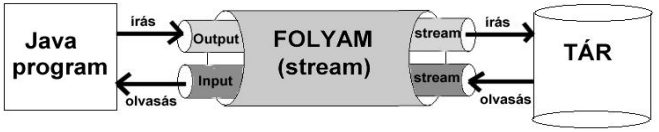
Programozás III

FÁJLKEZELÉS, I/O

FOLYAMOK

Adatállományok kezelése: folyamok segítségével

A folyam olyan objektum, amely képes egy meghatározott célhelyre adatokat sorosan írni, vagy onnan sorosan olvasni.



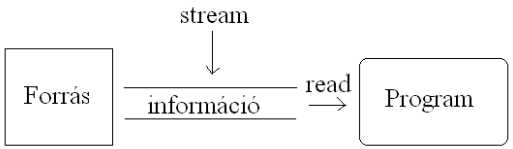
Iránytól függően: beviteli – kiviteli folyam

VÁZLAT

1. „sima” I/O
2. állományok
   - Scanner
   - BufferedReader

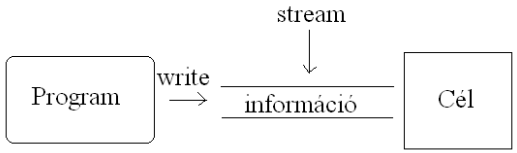
OLVASÁS – ÍRÁS

Olvasás
- adatfolyam megnyitása
- amíg van újabb információ
  - olvasás
- adatfolyam bezárása



OLVASÁS – ÍRÁS

Írás
- adatfolyam megnyitása
- amíg van újabb információ
  - írás
- adatfolyam bezárása



FOLYAMOK

Pufferező folyam: szerepe, hogy csökkentse az író/olvasó műveletek számát a memória és a külső erőforrás között.

Bevitelkor a folyam előre beolvas egy bájtsorozatot,

kivitelkor egyszerre írja ki az összegyűjtött bájtokat.

Az adatok átmeneti tárolását a puffer (buffer) végzi.

Pufferező folyamok: BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter.

Bármelyik folyamtípus összeköthető pufferezéssel.

A szükséges csomag: **java.io**

FOLYAMOK

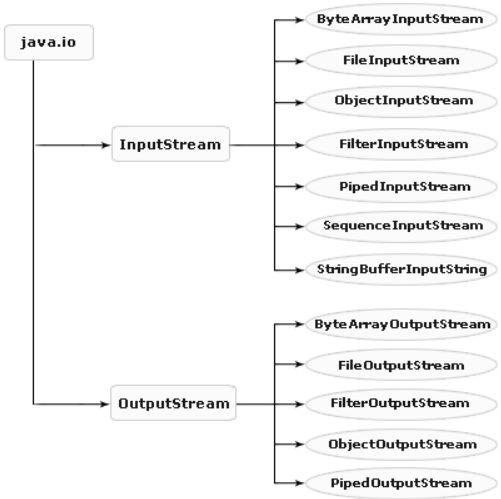
Bájtfolyam (byte stream): Az írás/olvasás egysége a bájt. Minden adatsorozat feldolgozható bájtfolyamként. Ősosztályai: InputStream, OutputStream.

Karakterfolyam (character stream): Az írás/olvasás egysége az unikód karakter, azaz két bájt. A szöveges fájlokat tipikusan karakterfolyammal szokás feldolgozni. Ősosztályai: Reader, Writer.

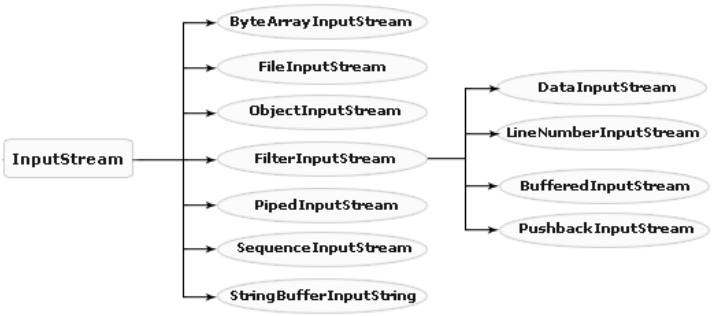
Adatfolyam (data stream): Az írás/olvasás egységei a primitív adatok (boolean, byte, short, char, int, long, float, double) és a String. Adatok írására/olvasására használják. Ősosztálya: DataInputStream, DataOutputStream.

Objektumfolyam (object stream): Egység: az objektum

JAVA.IO – FOLYAMOK



JAVA.IO – FOLYAMOK



SZÖVEG, SZÖVEGES ÁLLOMÁNY OLVASÁSA

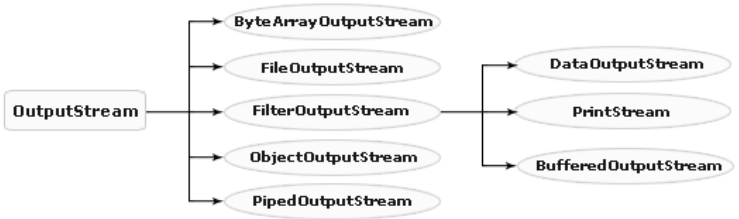
Szöveges állomány:
A csak olvasható karaktereket tartalmazó állományok. A szöveges állományok sorokból, a sorok karakterekből állnak. Minden sor végén sor vége jel található. Az állományt az állomány vége jel zárja.

A szöveges állományok tárolási formátuma függ az operációs rendszertől, de a Java platformfüggetlen

↓

A karakterek és a Java karakterei között kódolás/dekódolás szükséges (billentyűzetről való olvasás esetén is).
Ezt a feladatot látják el az **InputStreamReader** és az **OutputStreamWriter** osztályok. Ezek az osztályok alakítják át a beolvasott bájtokat karaktersorozattá.

JAVA.IO – FOLYAMOK



SZÖVEG, SZÖVEGES ÁLLOMÁNY OLVASÁSA

Szükséges osztályok, konstruktorok, metódusok (java.io csomag)

public class **BufferedReader** extends Reader
Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

BufferedReader(Reader in)
Creates a buffering character-input stream that uses a default-sized input buffer.

public String **readLine**() throws IOException
Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

SZÖVEG, SZÖVEGES ÁLLOMÁNY OLVASÁSA

Szükséges osztályok, konstruktorok, metódusok (java.io csomag)

public class **InputStreamReader** extends Reader

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

InputStreamReader(InputStream in)

Creates an InputStreamReader that uses the default charset.

System.in

The "standard" input stream.

OLVASÁS SCANNERREL

A Scanner „értelmezni” tudja a szöveges inputot és fel tudja bontani primitív típusokra.

Tetszőleges határolók (delimiter) megadhatóak hozzá, alapértelmezett: fehér szóköz (whitespace).

Pl.: Egy egész szám beolvasása a standard input stream-ről:

```
Scanner scanner = new Scanner(System.in);
int n = scanner.nextInt();
```

Határoló megadása pl.: scanner.useDelimiter(";");

BUFFERELT OLVASÁS

Pl.:

```
BufferedReader bevitel = new BufferedReader(
                         new InputStreamReader(System.in));

int szam;

try {
        szam = Integer.parseInt(bevitel.readLine());

    } catch (IOException ex) {
        …
    }
```

SZÖVEG OLVASÁSA, KONVERTÁLÁSA – PÉLDA

```
Scanner sc = new Scanner(System.in);

BufferedReader bevitel = new BufferedReader(
        new InputStreamReader(System.in));

int egyik = 0, masik, osszeg = 0;
System.out.print("egyik: ");
try {
    egyik = Integer.parseInt(bevitel.readLine());
} catch (IOException ex) {
    Logger.getLogger(Main.class.getName()).log(Level.SEVER
}
System.out.print("másik: ");
masik = sc.nextInt();
osszeg = egyik + masik;
System.out.println("az összeg: " + osszeg);
```

Lehetőleg ne kavarjuk a kettőt. ☺

OLVASÁS SCANNERREL – KÓDOLÁS

```
try(InputStream ins = this.getClass().getResourceAsStream(path);
Scanner fScanner = new Scanner(ins,StandardCharsets.UTF_8.name())){
```

vagy

```
try(InputStream ins = this.getClass().getResourceAsStream(path);
Scanner fScanner = new Scanner(ins,CHAR_SET)){
```

ahol

```
private static final String CHAR_SET = "UTF-8";
```

SCANNER vs BUFFEREDREADER

A BufferedReader

- biztonságosabb (kötelező kivételkezelés)
- általánosabb

A Scanner értelmezi (részekre szedi) az inputot,

a BufferedReader akár karakterenként is enged olvasni.

Írni csak folyamokon (stream) keresztül lehet.

JAVA.IO – FILE OSZTÁLY

A File osztály egy platform-független elérési útvonalat jelenít meg.
Az osztály példányosításakor egy elérési útvonalat hozunk létre, amelyen az osztály által biztosított metódusok végzik a manipulációkat (könyvtárrendszer felderítése, állományok létre-hozása, törlése, átnevezése, stb.).

Az osztály egyik – és leggyakrabban használt – konstruktorának paramétere a fájl vagy könyvtár elérési útvonala:

File fajl_nev = new File(eleresi_utvonal);

Pl.:

File fajl = new File(”c:/adatok.txt”);

Figyelem! Útvonalelválasztó: ”/” vagy ”\\”

JAVA.IO – FILE OSZTÁLY

Néhány metódus:

canRead()
– ellenőrzi, hogy a File objektum olvasható-e

canWrite()
– ellenőrzi, hogy a File objektum írható-e

createNewFile()
– létrehozza az objektum elérési útvonalában megadott fájlt

delete()
– törli az állományt vagy könyvtárat

exists()
– ellenőrzi, hogy a fájl vagy könyvtár létezik-e

mkdir()
– létrehozza az objektum elérési útvonalában meghatározott könyvtárat

…stb.

FÁJLKEZELÉS – PÉLDA

Ellenőrizzük egy fájl létezését. Ha nem létezik, akkor hozzuk létre.

```
import java.io.*;

public class Pelda{
   public static void main(String args[]){
       File fajl=new File("c:\\ujfajl.java");

       if(fajl.exists()){
           System.out.println("A fájl létezik!");
       }
       else{
           try{
               fajl.createNewFile();
               System.out.println("Létrehoztam");
           }catch(Exception e){
               System.out.println("Hiba: " + e.getMessage());}
       }
   }
}
```

„try” nélkül: unreported exception java.io.IOException; must be caught or declared to be thrown

ÍRÁS FOLYAMOKON KERESZTÜL, APPEND

```
private void textFaljba() throws Exception {
    try(PrintWriter out =
        new PrintWriter(new BufferedWriter(new FileWriter(path, true)))){
        for (Diak diak : diakok) {
            out.println(diak);
        }
    }
}
```

FileWriter paraméterezése:

FileWriter(File file)
Constructs a FileWriter object given a File object.
FileWriter(File file, boolean append)
Constructs a FileWriter object given a File object.

ÍRÁS FOLYAMOKON KERESZTÜL

```
File file = new File(path );

if (file.exists()) {
    //...
} else {
    file.createNewFile();
    writeFile(file);
}

private void writeFile(File file) {
    try (PrintWriter printWriter = new PrintWriter(file)) {
        for (Student student : this.students) {
            printWriter.println(student.studentToFile());
        }
    } catch (FileNotFoundException e) {
        // ...
    }
}
```

http://stackoverflow.com/questions/14067843/difference-between-printwriter-and-filewriter-class

FOLYAMOK – OSZTÁLYOK

Bájtfolyam:
FileInputStream – FileOutputStream

Karakterfolyam:
FileReader – FileWriter – PrintWriter

Adatfolyam:
DataInputStream – DataOutputStream

Objektumfolyam (object stream): – ezt most részletezzük.

FOLYAMOK – OBJEKTUMFOLYAM

Szükség lehet objektumok
- – háttértárra mentésére
- – adatbázisba mentésére
- – hálózaton való továbbítására
- – …

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

A **szerializáció** során az objektumból egy bájtfolyam lesz, amelyből később vissza lehet állítani az eredeti objektumot.

FOLYAMOK – OBJEKTUMFOLYAM

Az objektumokat **ObjectOutputStream** típusú objektumon keresztül lehet kiírni és **ObjectInputStream** típusú példányon keresztül beolvasni.
A kiírást a **writeObject**, a beolvasást a **readObject** metódus végzi. Ezek a metódusok automatizálva vannak – tudják, hogyan kell az egyes objektumokat kezelni.

Az elnevezés eredete:
Az algoritmus egészen bonyolult fastruktúra esetén is szigorú sorrendbe állítja az objektumokat.

FOLYAMOK – OBJEKTUMFOLYAM – PÉLDA

Definiáljunk egy Diak osztályt, hozzunk létre a diákok listáját, és ezt a példányt objektumként írjuk fájlba, majd olvassuk vissza.

A List nem szerializálható, de az ArrayList igen. Viszont a Diak osztályt szerializálhatóvá kell tennünk.

A Diak osztálynak implementálnia kell a Serializable interfészt.

FOLYAMOK – OBJEKTUMFOLYAM – PÉLDA

```
public class Diak implements Serializable {
    private String nev;
    private String kod;
    private float atlag;

    public Diak(String nev, String kod, float atlag) {
        this.nev = nev;
        this.kod = kod;
        this.atlag = atlag;
    }

    @Override
    public String toString() {
        return nev + ";" + kod + ";" + atlag;
    }
```

FOLYAMOK – OBJEKTUMFOLYAM – PÉLDA

```
private String path = "adatok.txt";
private List<Diak> diakok = new ArrayList<>();

private void fajlba() throws Exception {
    try(FileOutputStream fStream = new FileOutputStream(path);
            ObjectOutputStream oStream =
                    new ObjectOutputStream(fStream)){
        oStream.writeObject(this.diakok);
    }
}

private void fajlbol() throws Exception {
    try(FileInputStream fStream = new FileInputStream(path);
            ObjectInputStream oStream =
                    new ObjectInputStream(fStream)){
        List<Diak> diakLista = (List<Diak>) oStream.readObject();
        System.out.println("a fájlból olvasott adatok: \n"
                                                + diakLista);
    }
}
```

VÉLETLEN ELÉRÉSŰ ÁLLOMÁNYOK

Szükséges osztályok, konstruktorok, metódusok
(java.io csomag)

public class **RandomAccesFile**
extends Object
implements DataOutput, DataInput, Closeable

public **RandomAccessFile**(File file, String mode) throws FileNotFoundException

módok: „rw”, „r”, „rws”, „rwd”

s és d szinkronizálásra utal – ld. Help

Metódusok:
write-ok, read-ek, seek, stb.

VÉLETLEN ELÉRÉSŰ ÁLLOMÁNYOK

A véletlen elérésű állományt – a folyammal ellentétben – nem csak sorosan és egy irányban tudjuk kezelni, hanem a fájlban tetszőleges helyre pozícionálhatunk, és az adott pozíciótól kezdve felvehetünk vagy olvashatunk adatokat.

Szükséges osztály: **RandomAccessFile**

Segítségével egy állomány bármelyik (véletlen elérésű) részét fel lehet dolgozni.

Mindig van egy aktuális mutatója, amelynek értéke az állomány elejétől számított bájtsorszám. Az író/olvasó utasítások mindig az állománymutató által meghatározott pozíciótól írnak/olvasnak.

VÉLETLEN ELÉRÉSŰ ÁLLOMÁNYOK

Hozzunk létre egy véletlen elérésű állományt,
- írjunk bele n db véletlen egész számot, (semmi köze ennek a véletlennek a véletlen eléréshez)
- olvassuk ki az adatokat, és írassuk ki a képernyőre
- keressünk meg egy adott sorszámú elemet!

VÉLETLEN ELÉRÉSŰ ÁLLOMÁNYOK – PÉLDA

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomFileDemo {

    RandomAccessFile fajl;

    public RandomFileDemo(String utvonal){
        try{
            fajl = new RandomAccessFile(utvonal,"rw");
        }
        catch(FileNotFoundException e){
            System.out.println("Hiba: " + e.getMessage());}
        kiir();
        beolvas();
        keres();
    }

    public static void main(String[] args) {
        new RandomFileDemo("pelda.dat");
    }
```

Véletlen elérésű állományok - példa

```
    public void keres(){
        int k = 3, byteSzam = 4;
        try{
            fajl.seek((k-1)*byteSzam);
            System.out.println("A " + k + ". szám: " + fajl.readInt());
            fajl.close();
        } catch (IOException e) {
            System.out.println("Hiba: " + e.getMessage());}
    }
}
```

(Az int 4 byte-os)

Véletlen elérésű állományok - példa

```
public void kiir(){
    int n=10, hatar = 100;
    for(int i=1; i<=n; i++){
        try {
            fajl.writeInt((int) (Math.random()*hatar));
        } catch (IOException e) {
            System.out.println("Hiba: " + e.getMessage());}
    }
}

public void beolvas(){
    int sorszam=1, szam = 0;
    try {
        fajl.seek(0);
        while(true){
           szam = fajl.readInt();
           System.out.println(sorszam + ". adat: " + szam);
           sorszam++;
       }}
       catch(EOFException e2){
           System.out.println("Vége a fájlolvasásnak.");
       }
       catch (IOException e1) {
           System.out.println("Hiba: "); e1.printStackTrace();}
 }
```