csomagban, és mindkét csomag importálva van, akkor a tagra teljes nevével kell hivatkoznunk. Például a `Rectangle` osztály definiálva van a `graphics` csomagban, és a `java.awt` csomagban is. Ha a `graphics` és `java.awt` is importálva van, akkor a következő nem egyértelmű:

```
Rectangle rect;
```

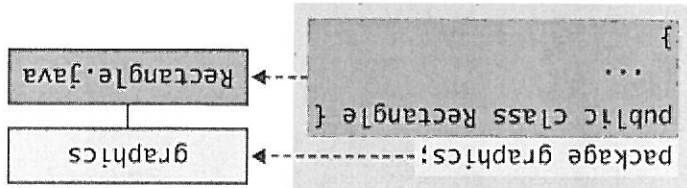
Az ilyen esetekben a minősített nevet kell használnunk, hogy pontosan meg tudjuk adni, melyik `Rectangle` osztályt akarjuk használni:

```
graphics.Rectangle rect;
```

17.4. Forrás és osztály fájlok menedzselése

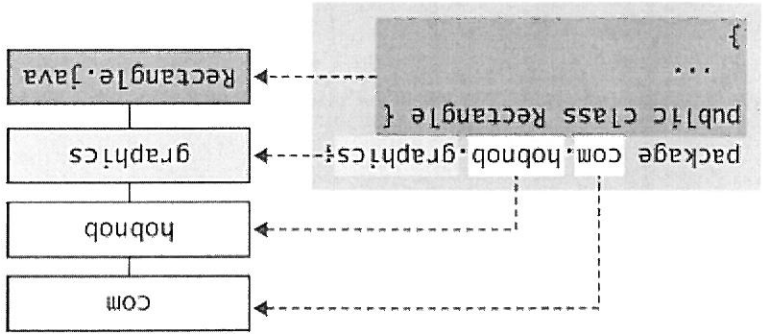
A Java környezet sok megvalósításánál számít a hierarchikus fájltrendszer a forrás és osztály állományok menedzseléséhez, bár maga a Java nem igényli ezt. A stratégia a következő.

A forráskód egy osztályt, interfészt vagy felsorolt típust tartalmaz, amelynek neve a típus egyszerű neve és a kiterjesztése: `.java`. A forrásfájl egy könyvtárba tesszük, amelynek a neve kifejezi a csomag nevét, amelyikhez a típus tartozik. Például, a forráskódban lévő `Rectangle` osztály adja a fájl nevét, `Rectangle.java`, és a fájl egy `graphics` nevű könyvtárban lesz. A `graphics` könyvtár bárhol lehet a fájl rendszeren. A lenti ábra mutatja, hogyan működik.

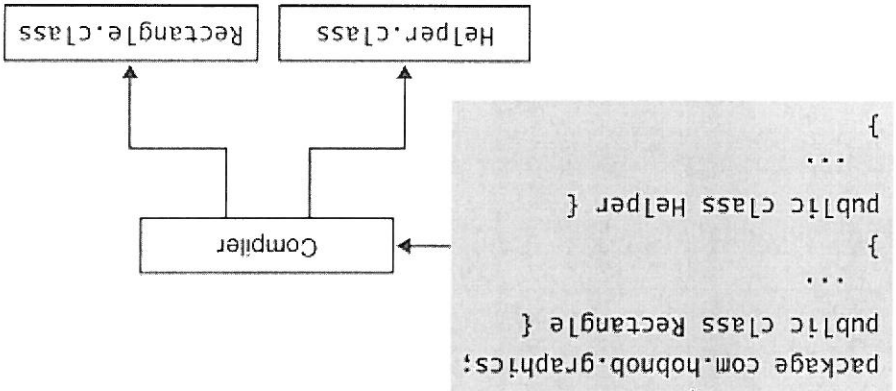


A csomag tag teljes neve és a fájl elérési útja hasonló:
 Osztály név `graphics.Rectangle`
 Fájl elérési út `graphics\Rectangle.java`

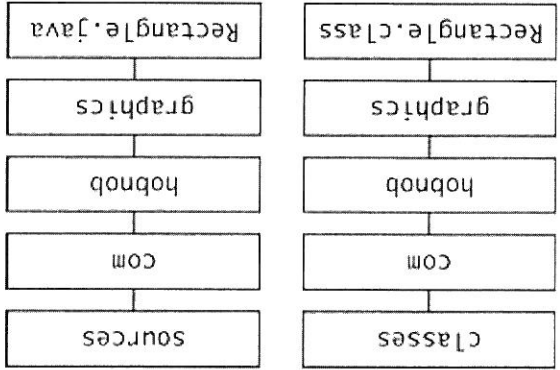
Emlékezzünk vissza, hogy a megállapodás értelmében a cégek fordított internet domén nevet használnak a csomagok elnevezésére. A példában a cég domén neve *hobnob.com*, ezért mindenképp csomagnak a nevet *com.hobnob* fogja megjelölni. A csomag névnek mindegyik összetevője meggyezzik egy-egy alkönyvtárral. Így ha Hobnob-nak van egy *graphics* csomagja, amely tartalmazza a *Rectangle.java* forrásfájljt, akkor ezek alkönyvtárak sorozatában tárolódnak:



Mikor lefordítjuk a forrásállományt, a fordító létrehoz különböző kimeneti állományt minden egyes definiált osztálynak és interfésznek. A kimeneti állománynak az alap neve az osztály vagy interfész neve és a kiterjesztése: *.class*, ahogyan a következő ábra is mutatja:



A *.class* fájlnak ugyanúgy, mint egy *.java* fájl, is kell a csomagokat kifejező könyvtár szerkezet. Azonban nem lehet ugyanabban a könyvtárban, mint a forrás. Külön-külön könyvtárba kell rendezni a forrást és az osztályt:



Ha ez alapján csináljuk, oda tudjuk adni az osztály könyvtárakat más programozóknak anélkül, hogy megmutatnánk a forrásunkat. Ekkor használhatjuk a Java fordítónak a *-d* opcióját az osztály fájlok megadásához, ehhez hasonlóan:

```
| java -d classes sources\com\hobnob\graphics\Rectangle.java
```


A forrás és osztály állományok kezeléséhez erre a módszerre van szükségünk, így a fordító és a Java virtuális gép (JVM) megtalálja az összes típust, amit a programunk használ. Mikor a fordító összeakad egy új osztállyal a program fordításakor, meg fogja találni az osztályt, úgy hogy feloldja a neveket, típusvizsgálatot hajt végre és így tovább. Hasonlóan, amikor a JVM összeakad egy új osztállyal program futáskor, meg fogja találni az osztályhoz tartozó metódusokat. A fordító és a JVM az osztályokat a *class path*-ban listázzott könyvtárakban és JAR fájlokban keresi.

Definíció: A *class path*, könyvtárak vagy JAR fájlok egy rendezett listája, amelyben *class* fájlokat kereshetünk. Fizikailag ez egy CLASSPATH nevű környezeti változót jelent.

Mind egyik könyvtár listázva van a *class path*-ban egy legfelső szinten lévő könyvtárban, amelyben csomag könyvtárak láthatók. A legfelső szintű könyvtártól a fordító és a JVM összeállítja az elérési útnak a maradvékát, az alaphoz a csomagot és az osztály névvel az osztályt. A fordító és a JVM is összeállítja az elérési út nevet a *class* fájlhoz a csomag teljes nevével.

Alapértelmezésként a fordító és a JVM az aktuális könyvtárban és a JAR fájlban keres, amely magába foglalja a Java környezeti *class* fájlokat. Más szóval, az aktuális könyvtár és a Java környezeti *class* fájlok automatikusan benne vannak a *class path*-ban. Legtöbbször az osztályokat megtalálják ezen a két helyen. Így nem kell aggódnunk az osztályaink elérési útját.

17.5. Ellenőrző kérdések

- Mi a hasonlóság az alkönyvtárak és a csomagok között?
- Hogyan kell egy Java osztályt egy adott csomagba tenni?
- Mit jelent a *package* direktíva és mi a parancsere?
- Hogyan lehet egy csomagban lévő osztályt elérni?
- Mit egyeztet az import direktíva?
- Hogyan leli fel a virtuális gép a *class* fájlokat?
- Mi a CLASSPATH környezeti változó?

Igaz vagy hamis? Indokolja!

- Különböző csomagokban lehetnek azonos nevű típusok.
- Csomagot létrehozni a *create package* utasítással lehet.
- Egy csomag típusait a *uses* kulcsszó segítségével lehet elérhetővé tenni.
- Ha egy forrásállományban nem nevezünk meg tartalmazó csomagot, akkor a forrásállományban szereplő osztályok semelyik csomagnak nem lesznek részei.
- Ha két csomagban van azonos nevű típus, akkor nem importálhatjuk egyezsere mindkettőt.
- A *package* utasítás csak a forrásállomány legelső utasítása lehet.
- A csomag kizárólag logikai szintű csoportosítás.

- Egy csomag csak publikus osztályokat tartalmazhat.
 - Egy fordítási egységben csak egy osztályt lehet deklarálni.
 - Elnevezési konvenció szerint a csomag neve nagybetűvel kezdődik, a többi kicsi.
 - Egy fordítási egység kötelezően tartalmaz *package* deklarációt.
 - Egy fordítási egység kötelezően tartalmaz *import* deklarációt.
 - Ha az osztály deklarációjánál nem adunk meg láthatóságot, akkor arra a csomag más osztályaiból lehet hivatkozni.
- Melyik a helyes sorrend egy forrásállomány esetén?**
- *package, import, class*
 - *package, import, class, import*
 - *import, package, class*
 - *class, import, package*
 - *package, import, class*

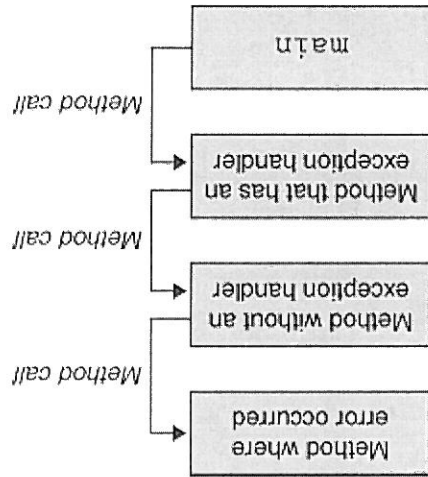
18. Kivételkezelés

A *kivétel* a „kivételes esemény” kifejezés rövidítése.

Definíció: A kivétel egy olyan esemény, amely a program végrehajtásakor keletkezik, megszakítva az utasítások végrehajtásának normális folyamatát.

Ha egy módszerben hiba keletkezik, a módszer egy objektumot hoz létre, melyet átad a futtatási környezetnek. Az objektum – melyet kivétel objektumnak neveznek – tartalmazza az információt a hibáról, annak típusáról és a program állapotáról, amikor a hiba létrejött. Kivétel objektum létrehozását és futtatási rendszer által történő kezelését kivétel dobásnak hívják.

Miután egy módszer eldob egy kivételt, a futtató környezet megpróbál a kezelésére találni valamit. A lehetséges dolgok, melyek a kivételt kezelik a meghívott módszerok rendezett listája abban a módszerben, ahol a hiba keletkezett. A módszerok listáját hívási vezetéknek nevezzük.



A futtató rendszer átkutatója a hívási vermet olyan módszer után, mely tartalmaz kivételkezelésre alkalmas kódblokkot. Ezt a blokkot kivételkezelőnek nevezzük. A keresés abban a módszerban kezdődik, ahol a hiba generálódott, majd a hívási verem módszerainak fordított sorrendjében folytatódik. Mikor egy megfelelő kezelőt talál, a futtató rendszer, átadja a kivételt a kezelőnek. Egy kivételkezelő megfelelő, ha az eldobott kivétel objektum típusa megegyezik azzal a típussal, melyet a kezelő kezelni tud. A kivételkezelő kiválasztását úgy is nevezzük, hogy elkapni a kivételt. Ha a futtatókörnyezet a módszerok átkutatása után sem talál megfelelő kivételkezelőt, mint ahogy a következő ábra mutatja, a futtató rendszer (és ez által a program) leáll.

Hibák kezelésére használt kivételeknek van néhány előnye a hagyományos hibakezelési technikákkal szemben. (Később visszatérünk a témára.)

18.1. Kivételek elkapása vagy továbbengedése

A Java futtató rendszer megköveteli, hogy a módszer elkapja, vagy felsorolja az összes ellenőrzött kivételt, melyet a módszer eldobhat (másként fogalmazva a kivételt tovább- enged). Először is tekintünk át néhány kifejezést.

Elkapás (catch)

A módszer elkaphat egy kivételt, ha rendelkezik ilyen típusú kivételkezelőjével.

Továbbengedés (throws záradék)

A módszer a deklarációjában írja le, hogy milyen kivételeket dobhat.

Ellenőrzött kivételek

Kétfajta kivétellel létezik: futási időben keletkezett kivétel és nem futási időben keletkezett kivétel. Futási idejű kivétel a Java futtatórendszerében keletkezik: aritmetikus kivételek (például nullával való osztás), referenciával kapcsolatos kivételek (mint például egy ob- jektum tagjaihoz való hozzáférés *null* hivatkozással) és indexeléssel kapcsolatos kivéte- lek (mint például egy tömb eleméhez való hozzáférés olyan indexszel, mely túl nagy vagy túl kicsi). Egy módszernek nem kötelező előírnia futási idejű kivételeket, de aján- lott.

Nem futási időben keletkezett kivételek olyan kivételek, melyek a Java futási rendszeren kívül keletkeznek. Például: kivételek, melyek I/O során keletkeznek. A fordító biztosítja, hogy a nem futási időben keletkezett kivételeket elkapják, vagy továbbengedjék; ezért ezeket ellenőrzött kivételeknek is nevezzük.

Sok programozó inkább futási időben keletkezett kivételeket használ az ellenőrzött kivé- telekkel szemben, hogy ne keljen elkapniuk vagy továbbengedniük őket. Ez általában nem ajánlott.

Kivételek, melyeket a módszer eldobhat

- Minden a módszer által közvetlenül eldobott kivétel
- Minden közvetetten eldobott kivétel másikkal hívásával, mely kivételt dob.

Kivételek elkapása és kezelése

Ez a fejezet bemutatja, hogyan kell használni a kivételkezelő három komponensét – a *try*, *catch*, és *finally* blokkokat – egy kivételkezelő megírásához. A fejezet utolsó része végigmegy egy példán, bemutatva, mi történik különböző esetekben.

A következő példa a *ListOfNumbers* osztályt definiálja és implementálja. A *ListOfNumbers* konstruktor egy *Vector* objektumot hoz létre, mely *int Integer* elemet tartalmaz a nullától kilencig terjedő index értékekhez. A *ListOfNumbers* osztály egy *writeList* metódust is definiál, mely a számok listáját kiírja egy szövegfájlba, melynek neve *OutFile.txt*. A példa program a *java.io* csomagban definiált kimeneti osztályokat használja, melyekről részletesebben később lesz szó.

```
import java.io.*;
import java.util.*;

public class ListOfNumbers {
    private Vector vector;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        vector = new Vector(SIZE);
        for (int i = 0; i < SIZE; i++) {
            vector.addElement(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = new PrintWriter(
            new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " +
                vector.elementAt(i));
        }
        out.close();
    }
}
```

Az első vastag betűs sor egy konstruktor hívása. A konstruktor egy kimeneti folyamatot inicializál. Ha a fájl nem lehet megnyitni, a konstruktor *IOException*-t dob. A második vastag betűs sor a *Vector* osztály *addElement* metódusát hívja meg, mely egy *ArrayIndexOutOfBoundsException*-t dob, ha az paramétereinek értéke túl kicsi (kisebb, mint nulla), vagy túl nagy (nagyobb, mint a változók száma, melyeket a *Vector* tartalmaz). Ha megpróbáljuk lefordítani a *ListOfNumbers* osztályt, a fordító egy hibaüzenetet ír ki a *FileNotFoundException* által dobott kivételről, de nem fog hibaüzenetet megjeleníteni az *elementAt* által dobott hibáról. Ennek oka, hogy a konstruktor által dobott kivétel, *IOException*, ellenőrzött kivétel, míg a másik, *ArrayIndexOutOfBoundsException*, egy futási időben keletkezett kivétel. A Java programozási nyelv csak az ellenőrzött kivételek kezelését követeli meg, tehát a felhasználó csak egy hibaüzenetet fog kapni.

Most, hogy megismerkedünk a *ListOfNumbers* osztállyal és az abban eldobott kivételekkel, készen állunk egy kivételkezelő megírására, mely ezeket a hibákat elkapja, és kezelni tudja.

A try blokk

Egy kivételkezelő elköszítésének első lépése, hogy elhatározzuk a kódot, ami hibát dobhat a try blokkban. A try blokk általában így néz ki:

```
try {
    code
}
catch and finally blocks ...
```

A példakódban található szegmens tartalmaz egy vagy több olyan sort, amely kivételt dobhat. (A *catch* és a *finally* blokkokról részletes magyarázatot a következő részben találhatunk.)

Ahhoz, hogy egy kivételkezelőt készítsünk a *ListOfNumbers* osztály *writeList* módszerához, a *writeList* módszer kivétel-dobó részét el kell határolnunk a try blokkal. Ezt többféleképpen tehetjük meg. A programkód azon sorait, melyekről feltételezzük, hogy kivétel-dobhatnak, try blokkba tesszük, és mindannyiuknál gondoskodunk a kivételek kezeléséről. Vagy pedig betehetjük az összes *writeList* kódot egy egyszerű try blokkba, és ehhez hozzákapcsolhatunk többféle kivételkezelőt. A következőkben láthatjuk, hogy az egész módszerára használjuk a try blokkot, mivel a szóban forgó kód nagyon rövid:

```
...
private Vector vector;
private static final int SIZE = 10;
...
PrintWriter out = null;
try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " +
            vector.elementAt(i));
    }
} // catch and finally statements ...
```

Amennyiben a kivétel bekövetkezik a try blokkon belül, a kivétel lekezelésre kerül a kivételkezelő által. Ahhoz, hogy a kivételkezelést hozzá tudjuk kapcsolni a try blokkhoz, utólag használunk kell *catch* blokkot is. A következő rész megmutatja, hogyan is kell ezt használnunk.

Megjegyzés: A kivételkezeléssel ismerkedők esetén gyakori hiba, hogy csak a kivételt eldobó függvényhívást tesszük a try blokkba. Erdemes ezen a példán átgondolni, hogy ha nem sikerülne az állomány megnyitása, akkor semmi értelme nem lenne a *for* ciklus lefutásának. A fenti megoldásnál ez nem is fog bekövetkezni, hiszen a kivétel létrejöttekor a vezérlés a teljes try blokkból kilép.

A catch blokk

A try blokkhoz hozzáállíthatjuk a kivételkezelést, amennyiben egy vagy több *catch* blokkot használunk közvetlenül a try blokk után. Semmilyen programkód nem lehet a try blokk vége és az első *catch* blokk között!

```
try {
    ...
}
catch (ExceptionType name) {
    ...
}
catch (ExceptionType name) {
    ...
}
```

Minden *catch* blokk egy kivételkezelő, és azt a típusú kivételt kezeli, amilyet a paraméter tartalmaz. A paraméter típusa (*ExceptionType*) deklarálja a kivétel típusát, amit a kezelő lekezel.

A *catch* blokk tartalmazza azt a programkódot, amely végrehajtásra kerül, amikor a kivételkezelőt meghívjuk. A futtatórendszer meghívja azt a kivételkezelőt, amelyik esetén az *ExceptionType* megfelel a dobott kivétel típusának.

Itt látható két kivételkezelő a *writeList* metódushoz – az ellenőrzött kivételek két típusához, melyeket a *try* blokk dobhat:

```
try {
    ...
    catch (FileNotFoundException e) {
        System.err.println("FileNotFoundException: "+e.getMessage());
    }
    catch (IOException e) {
        System.err.println("Caught IOException: "+e.getMessage());
    }
}
```

Mindkét kezelő egy hibaüzenetet ír ki. A második kezelő semmi más nem hajt végre. Bármilyen *IOException* (I/O kivétel) elkapása esetén (amit az első kezelő nem kapott el), megengedi a programnak, hogy folytassa a futtatást.

Az első kezelő a kirandó szövegek összeillesztése során egy saját definiált kivételt dob. Ebben a példában a *FileNotFoundException* kivétel létrejöttékor egy saját definiált kivételt hoz létre, aminek *SampleException* a neve, és ezt dobja a program.

A kivételkezelők többet is tudnak, mint hogy hibaüzeneteket írnak ki, vagy pedig megállítják a program futását. Hibajavításra is képesek, amint a felhasználó hoz egy rossz döntést, vagy pedig a hibák túlnőnek a legmagasabb szintű kezelőn láncolt kivételeket használva.

A *finally* blokk

A kivételkezelő beállításának utolsó lépése, hogy rendet tegyünk magunk után, mielőtt átadjuk az irányítást a program különböző részeinek. Ezt a takarító programkódot a *finally* blokkba kell beírunk. A *finally* blokk tetszőlegesen használható. Olyan mechanizmust nyújt, ami kiküszöböli azokat a figyelmenytelenségeket, amik a *try* blokkban történnek. A *finally* blokkot például arra használhatjuk, hogy bezárjuk a fájlokat, amelyekre se a hiba nélküli futás, se a hiba dobása esetén nem szükségességek már.

A *writeList* metódus *try* blokkja (amivel eddig dolgoztunk) megnyitja a *PrintWriter*-t. A *writeList* metódusból való kilépés előtt programnak be kellene zárnia ezt a folyamatot. Ez felvet egy némiképp komplikált problémát, mivel *writeList* metódus *try* blokkja a következő három lehetőség közül csak egyfelelekeppen tud kilépni:

- A *new FileWriter* hibát jelez és *IOException*-t dob.
- A *vector.elementAt(i)* hibát jelez és *ArrayIndexOutOfBoundsException*-t dob.
- Minden sikerül és a *try* blokk zökkenőmentesen kilép.

A futtatórendszer mindig végrehajtja azokat az utasításokat, amelyek a *finally* blokkban vannak, függetlenül attól, hogy történt-e kivétel dobása, vagy nem. Így ez a legmegfelelőbb hely, hogy kitakarítsunk magunk után.

A következő *finally* blokk a *writeList* metódust takarítja ki, és bezárja a *PrintWriter*-t.


```

finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}

```

A *writeList* példában, a *finally* blokkba való beavatkozás nélkül tudunk gondoskodni a kitakarítástól. Például, a *PrintWriter* bezárását végző programkódot a *try* blokk végéhez tudjuk illeszteni, továbbá az *ArrayIndexOutOfBoundsException* kivétel-kezelőhöz:

```

try {
    ...
    out.close();
} catch (FileNotFoundException e) {
    out.close();
    System.err.println(
        "Caught: FileNotFoundException: " + e.getMessage());
    throw new RuntimeException(e);
} catch (IOException e) {
    System.err.println("Caught IOException: " +
        e.getMessage());
}

```

Azonban ez mégis lemásolja a kódot, így amennyiben később módosítjuk a programkódot, nehéz lesz olvasni benne, és a hibák kiterjedhetnek. Például, ha a *try* blokkhoz egy olyan programkóddal bővítjük, ami egy új típusú kivételt dobhat, emlékeznünk kell arra, hogy bezárjuk a *PrintWriter*-t az új kivétel-kezelőben.

Az összeillesztés

Az előző részekben volt arról szó, hogyan építsünk fel a *ListOfNumbers* osztályban *try*, *catch* és *finally* blokkokat a *writeList* metódus számára. A következőkben a forráskódot nézzük meg, vizsgálva a végeredményt.

Mindent egybevéve, a *writeList* metódus a következőképp néz ki:

```

public void writeList() {
    PrintWriter out = null;

```

```

try {
    System.out.println("Entering try statement");
    out =
        new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++)
        out.println("Value at: " + i + " = "
            + vector.elementAt(i));
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught " +
            "ArrayIndexOutOfBoundsException: " +
            e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " +
            e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}

```

Ahogy azt előzőekben említettük, a metódusban a *try* szerkezetnek három lehetséges ki-
menete lehet:

- A kódunk hibás és kivételt dob. Ez lehet egy *IOException* kivétel, amit a *FileWriter* okoz vagy egy *ArrayIndexOutOfBoundsException* kivétel egy rosszul megadott érték esetén a *for* ciklusban, esetleg egy *RuntimeException* kivétel bármilyen futási hiba esetén.
- Minden rendben, a várt kimenetet kapjuk.

Nézzük meg, mi történik a *writeList* metódusban ezen esetekben:

Kivétel történik

Számos oka lehet annak, hogy a *FileWriter* hibás működést fog eredményezni.

Ökozhatja a konstruktor (ha nem tudjuk létrehozni a fájlt, vagy nem tudunk beírni).

Amikor a *FileWriter* az *IOException* kivételt dobja, a program azonnal leállítja a *try* blokk végrehajtását. Ezek után a rendszer a metódus elejétől indul, és meghívja a megfelelő kivételkezelőt. Habár a *FileWriter* konstruktorának nincs megfelelő kivételkezelője, a rendszer megnezi a *writeList* metódust.

A *writeList* metódusnak két kivételkezelője van: egyik az *IOException*, másik, pedig az *ArrayIndexOutOfBoundsException*.

A rendszer megnezi a *writeList* kivételkezelőt, de csak a *try* blokk után. Az első kivételkezelő nem fogalkozik a konkrét kivétel típussal, ezért a rendszer automatikusan a második kivételkezelőt használja (*IOException*). Ez be tudja azonosítani a típusát, ennek köszönhetően a rendszer már megtalálja a megfelelő kivételkezelőt, így a *catch* blokk végrehajtható.

Miután a kivételkezelő lefutott, a rendszer a *finally* blokkra ugrik, melynek futása elindul a kivételeket figyelmen kívül hagyva. Miután a *finally* blokk lefutott a rendszer normális útemben fut tovább.

Az *IOException* által dobott kimenet:

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

A try blokk normális működése

Minden, ami a *try* blokkon belül van, sikeresen lefut és megfelelő kimenetet ad eredményeképpen a *PrintWriter* megnyílik, majd amikor a rendszer elér a *finally* blokkot, bezárja. Miután a *finally* blokk lefutott, a rendszer normális útemben fut tovább.

A kivételek nélküli kimenet:

```
Entering try statement
Closing PrintWriter
```

Metódusok által dobott kivételek

Az előzőekben láthattuk, hogyan írhatunk kivételkezelőt a *ListOfNumbers* osztályban a *writeList* módszernek. Olykor ez megfelelő a működést illetően, de vannak esetek, amikor jobb a veremkezelőt használni. Például, ha a *ListOfNumbers* osztályt egy másik osztály csomagjaként hozunk létre, ebben az esetben jobban tesszük, ha más módon kezeljük a kivételt.

Bizonyos esetekben szükség lehet az eredeti *writeList* módszer módosítására, melynek hatására már a kívánt működést kaphatjuk. Az eredeti *writeList* módszer, ami nem fordul le:

```
public void writeList() {
    PrintWriter out =
        new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " +
            victor.elementAt(i));
    }
    out.close();
}
```

A *writeList* módszer deklarációjában a *throws* kulcsszó hozzáadásával két kivételt adunk meg. A *throws* kulcsszó egy vesszővel elválasztott lista követ a kivételosztályok-
kal.

A *throws* a módszer neve és a paraméter lista után áll, utána következik a kapcsos záró-
jelben a definíció:

```
public void writeList() throws IOException,
    ArrayIndexOutOfBoundsException {
```

Az *ArrayIndexOutOfBoundsException* futásidejű kivétel, nem kötelező lekezelni, így
elegendő az alábbi forma használata:

```
| public void writeList() throws IOException {
```

18.2. Kivételek dobása

Bármelyik forráskód tud kivételt dobni, legyen az saját kódunk, egy csomagból származó, más által írt kód, bármi is annak a kimenete, a *throws* minden esetben jelen van.

A Java platform számos osztályt nyújt a kivételek kezelésére, melyek egyig leszár-
mazottjai a *Throwable* osztálynak. Ezek az osztályok lehetővé teszik a programok szá-
mára, hogy megkülönböztessék az eltérő típusú kivételeket, melyek a program futása
közben keletkeznek.

Saját magunk is létrehozhatunk speciális osztályokat a leendő problémák reprezentáció-
jával. Rejlesztőként saját magunknak kell létrehozunk az osztályt és elvégezni a szükse-
ges beállításokat.

18.2.1 A *throw* használata

A kivétel dobásánál minden módszer a *throw* kulcsszót használja, melynek a következő
formai követelménynek kell, hogy megfeleljen:

```
throw someThrowableObject;
```

Az alábbi *pop* módszer egy osztály által létrehozott közös veremobjektumból jön létre. A
módszus eltávolítja a verem felső rekeszt és az objektummal tér vissza:

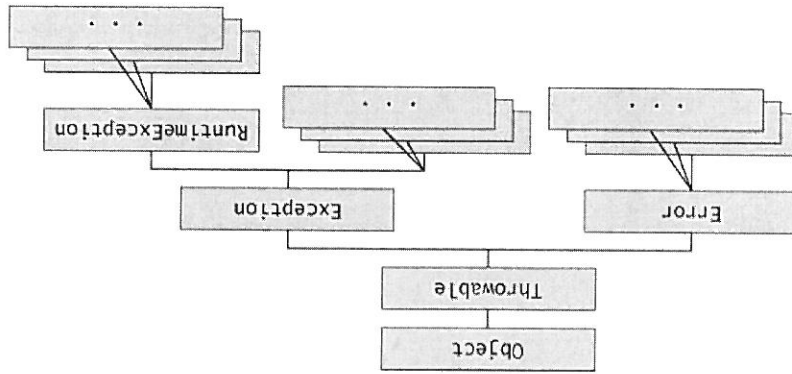
```
public Object pop() throws EmptyStackException {
    Object obj;
    if (size == 0) {
        throw new EmptyStackException();
    }
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

A *pop* módszer megnevezi, van-e valami a veremben. Ha a verem üres (mérete 0), akkor a
pop által létrejön egy új *EmptyStackException* objektumot (a *java.util* tagja) és eldobja
azt. Ezek az objektumok a *java.lang.Throwable* osztály leszármazottjai kell, hogy legye-
nek, a fejezet későbbi része a kivétel osztályok létrehozásával foglalkozik.

A *pop* módszer deklarációja magában foglalja a *throws* kulcsszót. Az *EmptyStackEx-
ception* kivételt a *pop* módszer nem kapja el, így a módszernek a *throws* kulcsszót kell,
hogy használja a deklarációhoz.

18.3. Eldobható osztályok és leszármazottjai

Az ábra a *Throwable* osztály és annak legjelentősebb leszármaztatott osztályait (osztályhi-
erarchiáját) reprezentálja. Amint láthatjuk, a *Throwable* osztálynak kettő direkt módon
származtatott utódja van: *Error* és *Exception*.



Error osztály

Ha a művelet valamilyen okból kifolyólag nem hajtható végre, sikertelen a végrehajtás, a Java virtuális gép *Error*-t fog dobni. Egyszerű programok nem kapnak el, illetve nem dobnak *Error*-t.

Exception osztály

A legtöbb program elkap és dob objektumokat, melyek az *Exception* osztályból származnak. A legtöbb program elkap és dob kivételeket (jelzi a hibát). Az *Exception* osztálynak számos leszármazottja definiált a Java Platformban, ezek a leszármazottak jelzik a különböző típusú kivételeket. Például *IllegalAccessException* jelzi, ha egy különös módszer nem található, a *NegativeArraySizeException* pedig, ha egy tömb méretét helytelenül adjuk meg (negatív szám). A *RuntimeException* kivételek futás közben generálódnak a Java Virtuális gép által. Ennek egyik példája a *NullPointerException*, mely akkor jön létre, ha egy módszer megpróbál hozzáférni *null* hivatkozással valamely objektumhoz.

18.4. Láncolt kivételek

A kivételeknek sok esetben egy másik kivétel az oka. Ilyen esetben célszerű az eredeti (okozó) kivételt és az új kivételt együtt kezelni. Erre a problémára jelent megoldást a Java 1.4 verzió óta rendelkezésre álló kivételláncolás. Ennek lényege, hogy egy kivétel létrehozásakor becsomagoljuk az okozó kivételt az új kivétel objektumba.

Ennek érdekében két új módszer és két új konstruktor lett hozzáadva a *Throwable* ti-pushoz.

A következő új módszerek és konstruktorok támogatják a láncolt kivételeket a *Throwable* osztályban:

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

Az *initCause* és a *Throwable* konstruktorok *Throwable* paraméterre egy kivétel, amik okozták az aktuális kivételt. A *getCause* visszaad egy kivételt, ami okozta az aktuális kivételt, az *initCause* pedig visszaadja az aktuális kivételt.

A következő példa megmutatja, hogyan használjuk a láncolt kivételeket:

```
try {
    ...
} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

Ebben a példában, amikor az *IOException*-t kap el a program, akkor egy új *SampleException* kivétel jön létre az eredeti okkal egybekötve, és a kivételek lánc feloldódik egy következő, magasabb szintű kivételkezelőhöz.

Hívási verem információk kinyerése

Most feltételezzünk, hogy egy magasabb szintű kivételkezelő akarja a kimenetre írni a hívási verem tartalmát egyedi formátumban.

Definíció: a hívási verem információt szolgáltat az aktuális szál futási történetéről, az osztályok és a metódusok neveiről, amik akkor hívódnak meg, amikor a kivétel bekövetkezett. A hívási verem egy nagyon hasznos hibakereső eszköz.

A következő kód megmutatja, hogyan kell a `getStackTrace` metódust használni a kivétel objektumon:

```
catch (Exception cause) {
    StackTraceElement[] = cause.getStackTrace();
    for (int i = 0; i < elements.length; i++) {
        System.err.println(elements[i].getFileName() + " : "
            + elements[i].getLineNumber() + ">>"
            + elements[i].getMethodName() + "()"");
    }
}
```

API naplózás

A következő kódrészlet `catch` blokkjában naplózásra látunk példát. Barmennyire célszerűnek tűnik is az előző példa megközelítése, érdemes inkább a naplózás mechanizmusát alkalmazni a `java.util.logging` segítségével:

```
try {
    Handler handler = new FileHandler("LogFile.log");
    Logger.getLogger("").addHandler(handler);
    catch (IOException e) {
        Logger logger = Logger.getLogger("package");
        StackTraceElement[] = e.getStackTrace();
        for (int i = 0; i < elements.length; i++) {
            logger.log(Level.WARNING, elements[i].getMethodName());
        }
    }
}
```

18.5. Saját kivétel osztályok létrehozása

Amikor szükségessé válik egy kivétel dobása, az egyik lehetőség egy valaki által már megírtat használni – a Java Platform rengeteg kivétel osztállyal van ellátva – vagy magunk is tudunk írni a céljainknak megfelelőt.

Saját kivétel osztályt kell írunk, ha a következő kérdések bármelyikére igennel válaszolhatunk. Különböz valószerűleg egy valaki más által megírtat kell használnunk.

- Olyan típusú kivétel osztályra van-e szükség, ami nem áll rendelkezésre a Java Platformban?

- Fog segíteni a felhasználóknak, ha képesek lesznek megkülönböztetni a mi kivételünket a mások által megírt kivételosztályoktól?

- A kódunk több helyen fog-e kivételt dobni?

- Ha valaki más kivételt használjuk, a felhasználó hozzá fog-e tudni férni azokhoz a kivételéhez?

Egy hasonló kérdés: a csomagunknak függetlennek és önállóknak kell-e lennie? Tegyük fel, hogy írunk egy láncolt lista osztályt, amit szeretnénk megosztani nyílt forrási programként. A láncolt lista osztályunk többek között a következő metódusokat támogatja:

objectAt(int n)

Visszaadja a lista n -dik pozíciójában levő objektumot. Kivételt dob, ha a paraméter kisebb, mint 0, vagy nagyobb, mint az aktuális objektumok száma a listában.

firstObject()

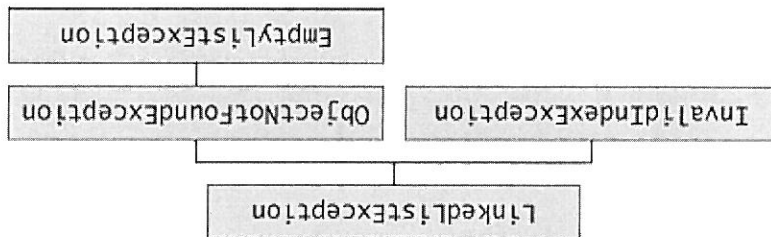
Visszaadja az első objektumot a listában. Kivételt dob, ha a lista nem tartalmaz objektumot.

indexOf(Object o)

Átnézi a listát egy speciális objektumért és visszaadja a pozícióját a listában. Kivételt dob, ha az objektum nincs a listában.

A láncolt lista osztály képes különböző kivételeket dobni, ugyanakkor lehetőséget ad arra is, hogy a felhasználó program el tudja kapni a láncolt lista által dobott összes kivételt egy kivételkezelővel. Ha a láncolt listánkat egy csomagban tervezzük megosztani, minden kapcsolódó kódnak egy közös csomagban kell lennie.

A következő ábra illusztrál egy lehetséges osztályhierarchiát a láncolt listák által dobható kivételekről:

**Szülőosztály választása**

Az *Exception* valamennyi leszármazott osztályát lehet a *LinkedListException* szülőosztálynak választani. Azonban könnyen látható, hogy ezek a leszármazott osztályok nem odaillők, mert vagy túl specializáltak, vagy nem függenek össze a *LinkedListException*-al. Ezért a *LinkedListException* szülő osztályának az *Exception*-nek kell lennie.

A legtöbb applet és alkalmazás, amit írunk, *Exception* objektumokat dob.

Figyelem: az olvasható kód érdekében egy jó gyakorlat, hogy hozzáfűzzük az *Exception* szót minden osztály nevéhez, amelyik közvetlenül vagy közvetve öröklődik az *Exception* osztályból.

18.6. Ellenőrző kérdések

- Mi a kivétel?
- Hogyan lehet kivételeket létrehozni?
- Kell-e deklarálni a metódusban létrehozott kivételeket, és ha igen, hogyan?
- Mire szolgál a *try-catch*?
- Hány *catch* ága lehet egy *try-catch* szerkezetnek?
- Hány *finally* ága lehet egy *try-catch-finally* szerkezetnek?
- Mi a teendő, ha metódusunk olyan másik metódust hív meg, ami kivételt generálhat?
- Mik a nem ellenőrzött kivételek, és hogyan lehet ellátni őket?

Korrektt a következő kód?

```
try {  
    ...  
} finally {  
}
```

Milyen típusú kivételeket fog elkapni a következő kezelő?

```
catch (Exception e) {  
    ...  
}
```

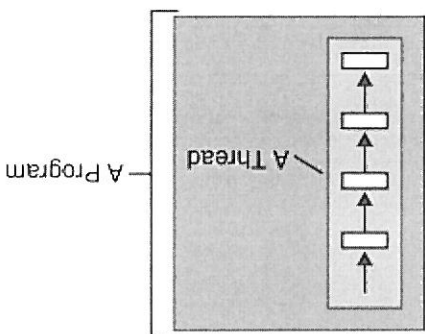
Igaz vagy hamis? Indokolja!

- A metódusban létrejövő kivételt kötelesek vagyunk a metódusban elkapni.
- Kivételt dobni a *throws* utasítással lehet.

19. Programszálak kezelése

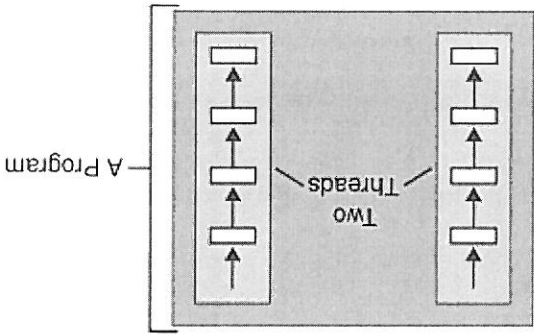
A programok jelentős része soros végrehajtású. Van kezdetük, egy végrehajtandó rész és egy befejezés. A program csak bizonyos időszakaszokban fut, hiszen a processzor már programokat is futtat.

A szál (*thread*) hasonló a soros programhoz. A szálnak is van kezdete, egy végrehajtandó része és egy befejezése. A szál is csak bizonyos időszakaszokban fut. De a szál maga nem egy program, a szál nem lehet önállóan futtatni. Inkább úgy tekinthetjük, hogy a program részeként fut. A következő ábra megmutatja a kapcsolatot.



Definíció: A szál egy egyedülálló irányító folyamat a programon belül.

Persze nem arra használjuk a szálakat, hogy csak egyet alkalmazzunk, inkább több szálát futtatva egy időben különböző feladatokat elvégezve egyetlen programban. A következő kép ezt illusztrálja.



A böngező programunk is egy példa a többszálú alkalmazásra. Egy általános böngészőben egy oldalon belül egyszerre történik egy kép, applet letöltése, mozgó animáció vagy hang lejátszása, az oldal nyomtatása a háttérben, miközben egy új oldalt tölt be vagy éppen három rendező algoritmus versenyt tekintetheti meg.

Néha a szálak könnyűsúlyú processzornak nevezik, mert egy teljes programon belül fut, annak lefoglalt erőforrásait és a futtató környezetet használja.

Mint irányító folyamat, a szálnak is kell saját erőforrásokkal rendelkeznie. Rendelkeznie kell egy végrehajtó veremmel és a programszámlálóval. Ebben a környezetbe fut a szál kódja. Nehol ezt a környezetet a szál fogalom szinonimájának nevezik.

A szál programozás nem egyszerű. Ha mégis szálakat kell használni, akkor érdemes a magas-szintű szál API-kat használni. Példaképpen, ha a programban egy feladatot többször kell elvégezni, érdemes a *java.util.Timer* osztályt alkalmazni. A *Timer* osztály időzített feladatokról is hasznos lehet. Erre hamarosan látnunk példát.

19.1. A *Timer* és a *TimerTask* osztály

Ebben a részben az időzítők használatát tárgyaljuk. A *Timer* osztály a *java.util* csomag része, a *TimerTask* osztály példányai ütemezését végzi. *Reminder.java* egy példa arra, hogy használjuk az időzítőket a késleltetett futtatáshoz:

```
import java.util.Timer;
import java.util.TimerTask;

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new ReminderTask(), seconds*1000);
    }

    class ReminderTask extends TimerTask {
        public void run() {
            System.out.println("Lejárt az idő!");
            timer.cancel(); //A timer szál megszüntetése
        }
    }

    public static void main(String args[]) {
        new Reminder(5);
        System.out.println("Munka ütemezve.");
    }
}
```

Indulás után ezt látjuk:

```
Munka ütemezve.
```

Öt másodper múlva ezt látjuk:

```
Lejárt az idő!
```

A program egy példa arra, hogyan kell példányosítani és ütemezni a szálakat:

- *TimerTask* leszármazott osztály példányosítása. A *run* metódus tartalmazza a futtatás során végrehajtandó kódot. A példába ezt az leszármazott osztály *ReminderTask*-ként neveztük el.

- Létrehozzuk a *Timer* osztály egy példányát.

- Létrehozzunk egy időzítő objektumot (*new ReminderTask()*).

- Beütemezzük az időzítőt. Ez a példa a *schedule* metódust használja, amelynek paraméterei az időzítő és késleltetés ezredmásodpercben (5000).

Egy másik lehetőség az ütemezésre, hogy az indítási időpontot adjuk meg. Példaképpen a következő kód 23:01-re ütemezi a végrehajtást:

```
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();
timer = new Timer();
timer.schedule(new ReminderTask(), time);
```

19.1.1 Időzített szálak leállítása

Alapesetben a program addig fut, amíg az időzítő szál is fut. Négy leállítási módszer közül választhatunk:

- A *cancel* metódust hívjuk meg. Ezt a program bármelyik pontján megtehetjük, mint a példa *run* metódusában.
- Az időzítő szálát démon szálá kell tenni a következőképpen: *new Timer(true)*. Ha a programban csak démon szálak maradnak, akkor a program kilép.
- Ha befejeződtek az ütemezések, el kell távolítani a *Timer* objektumhoz tartozó mutatókat. Ezzel a szál is megszűnik.
- A *System.exit* metódust hívjuk meg, amely leállítja a programot (benne a szálakat is). A *Reminder* példa az első módszerrel használja, a *cancel* metódust hívja meg a *run* metódusból. A démon szálként való létrehozás a példánkban nem működne, mert a programnak futni kell a szál lefutása után is.

19.1.2 Ismételt futtatás

Ebben a példában másodpercenként ismétli a kód futását:

```
public class AnnoyingBeep {
    Toolkit toolkit;
    Timer timer;

    public AnnoyingBeep() {
        toolkit = Toolkit.getDefaultToolkit();
        timer = new Timer();
        timer.schedule(new ReminderTask(),
            0, //kezdeti késleltetés
            1*1000); //ismétlési ráta
    }

    class ReminderTask extends TimerTask {
        int numWarningBeeps = 3;

        public void run() {
            if (numWarningBeeps > 0) {
                toolkit.beep();
                System.out.println("Slip!");
                numWarningBeeps--;
            } else {
                toolkit.beep();
                System.out.println("Idő lejárt!");
                timer.cancel(); // Nem szükséges
                // mert van System.exit is.
                System.exit(0); // Leállítja AWT szálakat
                // (és minden mást)
            }
        }
    }
}
```

Futtatás közben ez lesz a kimenet:

```
Munka utemezve.
s1p!
s1p!
s1p!
s1p!
Idő lejárt!
```

Az *AnnoyingBeep* program három paraméteres *schedule* metódust használ, hogy meghatározza a task másodpercenkénti indítását. A *Timer* metódus változatai:

- *schedule(TimerTask task, long késleltetés, long gyakoriság)*
- *schedule(TimerTask task, Date idő, long gyakoriság)*
- *scheduleAtFixedRate(TimerTask task, long késleltetés, long gyakoriság)*
- *scheduleAtFixedRate(TimerTask task, Date kezdetIdő, long gyakoriság)*

A *schedule* metódust akkor használjuk, ha a többszörösen futtatott task ismétlési ideje számít, a *scheduleAtFixedRate* metódust, ha az ismétlések időben kötődnek egy pontos időhöz. A példában is a *schedule* metódust alkalmazzuk, amitől 1 másodperces intervallumokban sípol a gép. Ha valamelyik sípszó késik, akkor a következők is késlekednek. Ha úgy döntünk, hogy a program 3 másodperc múlva kilép az első sípszó után – ami azt is eredményezheti, hogy a két sípszó kisebb időközzel szólal meg, ha késlekedés lép fel – akkor a *scheduleAtFixedRate* metódust használjuk.

19.2. Szálak példányosítása

Ha a fenti megközelítés nem alkalmas a feladat ellátására, akkor saját szál példányosításra lehet a megoldás. Ez a fejezet elmagyarázza, hogyan lehet a szál *run* metódusát egyé-nivé alakítani.

A *run* metódusban van, amit a szál ténylegesen csinál, ennek a kódja határozza meg a futást. A szál *run* metódussal mindent meg lehet tenni, amit a Java programnyelvben lehet írni. Pl.: primszámmal való számolás, rendezés, animáció megjelenítés.

A *Thread* osztály példányosításával egy új szál jön létre, ami alaptól nem csinál semmit, de lehetőség van a leszámrazott osztályban tartalommal megtölteni.

19.2.1 *Thread* leszámrazott és a *run* felülírása

Az első lépés a szálak testreszabásánál, hogy *Thread* osztály leszámrazottját létrehozzuk, és az üres *run* metódusát felülírjuk, hogy csináljon is valamit. Nézzük meg a *SimpleThread* osztályt, amely ezt teszi:

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
}
```

```

public void run() {
    for (int i = 0; i < 10; i++) {
        System.out.println(i + " " + getName());
        try {
            sleep((Long)(Math.random() * 1000));
        } catch (InterruptedException e) {}
    }
    System.out.println("KESZ! " + getName());
}
}

```

A *SimpleThread* osztály első metódusa egy konstruktor, amely paraméterként egy *String*-et fogad. A konstruktor meghívja az *osztály* konstruktorát, ami azért érdekes számunkra, mert az beállítja a szál nevét. Ezt a nevet használjuk a későbbi programban a szál felhasználói azonosítására. A nevet később a *getName* metódussal tudjuk lekérdezni.

A *SimpleThread* osztály következő metódusa a *run*. Itt történik a *Thread* érdemi működése. Ebben az esetben egy tízes *for* ciklust tartalmaz. Minden ciklusban kiírja az iteráció számát, a *Thread* nevét, és utána alaptársa kerül véletlenszerűen 1 másodperces határon belül. Ha végzett, a *KESZ!* feliratot írja ki a szál nevével együtt. Ennyi a *SimpleThread* osztály. Használjuk fel ezt a *TwoThreadsTest*-ben.

A *TwoThreadsTest* osztály *main* metódusába két *SimpleThread* szálát hozunk létre: Jamaica és Fuji. (Ha valaki nem tudná eldönteni hova menjen nyaralni, akkor használja ezt a programot.)

```

public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fuji").start();
    }
}

```

A *main* metódus azonnal mindkét szálát létrehozza a *start* metódus meghívásával, amely a *run* metódust hívja meg. Fordítás és futtatás után kibontakozik az úti cél. A kimenet hasonló kell, hogy legyen:

```

0 Jamaica
0 Fuji
1 Jamaica
1 Fuji
2 Jamaica
2 Fuji
3 Fuji
3 Jamaica
4 Jamaica
4 Fuji
5 Jamaica
5 Fuji
6 Fuji
6 Jamaica
7 Jamaica
7 Fuji
8 Fuji
8 Fuji
9 Fuji
8 Jamaica
DONE! Fuji! — Look out, Fuji, here I come!
9 Jamaica
DONE! Jamaica

```

Figyeljük meg, hogy a szálak kimenete mennyire független egymástól. Ennek az oka, hogy a *SimpleThread* szálak egyidejűleg futnak. Mindkét *run* metódus fut, és mindkét tőnek van saját kimenete ugyanakkor. Ha a ciklus befejeződik, a szálak leállnak és meghalnak.

19.2.2 Runnable interfész példányosítása

A következő *Clock* applet a jelenlegi időt mutatja, és azt másodpercenként frissíti. A frissítés állandó, mivel az óra kijelzése egy saját szálon fut.

Az applet más módszert használ, mint a *SimpleThread*. Itt ugyanis a száloosztály csak az *Applet* lehet, de így az egyszerűsítés érdekében az *Thread* már nem lehet szülő. Ezért a példát *Thread* leszármazott létrehozása helyett egy *Runnable* interfészt implementál, ezért a *run* metódust ebben definiálja. A *Clock* létrehoz egy szálat, amelynek a frissítés a célja.

```
import java.awt.*;
import java.util.*;
import java.applet.*;
import java.text.*;

public class Clock extends java.applet.Applet implements
    Runnable {
    private volatile Thread clockThread = null;
    DateFormat formatter;
    String lastdate;
    Date currentDate;
    Color numberColor;
    Font clockFaceFont;
    Locale locale;

    public void init() {
        setBackground(Color.white);
        numberColor = Color.red;
        locale = Locale.getDefault();
        formatter =
            DateFormat.getInstance(DateFormat.FULL,
                DateFormat.MEDIUM, locale);
        currentDate = new Date();
        lastdate = formatter.format(currentDate);
        clockFaceFont = new Font("Sans-Serif",
            Font.PLAIN, 14);
        resize(275, 25);
    }

    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
}
```



```

public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
    }
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}
}

public void paint(Graphics g) {
    String today;
    currentDate = new Date();
    formatter =
        DateFormat.getInstance(DateFormat.FULL,
            DateFormat.MEDIUM, locale);
    today = formatter.format(currentDate);
    g.setFont(clockFaceFont);
    // Erase and redraw
    g.setColor(getBackground());
    g.drawString(lastdate, 0, 12);
    g.setColor(numberColor);
    g.drawString(today, 0, 12);
    lastdate = today;
    currentDate=null;
}

public void stop() {
    clockThread = null;
}
}

```

A *Clock* applet *run* metódusa addig nem lép ki a ciklusból, amíg a böngésző le nem állítja. Minden ciklusban az óra kimenetele újrarajzolódik. A *paint* metódus lekérdezi az időt, formázza és kijelzi azt.

Kell-e a *Runnable* interfész?

Két módszert ismertettünk a *run* metódus előállítására.

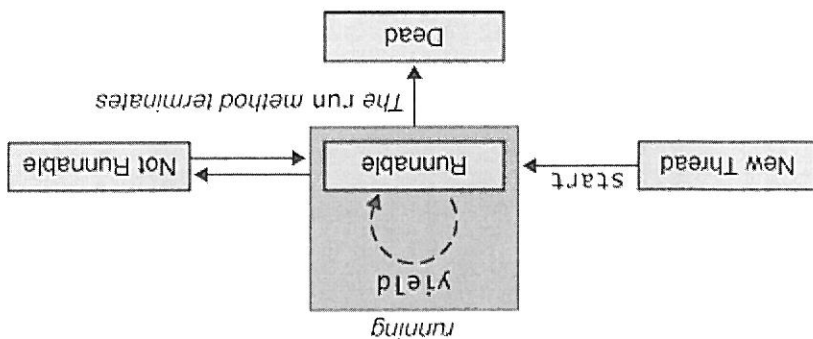
- Leszámrazott *Thread* osztály létrehozása és a *run* metódus felülírása.
- Egy olyan osztály létrehozása, amelyik megvalósítja a *Runnable* interfészt és így a *run* metódust is. Ebben az esetben a *Runnable* objektum szolgáltatja a *run* metódust a szálnak. Ezt feljebb láthatjuk.

Megjegyzés: A *Thread* is megvalósítja a *Runnable* interfészt.

Ahhoz, hogy egy böngészőben futtasson a *Clock* osztálynak az *Applet* osztály leszámrazottjának kell lennie. Továbbá a *Clock* applet folyamatos kijelzése miatt egy szála van szükség, hogy ne a processzt vegye át, amiben fut. (Néhány böngésző figyelhet arra, hogy az appletek saját szálát kapjanak, elkerülendő, hogy a hibás appletek elvegyék a böngésző szálát. De erre nem építhetünk egy applet írásánál. Az általunk írt appletnek tudnia kell a saját szál létrehozását, ha olyan munkát végez.) De mivel a Java nyelv nem engedélyezi a több osztályokon át való öröklődést, a *Clock* nem lehet egyszerre a *Thread* és az *Applet* osztály leszámrazottja. Így a *Clock* osztálynak egy *Runnable* interfészen keresztül kell a szálak viselkedését felvennie.

19.3. Programszál életciklusa

A következő ábra bemutatja a szál különböző állapotait, amibe élete során tud kerülni, és illusztrálja, hogy melyik metódusok meghívása okozza az átmenetet egy másik fázisba. Az ábra nem egy teljes állapotdiagram, inkább csak egy áttekintés a szál életének fontosabb és egyszerűbb oldalairól. A továbbiakban is a *Clock* példát használjuk, hogy bemutassuk a szál életciklusát különböző szakaszokban.



19.3.1 Programszál létrehozása

Az alkalmazás, melyben egy szál fut, meghívja a szál `start` metódusát, amikor a felhasználó a programot elindítja. A *Clock* osztály létrehozza a *clockThread* szálát a `start` metódusában, ahogy a lentí programkód bemutatja:

```

public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}

```

Miután a kiemelt utasítás végrehajódik, a *clockThread* az „új szál” (*New Thread*) állapotban van. Egy szál ebben az állapotban csupán egy üres szál, semmiféle rendszeres erőforrás nincs hozzárendelve. Ebben az állapotban csak elindítható a szál. A `start` metóduson kívül bármilyen metódus meghívása értelmetlen, és *IllegalThreadStateException* kivételt eredményez. (Alapértelmezetten a rendszer mindig ezt a hibüzenetet dobja, ha egy olyan metódus kerül meghívásra, amelyik a szál pillanatnyi állapotában nem engedélyezett.)

Erdemes megjegyezni, hogy a *Clock* típusú példány az első paraméter a szál konstruktor paraméter. Ez a paraméter meg kell, hogy valósítsa a *Runnable* interfészt, csak így lehet a konstruktor paramétere. A *Clock* szál a `run` metódust a *Runnable* interfészről örökli. A konstruktor második paramétere csupán a szál neve.

19.3.2 Programszál elindítása

Most gondoljuk át, mit eredményez a programkódban a `start` metódus:

```

public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}

```

A *start* módszer létrehozza a szükséges rendszererőforrásokat a szál futtatásához, előkészíti a szálát a futáshoz, és meghívja a szál *start* módszerét. A *clockThread.run* módszer a *Clock* osztályban van definiálva.

Miután a *start* módszer visszatér, a szál fut. Mégis, ennél azért kicsit komplexebb a do-log. Ahogy az előző ábra is mutatja, egy elindított szál a futtatható állapotba kerül. Sok számítógép csak egy processzorral rendelkezik, így lehetetlen, hogy minden futási állapotban lévő szál egyszerre fusson. Ezért a Java futtatórendszernek implementálnia kell egy ütemezési sémát, ami elosztja a processzor erőforrásait az összes futó szál között. Szóval egy bizonyos időpontban, egy futó szál valószínűleg vár, hogy ő kerüljön sorra a CPU ütemezésében.

Még egyszer a *Clock.run* módszer:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // The VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

A *Clock.run* módszer addig lépked a ciklusban, amíg a *clockThread == myThread* feltétel nem lesz igaz.

A cikluson belül a szál újra kirajzolja az apple-t, majd utasítja a szálát, hogy kerüljön al-vó állapotba 1 másodpercere. A *repaint* módszer végén a szál meghívja a *paint* metódust, ami az aktuális frissítéseket végzi a program megjelenítési felületén. A *Clock.paint* metódus eltávolítja az aktuális rendszeridőt, formázza, majd megjeleníti:

```
public void paint(Graphics g) {
    //get the time and convert it to a date
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    //format it and display it
    DateFormat dateFormatter = DateFormat.getInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
}
```

19.3.3 Programszál nem futtatható állapotba állítása

Egy szál akkor válik nem futtathatóvá, ha a lenti események közül bármelyik is bekövetkezik:

- Meghívásra kerül a *sleep* metódusa.
- A szál meghívja a *wait* metódust, hogy az várjon egy bizonyos feltétellel teljesülésére.
- A szál blokkolt egy I/O művelet miatt.

A *clockThread* a *Clock* szálon belül akkor kerül nem futtatható állapotba, mikor a futás metódus meghívja a *sleep*-et a jelenlegi szálban:

```

public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
    }
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // A VM nem akarja, hogy tovább aludjunk,
        // szöveg vissza dolgozni ©
    }
}
}

```

Az alatt a másodperc alatt, amíg a *clockThread* alvó állapotban van, a szál nem fut, még akkor sem fog, ha a processzor erőforrásai elérhetővé válnak. Miután letelik az egy másodperc, a szál újra futtathatóvá válik; ha a processzor erőforrásai elérhetővé válnak, a szál újra futni kezd.

Minden egyes nem futtatható állapotba kerüléskor, egy specifikus és megkülönböztethető *exit* módszer téríti vissza a szálát a futtatható állapotba. Egy *exit* hívás után csak a megfelelő feltételek teljesülése esetén fog ismét futni. Például: miután egy szál alvó állapotba lett helyezve, a meghatározott ezredmásodpercek lejártá után újból futtatható állapotba kerül. A következő felsorolás leírja a nem futtatható állapot kilépési feltételeit:

- Ha a szál alvó állapotba lett helyezve, a meghatározott időnek le kell telnie.
- Ha egy szál egy feltételre vár, akkor egy másik objektumnak kell értesítenie a várakozó szálát a feltétel teljesüléséről a *notify* vagy *notifyAll* módszer meghívásával.
- Ha egy szál blokkolva volt egy I/O művelet miatt, az I/O műveletnek be kell fejeződnie.

19.3.4 Programszál leállítás

Habár a *Thread* osztályban rendelkezésre áll a *stop* módszer, ezen módszer alkalmazása nem javallott, mert nem biztonságos, pl. adatvesztés léphet fel. Egy szálnak inkább saját magának kell befejezni a futását úgy, hogy a *run* módszer természetesen áll le. Például a *while* ciklus a *run* módszerban egy véges ciklus 100-szor fut le, majd kilép:

```

public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}

```

A szál a *run* módszerban természetesen „hal meg” mikor a ciklus befejeződik és a *run* módszer kilép.

Nézzük meg, hogyan hajtja végre a *Clock* szál a saját halálát. Nézzük meg újra a *Clock* *run* módszerát:

```

public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            //the VM doesn't want us to sleep anymore,
            //so get back to work
        }
    }
}

```

A kilépési feltétel ebben a `run` módszerben ugyanaz a kilépési feltétele, mint a `while` ciklusnak, mert a ciklus után nincs kód:

```
while (clockThread == myThread) {
```

E feltétel jelzi, hogy a ciklus kilép, ha a jelenleg futásban lévő szál nem egyezik a `clockThread`-el. Mikor lesz ez a helyzet? Akkor, amikor a felhasználó elhagyja az oldalt, az alkalmazás, melyben a szál fut, meghívja a szál `stop` módszerét. Ez a módszer ezután beállítja a `clockThread`-et `null` értékre, így utasítva a fő ciklust, hogy szakítsa meg a futás módszerét:

```

public void stop() { // applets' stop method
    clockThread = null;
}

```

19.3.5 Programszál státusz tesztelése

Az 5.0-ás verzióban került bevezetésre a `Thread.getState` módszer. Mikor ez kerül meg-

hívásra, az alábbi `Thread.State` értékek valamelyike tér vissza:

- `NEW`
- `RUNNABLE`
- `BLOCKED`
- `WAITING`
- `TIMED_WAITING`
- `TERMINATED`

A `Thread` osztályhoz tartozó API tartalmazza az `isAlive` módszert is. Az `isAlive` módszer igaz visszatérési értéket generál, ha a szálát elindították, de még nem lett leállítva. Ha az `isAlive` visszatérési értéke hamis, akkor tudhatjuk, hogy a szál vagy új szál (`NEW`), vagy halott állapotban van (`TERMINATED`). Ha a visszatérési érték igaz, akkor viszont a szál vagy futtatható (`RUNNABLE`), vagy nem futtatható állapotban van.

Az 5.0-ás verziót megelőzően nem lehetett megkülönböztetni a futtatható, vagy nem futtatható lak között. Ugyancsak nem lehetett megkülönböztetni a futtatható, vagy nem futtatható állapotban lévő száltól.

19.3.6 A processzor használatának feladása

Ahogy an el tudjuk képzelni, CPU-igényes kódok negatív hatással vannak más szálakra, amelyek azonos programban futnak. Vagyis próbáljunk jól működő szálakat írni, ame-

Ilyek bizonyos időközönként önkéntesen felhagynak a processzor használatával, ezzel megadva a lehetőséget minden szálnak a futásra.
Egy szál önkéntesen abbahagyhatja a CPU használatát a *yield* módszer meghívásával.

19.4. Ellenőrző kérdések

Melyik az az interfész, amelyet megvalósítva több szálu programfutás érhető el?

- *Runnable*
- *Run*
- *Threadable*
- *Thread*
- *Executable*

Mi annak a módszernek a neve, amellyel a külön programszál futását kezdeményezni tudjuk?

- *init*
- *start*
- *run*
- *resume*
- *sleep*

Mi annak a módszernek a neve, amellyel a programszál futását le tudjuk állítani?

(Minden helyes választ jelöljön meg!)

- *sleep*
- *stop*
- *yield*
- *wait*
- *notify*
- *notifyAll*
- *synchronized*

Mit tapasztalunk, ha fordítani és futtatni próbáljuk a következő programot?

```
public class Background extends Thread{
    public static void main(String argv[]){
        Background b = new Background();
        b.run();
    }
    public void start(){
        for (int i = 0; i < 10; i++){
            System.out.println("Value of i = " + i);
        }
    }
}
```

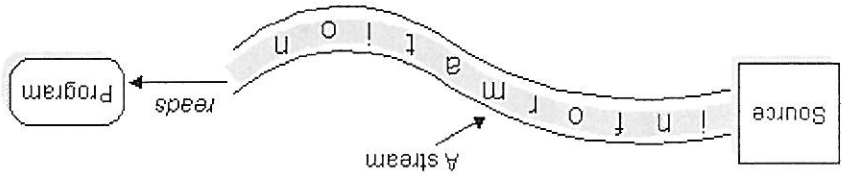
- Fordítási hiba, mert a *run* metódus nincs definiálva a *Background* osztályban
- Futási hiba, mert a *run* metódus nincs definiálva a *Background* osztályban
- A program lefut, és 0-tól 9-ig ír ki számokat
- A program lefut, de nincs kimenete

20. Fájlkészítés

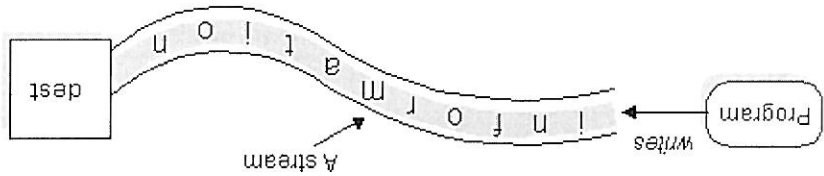
A legtöbb programnak szüksége van arra, hogy külső forrásból olvasson adatokat, vagy külső célba írja a működése (rész)eredményeit. A forrás és a cél sokféle lehet: konzol, háttérár, másik program, vagy egyéb speciális eszközök. Az adatok is különbözőek lehetnek: számok, szövegek, vagy akár tetszőleges objektumok. Ez a fejezet bemutatja, hogyan lehet Java nyelven kezelni az adatállományainkat.

20.1. Adatfolyamok

Információ kinyeréséhez a program megnyit egy adatfolyamot az információforráshoz (fájl, memória, socket), és sorban kiolvassa az információt, amint az ábrán látszik:



Hasonlóképpen, egy program külső célba is küldheti az információkat adatfolyam megnyitásával, és információkat írhat ki sorban, mint itt:



Nem számít, mekkora adat jön be, vagy megy ki, és nem számít a típusa, az algoritmus sorban olvassa és írja, a következők köz hasonlóan:

Olvasás

```

adatfolyam megnyitása
amíg van újabb információ
  olvasás
adatfolyam bezárása
  
```

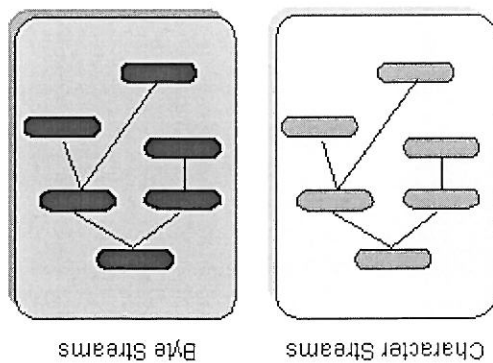
Írás

```

adatfolyam megnyitása
amíg van újabb információ
  írás
adatfolyam bezárása
  
```

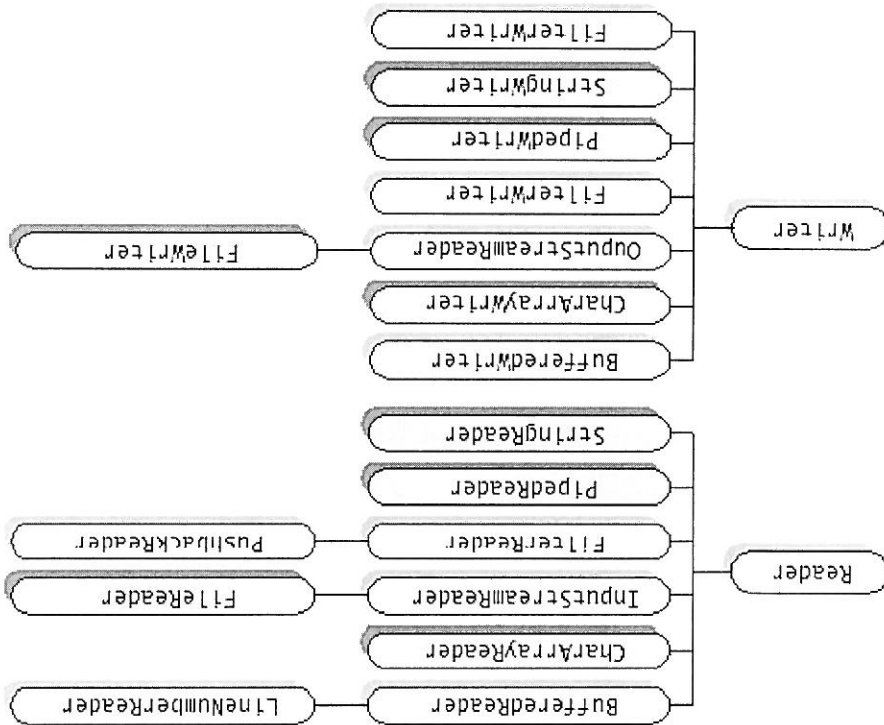
A `java.io` csomag tartalmazza azon osztályokat, melyek lehetővé teszik az írást és olvasást. Az osztályok használatához importálni kell a `java.io` csomagot.

Az adatfolyam osztályok két hierarchiára oszthatók az adattípusuk alapján (karakter vagy bájt):



Karakter folyamok

Reader és *Writer* a *java.io* absztrakt osztályai. 16 bites karakterekkel való műveleteket teszük lehetővé. A *Reader* és *Writer* leszármazott osztályai speciális műveleteket végeznek, és két külön kategóriára oszthatók: az adat olvasás és írás (szűrőkkel jelezve) és feldolgozások végrehajtása (fűtők). Az utóbbiakat szűrő folyamoknak is nevezzük. A *Reader* és *Writer* osztályok osztályhierarchiáját mutatja az ábra.

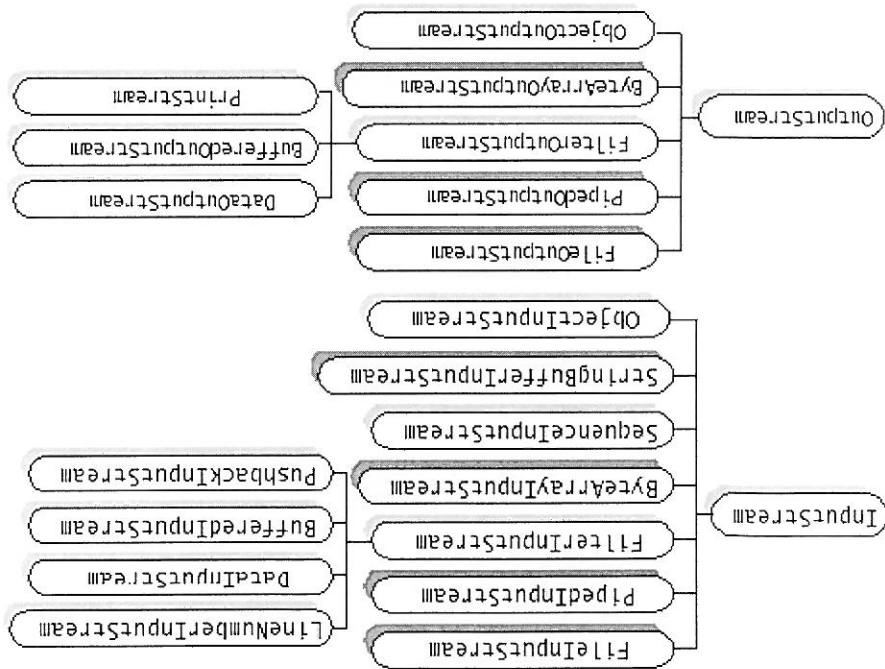


A legtöbb programnak szüksége van a *Reader* és *Writer* osztályokra szöveges információk olvasásához és írásához. Ezek bármilyen karaktert képes lekezelni Unicode karakterként.

Bináris folyamok

A 8-bites bájt írásához és olvasásához a programnak használnia kell a bináris folyamokat, melyek az *InputStream* és *OutputStream* leszármazottai. Ezek a folyamok jellemzően bináris adatok olvasására és írására alkalmasak, mint egy kép vagy hang. A bájt folyam két osztálya, *ObjectInputStream* és *ObjectOutputStream* használható objektumok soros folyammon keresztüli kezeléshez.

Az *InputStream* és *OutputStream* osztályok a *Reader* és *Writer* osztályokhoz hasonlóan két kategóriára oszthatók, a következő osztályhierarchia alapján: adatkezelő folyamatok (szűrke) és feldolgozó vagy más néven szűrő folyamatok (fehér).



Az I/O szülőosztályok

Reader és *InputStream* azonos API-kat határoznak meg, de különböző adattípusra. A *Reader* metódusai karakterteremből olvasására alkalmasak:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

Az *InputStream* hasonló metódusai bináris adatok és tömbök olvasására alkalmasak:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

A *Reader* és *InputStream* is megvalósítja az adatfolyamon belüli állapot jelölését, ugrást, és az aktuális pozíció visszaállítását.

A *Writer* és *OutputStream* hasonló szolgáltatásokkal rendelkeznek. A *Writer* karakterek, és karakterteremből írását valósítja meg:

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Az *OutputStream* azonos metódusú de bináris adatra:

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

Mind a négy osztály esetén igaz, hogy automatikusan megnyílik az adatfolyam, ha létre van hozva az objektum. Minden adatfolyam bezárható a *close* meghívásával. A szemét-

gyűjtő is bezárja, ha már nincs rá hivatkozás, de célszerű inkább direkt módon meghívni a `close` metódust.

Megjegyzés: A program „kötelessége”, hogy minden erőforrást csak a ténylegesen szükséges ideig foglaljon le. Eppen ezért az állomány megnyitását olyan későn kell megtenni, amikor csak lehet, és ugyanilyen hamar be is kell zárni.

Az adatfolyam osztályok használata

A táblázat a `java.io` adatfolyamokat és feladatuk leírását tartalmazza.

I/O típus	Osztályok	Leírás
Memória	CharArrayReader CharArrayWriter ByteArrayInputSteam ByteArrayOutputSteam	Az adatfolyamok kiolvasására és a memóriába írására használható. Az adatfolyamok létrehozhatók egy meglévő tömbbel, és az olvasást és írást használva kiolvas és a tömbbe ír.
	StringReader StringWriter StringBufferInputSteam	A <code>StringReader</code> karaktertereket olvas a memóriából. A <code>StringWriter</code> egy <code>String</code>-be ír. A <code>StringWriter</code> kiírandó karaktertereket gyűjt egy <code>StringBuffer</code>-be, ami <code>String</code>-be konvertál. <code>StringBufferInputSteam</code> hasonló a <code>StringReader</code>-hez, kivéve, hogy egy <code>StringBuffer</code>-ből karaktertereket olvas.
Cső	PipedReader PipedWriter PipedInputSteam PipedOutputSteam	A bemenet és kimenet között valósít meg egy összekötő csövet. A csövek a kimenettől a másik bemenetig tartó folyamot valósítanak meg.
Fájl	FileReader FileWriter FileInputSteam FileOutputSteam	Együttesen fájl folyamatoknak hívják, melyekkel fájlból olvashatunk, vagy fájlba írhatunk.
Lánc	SequenceInputSteam	Több bemenet adatfolyamainak összefűzése egy bemeneti adatfolyamba.
Objektum sorosítása	ObjectInputSteam ObjectOutputSteam	Objektumok soros szervezésű tárolásához és visszaállításához.
Adat konverzió	DataInputSteam DataOutputSteam	Primitív adattípusok írása és olvasása géptüggetlen formátumban.
Számláló	LineNumberReader LineNumberInputSteam	Nyomon követhető a sorok száma olvasáskor.
Előre olvasás	PushbackReader	A bemeneti adatfolyamok egy vissza-

tehető pufferbe kerülnek. Az adatok olvasásakor néha hasznos a következő néhány bájt vagy karakter ismerete a következő lépés eldöntéséhez.

Kinyomtatás	<i>PrintWriter</i>	Nyomtatási metódusokat tartalmaz. Az adatfolyamok tartalmának kinyomtatására használható.
	<i>PrintStream</i>	
Pufferelés	<i>BufferedReader</i>	Olvasásakor és írásakor az adatok pufferelődnek, ezáltal csökkentve a forrásadat elérésének számát. A pufferelt adatfolyamok hatékonyságuk, mint a nem pufferelt adatfolyamok, és gyakran használhatók más adatfolyamokkal is.
	<i>BufferedInputStream</i> <i>Buf-feredInputStream</i>	
	<i>BufferedOutputStream</i>	

Szűrés	<i>FilterReader</i>	Ezek az absztrakt osztályok a szűrő adatfolyamok vázát adják meg, milyen szűrő használható írásakor vagy olvasásakor.
	<i>FilterInputStream</i> <i>FilterOutputStream</i>	

Bájtok és karakterek közti konverzió	<i>InputStreamReader</i>	Az olvasó és író páros hidat képez a bájt és a karakterfolyamok között.
	<i>OutputStreamWriter</i>	Az <i>InputStreamReader</i> bájtokat olvas egy <i>InputStream</i> -ből és karakterbe konvertálja, az alapértelmezett karakterkódolást vagy egy megnevezett karakterkódolást használva.

Az *OutputStreamWriter* karaktereket konvertál bájtba, az alapértelmezett karakterkódolást vagy egy megnevezett karakterkódolást használva és a bájtokat egy *OutputStream*-be írja.

20.1.1 Fájladatfolyamok használata

A fájladatfolyamok talán a legkönnyebben megérthető adatfolyamok. A fájladatfolyamok (*FileReader*, *FileWriter*, *FileInputStream*, és *FileOutputStream*) a natív fájltrendszert fájljait olvassák, vagy írják. Létrehozhatunk fájladatfolyamot fájlnév *String*-ből, *File* objektummal, vagy *FileDescriptor* objektummal.

A következő *Copy* program *FileReader* és *FileWriter* segítségével az *input.txt* tartalmát átírja (átmásolja) az *output.txt* fájlba:

```
import java.io.*;

public class Copy {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("input.txt");
        File outputFile = new File("output.txt");
    }
}
```

```

FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);

int c;
while ((c = in.read()) != -1)
    out.write(c);

in.close();
out.close();
}
}

```

A program nagyon egyszerű. A *FileReader* megnyitja a *farvago.txt* fájlt, a *FileWriter* pedig az *outagin.txt* fájlt. A program az *in* objektummal beolvas minden karaktert a bemeneti fájlból, és az *out* objektummal kírja a karaktereket. Ha a bemenet elfogyott (-1), a program beárja az olvasót és az író is.

Megjegyzés: Erdemes megfigyelni, a *read* és *write* metódusok nem *char*, hanem *int* típusú dolgoknak. Ennek mindössze az az oka, hogy egy speciális, a *char* megengedett értéktartományán kívüli visszatérési értéket (a -1-et) is kell nyújtani abban az esetben, ha nincs több olvasható karakter, így a *char* nem lenne alkalmazható.

A *Copy* program ezt a kódot használja a *FileReader* létrehozásához:

```

File inputFile = new File("input.txt");
FileReader in = new FileReader(inputFile);

```

A kód létrehoz egy *File* objektumot, ami a fájl logikai leírására szolgál. A *File* segédosztályt a *java.io* tartalmazza. A *Copy* program ezt az objektumot használja a *FileReader* felépítéséhez. Azonban a program az *inputFile*-t akkor is használhatja, ha információt szeretnénk szerezni a fájl egészéről (pl. olvasható-e, mekkora a mérete stb).

A program futása után ugyanannak kell szerepelnie az *input.txt* és az *output.txt* fájlban is. Ne feledjük, hogy a *FileReader* és a *FileWriter* 16-bites karaktert olvas és ír. Azonban, a legtöbb natív fájlrendszer 8-biten alapszik. Az adatfolyamok kódolják a karaktereket, az alapértelmezett karakterkódolási séma alapján. A karakterkódolás megkapható a *System.getProperty("file.encoding")* használatával. Új alapértelmezett kódolás megadása-hoz létre kell hozni egy *OutputStreamWriter* vagy egy *FileOutputStream* objektumot, és megadni a kódolást.

20.1.2 Szűrő adatfolyamok

A *java.io* csomag tartalmazza az absztrakt osztályok speciális beállításainak lehetőségét, melyel definiálhatunk, és részlegesen implementálhatunk szűrőket, olvasás, illetve írás céljából. A használatos filter adatfolyamok a *FilterInputStream*, *FilterOutputStream*, Egy filter adatfolyam egy másik alap-adatfolyamra épül, melybe a *write* metódus fog adatokat menteni, de csak a szűrés után. Vannak adatfolyamok, melyek speciális feladatokat látnak el, mint például a konvertálás vagy számlálás folyamata.

A legtöbb filter adatfolyamot a *java.io* csomag által szolgáltatott származtatott osztályok nyújtják, ezek a következők:

- *DataInputStream* és *DataOutputStream*
- *BufferedInputStream* és *BufferedOutputStream*
- *LineNumberInputStream*
- *PushbackInputStream*

- *PrintStream*

A *java.io* csomag csak egy származtatott osztályt tartalmazza a *FilterReader*-nek, ez nem más, mint a *PushbackReader*. A következő fejezet bemutatja egy példán szemléltetve, hogyan használjuk a szűrő adatfolyamokat a *DataInputStream* és a *DataOutputStream* használatával. Ugyancsak tárgyalja, hogyan hozhatunk létre származtatott osztályokat a *FilterInputStream* és *FilterOutputStream*-ből, melyek segítségével saját szűrő adatfolyamunkat valósíthatjuk meg.

Szűrő adatfolyamok használata

A szűrő objektumokat csatolni kell egy másik I/O adatfolyamhoz, példaként csatolhatunk szűrő adatfolyamot a standard input adatfolyamra:

```
BufferedReader d = new BufferedReader(new
    DataInputStream(System.in));
String input;
while ((input = d.readLine()) != null) {
    ... //do something interesting here
}
```

Megjegyzés: A *readLine* metódus nem használható a *DataInputStream*-ben, ezért hoztuk létre a *BufferedReader* típusú d szűrő objektumot.

A következőkben egy példán szemléltetjük a *DataInputStream* és *DataOutputStream* használatát. Ezek olyan szűrők melyek képesek írni/olvasni elemi adattípusokat.

Előfordul, hogy a feldolgozás független a kezelt adat formátumától, de néha szorosan összefügg az illető adatokkal vagy annak formátumaival, úgyminth adatok írása, olvasása sorokból vagy cellákból.

DataInputStream és *DataOutputStream* használata

Ez a fejezet a *DataInputStream* és *DataOutputStream* osztályok használatát mutatja be egy *DataIO Demo* példaprogramon keresztül, mely olvas és ír táblázatba (cellákba) foglalt adatokat. A cellák tartalmazzák az eladási árat, a rendelt áru mennyiségét, és a leírt. Elméletileg az adat a következőképpen néz ki, habár a gép ezt bináris adatként kezeli:

19.99	12	Java T-shirt
9.99	8	Java Mug

DataOutputStream-et, mint ahogyan más kimeneti adatfolyamokat, csatolni kell egy másik *OutputStream*-hez. Ebben az esetben a *FileOutputStream*-hez csatoltuk. Példa:

```
DataOutputStream out = new DataOutputStream(
    new FileOutputStream("invoice1.txt"));
```

A *DataIO Demo* a *DataOutputStream* speciális *write* metódusait használja írásra, amennyiben az megfelelő típusú.


```

for (int i = 0; i < prices.length; i++) {
    out.writeDouble(prices[i]);
    out.writeChar('\t');
    out.writeInt(units[i]);
    out.writeChar('\t');
    out.writeChars(descs[i]);
    out.writeChar('\n');
}
out.close();

```

A *DataIDemo* program beolvasa az adatokat a *DataInputStream* speciális metódusai segítségével:

```

try {
    while (true) {
        price = in.readDouble();
        in.readChar(); //throws out the tab
        unit = in.readInt();
        in.readChar(); //throws out the tab
        char chr;
        desc = new StringBuffer(20);
        char lineSep =
            System.getProperty("line.separator").charAt(0);
        while ((chr = in.readChar()) != lineSep) {
            desc.append(chr);
        }
        System.out.println("You've ordered " + unit
            + " units of "
            + desc + " at $" + price);
        total = total + unit * price;
    } catch (EOFException e) { }
    System.out.println("For a TOTAL of: $" + total);
    in.close();
}

```

Mihelyst beolvasa az adatokat, a *DataIDemo* megjeleníti az összegzést, a rendelt és az összes mennyiséget, majd bezárul.

Megjegyzés: a *DataIDemo* által használt ciklus a *DataInputStream*-ből olvassa az adatokat:

```

while ((input = in.read()) != null) {
    .
    .
    .
}

```

A *read* metódus *null* értékkel tér vissza, mely a fájl végét jelenti.

Számos alkalommal azonban ezt nem tehetjük meg, a szabálytalanul megadott értékek esetén hibajelzést kapunk. Tegyük fel, hogy -1 értéket szeretnénk ugyanerre a célra felhasználni. Ezt nem tehetjük meg, ugyanis a -1 nem szabályos érték, kiolvasása a *readDouble* vagy *readInt* használatával történik. Tehát *DataInputStream*-ek esetén nem kapunk *EOFException* kivételt.

A program futása a következő kimenetet eredményezi:

```

You've ordered 12 units of Java T-shirt at $19.99
You've ordered 8 units of Java Mug at $9.99
You've ordered 13 units of Duke Juggling Dolls at $15.99
You've ordered 29 units of Java Pin at $3.99
You've ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.8800000000000001

```

20.2. Objektum szerializáció

A *java.io* két adatfolyama - az *ObjectInputStream* és az *ObjectOutputStream* - abban más az átlagos bátfolyamoktól, hogy objektumokat tudnak írni és olvasni.

Az objektumok írásának az alap feltétele, hogy egy olyan szerializált formára hozzuk az objektumot, ami elegendő adatot tartalmaz a későbbi visszaalakításhoz. Ezért hívjuk az objektumok írását és olvasását objektum szerializációnak.

Objektumokat a következő két módon lehet szerializálni:

- Távoli eljárás hívás (*Remote Method Invocation*, RMI)
Kliens és szerver közötti objektumátvitel esetén
- Könnyűsúlyú perzisztencia (*Lightweight persistence*)
Egy objektum archiválása későbbi használatra

20.2.1 Objektumok szerializálása

Az *ObjectOutputStream* írása

A következő kódreszt megkapja a pontos időt a *Date* objektum konstruktorának a meghívásával, és aztán szerializálja az objektumot:

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

Az *ObjectOutputStream* konstruktorát egy másik adatfolyamra kell meghívni. Az előző példában az *ObjectOutputStream* konstruktora egy *FileOutputStream*-re hívódik, így egy 'theTime' nevű fájlba szerializálja az objektumot. Ezek után egy *String* ("Today") és egy *Date* objektum íródik az adatfolyamra az *ObjectOutputStream* *writeObject* nevű módszerével.

A *writeObject* módszer szerializálja az adott objektumot, rekurzívan átmenti a más objektumokra való hivatkozásait, így megtartja az objektumok közötti kapcsolatot.

Az *ObjectOutputStream* implementálja a *DataOutput* interfészt, ami primitív adattípusok írására való módszerokat tartalmaz (*writeInt*, *writeFloat*, *writeUTF*...). Ezeket a módszereket használjuk primitív adattípusok *ObjectOutputStream*-re való írásához.

A *writeObject* módszer *NotSerializableException*-t dob, ha a megadott objektum nem szerializálható. Egy objektum csak akkor szerializálható, ha implementálja a *Serializable* interfészt.

Az *ObjectInputStream* olvasása

A következő példaprogram kiolvassa a *theTime* fájlba előzőleg kiírt *String* és *Date* objektumokat:

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Az *ObjectOutputStream*-hez hasonlóan az *ObjectInputStream* konstruktorát is egy másikkal adatközlőre kell hívni. Mivel egy fájlba írunk az objektumokat, itt is egy *FileInputStream*-re kell hívni az *ObjectInputStream*-et. Az olvasást az *ObjectInputStream.readObject()* metódusa végzi. Az objektumokat mindenképpen abban a sorrendben kell kiolvasni, ahogy az adatközlőben lettek írva.

A *readObject()* metódus deszerializálja a soron következő objektumot. Rekurzívan vizsgálja a hivatkozásokat, hogy az összes hivatkozott objektumot is deszerializálja, így megtartva az objektumok közötti kapcsolatot.

Az *ObjectInputStream* adatközlő implementálja a *DataInput* interfészt, ami primitív adattípusokat olvasó metódusokat tartalmaz. A *DataInput* metódusai a *DataOutput* metódusainak olvasó megfelelői (*readInt*, *readFloat*, *readUTF*,...).

20.2.2 Objektum szerializáció a gyakorlatban

Egy objektum csak akkor szerializálható, ha az osztály implementálja a *Serializable* interfészt. Ezt könnyen megtehetjük:

```
public class MySerializableClass implements Serializable {
    ...
}
```

A *Serializable* egy üres interfész (vagyis nincs egyetlen tagja sem), csak a szerializálható osztályok elköteleződésére való. Az osztály példányainak a szerializálását az *ObjectOutputStream* osztály végzi. A deszerializáció az *ObjectInputStream* osztály *defaultWriteObject()* metódusa végzi. A legtöbb objektumhoz ez a módszer elég is. Néhány esetben viszont lassú lehet, és az osztálynak is szüksége lehet a szerializáció feletti határozottabb vezérlésre.

A szerializáció testreszabása

Az objektum szerializáció testreszabásához két metódus felülírására van szükségünk: a *writeObject()* metódus (írásra) és a *readObject()* (olvasásra vagy adattípusításra).

A *writeObject()* deklarálása:

```
private void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
    // testreszabott szerializáló kód
}
```

A *readObject()* metódus deklarálása:

```
private void readObject(ObjectInputStream s) throws IOException {
    s.defaultReadObject();
    // testreszabott deszerializáló kód
    ...
    // objektumokat frissítő kód (ha szükséges)
}
```

Mindkét metódust pontosan a bemutatott módon kell deklarálni.

A *writeObject()* és a *readObject()* metódusok csak az adott osztály szerializálásáért felelősek, a szülő osztályok esetleges szerializálása automatikus. Ha egy osztálynak a szülőosztályaival teljesen összehangban kell lennie a szerializációhoz, akkor az *Externalizable* interfészt kell implementálni.

20.2.3 Az *Externalizable* interfész implementálása

Hogy egy osztály teljes mértékben kontrollálni tudja a serializációs folyamatot, implementálnia kell az *Externalizable* interfészt. *Externalizable* objektumoknál csak az objektum egy 'reprezentációja' kerül az adatfolyamra. Az objektum maga felelős a tartalom írásáért, olvasásáért és a szülőobjektumokkal való együttműködésért.

Az *Externalizable* interfész definíciója:

```
package java.io;

public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
}
```

Egy *Externalizable* osztállyal a következőket kell szem előtt tartani:

- implementálnia kell a *writeExternal* metódust az objektum állapotának az elmentéséhez.
- implementálnia kell a *readExternal* metódust a *writeExternal* által az adatfolyamra írt tartalom visszaállításához.
- külső adatformátum írásáért és olvasásáért kizárólag a *writeExternal* és a *readExternal* metódusok felelősek.

A *writeExternal* és a *readExternal* metódusok publikusak, így lehetőséget nyújtanak arra, hogy egy kliens az objektum metódusai nélkül is tudja olvasni/írni az objektum adatait, ezért csak kevésbé biztonság-igényes esetekben alkalmazhatóak.

20.2.4 Az érzékeny információ megvédése

Ha egy osztályunk elérést biztosít bizonyos forrásokhoz, gondoskodnunk kell a kényes információk és függvények védelméről. A deszerializáció alatt az objektum *private* állapota is helyreáll. Például egy fájlleíró eljárás hozzáférést biztosít egy operációs rendszer forráshoz. Ha ez egy adatfolyamon keresztül áll helyre, a sebezhetősége miatt visszaélésekre ad lehetőséget. Ezért a valós időben történő serializáció esetében nem szabad feltétlenül megbízni az objektum-reprezentációk valódiságában. Két módon oldhatjuk meg a problémát: vagy nem szabad az objektum érzékeny tulajdonságait az adatfolyamról visszaállítani, vagy ellenőrizni kell azokat.

A védelem leegyszerűbb módja az érzékeny adatok *private transient*-ként való feltüntetése. A *transient* és a *static* adattagok nem kerülnek serializálásra, így nem is jelennek meg az adatfolyamon. Mivel a *private* adattagok olvasását és írását nem lehet az osztályon kívülről helyettesíteni, az osztály *transient* tagjai így biztonságban vannak.

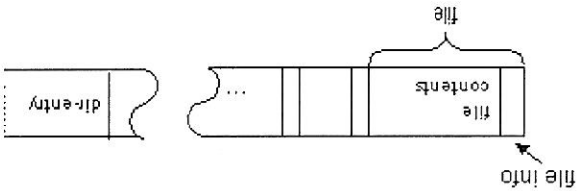
A különösen érzékeny osztályokat egyáltalán nem szabad serializálni. Ez esetben egy-egy szerűen nem szabad implementálni sem a *Serializable*, sem az *Externalizable* interfészt. Nehány osztállyal különösen előnyös az írás és az olvasás arra való használata, hogy a deszerializálás közben kezelni és ellenőrizni tudjuk az érzékeny adatot. A *writeObject* és a *readObject* metódusok itt csak a helyes adatot menthetik, vagy állíthatják helyre. Ha a

hozzáférés tiltására van szükség, egy *NotSupportedException* dobása elég, hogy megta-
gadjuk a hozzáférést.

20.3. Közvetlen elérésű állományok

Az ki és bemenő adatfolyamok eddig a leckéig soros hozzáférések voltak, melyek írása és olvasása az elejétől a végéig sorban történik. Bár az ilyen adatfolyamok hihetetlenül hasznosak, de ezek csak a soros hozzáférésű médiumok (papír és mágnesszalag) követ-keztében jöttek létre. Másrésztől a közvetlen hozzáférésű fájlok, nem-soros, hanem vé-letlenszerű (közvetlen) hozzáférést biztosítanak a fájl tartalmához.

Tehát miért van szükség közvetlen elérésű fájlokra? Gondoljunk csak a tömörített ZIP formátumra. A ZIP állományok fájlokat tartalmaznak, tömörítve a helytakarékosság mi-att. A ZIP állományok tartalmaznak egy az állomány végén elhelyezett *dir* bejegyzést, mely megmutatja, hogy a különböző fájlok hol találhatók a ZIP belsejében.



Felteve, hogy egy meghatározott fájlt akarunk kicsomagolni a ZIP-ből, soros hozzáférés esetén a következőket kell tennünk:

- Megnyitjuk a ZIP fájlt
- Végigkeressük a ZIP fájlt a keresett állománnyért
- Kicsomagoljuk az állományt
- Bezárjuk a ZIP-et

Ezt az algoritmust használva, átlagban legalább a ZIP állomány felét végig kell olvasni a keresett állománnyért. Kitémöríthetjük ugyanezt a fájlt a ZIP-ből sokkal hatékonyabban, ha a közvetlen elérésű fájlok *seek*, azaz keresési funkcióját használjuk a következő lépések szerint:

- Megnyitjuk a ZIP fájlt
- Megkeressük a *dir* bejegyzést és azon belül a kitémörítendő fájl helyét
- Megkeressük (visszafele) a fájl pozícióját a ZIP-en belül
- Kicsomagoljuk az állományt
- Bezárjuk a ZIP-et

Ez a módszer sokkal hatékonyabb, mert csak a *dir* bejegyzést és kicsomagolandó fájlt kell beolvasni. A *java.io* csomagban található *RandomAccessFile* osztály implementálja közvetlen elérésű fájlokat.

20.3.1 A közvetlen elérésű állományok használata

A `RandomAccessFile` osztály mind írásra, mind olvasásra alkalmas, ellentétben a `java.io` csomag által tartalmazott ki-, és bemeneti adatfolyamokkal. Egy `RandomAccessFile` objektumot a szerint hozunk létre különböző paraméterekkel, hogy írni vagy olvasni akarunk vele. A `RandomAccessFile` nincs kapcsolatban a `java.io` csomag ki-, és bemeneti adatfolyamaival, és ez az, amiért nem az `InputStream` és `OutputStream` leszármazottja. Ennek az a hátránya, hogy a `RandomAccessFile`-ra nem alkalmazhatók azok a szűrők, amelyek az adatfolyamokra igen. Habár a `RandomAccessFile` implementálja a `DataInput` és `DataOutput` interfészt, tehát ha készítnék egy olyan filtert, mely vagy az `DataInput` vagy a `DataOutput` együttműködik, ez működni fog néhány soros elérési állományal valamint az összes közvetlen eléréssel.

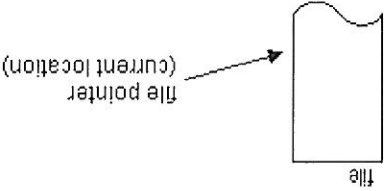
Ha meg akarjuk nyitni olvasásra az `input.txt` állományt, akkor a következő kóddal tehetjük meg:

```
new RandomAccessFile("input.txt", "r");
```

Ha olvasni és írni is akarjuk:

```
new RandomAccessFile("input.txt", "rw");
```

Ezek után a `read` és `write` metódusok a szokásos módon állnak rendelkezésre.



Fontos további szolgáltatás, hogy ezen kívül az aktuális állomány pozíció közvetlenül is manipulálható a következő metódusokkal:

<code>int skipBytes(int)</code>	előre mozgatja az aktuális pozíciót
<code>void seek(long)</code>	a megadott pozícióra mozgatja az aktuális pozíciót
<code>long getFilePointer()</code>	lekérdezi a pillanatnyi pozíciót

20.4. További osztályok és interfészek

Az előzőekben bemutatott osztályokon és interfészeken kívül a `java.io` csomag többek között a következő osztályokat és interfészeket tartalmazza:

File

Egy fájlt képvisel a natív fájlrendszerben. Létréhozhatunk a `File` objektumot egy állományhoz a natív fájlrendszerben, és adatokat kérdezhetünk le az állományról. Például a teljes elérési útit.

FileDescriptor

Egy fájlkezelőt valósít meg nyitott állományok és portok esetén. Nem sokszor használt.

StreamTokenizer

Adatfolyam tartalmát tördeli részekre (tokenekre). A tokenek a legkisebb egységek (szavak, szimbólumok), melyeket a szövegfelismerő algoritmusok képesek felismerni. A *StreamTokenizer* objektumok bármely szöveges fájl elemzésére használhatók. Például használható a forrásállományoktól a változónevek, operátorok, illetve HTML fájlokból HTML tagok kigyűjtésére. (A *StringTokenizer*-hez hasonló feladatot lát el.)

FileNameFilter

A *File* osztály *list* metódusa használja, hogy mely állományok vannak egy listázandó könyvtárban. A *FileNameFilter* használható az egyszerű kifejezés stílusú állomány kereső minták implementálására, mint például *.doc, vagy akár sokkal bonyolultabb szűrősekre is.

Találhatóak még egyéb input osztályok a *java.util.zip* csomagban:

CheckedInputStream és CheckedException

Egy ki- és bementi adatfolyam pár, mely egy ellenőrző összeget számít az adatok írásakor és olvasásakor.

DeflaterOutputStream és InflaterInputStream

Betömöríti és kitömöríti az adatokat íráskor és olvasáskor.

GZIPInputStream és GZIPOutputStream

GZIP formában tömörített adatokat ír és olvas.

ZipInputStream és ZipOutputStream

ZIP formában tömörített adatokat ír és olvas.

20.5. Ellenőrző kérdések

- Mi az író karaktercsatornák absztrakt öse?
- Mi az olvasó karaktercsatornák absztrakt öse?
- Mi az író bajtcsatornák absztrakt öse?
- Mi az olvasó bajtcsatornák absztrakt öse?
- Melyik módszer használható az összes olvasó csatorna esetén?
- Melyik módszer használható az összes író csatorna esetén?
- Mondjon példát memória-csatornára!
- Írjon példát a *File* osztály szolgáltatásaira!
- Milyen a szűrő folyamatok?

Igaz vagy hamis? Indokolja!

- A *RandomAccessFile* egy szűrő folyamat.

- A véletlen elérési állomány mutatóját lehet állítani.
- A véletlen elérési állomány segítségével egy fájlban bármelyik bájtja módosítható.
- Az objektumfolyamra írható primitív adat is.
- Az objektumfolyamra írt objektumnak implementálnia kell a *Serializable* interfészt.
- A pufferező folyamathoz mindig csatolni kell egy másik folyamatot.

A következők közül melyik hoz létre *InputStreamReader* objektumot?

(Minden helyes választ jelöljön meg!)

- `new InputStreamReader(new FileInputStream("data"));`
- `new InputStreamReader(new FileReader("data"));`
- `new InputStreamReader(new BufferedReader("data"));`
- `new InputStreamReader("data");`
- `new InputStreamReader(System.in);`

A következők közül melyik korrekt?

- `RandomAccessFile("data", "r");`
- `RandomAccessFile("r", "data");`
- `RandomAccessFile("data", "read");`
- `RandomAccessFile("read", "data");`

Melyikre nem képes a *File* osztály?

- Aktuális könyvtár beállítása
- Szülő könyvtár nevének előállítás
- Állomány törlése
- Állomány(ok) keresése

21. Gyűjtemények

A gyűjtemények (vagy más néven tárolók, konténerek, kollekciók) olyan típuskonstrukciós eszközök, melynek célja egy vagy több típusba tartozó objektumok memóriában történő összefoglaló jellegű tárolása, manipulálása és lekérdezése.

21.1. A gyűjtemény keretrendszer

A gyűjtemény keretrendszer (*Java Collections Framework*, JCF) egy egységes architektúra, ami a gyűjtemények használatára és manipulálására szolgál. A gyűjtemény keretrendszer tartalmazza:

- **interfészek:** absztrakt adattípusok reprezentációja. Az interfészek lehetővé teszik a gyűjtemények szolgáltatásainak (publikus interfészeknek) megvalósítását függetlenül a rázolásától.
- **implementációk:** a gyűjtemény interfészek konkrét implementációi. Főként ezek az újrafelhasználható adatstruktúrák.
- **algoritmusok:** azok a metódusok, amelyek hasznos műveleteket valósítanak meg, mint például a keresés vagy a rendezés egy objektumon, ami implementálható különböző gyűjtemény interfészekben. Ezeket az algoritmusokat többalakúnak hívjuk: azonos metódus használható különböző implementációk esetén is. Elsősorban az algoritmusok újrafelhasználható funkcionálitása.

A JCF-hez hasonló legismertebb megoldás a C++ nyelvű szabványos minta-könyvtár (*Standard Template Library*, STL).

21.1.1 A Gyűjtemény keretrendszer használatának előnyei

A Java Gyűjtemény keretrendszer az alábbi előnyökkel rendelkezik:

Csökkent a fejlesztési időt

Mivel kész adatstruktúrák és algoritmusok állnak rendelkezésünkre, a Gyűjtemény keretrendszer lehetővé teszi, hogy a program fontosabb részével foglalkozzunk, ahelyett, hogy alacsony szintű programozással kelljen foglalkoznunk.

Növeli a programozás sebességét és minőségét

A JCF gyors és jó minőségű algoritmus-, és adatstruktúra-implementációkkal rendelkezik. A különböző implementációk minden egyes interfésznel felcserélhetők, így a programokat könnyen össze lehet hangozni a gyűjtemény implementációkkal. Több időt tud arra fordítani, hogy növelje a programok minőségét és sebességét, mivel megszabadítja attól a rabszolgamunkától, hogy saját adatstruktúrákat kelljen írnia.

Megengedi az együttműködést a nem kapcsolódó API-k között

Ha két különböző függvénykönyvtár nem illeszthető egymáshoz közvetlenül, akkor lehet akár a keretrendszer a közös nevező az illesztés megteremtése érdekében.

Csökkeni az új API-k használatának és tanulásának nehézségét

Régen minden egyes API-nál egy kis segéd API-t készítettek arra, hogy manipulálja az egyes gyűjteményeket. Kéves összefüggés volt az erre a célra készült gyűjtemények segéd API-jai között, így külön-külön meg kell tanulni azokat, így könnyen hibát ejthetünk ezek használatával. Az általános gyűjtemény interfészek megjelenésétől ez a probléma már a múlté.

Megkönnyíti az új API-k tervezését

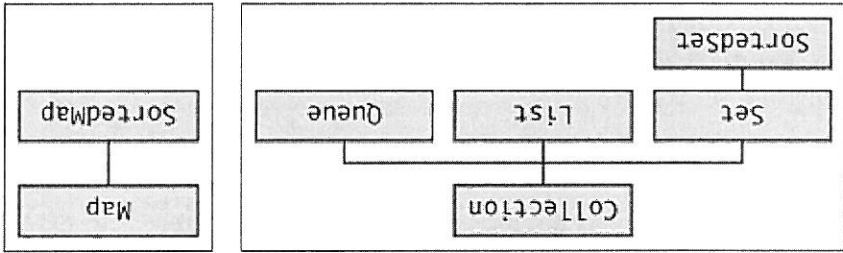
A tervezőknek és a kivitelezőknek nem kell újra kidolgozni az API-t, valahányszor készítenek egy gyűjteményekre alapozott API-t, hanem az általános gyűjtemény interfészt használhatják.

Elősegíti a szoftver újrafelhasználhatóságát

Az gyűjtemény interfészekkel összhangba hozott új adatstruktúrák természetesen újra felhasználhatók. Hasonlóan, új algoritmus fejlesztésekor könnyen hozzákapcsolható lesz az összes létező eddigi megvalósításhoz.

21.2. Interfészek

A gyűjtemény interfészek különböző típusú gyűjtemények leírását teszik lehetővé, amint az alábbi ábra mutatja. Ezek az interfészek lehetővé teszik a különböző gyűjtemény-reprezentációk együttes kezelését. A gyűjtemény interfészek a Gyűjtemény keretrendszer alapjainak tekinthetők.



Ahogy az ábrán látható, a gyűjtemény interfészek hierarchiát alkotnak: a *Set* egy speciálisat fajtája a *Collection*-nek, a *SortedSet* specializált fajtája a *Set*-nek, és így tovább. A hierarchia két egymástól független fából áll: a *Map* nem *Collection* leszármazott. Vegyük figyelembe, hogy a gyűjtemény interfészek (általános, generikus) típus paraméterekkel dolgoznak. Például a *Collection* interfész deklarációja:

```
public interface Collection<E> ...
```

Az *<E>* szintaxis azt jelenti, hogy az interfész általános (generikus) típusal működik. Amikor deklarálunk egy *Collection*-t, meg tudjuk határozni, hogy milyen típusú objektumot tartalmaz a gyűjtemény. A típus paraméter a fordítóprogram számára lehetővé teszi, hogy csak az adott típusú objektumot engedje beírni a gyűjteménybe, így csökkenti a futásidejű hibák számát.

Mikor megértjük, hogyan használjuk ezeket az interfészeket, akkor megértettük a Gyűjtemény keretrendszer lényegét. Ez a fejezet be fogja mutatni az általános irányelveket, hogy hogyan lehet hatékonyan használni ezeket az interfészeket. Továbbá irányt ad, hogy mikor melyik interfészt használjuk. Minden egyes interfészhez tanulni fogunk programozási trükköket, hogy segítsen megoldani a legtöbb későbbi problémát.

21.2.1 A gyűjtemény interfészek

Collection

A gyűjtemény hierarchia gyökere. A *Collection* objektumok csoportját reprezentálja, amiket elemeknek hívjuk. A gyűjtemény interfész a legkisebb közös nevező a gyűjtemény implementációk között, és akkor érdemes ezt választani, ha a lehető legnagyobb rugalmasságra van szükség. A gyűjtemények néhány típusa engedélyezi az elemek duplikálását, a többi pedig nem. A gyűjtemények néhány típusa rendezett, a többi nem rendezett. A Java platform nem biztosít közvetlen implementációt ehhez az interfészhez, ehelyett a sokkal specifikusabb al-interfészeket implementálja.

Set

Ez a gyűjtemény nem tartalmazhat duplikált elemeket. Ezt az interfészt halmazok tárolására szokták használni, mint például a futó processzek halmaza.

List

A *List* rendezettséget biztosító gyűjtemény, néha szekvenciának is hívják. A listák tartalmazhatnak duplikált elemeket. A lista felhasználója egész index segítségével hozzá tud férni az elemekhez (hasonlóan a tömbök indexeléséhez), valamint lehetőséget kap a lista egyszerű bejárásához.

Queue

Típkusan ez a gyűjtemény szokta tárolni a feldolgozásra váró elemeket. A *Collection* alapműveletein felül lehetőséget ad további beszurási, kivételi, és megtekintési műveletekre.

Legtöbbször, de nem szükségképpen az elemeket FIFO (*first-in-first-out*) elv szerint rendezi. A kivételek között van a prioritási sor, amelyik rendezi az elemeket valamilyen szempont alapján. Bármilyen rendezést használunk, a sor elején van az az elem, amit legelőször el szeretnénk távolítani a *remove* vagy a *poll* metódus meghívásával. A FIFO sor minden új elemet a sor végére illeszt be. Más fajta sorok használhatnak más elhelyezési szabályokat.

Megjegyzés: A *Queue* gyűjteményekkel terjedelmi okokból a továbbiakban nem foglalkozunk.

Map

A kulcsokat értékek képezi le. A leképezések nem tartalmazhatnak duplikált kulcsokat. Minden egyes kulcs legfeljebb egy értéket tud leképezni. Másként fogalmazva a *Map* kulcs-érték párokat tárol.

Az utolsó két gyűjtemény interfész csupán rendezett verziója a *Set*-nek és a *Map*-nek:

SortedSet

Az elemek növekvő sorrendben tárolását teszi lehetővé. Számos kiegészítő művelettel biztosítja, hogy kihasználhassuk a rendezettséget. Ezt az interfészt szokták használni olyan rendezett halmazként, mint például egy szöveges lista vagy tagsági lista.

SortedMap

A leképezések kulcs szerinti növekvő sorrendbe tárolását teszi lehetővé. A rendezett leképezést használjuk a rendezett gyűjtemények kulcs/érték páryainál. Ilyen például a szótár vagy a telefonkönyv.

21.2.2 A *Collection* interfész

A *Collection* az objektumok (elemek) csoportját tárolja.

```
public interface Collection<E> extends Iterable<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // optional
    boolean remove(Object element); // optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection<? c>);
    boolean addAll(Collection<? extends E> c); // optional
    boolean removeAll(Collection<? c>); // optional
    boolean retainAll(Collection<? c>); // optional
    void clear(); // optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Az interfésznek van metódusa, ami megmondja hány elem van a gyűjteményben (*size*, *isEmpty*), leellenőrizi, hogy az adott objektum benne van-e a gyűjteményben (*contains*), hogy hozzáadjon elemet gyűjteményhez vagy elvegye belőle (*add*, *remove*), bejártot (más néven iterátort) szolgáltat a bejáráshoz (*iterator*). Az *add* metódus *boolean* visszatérési értéke azt jelzi, hogy történt-e a gyűjteményben változás. Ugyanis olyan implementáció esetén, ahol az elemek többszörös tárolása nem megengedett, nem történik változás, ha az elemet már eleve tartalmazta a gyűjtemény. Hasonlóan, a *remove* metódus esetén csak akkor változik a gyűjtemény állapota, ha benne volt az adott elem, és így sikeres volt a kivétel.

A gyűjtemények bejárása

Két módja van hogy bejárjuk a gyűjteményeket. Az egyik a *for-each* ciklussal, a másik az iterátorok használatával.

For-each ciklus

A *for-each* ciklus megengedi, hogy bejárjuk a gyűjteményt vagy tömböt a *for* ciklus használatával. A következő kód a *for-each* ciklust használja, és minden elemet külön sorban jelenít meg:

```
For (Object o : collection)
    System.out.println(o);
```

Iterátorok

Az iterátor egy olyan objektum, ami megengedi, hogy bejárjuk a gyűjteményt és eltávolítsuk az elemeket a gyűjteményből, ha akarjuk.

Az *Iterator* interfész:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional
}
```

A *hasNext* módszer visszatérési értéke *true*, ha az iterációnak van még be nem járt eleme. A *remove* módszer eltávolítja a gyűjteményből az utolsó elemet, amit a *next* hívásra kapunk. A *remove* módszer kivételt dob, ha még nem volt kiolvasott elem.

Az *Iterator.remove* az egyetlen biztonságos út, hogy módosítsuk a gyűjteményt az iteráció folyamán. A viselkedés meghatározatlan, ha a gyűjtemény más egyéb úton módosítva lett, amíg az iteráció folyamatban volt.

Iterátor használata célszertű a *for-each* ciklus helyett a következő esetekben:

- Törölni szeretnénk a bejárás közben.
- Ki szeretnénk cserélni az elemeket.
- Egyyszerre többféle bejárásra is szükség van.

A következő módszer mutatja, hogyan használjuk az iterátort szűrőnek egy tetszőleges gyűjteményben (ezzel áttekinthetjük, hogy a gyűjtemény hogyan távolítja el a specifikus elemeket):

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (!cond(i.next()))
            i.remove();
}
```

Ez az egyszerű kis kódreszt sokoldalú, ami azt jelenti, hogy működik bármely gyűjteményben, kivétel minden implementációban. Ez a példa bemutatja, milyen könnyű sokoldalú algoritmust írni a Gyűjtemény keretrendszer segítségével.

A Collection interfész tömeges módszerei

A tömeges módszerek nem egy elemmel, hanem egész gyűjteménnyel hajtanak végre műveleteket. Implementálni lehet ezeket a módszereket az alap módszereket használva, bár a legtöbb esetben az ilyen implementációk kevésbé hatékonyak.

A tömeges módszerek

- *containsAll* *true* a visszatérési értéke, ha a cél gyűjtemény tartalmaz minden elemet a meghatározott gyűjteményben (részhalmoz)
- *addAll* Minden elemet hozzáad a meghatározott gyűjteményhez (unió)
- *removeAll* Eltávolítja a célgyűjteményből az összes olyan elemet, amit a meghatározott gyűjtemény tartalmazott (különbség)
- *retainAll* Eltávolítja a célgyűjteményből az összes olyan elemet, amit a meghatározott gyűjtemény nem tartalmazott. Azaz, csak azokat tartja meg a célgyűjteményben, amiket a meghatározott gyűjtemény is tartalmazott. (metszet)
- *clear* Eltávolítja az összes elemet a gyűjteményből.

Az *addAll*, *removeAll*, és a *retainAll* metódus mindig *true*-val tér vissza, ha a célgyűjtemény módosítva volt a művelet végrehajtása alatt.

Egy egyszerű példa tömeges metódusok erejéről: a következők kifejezés eltávolítja az *e* elem minden példányát a *c* gyűjteményből:

```
c.removeAll(c.contains(e));
```

Különösen praktikus, ha le akarjuk törölni az összes *null* elemet a gyűjteményből:

```
c.removeAll(c.contains(null));
```

A kifejezés használja a *Collections.singleton* metódust, amelyik egy statikus gyártó metódus, ami visszaad egy olyan gyűjteményt, amely csak egy meghatározott elemet tartalmaz.

A Collection interfész tömb metódusai

A *toArray* metódusok olyanok, mint egy híd a gyűjtemények és a régebbi API-k között, amik tömböket várnak a bemeneten. A tömb metódusok lehetővé teszik, hogy a gyűjtemény tartalmát tömbként is elérhetővé tegyék. A paraméterek nélküli változat készíti el egy új *Object* tömböt. A bonyolultabb formája lehetővé teszi, hogy a hívó paraméterként egy tömböt adjon, és az eredmény is ilyen típusú tömb lesz.

Például, feltételezzük, hogy *c* egy gyűjtemény. Ekkor a következő metódus létrehoz egy tömböt, amelynek métere megegyezik a gyűjtemény méretével, elemei pedig a gyűjtemény elemei:

```
Object[] a = c.toArray();
```

Tegyük fel, hogy *c* ismert és csak sztringet tartalmaz (c típusa *Collection<String>*). Ekkor használható a következő változat is:

```
String[] a = c.toArray(new String[0]);
```

21.2.3 A Set interfész

A *Set* egy speciális *Collection*, amely nem tartalmaz duplikált elemeket. A *Set* nem tartalmaz új metódust a *Collection*-tól örököltékhez képest. A *Set* a tartalmazott objektumok

equals és a *hashCode* módszerét használja arra, hogy a többszörös tárolást kiszűrje. Két *Set* objektum akkor egyenlő, ha ugyanazon elemeket tartalmaznak. A *Set* interfész:

```
public interface Set<E> extends Collection<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // optional
    boolean remove(Object element); // optional
    Iterator iterator();
    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // optional
    boolean removeAll(Collection<?> c); // optional
    boolean retainAll(Collection<?> c); // optional
    void clear();
    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

A Java platform három általános *Set* implementációt tartalmaz: a *HashSet*, *TreeSet*, és *LinkedHashSet*. *HashSet* esetén az elemeket egy hash táblában tárolja, ez a legjobb választás, ha az elemek bejárása nem fontos. A *TreeSet* az elemeket egy piros-fekete fában tárolja, ami az elemek bejárását hatékonyan meg tudja valósítani, de egy elem elérése, beszűrése lassabb, mint a *HashSet* esetén. A *LinkedHashSet* ötvözi a láncolt listákat a hash táblával.

Nézzünk egy egyszerű példát. Feltételezzük, hogy van egy *c* nevű *Collection*-ünk, és szeretnénk még egy *Collection*-t létrehozni, amely ugyanazokat az elemeket tartalmazza, de minden elemet csak egyszeresen.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

Létejön egy *Set*, amely nem tartalmaz duplikált elemeket, de a *c* összes eleme benne van. Az alábbi lemezt konverziós konstruktor használjuk a tényleges létrehozásra.

A Set interfész alapvető műveletei

A *size* módszer visszatér az elemek számával. Az *add* módszer a *Set*-hez adja a megadott elemet, ha nem tudja, akkor *false* értékkel tér vissza. A *remove* módszer eltávolítja a megadott elemet a listából, ha ez sikerült, akkor *false* értékkel tér vissza. Az *iterator* módszer visszaad egy *Iterator* objektumot.

A következő példa a paraméterként kapott szavakat eltávolítja az *s* objektumban, kinyíja a duplikált szavakat, végül az eltérő szavak számát, és a szavakat:

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate: " + a);
        System.out.println(s.size()+" distinct words: "+s);
    }
}
```

Futtassuk a programot:

```
Java FindDups t came t saw t left
```

A kimenet a következő lesz:

```
Duplicate: t
Duplicate: t
4 distinct words: [t, left, saw, came]
```

Megjegyezzük, hogy a referencia típusát szokás interfész típussal (*Set*) leírni, míg az objektum típusa egy implementációs osztály lesz (*HashSet*). Ez egy nagyon ajánlott programozási gyakorlat, amivel könnyebb rugalmas kódot készíteni. Ha később esetleg abc sorrendben akarjuk a szavakat kiírni, csupán le kell cserélni *HashSet*-et *TreeSet*-re. Ez az egyszerű, egy soros változtatás ilyen eredményt produkál:

```
Duplicate word: t
Duplicate word: t
4 distinct words: [came, t, left, saw]
```

A Set interfész tömeges módszerei

A tömeges módszerek különösen jó szolgálatot tesznek a *Set* esetén, hiszen így egyszerűen megvalósíthatók az algebrából ismert halmazműveletek:

- *si.containsAll(s2):* Igaz logikai értékekkel tér vissza, ha *s2* részhalmaza *si*-nek, ha *si* tartalmazza az *s2* összes elemét).

- *si.addAll(s2):* Az *si*-be *s1* és *s2* uniója kerül.

- *si.retainAll(s2):* Az *si*-be *si* és *s2* metszete kerül.

- *si.removeAll(s2):*

Az *si*-be *si* és *s2* különbsége (*si \ s2*) kerül.

Ha azt szeretnénk, hogy az unió, metszet vagy különbség képzés során egyik halmaz se módosuljon (ahogy az matematikailag is várható), akkor a következő módon kell alkalmazni a módszereket:

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);
Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

Nézzük meg újra a *FindDups* programot. Azt akarjuk tudni, hogy melyik szavak fordulnak elő egyszer a vitatott listában, és mely szavak fordulnak elő többször, de nem akarnunk duplán kiírt szavakat látni. Ezt úgy tudjuk elérni, hogy két halmazt hozunk létre: az egyik minden elemet tartalmaz, a másik csak a duplikáltakat. Nézzük a programot:

```
import java.util.*;
public class FindDups2 {
    public static void main(String args[]) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();
```

```

for (String a : args)
    if (!uniques.add(a))
        dups.add(a);
// Destructive set-difference
uniques.removeAll(dups);
System.out.println("Unique words: " + uniques);
System.out.println("Duplicate words: " + dups);
}
}

```

Ha lefuttatjuk az előző parancssori paraméterlistával (*i came i saw i left*), akkor a következő eredményt kapjuk:

```

Unique words: [left, saw, came]
Duplicate words: [i]

```

A halmazelméleti szimmetrikus differencia a következőképpen állítható elő:

```

Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2);
symmetricDiff.removeAll(tmp);

```

A *symmetricDiff*-ben először a két halmaz unióját állítjuk elő az *addAll* módszerrel, majd a *tmp*-ben előállított metszetet kivonjuk belőle a *removeAll* módszerrel.

A Set interfész tömb módszerei

Ezek a módszerek semmilyen nem térnek el a *Collection*-nál leírtaktól.

21.2.4 A List interfész

A *List* egy rendezett *Collection*, a *List* duplikáltan is tartalmazhat elemeket. A *Collection*-ból örökölt módszerekön kívül a *List* interfész a következő lehetőségeket tartalmazza:

- Pozíció szerinti elérés: Az elemek a listában betöltött helyük alapján is elérhetők.
- Keresés: A megadott elemet kikeresi a listából, és visszaadja a pozícióját.
- Bejárás: Az *Iterator* alapú szolgáltatások bővültek.
- Részlista: Lehetővé tesz részlista műveleteket.

A *List* interfész:

```

public interface List<E> extends Collection<E> {
    // Positional Access
    E get(int index);
    boolean add(E element); // Optional
    void add(int index, E element); // Optional
    E remove(int index); // Optional
    abstract boolean addAll(int index,
        Collection<E> c); // Optional
    int indexOf(Object o);
    int lastIndexOf(Object o);
}

```

A Java platform két alapvető *List* megvalósítást tartalmaz: *ArrayList* és *LinkedList*.

```
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);
List<E> subList(int from, int to);
```

Collection metódusok

A metódusok öröklődnek a *Collection*-ből, és mindent az elvárásainknak megfelelően használhatunk.

A *remove* metódus mindig az első előforduló elemet törli a listából. Az *add* és *addAll* metódusnál az elem a lista végére kerül. Például a *list* végére másolja a *list2* elemét.

```
list1.addAll(list2);
```

A következő kód elkészíti a *list3* nevű listát, amelynek eredménye ugyanaz, mint az előzőnek, de nem rontja el *list1*-et:

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

Két *List* objektum akkor egyenlő, ha ugyanazok az elemek ugyanolyan sorrendben fordulnak elő benne.

A pozíció szerinti elérés és keresés metódusai

A *set* és *remove* metódusok egy adott elemhez pozicionálnak és felülről, vagy eltávolítják azt. A keresési metódusok (*indexOf* és *lastIndexOf*) viselkedése tökéletesen megegyezik *String* osztállyal megismertekkel.

Az *addAll* metódus az adott *Collection* elemét elhelyezi az adott pozíciótól kezdődően. Az elemeket sorrendben helyezi el egy *Iterator* segítségével.

A következő metódus megcseréli a két értéket a listában:

```
static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

A következő metódus arra használja a *swap*-et, hogy véletlenszerűen keverje az elemeket:

```
static void shuffle(List<? List> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

Az algoritmus találomra felcseréli az adott *list* elemeit. Ez egy egyszerű módszer a keverésre. Másrészt az összes keverés valószínűsége megegyezik, feltéve, hogy a forrás is véletlenszerű, valamint gyors (csak *list.size()-1* csere történt). A következő program bemutató egy példa felhasználást:

```

import java.util.*;
public class Shuffle {
    public static void main(String args[]) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}

```

A programot még rövidebbé és gyorsabbá tudjuk tenni. Az *Array* osztálynak van egy *asList* nevű statikus módszere, ami egy tömb elemeit *List*-ként is elérhetővé teszi. Ez a módszer nem másolja át a tömb elemeit, csupán referenciákkal dolgozik, és ugyanazok az elemek a listából és a tömbből is elérhetők.

A visszaadott *List* objektum nem tartalmazza az opcionális *add* és *remove* metódusokat, hiszen a tömb mérete nem változhat. Nézzük meg a következő kódot:

```

import java.util.*;
public class Shuffle {
    public static void main(String args[]) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}

```

Iterátorok

A *List* interfész esetén természetesen használható az őstől örökölt módszer a bejárásra, de van egy ennél több lehetőséget nyújtó megoldás is: a *ListIterator*. Ez az interfész megengedi a lista kétirányú bejárását és az ehhez kapcsolódó további műveleteket. A *ListIterator* interfész:

```

public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); // Optional
    void set(E o); // Optional
    void add(E o); // Optional
}

```

Három metódusát (*hasNext*, *next* és *remove*) az *Iterator*-tól örökölte, a *hasPrevious* és *previous* a visszatérő történet bejáratát támogatja. A következő példa visszafelé járja be a listát:

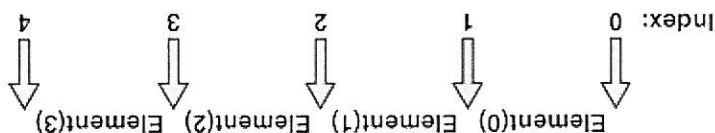
```

for (ListIterator<Type> i = list.listIterator(list.size());
     i.hasPrevious(); ) {
    Type t = i.previous();
    ...
}

```

A *List* interfész kétféle paraméterezéssel tud *ListIterator*-t gyártani. Paraméter nélkül a lista elejére pozicionál, míg az *int* paramétert váró verzió esetén a paraméter alapján pozicionál *ListIterator* tér vissza a *listIterator* módszer.

Az index vagy kurzor pozíció a következő alapján értendő: *n* hosszúságú lista esetén *n+1* érvenyes értéke van az indexnek, 0-tól *n*-ig minden egész. Az index mindig két elem között van a következő ábra szerint:



Nézzük meg az *indexOf* működését:

```

public int indexOf(E o) {
    for (ListIterator<E> i = listIterator(); i.hasNext(); )
        if (o==null ? i.next()==null : o.equals(i.next()))
            return i.previousIndex();
    return -1; // Object not found
}
  
```

Ha az *indexOf* módszer visszatérési értéke *i.previousIndex()*, akkor megtalálta a keresett objektumot.

Az *Iterator* interfész szolgáltatása a *remove* módszer, mely a gyűjteményből eltávolítja az adott elemet. A *ListIterator* interfész további két módszert nyújt, amely módosítja a listát: *set* és *add*. A *set* módszer felülrja az aktuális elemet. A következő módszer használja *set* módszert az adott elem összes előfordulásának cserélese során:

```

public static <E> void replace(List<E> s, E val, E newVal) {
    for (ListIterator<E> i = s.listIterator(); i.hasNext(); )
        if (val==null ? i.next()==null : val.equals(i.next()))
            i.set(newVal);
}
  
```

Egy kis trükk van ebben a példában a *val* és az *i.next()* egyenlőség vizsgálatában. Ha a *val* értéke *null*, akkor a *NullPointerException* kivétel nem váltódik ki, hiszen ha *val* értéke *null* lenne, akkor a feltételes operátor másik ága értékelődik ki.

Az *add* módszerrel új elemeket adhatunk a *List*-hez az aktuális kurzorpozíció elé. A következő példa bemutatja, hogy hogyan lehet a *val* minden előfordulását kicserélni a *newVals* listával:

```

public static <E> void replace(List<E> s, E val,
                               List<E> newVals) {
    for (ListIterator<E> i = s.listIterator(); i.hasNext(); )
        if (val==null ? i.next()==null : val.equals(i.next())) {
            i.remove();
            for (E e : newVals)
                i.add(e);
        }
}
  
```

Részlista művelet

A *subList(int fromIndex, int toIndex)* részlista módszer visszaadja a lista egy szeletét, a paraméterként megadott indexek figyelembevételével: *fromIndex* része, de *toIndex* nem

része a visszaadott szelvények. (Emlékeztetőül: a *String* osztály *substring* módszere is így működik.) Ezért gyakori példa a következő:

```
for (int i = fromIndex; i < toIndex; i++) {
    ...
}
```

A visszaadott lista kapcsolatban marad az eredeti listával. Így bármilyen módosítás történik a részlistán, az hatással lesz az eredetire is, sőt ez fordítva is igaz. Ezért nincs szükség további részlista módszerekre. Nézzünk erre egy olyan példát, amikor a lista egy részét akarjuk törölni:

```
list.subList(fromIndex, toIndex).clear();
```

A következő példák csak egy részlistában keresnek:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Arra érdemes figyelni, hogy itt a visszaadott indexek a talált elemek részlistabeli indexeit jelentik, nem az eredeti *list*-belieket.

A következő módszer is a *subList*-et használja, hogy könnyítse a munkát. A feladata az, hogy a lista egy adott méretű végét levágja, és egyben vissza is adja azt. Másként fogalmazva egy index mentén történő szétvágrásról van szó:

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```

A következő program az előző *dealHand* módszert használja a *Collection.shuffle*-vel együtt, hogy 52 lapból összesen 1e. A program 2 parametert használ: a kezek számát és az egy kézbe osztandó lapok számát:

```
import java.util.*;

class Deal {
    public static void main(String[] args) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);

        // Make a normal 52-card deck
        String[] suit = new String[] {
            "spades", "hearts", "diamonds", "clubs"
        };
        String[] rank = new String[] {
            "ace", "2", "3", "4", "5", "6", "7", "8",
            "9", "10", "jack", "queen", "king"
        };
        List<String> deck = new ArrayList<String>();
        for (int i = 0; i < suit.length; i++)
            for (int j = 0; j < rank.length; j++)
                deck.add(rank[j] + " of " + suit[i]);
        Collections.shuffle(deck);

        for (int i=0; i<numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }
}
```


A fűtő program ezeket az adatokat jeleníti meg:

```
% java Deal 4 5
[8 of hearts, jack of spades, 3 of spades, 4 of spades,
 king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,
 queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,
 9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,
 ace of hearts]
```

List algoritmusok

A *Collections* osztály nagyon hatékonyan használható algoritmusokat nyújt listák kezelésére. A következő lista csak felsorolja a fontosabbakat:

- *sort*: Rendezi a listát.
- *shuffle*: Véletlenszerűen felcserél elemeket a listában. (Permutál.)
- *reverse*: Megfordítja az elemek sorrendjét a listában.
- *rotate*: Megforgatja az elemeket.
- *swap*: Felcserél két meghatározott pozícióban lévő elemet.
- *replaceAll*: Az összes előforduló elemet kicseréli egy másikra.
- *fill*: Felülrja az összes elemet egy meghatározott elemre.
- *copy*: Átmásolja a forráslistát egy céllistába.
- *binarySearch*: Egy elemeket keres a bináris keresési algoritmust használva.
- *indexOfSubList*: Visszatér az első olyan indexszel, amelynél kezdődő részlista lő a másik listával.
- *lastIndexOfSubList*: Visszatér az utolsó olyan indexszel, amelynél kezdődő részlista egyenlő a másik listával.

21.2.5 Map interfész

A *Map* olyan tároló, ami kulcs-érték párokat tartalmaz. A *Map* nem tud tárolni duplikált kulcsokat, egy kulcshoz csak egy érték rendelhető. A *Map* interfész:

```
public interface Map {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    void putAll(Map<? extends K, ? extends V> t);
    void clear();
}
```

```

public Set<K> keySet();
public Collection<V> values();
public Set<Map.Entry<K,V>> entrySet();
public interface Entry {
    K getKey();
    V getValue();
    V setValue(V value);
}

```

A Java platform három különböző *Map* implementációt tartalmaz: *HashMap*, *TreeMap*, *LinkedHashMap*.

A *Map* interfész főbb metódusai

A metódusok (*put*, *get*, *containsKey*, *containsValue*, *size*, és *isEmpty*) hasonlóképpen működnek, mint a *Collection*-nál. A következő program kilistázza a szövegben szereplő szavakat és azok gyakoriságát.

```

import java.util.*;

public class Freq {
    public static void main(String args[]) {
        Map<String, Integer> m =
            new HashMap<String, Integer>();
        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}

```

A program a következőképpen futtatható:

```

Java Freq if it is to be it is up to me to delegate

```

A program kimenete:

```

8 distinct words:
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

Ha betűrendi sorrendbe akarjuk rendezni, akkor a HashMap-et TreeMap-re kell cserélni, és a következőket írja ki a képernyőre:

8 distinct words:
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}

Ha a felbukkanás sorrendjére vagyunk kíváncsiak, akkor ez a LinkedHashMap-el te-
gyük, és a kimenet a következő lesz:

8 distinct words:
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}

```

Két *Map* objektum könnyedén összehasonlítható. Akkor egyenlők, ha a kulcs-érték páro-
jaik megegyeznek. (A sorrend nem számít.)

A *Map* konstruktora egy új *Map* objektumot hoz létre egy már meglévő *Map* objektum
tartalmával.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

A Map interfész tömeges műveletei

A *clear* módszer törli a *Map* teljes tartalmát. A *putAll* metódussal átmásolható egy másik objektum tartalma. A következő példában a paraméterként kapott két *Map* unióját állítjuk elő:

```
static <K, V> Map<K, V> new AttributeMap(
    Map<K, V> defaults, Map<K, V> overrides) {
    Map<K, V> result = new HashMap<K, V>(defaults);
    result.putAll(overrides);
    return result;
}
```

Collection nézetek

A *Map* tartalmát háromféle szempont alapján nézhetjük vissza:

- *keySet*: kulcsok halmaza (*Set*)

- *values*: értékek gyűjteménye (*Collection*)

- *entrySet*: kulcs-érték párok halmaza (*Set*)

A következő példa a kulcsok halmazának kiírása:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

Ugyanez iterátorral:

```
for (Iterator<Type> i=m.keySet().iterator(); i.hasNext(); )
    if (i.next().isBogus())
        i.remove();
```

Kulcs-érték párok kiírása:

```
for (Map.Entry<KeyType, ValueType> e : m.entrySet())
    System.out.println(e.getKey() + " : " + e.getValue());
```

21.2.6 Objektumok rendezése

Rendezzünk egy tetszőleges tartalmú listát:

```
Collections.sort(l);
```

Ha a lista *String*-eket tartalmaz, akkor azt (Unicode-beli) abc sorrendbe rendezhetjük. Ha pedig Dátum (*Date*) tagokat tartalmaz, akkor időrendi sorrendbe. Hogyan történik ez? A *String* és a *Date* is megvalósítja a *Comparable* interfészt. Ez az interfész tartalmazza a legáltalánosabb rendezési eljárásokat, amely megegyezik az osztálynak, hogy automatikusan rendezve legyen valamely szempont szerint. A következő táblázat összefoglalja a legfontosabb osztályokat, amelyek az interfészt megvalósítják.

Osztály	Alapértelmezett sorrend
Byte	előjeles szám
Character	előjel nélküli szám
Long	előjeles szám
Integer	előjeles szám
Short	előjeles szám
Double	előjeles szám
Float	előjeles szám
BigInteger	előjeles szám
BigDecimal	előjeles szám
Boolean	false < true
File	rendszerfüggő abc szerinti a teljes név alapján
String	abc szerinti
Date	időrendi
CollationKey	abc szerinti a helyi jellemzők alapján

Ha a listánk olyan objektumokat tartalmaz, amelyek nem valósítják meg a *Comparable* interfészt, akkor a *Collections.sort(list)* hívás *ClassCastException* kivételt fog dobni.

Saját összehasonlítható osztály létrehozása

A *Comparable* interfész egyetlen metódust tartalmaz:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

A *compareTo* metódus összehasonlíja az objektumot az átvett objektummal, és vissza-
terési értéke negatív egész, nulla, vagy pozitív egész, ha az átvett objektum kisebb,
egyenlő vagy nagyobb, mint az átvett objektum. Ha a metódus nem tudja összehasonlí-
tani a két objektumot, akkor *ClassCastException*-t dob.

A következő osztály emberek nevét tartalmazza összehasonlító eszközökkel kiegészítve:

```
import java.util.*;  
public final class Name implements Comparable<Name> {  
    private final String firstName, lastName;
```

Ebben a rövid példában az osztály némileg korlátozott: nem támogatja a középső nevet, kötelező megadni a vezeték- és keresztnévet, és nem veszi figyelembe a nyelvi sajátosságokat. Azonban így is illusztrál néhány fontos pontot:

- A konstruktor ellenőrzi, hogy az paraméterei *null* értékek-e. Ez minden létrejövő *Name* objektum számára biztosítja, hogy jól formázott legyen, így semelyik más metódus nem dob *NullPointerException*-t.
- A *hashCode* metódus újradefiniált. (Azonos objektumoknak azonos hash kódjuknak kell lennie)
- Az *equals* metódus *false* értékkel tér vissza, ha a paraméterként kapott másik objektum *null*, vagy helytelen típus. Ilyen esetben a *compareTo* metódus futásidejétől kivétel dob.
- A *toString* metódus újradefiniálással a *Name* olvasható formában jeleníthető meg. Ez mindig jó ötlet, különösen olyan objektumok esetén, amiket gyűjteménybe helyezünk. A különböző típusú gyűjtemények *toString* metódusai jól olvasható formában jelenítik meg a gyűjtemények tartalmát.

- A *compareTo* metódus összehasonlítja az objektumok legfontosabb részét (*lastName*). A *compareTo* metódus összehasonlítja az objektumok legfontosabb részét (*lastName*). (Mint ahogy itt is, máskor is gyakran tudjuk használni a részek típusa szerinti alapértelmezett rendezést.) A *lastName* adattag *String* típusú, így az alapértelmezett rendezés pontosan megfelelő lesz. Ha az összehasonlítási eredmények nullától különbözőek, amely egyenlőséget jelent, kész vagyunk: csak vissza kell adni az eredményt. Ha a legfontosabb részek egyenlők, összehasonlítjuk a következőket (itt *firstName*). Ha ez alapján sem dönthető el a sorrend, továbblépünk.

```
public Name(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

public String firstName() { return firstName; }
public String lastName() { return lastName; }
public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name) o;
    return n.firstName.equals(firstName) &&
        n.lastName.equals(lastName);
}

public int hashCode() {
    return 31*firstName.hashCode() + lastName.hashCode();
}

public String toString() {
    return firstName + " " + lastName;
}

public int compareTo(Name n) {
    int lastCmp = lastName.compareTo(n.lastName);
    return (lastCmp != 0 ? lastCmp :
        firstName.compareTo(n.firstName));
}
```

A következő példa felépíti a nevek listáját:

```
import java.util.*;

public static void main(String[] args) {
    Name nameArray[] = {
        new Name("John", "Lennon"),
        new Name("Karl", "Marx"),
        new Name("Groucho", "Marx"),
        new Name("Oscar", "Groucho")
    };

    List<Name> names = Arrays.asList(nameArray);
    Collections.sort(names);
    System.out.println(names);
}
```

A program futása után a következőt írja ki:

```
[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]
```

A Comparator-ok

Hogyan tudjuk az objektumokat az alapértelmezett rendezéstől eltérő más sorrendbe rendezni? Ilyen esetben jön jól a *Comparator* interfész, amely egyetlen egyszerű metódust tartalmaz:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

A *compare* metódus összehasonlítja a két paramétert, visszatérési értéke negatív egész, nulla, vagy pozitív egész, ha az első paraméter kisebb, egyelő vagy nagyobb, mint a második. Ha valamelyik paraméter helytelen típusú a *Comparator* számára, a *compare* metódus *ClassCastException*-t dob.

Tegyük fel, hogy van egy *Employee* nevű osztályunk:

```
public class Employee implements Comparable<Employee> {
    public Name name() { ... }
    public int number() { ... }
    public Date hireDate() { ... }
    ...
}
```

Ha az *Employee* objektumok alapértelmezett rendezése a *Name* tag szerinti, akkor nem egyszerű a legmagasabb rangú dolgozókat kilikeresni a listából. A következő program elkészíti a listát:

```
import java.util.*;

class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER =
        new Comparator<Employee>() {
            public int compare(Employee e1, Employee e2) {
                return e2.hireDate().compareTo(e1.hireDate());
            }
        };
    // Employee Database
    static final Collection<Employee> employees = ... ;
}
```

```
public static void main(String[] args) {  
    List<Employee> e = new ArrayList<Employee>(employees);  
    Collections.sort(e, SENIORITY_ORDER);  
    System.out.println(e);  
}
```

21.3. Implementációk

Az implementációk az előző fejezetben leírt interfészeket valósítják meg (implementálják). Az alábbiakban az implementációk fajtáit láthatjuk.

- Általános célú implementációk:** a leggyakrabban használt implementációk, mindennapos használatra terveztek
 - Speciális célú implementációk:** arra terveztek, hogy speciális helyzetekben esetben használjuk. Nem szabványos teljesítmény karakterisztikák megjelentésére, használati és viselkedési korlátozásokra.
 - Konkurens implementációk:** erős konkurencia támogatására terveztek, többnyire az egyszerű teljesítmény rovására. Ezek az implementációk a `java.util.concurrent` csomag részét képezik.
 - Csomagoló implementációk:** a többi implementációval összekombinálva használhatók (gyakran az általános célúval), hogy extra funkcionálitást biztosítsanak, vagy bizonyos funkciókat korlátozzanak
 - Kényelmi implementációk:** mini implementációk, tipikusan a gyártó módszerek keresztlétrehozhatók el. Kényelmes és hatékony alternatívát biztosítanak az általános célú implementációk számára speciális esetekben.
 - Absztrakci implementációk:** vázlatos implementációk, amik megkönnyítik az egyéni implementációk létrehozását.
- Az általános-célú implementációkat az alábbi táblázat foglalja össze:

	Hash	Átméretezhető	Fa	Láncolt	Láncolt lista	Hasító tábla + Láncolt lista
Set	HashSet			TreeSet		LinkedHashSet
List		ArrayList			LinkedList	
Map	HashMap			TreeMap		LinkedHashMap

A táblázatban látható hogy a *Set*, a *List* és a *Map* interfészek többfajta általános célú implementációja használható. A *SortedSet* és a *SortedMap* interfészeknek nincs saját sora a táblázatban, ezeknek az interfészeknek egy implementációja van (*TreeSet* és *TreeMap*). Az általános célú implementációk mind biztosítják az összes opcionális műveletet, amit az interfészek tartalmaznak. Mind egyik megengedi a *null* elemet mind a kulcsok, mind az értékek esetén. Egyik sem szinkronizált. Mind egyik szerializálható (*Serializable*), és biztosít egy publikus *clone* metódust.

A korábbival ellentétes elközelést jelent az a tény, hogy ezek az implementációk nem szinkronizáltak. A régebbi *Vector* és *Hashtable* gyűjtemények szinkronizáltak. A jelenle-

gi megközelítés azért alakult ki, mert a gyűjteményeket sűrűn használják olyan helyeken, ahol a szinkronizálhatóság nem jelent előnyt. Ilyenek például az egyszerű használat, csak olvasási használat, és ha egy olyan nagyobb adat objektum részét képezi, amely megcsinálja a saját szinkronizációját. Általános API tervezési tapasztalat az, hogy ne kötelezzük a felhasználót olyan szolgáttatás használatára, amit nem sokszor használ. Ráadásul a szükségtelen szinkronizáció adott körülmények között akár holtponthoz is eredményezhet.

Általános szabály, hogy programírásakor az interfészek, és nem az implementációkon kell gondolkodni. Ez az oka annak, hogy ebben a fejezetben nincsenek példa programok. Legtöbb esetben az implementáció megválasztása csak a teljesítményt befolyásolja. Ahogy azt a korábbiakban említettük, egy előnyben részesített programozói stílus az, ha válasszunk egy implementációt a *Collection* létrehozásakor, és rögtön hozzá rendeljük az új *Collection*-t egy olyan változéhoz, ami adott interfész típusú (vagy egy olyan eljárásnak adjuk át a *Collection*-t, ami egy adott interfész típusú paramétert vár). Így a program nem függ majd egy adott implementáció esetén az ahhoz hozzáadott új metódusoktól, ezáltal a programozó bármikor szabadon változtathatja az implementációt, mikor jobb teljesítményt szeretne elérni, vagy a működési részleteket módosítani.

21.3.1 Általános célú *Set* implementációk

Három általános célú *Set* implementáció létezik: *HashSet*, *TreeSet*, és *LinkedHashSet*. Általában egyszerű eldönteni azt, hogy melyiket kell használni. A *HashSet* sokkal gyorsabb, mint a *TreeSet* (konstans végrehajtási idő szemben a logaritmikus idővel a legtöbb művelet esetén), de nem biztosít rendezhetőséget. Ha a *SortedSet* interfész műveleteire, vagy egy érték szerinti rendezett iterációra van szükség, akkor a *TreeSet* használatát javasolt, egyéb esetben a *HashSet*.

A *LinkedHashSet* bizonyos értelemben egy átmenetet képez a *HashSet* és a *TreeSet* között. Megvalósítását tekintve ez egy olyan hash-tábla, aminek elemei egy láncolt lista segítségével vannak összekötve, így rendezett-beszűrős iterációt biztosít (a leggyakrabban használt elemet szúrja be a leggyűrűbben használtba), és majdnem olyan gyorsan fut, mint a *HashSet*. A *LinkedHashSet* implementációja megkíméli a használatát az olyan nem specifikált, általában kaotikus rendezéstől, mint a *HashSet* esetében, valamint a *TreeSet* esetében felmerülő többlet költségektől is.

Figyelemre méltó, hogy *HashSet* használat esetén az iterációk száma lineárisan arányos a benne tárolt bejegyzések számával, valamint a beszűrősek számával (a kapacitással). Emiatt egy túl magas kezdeti kapacitás esetén feleslegesen pazaroljuk az időt és a memóriát. Másrészt a túl alacsony kezdeti kapacitás választás esetén szintén időt veszünk azzal, hogy az adatstruktúrát mindig át kell másolnunk, mikor a kapacitást növeljük. Ha nem specifikálunk kezdeti kapacitást, akkor az alapbeállítás lép érvénybe, ami 16. A kapacitást mindig felfele kerekítjük a legközelebbi kettő hatványra. A kezdeti kapacitást a konstruktor *int* paramétereként adhatjuk meg. A következő kód sor egy olyan *HashSet* példány létrehozását mutatja be, ahol a kezdeti kapacitás 64.

```
| Set<String> s = new HashSet<String>(64);
```

A *LinkedHashSet*-nek hasonló beállítási paraméterei vannak, mint a *HashSet*-nek, de az iteráció idejét nem befolyásolja a kapacitás. A *TreeSet* nem rendelkezik beállítási paraméterekkel.

21.3.2 Altalános célú List implementációk

Két általános célú List implementáció létezik: *ArrayList* és *LinkedList*. Leggyakrabban valószerűleg az *ArrayList*-et fogjuk használni. Konstans elérési időt biztosít a lista elemeinek pozicionált eléréskor, egyszerű és gyors. Nem szükséges csomópont objektumot foglalni minden lista elemhez, és képes kihasználni a *System.arraycopy* nyújtotta előnyöket, amikor egyszerűen több listaelemet kell mozgatunk.

Abban az esetben, ha gyakran kell elemeket hozzáadni egy lista elejéhez, vagy iterálni a listán, hogy valamelyik belső elemét kitöröljünk, akkor érdemes *LinkedList*-t használni. Ezek a műveletek konstans idő alatt végrehajthatóak egy *LinkedList*-ben, míg egy *ArrayList*-ben lineáris idejűek, de ezért cserébe nagy árat kell fizetni a teljesítmény terén. A *List*-ben lineáris idejűek, de ezért cserébe nagy árat kell fizetni a teljesítmény terén. A pozicionált elérés lineáris idejű *LinkedList* esetén, míg ugyanez konstans *ArrayList*-nél, továbbá ez a konstans-faktor a *LinkedList* esetében sokkal rosszabb.

Az *ArrayList*-nek egy állítható paramétere a kezdeti kapacitása. Ez azt határozza meg, hogy hány eleme legyen az *ArrayList*-nek, mielőtt növelni kéne az elemek számát. A *LinkedList*-nek nincs állítható paramétere.

21.3.3 Altalános célú Map implementációk

A három általános célú Map Implementáció a *HashMap*, a *TreeMap* és a *LinkedHashMap*. Ha szükség van összegyűjtött információra, a *TreeMap*-et használjuk, ha a legjobb sebesség kell, és nem szükségesek az iterációk, használhatjuk a *HashMap*-et, ha az utóbbihoz hasonló sebesség kell és iteráció is, akkor használjuk *LinkedHashMap*-et. Ha hasonlóképp, a *Set* implementáció blokkban minden ugyanúgy működik, mint a *Map* implementációban.

21.4. Algoritmusok

Ebben a fejezetben a Java platform által kínált újra felhasználható, sokalakú algoritmusok egy részével foglalkozunk. Mindegyikük a *Collections* osztály statikus metódusa. Az algoritmusok széles skálája a *List*-eknél működik. De egy pár tetszőleges kollekciók esetében is működik.

21.4.1 Rendezés

A *sort* algoritmus újarendezi a listát. A művelet két formája adott. Az egyszerűbb forma veszi a listát, és rendezi azt az elemek természeténél (alapértelmezett) sorrendjében. Itt egy egyszerű program, ami kiríja betűrendben a parancssori paramétereket:

```
import java.util.*;

public class Sort {
    public static void main(String args[]) {
        List<String> l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

Futtassuk a programot:

```
java Sort i walk the line
```

A következő kimenetet kapjuk:

```
[1, line, walk]
```

A program megmutatta, hogy ezt az algoritmust nagyon könnyű használni.

Tegyük fel, hogy vannak anagrammákat tartalmazó szövegsítek. Szeretnénk kiírni azokat a listák hossza szerinti csökkenő sorrendben, de csak az adott mértéknél (pl. 8) nem kevesebb elemű listákat. A következő példa megmutatja, hogy hogyan érhetjük el ezt a *sort* metódus második formájának segítségével.

Maguk a szövegsítek egy *m* objektumban vannak tárolva. Eről leválogatjuk a megfelelő hosszúságúakat. Ezután a kód rendezi a listát, összehasonlítást végez, és végrehajtja a fordított méret szerinti rendezést. Végül a kód ismét végigfut a listán és kiírja az elemeket (a szavak csoportját):

```
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);
Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
        return o2.size() - o1.size();
    }
});
// Szócsoportok kiírása
for (List<String> l : winners)
    System.out.println(l.size() + " : " + l);
```

Futtassuk a programot, a minimum anagramma csoport méretével (8) együtt a következő kimenetet kapjuk:

```
12: [apers, apres, asper, pares, parse, pears, prase, presa,
rapes, reaps, spare, spear]
11: [alter, alters, artels, estral, laster, ratels, salter,
slater, staler, stelar, talers]
10: [least, setal, slate, stale, steal, stela, tael, tales,
teals, tesla]
9: [estrin, inerts, insert, inters, niters, nitres, sinter,
trins, trines]
9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,
secpa, spacer]
9: [palest, palets, pastel, petals, plates, pleats, septal,
staple, tepals]
9: [anestr, antsier, nastier, ratines, retains, retinas,
retsina, stainer, stearin]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares, sparse,
spears]
8: [enters, nester, reneat, rentes, resent, tenser, ternes,
treens]
8: [arles, earls, lares, laser, lears, tales, reals, seral]
8: [earings, erasing, gainers, regains, regains, reginas,
searing, seingna]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
8: [ates, east, eats, etas, seat, seta, teas]
8: [cates, cartes, caster, caters, crates, reacts, recast,
traces]
```

21.4.2 Keverés

A keverés algoritmus a bizonyos értelemben az ellenkezőjét teszi, mint amit a rendezés tesz. Lerombol bármilyen rendezést, ami esetleg található a listában. Ez az jelenti, hogy ez az algoritmus újrarendezi a listát úgy, hogy a véletlenszerűséget használja az összes azonos valószínűséggel bekövetkező permutációhoz, feltételezve a ténylegesen rendezetlen forrást. Például tudjuk használni a keverést egy pakli kártya megkeveréséhez a kártya objektumainak listájában. Ezenkívül hasznos tesztesetek, helyzetek generálásához is.

21.4.3 Keresés

A bináris keresés algoritmus (*binarySearch* metódus) elemeket keres a rendezett listában. Ennek az algoritmusnak két formája van. Az első vesz egy listát és egy elemet, amit keres (egy kereső kulcsot). Ez a forma feltételezi, hogy a lista elemei rendezve vannak a természetes rendezés szerint a tárolóban. A második forma nem az alapértelmezett rendezést alkalmazza, hanem még egy *Comparator* paramétert is vár.

A visszatérési érték azonos mindkét formában. Ha a lista tartalmazza keresett elemet, visszatér az indexével. Ha nem, a visszatérési érték negatív, és egyben utal arra is, hogy hol kellene lennie az elemnek, ha szerepelne a listában.

A következő példa keresi egy adott elem meglétét, és beilleszti a helyére, ha még nem szerepel:

```
int pos = Collections.binarySearch(list, key);
if (pos > 0)
    l.add(-pos-1);
```

21.5. Ellenőrző kérdések

- Mik a gyűjtemények?
- Milyen részei vannak a gyűjtemény keretrendszernek? Mire szolgálnak?
- Írjon példát olyan gyűjtemény interfészre, amely engedélyezi a többszörös tárolást!
- Írjon példát olyan gyűjtemény interfészre, amely nem engedélyezi a többszörös tárolást!
- Mi a *Set* interfész specialitása más gyűjteményekhez képest?
- Mi a *List* interfész specialitása más gyűjteményekhez képest?
- Mi a *Map* interfész specialitása más gyűjteményekhez képest?
- Mi az iterátor? Mire használhatjuk? Hogyan jön létre?

Igaz vagy hamis? Indokolja!

- A *Set* elemei indexelhetők.
- Egy *Iterator* objektum segítségével a kollekció elemei többször is bejárhatók.
- A *Comparator* segítségével egy *TreeSet* rendezettségétől is megadható.
- A *Set* implementációi minden esetben rendezettek.

- A *TreeSet* osztály implementálja a *Set* interfészt.
- Minden kollekció implementálja az *Iterator* interfészt.
- A kollekcióban lehet primitív adatokat is tárolni.

Melyik interfészt érdemes alkalmazni?

Olyan tároló objektumra van szükségünk, amelyikbe egyedi elemeket akarunk tárolni. A sorrend nem lényeges, de a többszörös tárolás semmiképpen sem megengedett.

- *Set*
- *List*
- *Map*

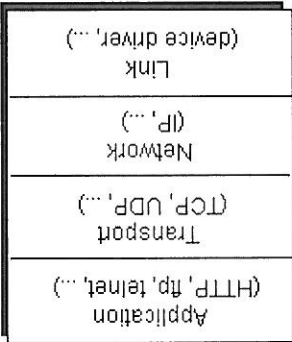
Mit tesz egy *Set* objektum annak érdekében, hogy ne legyenek duplikált elemek?

- Az *add* módszer kivétel dob, ha duplikált elemet akarunk beszúrni.
- Az *add* módszer *false* értéket ad vissza, ha duplikált elemet akarunk beszúrni.
- A duplikált értékeket a fordító kiszűri.

22.Hálózatkészítés

22.1. Hálózati alapok

Az Internetre kötött számítógépek vagy a TCP (*Transmission Control Protocol*) vagy az UDP (*User Datagram Protocol*) protokollt használják, amint azt az alábbi ábrán is látni lehet:



Ha egy Java programot írunk, ami használja a hálózatot, a felhasználói réteget programozzuk. Általában nincs szükség a TCP vagy az UDP protokollok közvetlen használatára, ehelyett használhatjuk az osztályokat a *java.net* csomagból. Ezek az osztályok rendszertüggelően hálózati kommunikációt biztosítanak. Ha szeretnénk tudni, hogy a programunk pontosan melyik osztályokat kell, hogy használja, meg kell érteni az alapvető külbőbbségeket a TCP és az UDP protokollok között.

22.1.1 TCP

Ha két program megbízhatóan akar egymással kommunikálni, létrehozhatunk egy kapcsolatot, és ezen keresztül küldik az adatokat, hasonlóan a telefon működéséhez. A TCP – akár egy telefonátvitel – garantálja, hogy az adat, amit küldünk, helyes sorrendben (i) megérkezik a vevőhöz. Ha ez nem sikerül, hibajelzést küld vissza.

A TCP egy megbízható pont-pont csatornát ad azoknak az alkalmazásoknak, amelyek megbízható kapcsolatot kívánnak meg. Ilyen alkalmazások például: *Hyper Text Transfer Protocol* (HTTP), *File Transfer Protocol* (FTP) és a *Telnet*. A hálózaton átküldött és fogadott adatok sorrendje elengedhetetlen az alkalmazások helyes működéséhez. Mikor a HTTP-t egy URL-ből való olvasáshoz használjuk, az adatokat olyan sorrendben kell kiolvasnunk, ahogy azt küldték, különben könnyen lehetne az eredmény egy összekuszált HTML fájl, hibás ZIP fájl, vagy valami más érvénytelen információ.

Definíció: a TCP (*Transmission Control Protocol*) egy kapcsolat alapú protokoll, ami az adatok megbízható folyamát adja két számítógép között.

22.1.2 UDP

Az UDP nem ad megbízható kapcsolatot egy hálózaton levő két számítógép között, mivel ez nem kapcsolat alapú, mint a TCP, inkább az adatok független csomagjait küldi az alkalmazások között. Ezeket a csomagokat adatcsomagoknak is hívjuk. A adatcsomagok küldése leginkább a levelek postán át való küldéséhez hasonlít: a szállítást nem garantált, a sorrend nem fontos, és mindegyik üzenet különbözik a másiktól.

Definíció: az **UDP** (*User Datagram Protocol*) egy olyan protokoll, ami az adatok olyan független csomagjait továbbítja egyik számítógépről a másikra, amiket adatcsomagoknak hívunk, és nincs garancia a megérkezésükre. Az UDP nem kapcsolat alapú, mit a TCP.

Számos alkalmazásnál a megbízhatóság garantálása kritikus, hogy az információ eljusson a hálózaton egyik végéről a másikra. Mindemellett a kommunikáció egyéb formái nem kívánnak meg olyan szigorú normákat.

Nézünk például egy órászervert, ami a pontos időt küldi el a klienseknek, ha azok igénylik. Ha a kliens hiányol egy csomagot, nem szükséges újraküldeni azt, mert az idő pontatlan lesz, ha második próbálkozásra kapja meg. Mikor a kliens két kérést irányít a szerver felé, és a csomagokat rossz sorrendben kapja, a kliens észreveszi ezt, és újabb kérést küld. A megbízható TCP protokoll ebben az esetben nem szükséges, mivel az a teljesítmény rovására mehet, és esetleg csak akadályozza a szolgáltatást.

A ping parancs egy másik kitűnő példa olyan szolgáltatásra, ami nem igényel megbízható csatornát. A parancs teszteli egy hálózaton levő két számítógép közötti kapcsolatot. A parancsnak tudnia kell, hogy a csomag megérkezett-e, hogy megállapítsa, működik-e a kapcsolat. Egy megbízható csatornán ez a szolgáltatás se működik megfelelően.

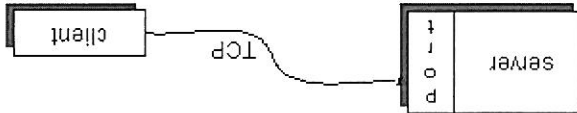
Megjegyzés: Sok tűzfal és router úgy van konfigurálva, hogy ne engedje át az UDP csomagokat. Ha problémánk van egy, a tűzfalon kívül levő szolgáltatáshoz való kapcsolódással, vagy a kliens nem tud hozzád csatlakozni, engedélyezzük az UDP kapcsolatokat.

22.1.3 A portokról általánosságban

Ha nagyvonalakban akarunk beszélni, azt mondjuk, a számítógép egy egyszerű fizikai kapcsolaton keresztül kapcsolódik a hálózatra. Minden adat ezen a kapcsolaton keresztül érkezik, bár az adatok a számítógép különböző programjait használják. Honnan tudja a számítógép, hogy melyik alkalmazásnak melyik adatot kell továbbítani? A portok használata által.

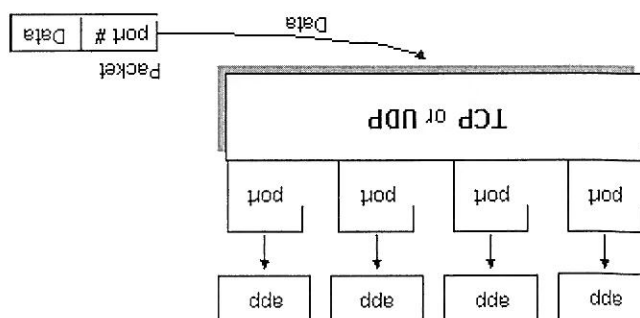
Az adat, amit az interneten keresztül küldenek, el van látva címzési információval, ami azonosítja a célszámítógépet és a portját. A számítógép a 32 bites IP címmel van azonosítva, melyet arra használunk, hogy az adat a megfelelő számítógépre érkezzen meg. A portot egy 16 bites számmal azonosítjuk, amit a TCP vagy UDP arra használ, hogy az adat a megfelelő programhoz jusson.

A kapcsolat alapú kommunikációnál, mint amilyen a TCP, a szerver program leköti egy foglalatot egy jellemző portnak. Ennek eredményeképpen a szerver megkap minden adatot, ami ezen a porton keresztül érkezik. A kliens ezután kommunikálhat a szerverrel a megadott porton keresztül, amit az alábbi ábra illusztrál:



Definíció: a TCP és az UDP protokollok portokat használnak, hogy a bejövő adatokat a számítógép megfelelő programjai felé irányítsák. Az adatcsomag alapú kommunikációnál, mit amilyen az UDP, az adatcsomag csomagok tartalmazzák a célállomás portszámát, és az UDP irányítja a megfelelő helyre a csomagot.

A portok a 0 - 65535 intervallumba kell, hogy essenek, mivel 16 bites számként vannak ábrázolva. A 0 és 1023 közötti portok fent vannak tartva olyan ismert szolgáltatásoknak, mint például a HTTP vagy az FTP vagy más rendszerszolgáltatás. Ezeket a portokat jól ismert portoknak hívjuk. A saját programjainknak nem szabad lekötni őket.



22.1.4 Hálózati osztályok a JDK-ban

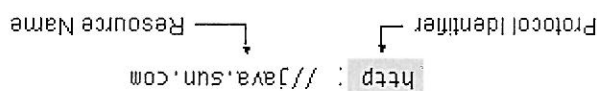
A Java programok használhatják a *java.net* osztályain keresztül a TCP vagy UDP protokollokat, hogy kommunikáljanak az Interneten. Az *URL*, *URLConnection*, *Socket* és *ServerSocket* osztályok mindegyike a TCP-n keresztül kommunikál, míg a *DataGramPacket*, *DataGramSocket* és *MulticastSocket* osztályok az UDP protokollt használják.

22.2. URL-ek kezelése

Ha már szűrőzött a weben, kétségtelenül hallott már az URL-ről, és használta is a HTML oldalakhoz való hozzáférésekhez. A legegyszerűbb arra gondolni, hogy az URL egy fájlra mutat a weben, azonban ez nem teljesen helyes megállapítás. Az URL más erőforrásokra is tud mutatni, például adatbázis lekérdezésekre és parancs kimenetekre.

Definíció: az **URL** (*Uniform Resource Locator*) az Interneten található erőforrásokra tud hivatkozni.

A következő ábra egy olyan URL-t mutat, ami a Java weboldalára mutat, amit a Sun Microsystems üzemeltet:



- Protokollazonosító (*Protocol identifier*)
- Erőforrás neve (*Resource name*)

Jegyezzük meg, hogy a protokoll azonosító és az erőforrás neve kettősponttal és két per-jelrel van elválasztva. A protokoll azonosító azt a protokollt jelzi, ami eléri a kívánt erőforrást. A fenti példa a *HyperText Transfer Protocol* (HTTP) használja, amit weboldalak eléréséhez használunk. A HTTP csak egy a sok különböző protokoll közül. Használjuk még a *File Transfer Protocol*-t (FTP), a *Gopher*, *File*, *News* protokollokat. Az erőforrás neve tartalmazza az erőforrás pontos címét. A formája függ a használt protokolltól, de a legtöbb protokoll esetében az erőforrás nevében benne van az alábbi komponensek közül legalább egy.

A legtöbb protokollnál a hosztnév és a fájlnev kötelező, míg a portszám és a hivatkozás csak opcionális. Például az erőforrás név egy HTTP típusú URL-ben megad egy szervert a hálózaton (hosztnév) és a dokumentum elérési útját a gépen (fájlnev), esetleg megad egy portszámot és egy hivatkozást (horgonyt). A Java weboldal URL-ben a *java.sun.com* a hosztnév.

22.2.1 URL objektum létrehozása

URL objektumot leggyakrabban egy olyan sztringből tudunk előállítani, ami ember számára olvasható formában tartalmazza az URL címet. Ez általában olyan alak, hogy más is felhasználhatja az URL-t. Például a Gamelan oldal URL-e, ami Java forrásokat használ, a következőképpen néz ki:

```
http://www.gamelan.com/
```

A Java programban a következőket kell használni, hogy URL objektumot hozzunk létre:

```
URL gamelan = new URL("http://www.gamelan.com/");
```

A fenti URL objektum egy abszolút URL. Az abszolút URL minden információt tartalmaz, ami szükséges lehet, hogy eljussunk a forrást. Ezen kívül létre lehet hozni objektumokat relatív URL címmel is.

Relatív URL létrehozása

A relatív URL-eket gyakran a HTML fájlokon belül használjuk. Például tegyük fel, hogy írtunk egy HTML fájlt, aminek a neve *JoeshomePage.html*. Az oldalon belül vannak linkek más oldalakra, pl. *PicturesOfme.html* és *MyKids.html*, amik ugyanazon a gépen és ugyanabban a könyvtárban vannak, mint a *JoeshomePage.html* fájl. A linkek meghatározhatóak fájlnevként, amit a következő programrészlet mutat be:

```
<a href="PicturesOfme.html">Pictures of Me</a>  
<a href="MyKids.html">Pictures of My Kids</a>
```

Ezek relatív URL címek, ahol az URL arra a szerverre hivatkozik, ahol a *JoeshomePage* oldal is található.

A Java programokban létrehozhatunk URL objektumokat relatív URL-ből is. Például tegyük fel, hogy ismerünk két URL-t a Gamelan oldalról.

```
http://www.gamelan.com/pages/Gamelan.game.html  
http://www.gamelan.com/pages/Gamelan.net.html
```

Hozzunk létre *URL* objektumot ezekből az oldalakból, hivatkozva a közös könyvtárakra (*http://www.gamelan.com/pages/*), mit itt:

```
URL gamelan = new URL("http://www.gamelan.com/pages/");  
URL gamelanGames = new URL(gamelan, "Gamelan.game.html");  
URL gamelanNetwork = new URL(gamelan, "Gamelan.net.html");
```

Ez a kódreszlet az *URL* azon konstruktorát használja, amivel létrehozhatunk egy *URL* objektumot egy másik alapján. A konstruktor általános alakja a következő:

```
URL(URL baseURL, String relativeURL)
```

Az első paraméter egy *URL* objektum, ami az új *URL* bázisát jelzi. A második paraméter egy sztring, ami specifikálja a bázisra hivatkozó forrás nevét. Ha a *baseURL* nincs megadva, a konstruktor abszolút *URL*-ként kezeli a *relativeURL* sztringet. Viszont, ha a *relativeURL* sztring egy abszolút *URL*, a konstruktor figyelmen kívül hagyja a *baseURL*-t.

Ez a konstruktor akkor is hasznos lehet, ha olyan *URL* objektumot akarunk létrehozni, amiben van horgony (*anchor*). Tegyük fel, hogy a *Gamelan.network.html* fájl tartalmaz egy horgonyt, amit *BOTTOM*-nak hívunk, és a fájl aljára mutat. A következő kód a relatív *URL* konstruktor segítségével használja az objektum létrehozására:

```
URL gamelanNetworkBottom = new URL(gamelanNetwork, "#BOTTOM");
```

Más *URL* konstruktorok

Az *URL* osztály tartalmaz két másik konstruktor is az *URL* objektumok létrehozásához. Ezek a konstruktorok akkor hasznosak, ha olyan *URL*-ekkel dolgozunk, mit például a HTTP, ami tartalmaz hosztnévet, fájlnevet, portszámot és hivatkozást (horgonyt) is. Ez a két konstruktor akkor is hasznos lehet, ha nincs egy olyan sztringünk, ami tartalmazza a komplett *URL*-t, de ismerjük néhány összetevőjét.

Például feltehetjük, hogy egy halózat-tallózó panelt fejlesztünk éppen, ami hasonlóan működik, mint egy fájl-tallózó, vagyis a felhasználó választhat protokollt, hosztnévet, portot és fájlnevet. Létrehozhatunk *URL*-t a panel komponenseiből. Az első konstruktor az *URL*-t a protokollból, a hosztnévből, és a fájlnevből állítja elő. Az alábbi példa a *Gamelan.network.html* fájlból (ami a *Gamelan* oldalon található meg) csinál *URL*-t.

```
new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");
```

Ez ekvivalens a következővel:

```
new URL("http://www.gamelan.com/pages/Gamelan.net.html");
```

Az első paraméter a protokoll, a második a hosztnév és az utolsó a fájl elérési útja. A fájl-nevnek perjellet kell kezdődnie. Ez jelzi, hogy a fájl gyökere a hosztnévben van megadva. A következő konstruktor egy portszámmal több paramétert tartalmaz az előzőhöz képest:

```
URL gamelan = new URL("http", "www.gamelan.com", 80, "pages/Gamelan.network.html");
```

Ez egy *URL* objektumot csinál a következő *URL*-nek:

```
http://www.gamelan.com:80/pages/Gamelan.network.html
```

Ha a fenti konstruktorok egyikével hozunk létre *URL* objektumot, a *toString* vagy *toExternalForm* paranccsal létrehozhatunk egy sztringet, amely tartalmazza a teljes *URL* címet.

MalformedURLException

A konstruktorok mindegyike *MalformedURLException* kivételt ad vissza, ha valamelyik paraméter null, vagy ismeretlen a protokoll. Ezt úgy tudjuk lekezelni, ha a konstruktor *try/catch* blokkba rakjuk, mit a következő kódban:

```

try {
    URL myURL = new URL( . . . )
} catch (MalformedURLException e) {
    . . . // Kivételkezelő kód
    . . .
}

```

Megjegyzés: Az URL objektumok egyszerűen írhatóak. Miután létrehoztunk egyet, nem lehet megváltoztatni az attribútumait (protokoll, hosztnev, fájlnev vagy portszám). Erdemes azt is megemlíteni, hogy az *URL* objektumok – hasonlóan a korábbi *File* objektumokhoz – a létrejöttükkor nem ellenőrzik az URL elérhetőséget, érvényességét stb., mivel az URL-nek egyelőre csak egy logikai reprezentációja.

22.2.2 URL elemzés

Az URL osztály több metódussal szolgál, amellyel kérdéseket tehetünk fel az *URL* objektumokrak. Egy *URL*-től megkaphatja a protokollt, a hoszt nevét, a port számát és a fájlnevet az alábbi metódusokat használva:

- *getProtocol*: Visszaadja az URL protokoll azonosító komponensét.
- *getHost*: Visszaadja az URL hoszt név komponensét.
- *getPort*: Visszaadja az URL port szám komponensét. A *getPort* metódus egy egészet ad vissza, ez a port szám. Ha a port nincs beállítva, akkor -1-et ad.
- *getFile*: Visszaadja az URL fájlnev komponensét.
- *getRef*: Visszaadja az URL hivatkozás komponensét.

Megjegyzés: Ne felejtse el, hogy egyes URL címek nem tartalmazzák ezeket a komponenseket. Az *URL* osztály azért nyújtja ezeket a metódusokat, mert a HTTP URL-ek és a leggyakrabban használt URL-ek tartalmazni szokták. Az *URL* osztály némiképp HTTP-centrikus.

Ezeket a *getXXX* metódusokat arra használhatja, hogy információt kapjon a URL-ről, tekintet nélkül a konstruktorra, amit az URL objektum elkészítésére használt.

Az *URL* osztály a metódusokkal együtt megszabadít az URL-ek állando újraelemzésétől. Ha adott egy tetszés szerinti *URL String*, csak készítenünk kell egy *URL* objektumot, és meghívni valamelyik metódusát a szükséges információkért. Ez a kis példaprogram készíti egy *URL*-t egy sztringből, és ezután az *URL* objektum metódusait használva elemzi az URL-t:

```

import java.net.*;
import java.io.*;

public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/books/tutorial/index.html#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = " + aURL.getHost());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("port = " + aURL.getPort());
        System.out.println("ref = " + aURL.getRef());
    }
}

```

Ez a program kimenete:

22.2.3 Közvetlen olvasás URL-ből

Miután sikeresen készítettünk egy *URL*-t, az *URL* *openStream* metódusát meghívva kap egy adatfolyamot, amiből kiolvashatja az *URL* tartalmát. Az *openStream* metódus viszszeresíti értéke egy *java.io.InputStream* objektum, így olyan könnyű az *URL*-ből olvasni, mint egy tetszőleges állományból.

A következő kis program az *openStream* használatával kap egy bemenő folyamot a <http://www.yahoo.com/URL-re>. Ezután nyit egy *BufferedReader*-t, ahonnan az *URL* tartalmát olvassa. Minden beolvasott sort átmásol a szabványos kimenetre:

```
import java.net.*;
import java.io.*;

public class URLReader {
    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                yahoo.openStream())));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

Amikor futtatjuk a programot, láthatjuk a parancs ablakában görgetve a HTML elemeket és a szöveges tartalmat a http://www.yahoo.com/címen_található_HTML_fájlból. Esetleg a program leállhat kivétellel. Ha a utóbbi esemény bekövetkezik, a programunkban be kell állítanunk a proxyt, hogy a program megtalálhassa a Yahoo szerveret.

22.2.4 Csatlakozás egy URL-hez

Miután sikeresen készítettünk egy *URL* objektumot, meghívhatjuk az *URL* objektum *openConnection* metódusát, hogy csatlakoztassuk. Amikor kapcsolódunk egy *URL*-hez, a hálózaton keresztül inicializálunk egy kommunikációs kapcsolatot a Java programja és az *URL* között. Például, nyithatunk egy kapcsolatot a Yahoo oldalra az alábbi kóddal:

```
try {
    URL yahoo = new URL("http://www.yahoo.com/");
    URLConnection yahooConnection = yahoo.openConnection();
    catch (MalformedURLException e) { // new URL() failed
    }
    catch (IOException e) {
        // openConnection() failed
    }
}
```

Ha lehetséges, az *openConnection* metódus készít egy új *URLConnection*-t, inicializálja, csatlakozik az *URL*-hez, és visszatér az *URLConnection* objektummal. Ha valami nem

megfelelő – például, a Yahoo szerver nem üzemel –, akkor az *openConnection* metódus egy *IOException*-t dob.

Miután sikeresen csatlakoztunk az URL-hez, írhatunk a csatlakozáshoz vagy olvashatunk róla az *URLConnection* objektum használatával.

Ha sikeresen használtuk az *openConnection*-t, hogy kommunikációt kezdeményezzen egy URL-lel, akkor van egy *URLConnection* objektumhivatkozásunk. Az *URLConnection* on egy HTTP-centrikus osztály; ennek sok metódusa csak akkor hasznos, ha HTTP URL-ekkel dolgozunk. Azonban a legtöbb URL protokoll megengedi, hogy olvasson és írjon a kapcsolatra.

Olvadás egy *URLConnection*-ről

Az alábbi program azonos funkciót hajt végre, mint az *URLConnection* program. De ahelyett, hogy bemenő folyamatot kapna közvetlenül az URL-ből, ez a program hatarozottan nyit egy kapcsolatot egy URL-hez, és egy bemenő folyamatot kap a kapcsolattól. Ekkor, mint az *URLConnection*, ez a program készít egy *BufferedReader*-t a bemenő folyamaton, és onnan olvas. A felkövető szakaszok eltérőek az előző és a mostani példákban.

```
import java.net.*;
import java.io.*;

public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

A program kimenete megegyezik az előző kimenettel. Használhatja mind a két módszer az URL-ből olvasásra. Azonban az *URLConnection*-ből olvasás sokkal használhatóbb lehet, mert az *URLConnection* objektumot használhatja más feladatokra is (például az URL-be írás) egy időben.

Még egyszer, ha a program leáll, vagy hibüzenetet lát, be kell állítani a proxyt, hogy a program megtalálhassa a Yahoo szerveret.

URLConnection-ba írás

Sok HTML lap tartalmaz URL-eket – szöveg mezőket és más GUI objektumokat –, hogy beküldje a kitöltött adatokat a szerverre. Miután begépelte a szükséges információkat és elküldi a kért gomb lenyomásával, a böngészőnk adatokat ír az URL-re a hálózaton keresztül. A szerver fogadja az adatokat, feldolgozza és visszaküld egy választ, általában egy új HTML oldal formájában.

Sok szkript használja a POST metódust a kliens adatainak továbbítására. A szerver-oldali szkriptek a POST metódust a bemenetük olvasására használják.

Egy Java program szintén kommunikálhat a szerver oldalon futó szkriptekkel. Egyszerűen lehet írni egy *URL*-be, így küldve adatokat a szerverre. Ezt a következő lépésekkel tudjuk megtenni:

- Készítünk egy *URL*-t.
- Nyitunk egy kapcsolatot az *URL*-hez.
- Beállítjuk a kimenő képességet az *URLConnection*-on.
- Kapunk egy kimenő folyamatot. Ez a kimenő folyamat összekapcsolt a szkript bemenő folyamatával a szerveren.
- Írunk a kimenő folyamatba.
- Bezárjuk a kimenő folyamatot.

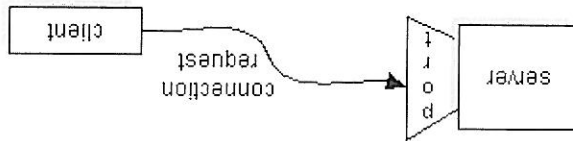
22.3. Socketek kezelése

A korábbi példák a kliens és szerver közötti magas szintű kommunikáció lehetőségét mutatták be. Gyakran van olyan szituáció, amikor a magas szintű megközelítés nem megfelelő. Ekkor alacsonyabb szintű, úgynevezett socket alapú kommunikációra van szükség.

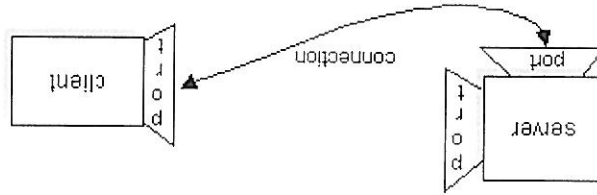
22.3.1 Mi az a socket?

Normális esetben a szerver egy speciális számítógépen fut és van egy socket-je, ami egy speciális port száma van köztve. A szerver várakozik, figyeli a socket-et hogy van-e felkérés egy kliensről a kapcsolódásra.

A kliens oldalon: A kliens ismeri annak a számítógépnek a hosztnevét, amelyiken a szerver fut, és annak a portnak a számát, amelyiken keresztül a szerver kapcsolódva van. A kapcsolatra való felkéréshez a kliensnek érintkeznie kell a szerverrel a szerver gépén és port-ján.



Ha minden jól megy, a szerver elfogadja a kapcsolatot. Az elfogadás során, a szerver egy új socket porthoz kapcsolódik. Kell egy új socket (és természetesen egy különböző port szám is), ezért folytatódik a figyelés az eredeti socket-en a kapcsolatkérésre, amíg a kliensnek kapcsolódnia kell.



A kliens oldalon, ha a kapcsolat elfogadásra került, a socket sikeresen létrejött, és a kliens használja a socket-et a szerverrel való kommunikálásra.

Megjegyzés: a kliens oldali socket nincs összekötve azzal a port számmal, amit a szerverrel való kapcsolattartásra használ. A kliens a helyi kijelölt port számmal azonosítható azon a gépen, amin fut. A kliens és a szerver kapcsolat tud létrehozni a socket-jeik által írásra, vagy olvasásra.

Definíció: A *socket* egy végpontja egy kétvégű kommunikációs hálózathoz, aminek két program fut. A *socket* egy port számhoz van kötve, ezért a TCP réteg azonosítani tudja a kérést, amit ahhoz kértek, hogy elküldhessék az adatot.

A *java.net* csomag tartalmaz egy *Socket* osztályt, ami alkalmas egy kétirányú kapcsolatra egyik oldalának vezérlésére egy, a hálózaton lévő másik program felé. Ezen kívül a *java.net* tartalmazza a *ServerSocket* osztályt, amely figyel és elfogadja a kapcsolatot a klientsektől.

Ha a webhez akarunk kapcsolódni, az *URL* osztály és a kapcsolódó osztályok (*URLConnection*, *URLDecoder*) talán alkalmasabbak, mint a *Socket* osztály. Valójában az *URL* viszonylag magas szintű kapcsolatot teremt a webbel, és használja a *socket*-eket.

22.3.2 Olvasás és írás a socket-ről

Nézzünk egy egyszerű példát arra, hogy hogyan lehet kiépíteni egy kapcsolatot a szerver programmal a *Socket* osztályt használva, azután megnézzük, hogy hogyan tud a klients adatot küldeni és fogadni a szerverről a *socket*-en keresztül.

A példaprogram része az *EchoClient*, ami össze van kötve az echo szerverrel. Az echo szerver elég egyszerűen tud adatot fogadni a klientsől és választ küldeni rá. Az echo szerver egy jól ismert kiszolgáló, amit a klients a 7-es port-on keresztül tud elérni.

Az *EchoClient* létrehoz egy *socket*-et azzal, hogy kapcsolódik az echo szerverhez. Ez beolvas a felhasználótól egy sort, elküldi az echo szervernek a *socket*-en keresztül. A szerver ezen keresztül választ a klientsnek. A klients program olvassa ezt, és kinyitja az adatot, amit visszaküldött neki a szerver:

```
import java.io.*;
import java.net.*;

public class EchoClient {
    public static void main(String[] args) throws IOException {
        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("taranis", 7);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Az alábbi host nem ismert: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for " + "kapcsolódás ehhez: taranis.");
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(
            new InputStreamReader(System.in));
        String userInput;

        while (true) {
            userInput = stdIn.readLine();
            out.println(userInput);
        }
    }
}
```

```

while ((userInput = stdin.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}

out.close();
in.close();
stdin.close();
echosocket.close();
}
}

```

Az *EchoClient* olvasni és írni is tud a socket-ről azáltal, hogy átküld és átvesz adatot az echo szerverről.

Nézzük a program érdekesebb részeit. A *main* módszer *try* blokkjában szereplő sorok kiépítik a kapcsolatot a socket-en keresztül a kliens és a szerver között, és megnyitják a *PrintWriter*-t és a *BufferedReader*-t a socket-en.

```

echosocket = new Socket("tarans", 7);
out = new PrintWriter(echosocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
    echosocket.getInputStream()));

```

Sorrendben az első rész létrehozza az új *Socket* objektumot, és nevet ad neki: *echosocket*. A *Socket* konstruktor használata igényli annak a gépnek a nevét és a port-jának a számát, amelyikkel a kapcsolatot létre akarjuk hozni. A példa program a *tarans* host nevet használja. Ez a helyi hálózatumk virtuális gépének a neve. Mikor futtatjuk a programunkat, akkor saját szerver gépünk host neve kerüljön ide. A 7-es porton figyel az echo szerver.

A második sor összekapcsolja a *Socket* kimenetét és a *PrintWriter*-t. A harmadik sor hasonló ehhez, összekapcsolja a *Socket* bemenetét és a *BufferedReader*-t. A példa Unicode karaktereket tud írni a socket-en át.

Az *EchoClient*-nek az adatok socket-en való elküldéséhez egyszerűen a *PrintWriter*-hez kell írnia. Ahhoz, hogy megkapja a szerver választ, olvasnia kell a *BufferedReader*-ről. A program hátralévő részét ez teszi ki.

A *while* ciklus a program következő érdekes része. A ciklus beolvas egy sort egy bizonyos idő alatt a szabványos bemenetről, és azonnal elküldi azt a szervernek a *PrintWriter*-be való írással, ami a sockethez kapcsolódik:

```

String userInput;
while ((userInput = stdin.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}

```

A *while* ciklus utolsó sorában beolvas egy sor szöveget a *BufferedReader*-ről, ami a socket-hez kapcsolódik. A *readLine* módszer addig vár, amíg a szerver választ az információra az *EchoClient*-nek. Mikor a *readLine* választ, az *EchoClient* kiírja a szöveget a szabványos kimenetre.

A *while* ciklus addig tart, amíg a felhasználó be nem gépel az utolsó karaktert. Vagyis az *EchoClient* olvas a felhasználó bemenetéről, majd elküldi ezt az echo szervernek, kap egy választ a szerverről, és utána kiírja azt, amíg az elér az utolsó karakterig. A *while* ciklus ekkor beárul, és a program folytatódik, futtatja a kód következő négy sort:

```
out.close();  
in.close();  
stdin.close();  
echoSocket.close();
```

A kód e sorai a takarítást végzik. A jó program mindig kitakarít maga után, és ez a program is ilyen. Ezek a részek bezárják az olvasót és az író, ami a socket-hez kapcsolódik és a szabványos kimenetet és bemenetet is, ami a szerverhez kapcsolódik. A sorrend itt nagyon fontos. Be kell zárunk minden folyamatot, ami a socket-hez kapcsolódik, mielőtt kilapcsoljuk magát a socket-et.

- Socket megnyitása.
- Kimeneti és bementi folyam megnyitása a socket-en.
- Olvasás és írás a szerver protokollja szerint.
- A folyamatok bezárása.
- Socket bezárása.