Completari pentru oameni disperati

Masina virtuala Java si codul binar (byte codes)

De obicei, compilatoarele traduc un limbaj de nivel inalt in limbaj masina pentru un tip particular de calculator. Cu toate acestea compilatorul Java nu traduce Java in limbaj masina, ci intr-un limbaj pseudo-masina numit cod binar Java. Codul binar (byte code) este limbajul masina pentru calculatorul Java imaginar. Pentru a rula cod binar Java pe un calculator particular, trebuie sa instalam o masina virtuala Java (JVM - Java Virtual Machine) pe acel calculator.

JVM este un program care se comporta ca un calculator. Un astfel de program se numeste interpretor. Un interpretor are si avantaje si dezavantaje.

Marele dezavantaj este ca un interpretor ruleaza programele mai lent decat un calculator actual. Cu toate acestea, anumite JVM-uri pot traduce cod binar in limbaj masina imediat - utilizind Just-In-Time compilere (JIT).

De asemeni, procesoarele noi de calculatoare sunt dezvoltate pentru a implementa JVM direct in hard pentru a nu mai exista penalizari de perfomanta.

Marele avantaj al unui interpretor este ca orice calculator il poate rula.

Asadar codul binar Java este foarte portabil. De exemplu multe din paginile ce se incarca de pe Web contin mici programe Java deja traduse in cod binar. Acestea se numesc applet-uri si ruleaza pe un JVM care este incorporat intr-un browser Web.

Deoarece programele Java ruleaza pe o masina virtuala se pot limita posibilitatile lor, deci nu trebuie sa ne facem griji ca un applet ne 'infecteaza' calculatorul. Java are un mecanism de securitate bine pus la punct.

Cum este executat un program Java?

Interpretorul Java transforma codul de octeti într-un set de instructiuni masina, întârzierea interpretarii fiind însa foarte mica datorita asemanarii dintre codul de octeti si limbajul de asamblare si din acest motiv executia se face aproape la fel de repede ca în cazul programelor compilate.

Cum este posibila portarea codului de octeti pe calculatoare diferite?

Codul sursa este compilat nu pentru calculatorul pe care se lucreaza ci pentru un calculator inexistent, acest calculator imaginar fiind numit Masina virtuala Java (Java Virtual Machine). Interpretorul actioneaza apoi ca un intermediar între Masina virtuala Java si masina reala pe care este rulat programul.

<table>
<tr><td colspan="3">Aplicatia utilizatorului</td></tr>
<tr><td colspan="3">Obiecte Java</td></tr>
<tr><td colspan="3">Masina virtuala Java</td></tr>
<tr><td>UNIX</td><td>Windows</td><td>Macintosh</td></tr>
<tr><td colspan="3">Sisteme de operare</td></tr>
</table>

Crearea unei aplicatii simple

Scriererea codului sursa:

```
class FirstApp {
        public static void main( String args[])
                { System.out.println("Hello world !");}
}
```

Toate aplicatiile Java contin o clasa principala (primara) în care trebuie sa se gaseasca metoda **main**. Clasele aplicatiei se pot gasi fie într-un singur fisier, fie în mai multe.

Salvarea fisierelor sursa

Se va face în fisiere cu extensia .java

Fiserul care contine codul sursa al clasei primare trebuie sa aiba acelasi nume cu clasa primara a aplicatiei (clasa care contine metoda main).

Obs: Java face distinctie între literele mari si mici (case sensitive).

C:/java/FirstApp.java

Compilarea aplicatiei (nativ la consola)

Se foloseste compilatorul Java, **javac**.

Apelul compilatorului se face pentru fisierul ce contine clasa principala a aplicatiei. Compilatorul creeaza câte un fisier separat pentru fiecare clasa a programului; acestea au extensia **.class** si sunt plasate în acelasi director cu fisierele sursa.

---

javac FirstApp.java -> FirstApp.class

Executarea aplicatiei

Se face cu interpretorul java, apelat pentru unitatea de compilare corespunzatoare clasei principale, fiind însa omisa extensia .class asociata acesteia.

---

java FirstApp

Executarea unei aplicatii care nu foloseste interfata grafica, se va face într-o fereastra sistem.

Crearea unui applet

Crearea structurii de fisere si compilarea applet-urilor sunt identice ca în cazul aplicatiilor. Difera în schimb structura programului si modul de rulare al acestuia.

Scrierea codului sursa si salvarea în fisier

```
import java.awt.* ;
import java.applet.* ;
public class FirstApplet extends Applet
{ Image img;
public void init() {
        resize(150,25); }
public void paint (Graphics g) {
g.drawString("Hello world!", 50, 25); }
 }
```

Salvarea se va face în fisierul FirstApplet.java

Compilarea applet-ului

---

javac FirstApplet.java -> FirstApplet.class

Executarea applet-ului

Applet-urile nu ruleaza independent. Ele pot fi rulate doar prin intermediul unui browser: Internet Explorer, Netscape sau printr-un program special cum ar fi appletviewer-ul din setul JDK (Java Development Kit).

Crearea unui fisier HTML pentru applet (exemplu.html)

```
<html>
    <head>
        <title>First Java Applet</title>
    </head>
    <body>
        <applet code=FirstApplet.class width=400 height=400>
        </applet>
    </body>
</html>
```

Vizualizarea appletului

---

appletviewer exemplu.html

Structura programelor

Pachete de clase

Clasele Java sunt organizate pe pachete. Aceste pachete pot avea nume ierarhice. Numele de pachete au forma următoare:

[NumePachet.]NumeComponentăPachet

Numele de pachete şi de componente ale acestora sunt identificatori Java. De obicei, aceste nume urmează structura de directoare în care sunt memorate clasele compilate. Rădăcina arborelui de directoare în care sunt memorate clasele este indicată de o variabilă sistem **CLASSPATH**. În DOS/Windows aceasta se setează în felul următor:

```
set CLASSPATH=.;c:\java\lib
```

În Unix se poate seta cu comanda:

```
CLASSPATH=.:/usr/local/lib/java; export CLASSPATH
```

dacă lucraţi cu **bash**. Din această rădăcină, fiecare pachet are propriul director. În director există codul binar pentru componentele pachetului respectiv. Dacă pachetul conţine subpachete, atunci acestea sunt memorate într-un subdirector în interiorul directorului pachetului.

Creatorii Java recomandă folosirea unei reguli unice de numire a pachetelor, astfel încât să nu apară conflicte. Convenţia recomandată de ei este aceea de a folosi numele domeniului Internet aparţinând producătorului claselor. Astfel, numele de pachete ar putea arăta ca în:

COM.Microsoft.OLE
COM.Apple.quicktime.v2 etc.

Importul claselor

Desigur, este nevoie ca o clasă să poată folosi obiecte aparţinând unei alte clase. Pentru aceasta, definiţia clasei respective trebuie să importe codul binar al celeilalte clase pentru a şti care sunt variabilele şi metodele clasei respective. Importul se face cu o instrucţiune specială:

import numeClasă;

unde numele clasei include şi pachetul din care aceasta face parte.

De exemplu:

```
import java.awt.Graphics;
import java.applet.Applet;
```

Se poate importa şi un pachet întreg, adică toate clasele aparţinând acelui pachet, printr-o instrucţiune de forma:

import numePachet.*;

De exemplu:

```
import java.awt.*;
```

Fişiere sursă

Codul sursă Java trebuie introdus cu un editor într-un fişier text pe care îl vom numi în continuare fişier sursă. Un fişier sursă poate să conţină declaraţia mai multor clase şi interfeţe, dar doar una dintre acestea poate fi declarată **publică** (detalii in laboratoarele urmatoare). Utilizarea celorlalte clase este limitată la fişierul respectiv. Mai mult, nu putem avea în acelaşi timp o interfaţă publică şi o clasă publică declarate în acelaşi fişier sursă.

Dacă dorim să înregistrăm codul clasei într-un anumit pachet, putem să includem la începutul fişierului sursă o declaraţie de forma:

package numePachet;

dacă această declaraţie lipseşte, clasa va fi plasată în pachetul implicit, care nu are nume.

Structura generală a unui fişier sursă este următoarea:

[DeclaraţiePachet]
[InstrucţiuneImport]
DeclaraţieDeTip

unde declaraţia de tip poate fi o declaraţie de clasă sau de interfaţă.

Compilare şi execuţie

Fişierele sursă Java au obligatoriu extensia **.java**. Numele lor este identic cu numele clasei sau interfeţei publice declarate în interior. În urma compilării rezultă fişiere cu nume identice cu numele claselor dar cu extensia **.class** indiferent dacă este vorba de o clasă sau o interfaţă. Fişierul **.class** este generat în directorul aferent fisierului **.java** şi **nu** direct la locaţia pachetului (chiar daca exista directiva **package** in fisier).

Compilarea se face cu o comandă de forma:

javac FişierSursă.java

Comanda aceasta, ca şi celelalte descrise în acest paragraf este specifică mediului de dezvoltare Java pus la dispoziţie de Sun, numit JDK (Java Development Kit). În viitor este probabil să apară multe alte medii de dezvoltare care vor avea propriile lor compilatoare şi interpretoare şi, posibil, propriile linii de comandă.

La compilare, variabila sistem CLASSPATH trebuie să fie deja setată.

Pentru lansarea în execuţie a unei aplicaţii Java, trebuie să introduceţi comanda:

java NumeClasă

unde numele clasei este numele aplicaţiei care conţine metoda **main**. Interpretorul va căuta un fişier cu numele NumeClasă.class şi va încerca să instanţieze clasa respectivă.

Pentru lansarea unui aplet veţi avea nevoie de un document **HTML** (Hyper Text Markup Language) care conţine tagul **APPLET** şi ca parametru al acesteia

name=NumeClasă.class

La lansarea unui aplet, clasele care sunt apelate de clasa principală sunt mai întâi căutate pe sistemul pe care rulează navigatorul (de exemplu: Netscape, Internet Explorer). Dacă nu sunt acolo, ele vor fi transferate din reţea (de pe calculatorul care contine pagina de internet in care se afla aplet-ul). Asta înseamnă că transferul de cod este relativ mic, trebuie transferat doar codul specific aplicaţiei.

Reluare: scrierea unei aplicatii simple

Cea mai simplă aplicaţie Java este declaraţia unei clase de pornire conţinând o singură metodă, **main**, ca în exemplul următor:

```
public class HelloWorld {
        public static void main( String args[] )
                { System.out.println( "Hello, world!" );
        }
}
```

Acest exemplu defineşte o funcţie principală care afişează un simplu mesaj la consola aplicaţiei. Afişarea este lăsată în sarcina clasei **java.lang.System** care conţine în interior

implementarea ieşirii şi intrării standard precum şi a ieşirii standard de eroare sub forma unor referinţe către obiecte de tip InputStream pentru **in** (intrarea standard) respectiv PrintStream pentru **out** şi **err** (ieşirea standard şi ieşirea standard de eroare).

Numele metodei **main** este obligatoriu, la fel şi parametrul acesteia. Atunci când lansăm interpretorul Java împreună cu numele unei clase care reprezintă clasa de pornire, interpretorul caută în interiorul acestei clase definiţia unei metode numite **main**. Această metodă trebuie să fie obligatoriu **publică** şi **statică** (detalii despre modificatori in laboratoarele ce urmeaza). În acelaşi timp, metoda main trebuie să nu întoarcă nici un rezultat (**void**) şi să accepte un singur parametru de tip tablou de şiruri de caractere (**String args[]**).

Dacă interpretorul găseşte această metodă în interiorul clasei apelate, el lansează în execuţie metoda **main**. Atenţie, metoda **main** fiind de tip **static**, nu poate apela decât variabile statice. De obicei însă, metoda **main** nu face nimic altceva decât să-şi prelucreze parametrul după care să creeze o serie de obiecte care vor controla execuţia ulterioară a aplicaţiei.

Folosirea argumentelor de pe linia de comanda

Singurul parametru al metodei **main** este un tablou care conţine argumentele aflate pe linia de comandă în momentul apelului. Nu este necesară transmiterea numărului de argumente care au fost găsite pe linia de comandă pentru că tablourile Java conţin în interior informaţii relative la numărul de elemente. Acest număr de elemente se poate obţine prin accesarea variabilei **length** din interiorul tabloului ca în exemplul următor care listează parametrii de pe linia de comandă la lansarea unei clase:

```
public class Arguments {
        public static void main( String args[] ) {
                for( int i = 0; i < args.length; i++ )
                        { System.out.println( args[i] );
                }
         }
}
```

Iată un exemplu de rulare a acestei aplicaţii:

```
> java Arguments unu doi trei unu
doi trei
>
```

O aplicatie Java poate primi oricâte argumente de la linia de comanda în momentul lansarii ei. Aceste argumente sunt utile pentru a permite utilizatorului sa specifice diverse optiuni legate de functionarea aplicatiei sau sa furnizeze anumite date initiale programului.

Atentie: *Programele care folosesc argumente de la linia de comanda nu sunt 100% pure Java deoarece unele sisteme de operare cum ar fi Mac OS nu au în mod normal linie de comanda.*

Argumentele de la linia de comanda sunt introduse la lansarea unei aplicatii, fiind specificate dupa numele aplicatiei si separate prin spatiu. De exemplu, sa presupunem ca aplicatia Sort ordoneaza lexicografic liniile unui fisier si primeste ca argument numele fisierului pe care sa îl sorteze. Pentru a ordona fisierul "persoane.txt" lansarea aplicatiei se va face astfel:

```
java Sort persoane.txt
```

Asadar, formatul general pentru lansarea unei aplicatii care primeste argumente de la linia de comanda este:

```
java NumeAplicatie [arg1 arg2 . . . argn]
```

In cazul în care sunt mai multe, argumentele trebuie separate prin spatii iar daca unul dintre argumente contine spatii, atunci el trebuie pus între ghilimele. Evident, o aplicatie poate sa nu primeasca nici un argument sau poate sa ignore argumentele primite de la linia de comanda.

In momentul lansarii unei aplicatii interpretorul parcurge linia de comanda cu care a fost lansata aplicatia si, în cazul în care exista, transmite aplicatiei argumentele specificate sub forma unui vector de siruri. Acesta este primit de aplicatie ca parametru al metodei **main**. Reamintim ca formatul metodei main din clasa principala este:

```
public static void main (String args[])
```

Vectorul primit ca parametru de metoda **main** va contine toate argumentele transmise programului de la linia de comanda. In cazul apelului *java Sort persoane.txt* vectorul **args** va contine un singur element *args[0]="persoane.txt"*.
Numarul argumentelor primite de un program este dat deci de dimensiunea vectorului args si acesta poate fi aflat prin intermediul atributului **length** al vectorilor:

```
numarArgumente = args.length;
```

Variabilele unei clase

În interiorul claselor se pot declara variabile. Aceste variabile sunt specifice clasei respective. Fiecare dintre ele trebuie să aibă un tip, un nume şi poate avea iniţializatori. Variabilele definite în interiorul unei clase pot avea definiţi o serie de modificatori care alterează comportarea variabilei în interiorul clasei, şi o specificaţie de protecţie (modificatori de acces) care defineşte cine are dreptul să acceseze variabila respectivă.
Modificatorii sunt cuvinte rezervate Java care precizează sensul unei declaraţii. Acestia sunt:

1. **static**
2. **final**
3. **transient**
4. **volatile**

Modificatorul **transient** este folosit pentru a specifica variabile care nu conţin informaţii care trebuie să rămână persistente la terminarea programului. Este folosit in special in cadrul aplicatiilor RMI (Remote Method Invocation).

Modificatorul **volatile** specifică faptul că variabila respectivă poate fi modificată asincron cu rularea aplicaţiei. În aceste cazuri, compilatorul trebuie să-şi ia măsuri suplimentare în cazul generării şi optimizării codului care se adresează acestei variabile.

Modificatorul **final** este folosit pentru a specifica o variabilă a cărei valoare **nu** poate fi modificată. *Variabila respectivă trebuie să primească o valoare de iniţializare chiar în momentul declaraţiei.* Altfel, ea nu va mai putea fi iniţializată în viitor. Orice încercare ulterioară de a seta valori la această variabilă va fi semnalată ca eroare de compilare.

Modificatorul **static** este folosit pentru a specifica faptul că variabila are o singură valoare comună tuturor instanţelor clasei în care este declarată. Modificarea valorii acestei variabile din interiorul unui obiect face ca modificarea să fie vizibilă din toate celelalte obiecte instantiate din clasa respectiva. Variabilele statice sunt iniţializate la încărcarea codului specific unei clase şi există chiar şi dacă nu există nici o instanţă a clasei respective. Din această cauză, ele pot fi folosite de metodele statice. *Variabilele si metodele statice mai sunt denumite si variabile si metode de clasa.*

Modificatori de acces

În Java există patru grade de protecţie pentru o variabilă sau o metoda aparţinând unei clase. Acestea grade de protectie sunt:
private
protected
public
friendly (default)

O variabilă **publică** este accesibilă oriunde este accesibil numele clasei. Cuvântul rezervat este **public**.

O variabilă **protejată** este accesibilă în orice clasă din pachetul căreia îi aparţine clasa în care este declarată. În acelaşi timp, variabila este accesibilă în toate subclasele clasei date, chiar dacă ele aparţin altor pachete. Cuvântul rezervat este **protected**.

O variabilă **privată** este accesibilă doar în interiorul clasei în care a fost declarată. Cuvântul rezervat este **private**.

O variabilă care nu are nici o declaraţie relativă la gradul de protecţie este automat o variabilă **prietenoasă**. O variabilă prietenoasă este accesibilă în pachetul din care face parte clasa în interiorul căreia a fost declarată, la fel ca şi o variabilă protejată. Dar, spre deosebire de variabilele protejate, o variabilă prietenoasă nu este accesibilă în subclasele clasei date dacă aceste sunt declarate ca aparţinând unui alt pachet. *Nu există un cuvânt rezervat pentru specificarea explicită a variabilelor prietenoase.*

O variabilă nu poate avea declarate mai multe grade de protecţie în acelaşi timp. O astfel de declaraţie este semnalată ca eroare de compilare.

Nota: *aceleasi reguli de protectie de la variabile se aplica si in cazul metodelor unui obiect*

Accesarea unei variabile sau a unei metode
Accesarea unei variabile declarate în interiorul unei clasei se face folosindu-ne de o expresie de forma:
ReferinţăInstanţă.NumeVariabilă

Referinţa către o instanţă trebuie să fie referinţă către clasa care conţine variabila. Referinţa poate fi valoarea unei expresii mai complicate, ca de exemplu un element dintrun tablou de referinţe.

În cazul în care avem o variabilă statică, aceasta poate fi accesată şi fără să deţinem o referinţă către o instanţă a clasei. Sintaxa este, în acest caz:

NumeClasă.NumeVariabilă

Nota: *aceleasi reguli de accesare de la variabile se aplica si in cazul metodelor unui obiect*

Vizibilitate
O variabilă poate fi **ascunsă** de declaraţia unei alte variabile cu acelaşi nume. De exemplu, dacă într-o clasă avem declarată o variabilă cu numele *var* şi într-o subclasă a acesteia avem declarată o variabilă cu acelaşi nume, atunci variabila din superclasă este ascunsă de cea din clasă. Totuşi, variabila din superclasă există încă şi poate fi accesată în mod explicit. Expresia de referire este, în acest caz:

super.NumeVariabilă

în cazul în care superclasa este imediată.
La fel, o variabilă a unei clase poate fi ascunsă de o declaraţie de variabilă dintr-un bloc de instrucţiuni. Orice referinţă la ea va trebui făcută în mod explicit. Expresia de referire este, în acest caz:
this.NumeVariabilă

Variabile predefinite: this şi super
În interiorul fiecărei metode non-statice dintr-o clasă există predefinite două variabile cu semnificaţie specială. Cele două variabile sunt de tip referinţă şi au aceeaşi valoare şi anume o referinţă către obiectul curent. Diferenţa dintre ele este tipul.

Prima dintre acestea este variabila **this** care are tipul referinţă către clasa în interiorul căreia apare metoda. A doua este variabila **super** al cărei tip este o referinţă către superclasa imediată a clasei în care apare metoda. În interiorul obiectelor din clasa Object nu se poate folosi referinţa super pentru că nu există nici o superclasă a clasei de obiecte Object. (*In Java, Clasa Object este clasa care sta la baza ierarhiei oricarui obiect*)

În cazul în care **super** este folosită la apelul unui constructor sau al unei metode, ea acţionează ca un cast către superclasa imediată.

Derivarea claselor

O clasă poate fi derivată din alta clasa prin folosirea în declaraţia clasei derivate a clauzei **extends**. Clasa din care se derivă noua clasă se numeşte superclasă imediată a clasei derivate. Toate clasele care sunt superclase ale superclasei imediate ale unei clase sunt superclase şi pentru clasa dată. Clasa nou derivată se numeşte subclasă a clasei din care este derivată.

Sintaxa generală este:

```
class SubClasă extends SuperClasă
```

*O clasă poate fi derivată **numai** dintr-o singură altă clasă, cu alte cuvinte o clasă poate avea o singură superclasă imediată.*

Clasa derivată moşteneşte toate variabilele şi metodele superclasei sale. Totuşi, ea nu poate accesa decât acele variabile şi metode care nu sunt declarate **private**.

Putem rescrie o metodă a superclasei declarând o metodă în noua clasă având acelaşi nume şi aceiaşi parametri. La fel, putem declara o variabilă care are acelaşi nume cu o variabilă din superclasă. În acest caz, noul nume ascunde vechea variabilă, substituindu-ise. Putem în continuare să ne referim la variabila ascunsă din superclasă specificând numele superclasei sau folosindu-ne de variabila **super**.

Exemplu:

```
class A {
    int var = 1;
    void m1() {
          System.out.println( var ); }
}

class B extends A {
  double var = 3.14;
  void m1() {
          System.out.println( var );  }
  void m2() {
          System.out.println( super.var ); }
   void m3() {
          m1();
          super.m1();
          }
}
```

Dacă apelăm metoda **m1** din clasa A, aceasta va afişa la consolă numărul 1. Acest apel se va face cu instructiunile:

```
A obiect = new A();
obiect.m1();
```

sau

```
( new A() ).m1();
```

Dacă apelăm metoda **m1** din clasa B, aceasta va afişa la consolă numărul 3.14. Apelul îl putem face de exemplu cu instrucţiunea:

```
B obiect = new B();
obiect.m1();
```

Observaţi că în metoda **m1** din clasa B, variabila referită este variabila **var** din clasa B. Variabila **var** din clasa A este ascunsă. Putem însă să o referim prin sintaxa **A.var** sau **super.var** ca în metoda **m2** din clasa B.

În interiorul clasei B, apelul metodei **m1** fără nici o altă specificaţie duce automat la apelul metodei **m1** definite în interiorul clasei B. Metoda **m1** din clasa B suprascrie (*overwrite*) metoda **m1** din clasa A. Vechea metodă este accesibilă pentru a o referi în mod explicit ca în metoda **m3** din clasa B. Apelul acestei metode va afişa mai întâi numărul 3.14 şi apoi numărul 1.

Dacă nu declarăm nici o superclasă în definiţia unei clase, atunci se consideră automat că noua clasă derivă direct din clasa Object, moştenind toate metodele şi variabilele acesteia.

Literali şir de caractere

Un literal şir de caractere este format din zero sau mai multe caractere între ghilimele. Caracterele care formează şirul de caractere pot fi caractere grafice sau secvenţe escape utilizind simbolul '\'. De exemplu, pentru a introduce o linie noua se foloseste secventa escape '\n', pentru introducerea " se foloseste secventa escape '\"', pentru introducerea ' se foloseste secventa escape '\'', pentru introducerea \ se foloseste secventa escape '\\' etc.
Dacă un literal şir de caractere conţine în interior un caracter terminator de linie va fi semnalată o eroare de compilare. Cu alte cuvinte, nu putem avea în sursă ceva de forma:

```
"Acesta este
greşit!"
```

chiar dacă aparent exprimarea ar reprezenta un şir format din caracterele A, c, e, s, t, a, spaţiu, e, s, t, e, linie nouă, g, r, e, ş, i, t, !. Dacă dorim să introducem astfel de caractere terminatoare de linie într-un şir va trebui să folosim secvenţe escape ca în:

```
"Acesta este\ngreşit"
```

Dacă şirul de caractere este prea lung, putem să-l spargem în bucăţi mai mici pe care să le concatenăm cu operatorul +.

Fiecare şir de caractere este în fapt o instanţă a clasei de obiecte **String** declarată standard în pachetul **java.lang**.

Exemple de şiruri de caractere:
""
"\""
"Şir de caractere"
"unu" + "doi"

Primul şir de caractere din exemplu nu conţine nici un caracter şi se numeşte şirul vid. Ultimul exemplu este format din două şiruri distincte concatenate.

Comentarii in Java

Un comentariu este o secvenţă de caractere existentă în fişierul sursă dar care serveşte doar pentru explicarea sau documentarea sursei şi nu afectează în nici un fel semantica programelor.

În Java există trei feluri de comentarii:

1. Comentarii pe mai multe linii, închise între /* şi */.
2. Toate caracterele dintre cele două secvenţe sunt ignorate.
3. Comentarii pe mai multe linii care ţin de documentaţie, închise între /** şi */. Textul dintre cele două secvenţe este automat copiat în documentaţia aplicaţiei de către generatorul automat de documentaţie.
4. Comentarii pe o singură linie care încep cu //. Toate caracterele care urmează acestei secvenţe până la primul caracter sfârşit de linie sunt ignorate.

În Java, nu putem să scriem comentarii în interiorul altor comentarii. La fel, nu putem introduce comentarii în interiorul literalilor caracter sau şir de caractere. Secvenţele /* şi */ pot să apară pe o linie după secvenţa // dar îşi pierd semnificaţia. La fel se întâmplă cu secvenţa // în comentarii care încep cu /* sau /**.
Ca urmare, următoarea secvenţă de caractere formează un singur comentariu:

/* acest comentariu /* // /* se termină abia aici: */

Fluxuri de Intrare / Iesire

Fluxurile Java pun la dispozitie modalitatea prin care doua sau mai multe procese pot comunica fara a avea informatii unul despre celalalt. Mai mult, prin fluxuri este posibila comunicarea între doua sau mai multe fire de executie ale aceleiasi aplicatii. Fluxurile sunt secvente de octeti.

Proces producator -> flux de iesire -> Proces consumator Proces consumator <- flux de intrare <- Proces producator

Caracteristicile fluxurilor:

1. fluxurile sunt unidirectionale, de la producator la consumator
2. fiecare flux are un singur proces producator si un singur proces consumator
3. între doua procese pot exista oricâte fluxuri, orice proces putând fi atât producator si consumator în acelasi timp, dar pe fluxuri diferite
4. consumatorul si producatorul nu comunica direct printr-o interfata de flux ci prin intermediul codului Java de tratare a fluxurilor

Importanta: *ajuta la citirea / scrierea informatiilor in dispozitive de intrare / iesire, fisiere, baze de date, etc.*
Toate interfetele pentru fluxuri implementeaza un set de metode de baza, comune tuturor categoriilor de fluxuri. Metodele standard pentru lucrul cu fluxuri se gasesc în pachetul java.io.

Fluxuri de intrare

Metode care functioneaza cu toate fluxurile de intrare:

1. read() - citeste date dintr-un flux de intrare
2. skip() - ignora unele date din fluxul de intrare
3. markAvailable() - testeaza daca metoda mark() este disponibila pentru fluxul de intrare respectiv
4. close() - închide un flux de intrare

Metode a caror functionare nu este garantata pentru toate fluxurile de intrare:

1. available() - determina cantitatea de date disponibile într-un flux de intrare
2. mark() - marcheaza în fluxul de intrare un punct la care se poate reveni ulterior
3. reset() - revine la un punct specificat în fluxul de intrare

Metoda **close()** închide un flux de intrare (Java închide automat fluxurile la terminarea aplicatiei)
Metoda **available()** determina daca o anumita cantitate de date poate fi citita fara blocarea fluxului de intrare. Returneaza numarul de octeti ce pot fi cititi din fluxul de intrare fara blocare.

Exemplu:

```
public boolean isRecordReady() {
  int recordSize = 512 ;
  boolean ret = false ;
  try {
        if (MyStream.available() >= recordSize)
        ret = true;
       }
  catch (IOException e) { ... }
return ret;
}
```

Fluxuri de iesire

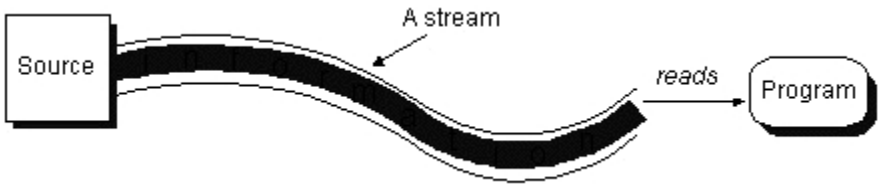
Metode pentru fluxurile de iesire:

1. write() - scrie date într-un flux de iesire
2. flush() - forteaza scrierea datelor într-un canal de redirectare
3. close() - închide un flux de iesire

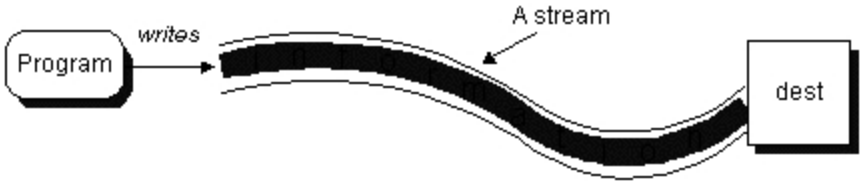
Metoda **flush()** forteaza scrierea catre dispozitivul de iesire a datelor stocate în zona tampon pentru un flux de iesire.
Metoda **close()** închide un flux de iesire (Java închide automat fluxurile la terminarea aplicatiei)
Operatiile read/write se recomanda a fi facute în fire de executie separate care sa nu blocheze programul.

Fluxuri Java. Introducere.

Adeseori programele necesita citirea unor informatii care se gasesc pe o sursa externa sau trimiterea unor informatii catre o destinatie externa. Informatia se poate gasi oriunde : într-un fisier pe disc, în retea, în memorie sau în alt program si poate fi de orice tip: date primitive, obiecte, imagini, sunete, etc. Pentru a aduce informatii dintr-un mediu extern, un progam Java trebui sa deschida un canal de comunicatie (flux) catre sursa informatiilor (fisier, memorie, socket,etc) si sa citeasca serial informatiile respective:



Similar, un program poate trimite informatii catre o destinatie externa deaschizând un canal de comunicatie (flux) catre acea destinatie si scriind serial informatiile respective:



Indiferent de tipul informatiilor, citirea/scrierea informatiilor de pe/catre un mediu extern respecta urmatorii pasi:

Citirea	Scrierea

deschide canal comunicatie
while (mai sunt informatii) {
citeste informatie
}
inchide canal comunicatie;
	deschide canal comunicatie
	while (mai sunt informatii) {
	scrie informatie
	}
	inchide canal comunicatie;

Pentru a generaliza, atât sursa externa a unor informatii cât si destinatia lor sunt vazute ca fiind niste procese care produc, respectiv consuma informatii.

Definitii.

1. Un flux este un canal de comunicatie unidirectional între doua procese.
2. Un proces care descrie o sursa externa de date se numeste proces producator.
3. Un proces care descrie o destinatie externa pentru date se numeste proces
4. consumator.
5. Un flux care citeste date se numeste flux de intrare.
6. Un flux care scrie date se numeste flux de iesire.

Observatii:

1. Fluxurile sunt canale de comunicatie seriale pe 8 sau 16 biti.
2. Fluxurile sunt unidirectionale, de la producator la consumator
3. Fiecare flux are un singur proces producator si un singur proces consumator ③ Intre doua procese pot exista oricâte fluxuri, orice proces putând fi atât producator si consumator în acelasi timp, dar pe fluxuri diferite
4. Consumatorul si producatorul nu comunica direct printr-o interfata de flux ci prin intermediul codului Java de tratare a fluxurilor
5. Clasele si intefetele standard pentru lucu cu fluxuri se gasesc în pachetul java.io.
6. Deci orice program care necesita operatii de intrare/iesire trebuie sa contina instructiunea de import a pachetului java.io: **import java.io.*;**

Clasificarea fluxurilor

Exista trei tipuri de clasificare a fluxurilor:

Dupa "directia" canalului de comunicatie deschis fluxurile se împart în:

a. fluxuri de intrare (pentru citirea datelor)
b. fluxuri de iesire (pentru scrierea datelor)

Dupa tipul de date pe care opereaza:

a. fluxuri de octeti (comunicare seriala se realizeaza pe 8 biti)
b. fluxuri de caractere (comunicare seriala se realizeaza pe 16 biti)

Dupa actiunea lor:

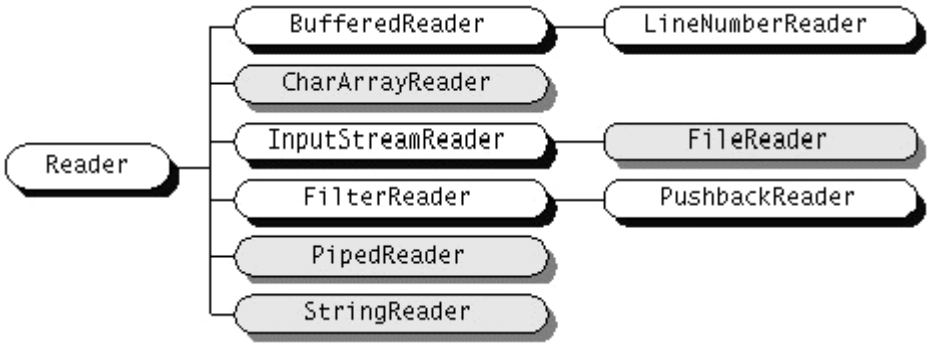
a. fluxuri primare de citire/scriere a datelor (se ocupa efectiv cu citirea/scrierea datelor)
b. fluxuri pentru procesarea datelor

Ierarhia claselor pentru lucrul cu fluxuri

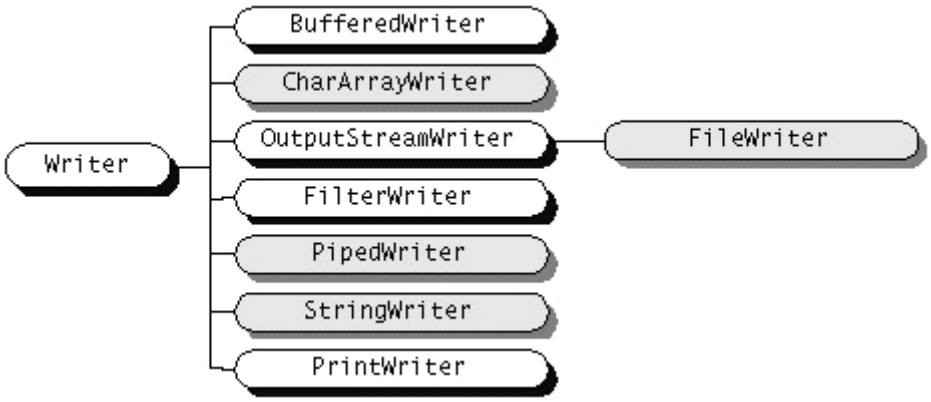
Fluxuri de caractere

Clasele radacina pentru ierarhia claselor ce se ocupa cu fluxurile de caractere sunt **Reader** (pentru fluxuri de intrare) si **Writer** (pentru fluxuri de iesire). Acestea sunt superclase abstracte pentru clase ce implementeaza fluxuri specializate pentru citirea/scrierea datelor pe 16 biti.

Ierarhia claselor pentru fluxuri de intrare pe caractere:



Ierarhia claselor pentru fluxuri de iesire pe caractere:

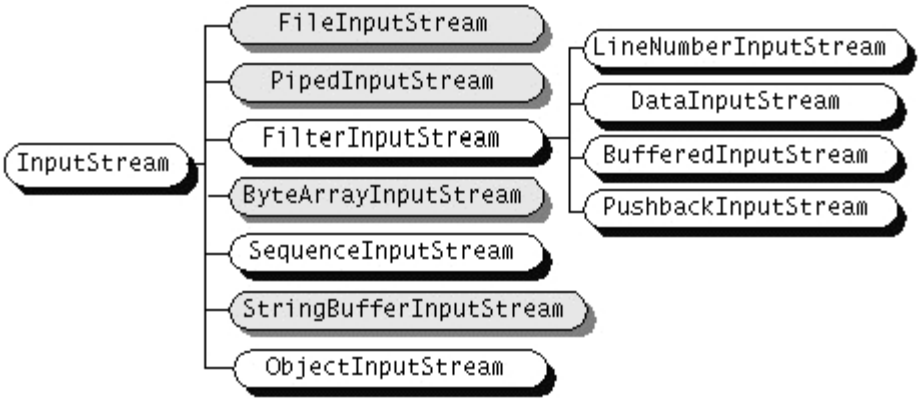


Au fost puse în evidenta (colorate cu gri) fluxurile care intra în categoria fluxurilor pentru procesarea datelor.

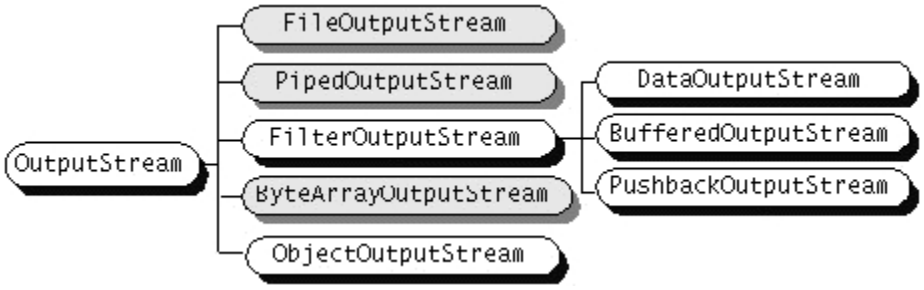
Fluxuri de octeti

Clasele radacina pentru ierarhia claselor ce se ocupa cu fluxurile de octeti sunt **InputStream** (pentru fluxuri de intrare) si **OutputStream** (pentru fluxuri de iesire). Acestea sunt superclase abstracte pentru clase ce implementeaza fluxuri specializate pentru citirea/scrierea datelor pe 8 biti.

Ierarhia claselor pentru fluxuri de intrare pe octeti:



Ierarhia claselor pentru fluxuri de iesire pe octeti:



Au fost puse în evidenta (colorate cu gri) fluxurile care intra în categoria fluxurilor pentru procesarea datelor.

Nota: Pentru majoritatea programelor scrierea si citirea datelor se vor face prin intermediul fluxurilor de caractere deoarece acestea permit manipularea caracterelor Unicode (16-biti), în timp ce fluxurile de octeti permit doar lucrul pe 8 biti - caractere ASCII.

Metode comune fluxurilor

Superclasele abstracte **Reader** si **InputStream** definesc metode similare pentru citirea datelor.

Citire	***Scriere***
int read()	int read()
int read(char buf[])	int read(byte buf[])
int read(char buf[],int offset,int length)	int read(byte buf[],int offset,int length)

De asemenea ambele clase pun la dispozitie metode pentru marcarea unei locatii într-un flux, saltul peste un numar de pozitii, resetarea pozitiei curente, etc.

Superclasele abstracte **Writer** si **OutputStream** sunt de asemenea paralele, definind metode similare pentru scrierea datelor.

Citire	*Scriere*
int write()	int write()
int write(char buf[])	int write(byte buf[])
int write(char buf[], int offset, int length)	int write(byte buf[], int offset, int length)

Inchiderea oricarui flux se realizeaza prin metoda **close**. In cazul în care aceasta nu este apelata explicit fluxul va fi automat închis de catre colectorul de gunoaie (garbage collector-ul masinii virtuale Java) atunci când nu va mai exista nici o referinta la el.

*Metodele referitoare la fluxuri pot genera exceptii de tipul **IOException**.*

Folosirea fluxurilor

Asa cum am vazut fluxurile pot fi împartite în functie de activitatea lor, în fluxuri care se ocupa efectiv cu citirea/scrierea datelor si fluxuri pentru procesarea datelor. In continuare vom vedea care sunt cele mai importante clase din cele doua categorii si la ce folosesc acestea.

Fluxuri pentru citirea/scrierea efectiva a datelor

Clasele ce descriu fluxuri pentru citirea/scrierea efectiva a datelor pot fi împartite în functie de tipul sursei datelor astfel:

Tip sursa	Fluxuri caractere	Fluxuri octeti
Memorie	**CharArrayReader** **CharArrayWriter**	**ByteArrayInputStream** **ByteArrayOutputStream**
	Aceste fluxuri folosesc pentru scrierea/citirea informatiilor în memorie si sunt create pe un vector existent deja. Cu alte cuvinte permit tratarea vectorilor ca sursa/destinatie pentru crearea unor fluxuri de intrare/iesire.	
	StringReader **StringWriter**	**StringBufferInputStream**
	Permit tratarea sirurilor de caractere aflate în memorie ca sursa/destinatie pentru crearea unor fluxuri de intrare/iesire. StringReader si StringWriter sunt folosite cu obiecte de tip String iar StringBufferInputStream cu obiecte de tip StringBuffer.	
Pipe	**PipedReader** **PipedWriter**	**PipedInputStream** **PipedOutputStream**
	Implementeaza componentele de intrare/iesire ale unei conducte de date (pipe). Pipe-urile sunt folosite pentru a canaliza iesirea unui program sau fir de executie catre intrarea altui program sau fir de executie.	
Fisier	**FileReader** **FileWriter**	**FileInputStream** **FileOutputStream**
	Numite si fluxuri fisier, acestea sunt folosite pentru citirea datelor dintr-un fisier, respectiv scrierea datelor într-un fisier.	

Crearea unui flux

Orice flux este un obiect al clasei ce implementeaza fluxul respectiv. Crearea unui flux se realizeaza asadar similar cu crearea obiectelor prin instructiunea new().

Exemple:

```
//crearea unui flux de intrare pe caractere
FileReader in = new FileReader("fisier_intrare.txt");

//crearea unui flux de iesire pe caractere
FileWriter out = new FileWriter("fisier_iesire.txt");

//crearea unui flux de intrare pe octeti
FileInputStream in = new FileInputStream("fisier_intrare.txt");

//crearea unui flux de iesire pe octeti
FileOutputStrem out = new FileOutputStream("fisier_iesire.txt");
```

Crearea unui flux primitiv de date care scrie/citeste informatii de la un dispozitiv extern are formatul general:

FluxPrimitiv numeFlux = new FluxPrimitiv(dispozitiv extern)

Fluxurile de procesare nu pot exista de sine statatoare ci se suprapun pe un flux primitiv de citire/scriere a datelor. Din acest motiv constructorii claselor pentru fluxurile de procesare nu primesc ca argument un dispozitiv extern de memorare a datelor ci o referinta la un flux primitiv responsabil cu citirea/scrierea efectiva a datelor.

Exemple:

```
//crearea unui flux de intrare printr-un buffer
BufferedReader in = new BufferedReader(new FileReader("fisier.in")); //echivalent cu
FileReader fr = new FileReader("fisier.in"); BufferedReader in = new BufferedReader(fr);

//crearea unui flux de iesire printr-un buffer
BufferedWriter out = new BufferedWriter(new FileWriter("fisier.out"))); //echivalent cu
FileWriter fo = new FileWriter("fisier.out"); BufferedWriter out = new BufferedWriter(fo);
```

Crearea unui flux pentru procesarea datelor are formatul general:

```
FluxProcesare numeFlux = new FluxProcesare( referintaFluxPrimitiv )
```

In general, fluxurile pot fi grupate în succesiuni oricât de lungi:

```
DataInputStream in =  new DataInputStream( new BufferedInputStream
                                                (new FileInputStream("fisier.in")));
```

Fluxuri pentru lucrul cu fisiere

Fluxurile pentru lucrul cu fisiere sunt cel mai usor de înteles. Clasele care implementeaza aceste fluxuri sunt urmatoarele:

Tip procesare	Fluxuri caractere	Fluxuri octeti
"Bufferizare"	**BufferedReader** **BufferedWriter**	**BufferedInputStream** **BufferedOutputStream**
	Sunt folosite pentru a introduce un buffer în procesul de scriere/citire a informatiilor, reducând astfel numarul de accese la dispozitivul ce reprezinta sursa originala de date. Sunt mult mai eficiente decât fluxurile fara buffer si din acest motiv se recomanda folosirea lor ori de câte ori eset posibil.	
Filtrare	**FilterReader** **FilterWriter**	**FilterInputStream** **FilterOutputStream**
	Sunt clase abstracte ce definesc o interfata pentru fluxuri care filtreaza automat datele citite sau scrise.	
Conversie octeti-caractere	**InputStreamReader,** **OutputStreamWriter**	
	Formeaza o punte de legatura între fluxurile de caractere si fluxurile de octeti. Un flux InputStreamReader citeste octeti dintr-un flux InputStream si îi converteate la caractere folosind codificarea standard a caracterelor sau o codificare specificata de program. Similar, un flux OutputStreamWriter converteste caractere în octeti si trimite rezutatul catre un flux de tipul OutputStream.	
Concatenare		**SequenceInputStream**
	Concateneaza mai multe fluxuri de intrare într-unul singur.	
Serializare		**ObjectInputStream** **ObjectOutputStream**
	Folosite pentru serializarea obiectelor.	
Conversie tipuri de date		**DataInputStream** **DataOutputStream**
	Folosite la scrierea/citirea datelor de tip primitiv într-un format independent de masina pe care se lucreaza.	
Numarare	**LineNumberReader**	**LineNumberInputStream**
	Numara liniile citite de la un flux de intrare.	
Citire în avans	**PushbackReader**	**PushbackInputStream**
	Fluxuri de intrare care au un buffer de 1-caracter(octet) în care este citit în avans si caracterul (octetul) care urmeaza celui curent citit.	
Afisare	**PrintWriter**	**PrintStream**
	Metode convenabile pentru afisarea informatiilor.	

FileReader, FileWriter - caractere
FileInputStream, FileOutputStream - octeti

Constructorii acestor clase accepta ca argument un obiect care sa specifice un anume fisier. Acesta poate fi un sir de caractere, un obiect de tip **File** sau un obiect de tip **FileDesciptor**.

Constructorii clasei FileReader:

```
public FileReader( String fileName )
    throws FileNotFoundException
public FileReader( File file )
    throws FileNotFoundException
public FileReader( FileDescriptor fd )
```

Constructorii clasei FileWriter:

```
public FileWriter( String fileName )
    throws IOException
public FileWriter( File file )
    throws IOException
public FileWriter( FileDescriptor fd )
public FileWriter( String fileName, boolean append )
    throws IOException
```

Cei mai uzuali constructori sunt cei care primesc ca argument numele fisierului. Acestia pot provoca exceptii de tipul **FileNotFoundException** în cazul în care fisierul cu numele specificat nu exista. Din acest motiv orice creare a unui flux de acest tip trebuie facuta într-un bloc **try** sau metoda în care sunt create fluxurile respective trebuie sa arunce exceptiile de tipul **FileNotFoundException** sau de tipul superclasei **IOException**.

Exemplu: un program care copie continutul unui fisier în alt fisier.

```
import java.io.*;
public class Copy {
public static void main(String[] args) throws IOException {
   FileReader in = new FileReader("in.txt");
   FileWriter out = new FileWriter("out.txt");
   int c;
   while ((c = in.read()) != -1)
            out.write(c);
   in.close();
   out.close();
   }
}
```

Nota: metoda **main** genereaza exceptii **IOException** care este superclasa pentru **FileNotFoundException**. Aceste exceptii nu vor fi tatate ("catch") decât de interpretor si va fi afisat un mesaj de eroare la aparitia lor.

Analiza lexicala pe fluxuri (clasa StreamTokenizer)

Clasa StreamTokenizer parcurge un flux de intrare de orice tip si îl împarte în "atomi lexicali".

Rezultatul va consta în faptul ca în loc sa se citeasca octeti sau caractere se vor citi, pe rând, atomii lexicali ai fluxului respectiv.

Printr-un atom lexical se întelege în general:

1. un identificator (un sir care nu este între ghilimele)
2. un numar
3. un sir de caractere
4. un comentariu
5. un separator

Atomii lexicali sunt despartiti între ei de **separatori**. Implicit acesti separatori sunt cei obisnuti (spatiu, tab, virgula, punct si virgula), însa pot fi schimbati prin diverse metode ale clasei.

Constructorii acestei clase sunt:

```
public StreamTokenizer( Reader r )
public StreamTokenizer( InputStream is )
```

Identificarea tipului si valorii unui atom lexical se face prin intermediul variabilelor clasei **StreamTokenizer**:

TT_EOF	- atom ce marcheaza sfârsitul fluxului
TT_EOL	- atom ce marcheaza sfârsitul unei linii
TT_NUMBER	- atom de tip numar
TT_WORD	- atom de tip cuvânt
nval	- valoarea unui atom numeric
sval	- sirul continut de un atom de tip cuvânt
ttype	- tipul ultimului atom citit din flux

Citirea atomilor din flux se face cu metoda **nextToken()**, care returneza tipul atomului lexical citit si scrie în variabilele **nval** sau **sval** valoarea corespunzatoare atomului.

Exemplul tipic de folosire a unui analizor lexical este citirea unei secvente de numere si siruri aflate într-un fisier sau primite de la tastatura:

```
//Citirea unei secvente de numere si siruri
import java.io.*;

public class TestTokenizer {
public static void main(String args[]) throws IOException {
    BufferedReader br = new BufferedReader( new InputStreamReader(
                                                    new FileInputStream("test.dat")));
    StreamTokenizer st = new StreamTokenizer(br);
```

```
    int tip = st.nextToken();                                   // citirea primul atom lexical
    while (tip != StreamTokenizer.TT_EOF) {
          switch (tip) {
              case StreamTokenizer.TT_WORD:                                          //cuvant
                    System.out.println(st.sval); break;
             case StreamTokenizer.TT_NUMBER:                                         //numar
                    System.out.println(st.nval); break;
           }
         tip = st.nextToken(); //urmatorul atom
       }
    }
}
```

Asadar, modul de utilizare tipic pentru un analizor lexical este într-o bucla "while" în care se citesc atomii unul câte unul cu metoda **nextToken()** pâna se ajunge la sfârsitul fluxului (**TT_EOF**). In cadrul buclei "while" se afla tipul atomului curent (întors de metoda **nextToken()**) si in functie de tip se afla valoarea numerica/sir de caractere corespunzatoare acestuia.

Alte clase pentru lucrul cu fisiere

Clasa RandomAccessFile

Fluxurile sunt, asa cum am vazut procese secventiale de intrare/iesire. Acestea sunt adecvate pentru scrierea/citirea pe medii secventiale de memorare a datelor (de exemplu: banda magnetica), dar sunt foarte utile si pentru dispozitive în care informatia poate fi accesata direct.

Clasa RandomAccesFile:

1. permite accesul nesecvential (direct) la continutul unui fisier.
2. este o clasa de sine statatoare, subclasa directa a clasei Object.
3. se gaseste în pachetul java.io.
4. implementeaza interfetele **DataInput** si **DataOutput**, ceea ce înseamna ca sunt disponibile metode de tipul **readXXX** si **writeXXX**.
5. permite atât citirea cât si scriere din/in fisiere
6. permite specificarea modului de acces al unui fisier (read-only, read-write)

Constructorii acestei clase sunt:

```
RandomAccessFile(String numeFisier, String mod_acces)
        throws IOException
RandomAccessFile(String fisier, String mod_acces)
        throws IOException
```

unde mod_acces poate fi:

"r" - fisierul este deschis numai pentru citire (read-only)
"rw" - fisierul este deschis pentru citire si scriere (read-write)

Exemple:

```
RandomAccesFile f1 = new RandomAccessFile("f.txt", "r"); //deschide fisierul f.txt pentru citire

RandomAccesFile f2 = new RandomAccessFile("f.txt", "rw");
                                    //deschide fisierul f.txt pentru scriere si citire
```

Clasa RandomAccesFile suporta notiunea de pointer de fisier. Acesta este un indicator ce specifica pozitia curenta în fisier. La deschiderea unui fisier pointerul are valoarea 0, indicând începutul fisierului. Apeluri de metode ***readXXX*** *sau* ***writeXXX*** *deplaseaza pointerul fisierului cu numarul de octeti cititi sau scrisi de metodele respective.*

In plus fata de metodele de citire/scriere clasa pune la dispozitie si metode pentru controlul pozitiei pointerului de fisier. Acestea sunt:

```
int skipBytes ( int n )    // Muta pointerul fisierului cu un numar specificat de octeti
void seek (long pozitie)  // Pozitioneaza pointerului fisierului pe octetului specificat.
long getFilePointer ( )     //Returneaza pozitia pointerului de fisier (pozitia de la care se citeste/la care se scrie)
```

Clasa File

Clasa File nu se refera doar la un fisier ci poate reprezenta fie un fisier anume, fie multimea fisierelor dintr-un director. O instanta a acestei clase poate sa reprezinte asadar: un fisier sau un director.

Specificarea unui fisier/director se face prin specificare caii absolute spre acel fisier sau a caii relative fata de directorul curent. Acestea trebuie sa respecte conventiile de specificare a cailor si numelor fisierelor de pe masina gazda.

Utilitatea clasei ***File*** *consta în furnizarea unei modalitati de a abstractiza dependentele cailor si numelor fisierelor fata de masina gazda precum si punerea la dispozitie a unor metode pentru lucrul cu fisiere si directoare la nivelul sistemului de operare.*

Astfel, aceasta clasa furnizeaza metode pentru testarea existentei, stergerea, redenumirea unui fisier sau director, crearea unui director, listarea fisierelor dintr-un director, etc.

Majoritatea fluxurilor care permit accesul la fisiere furnizeaza si constructori care accepta ca argument obiecte de tip File.

```
File f = new File("fisier.txt");
FileInputStream is = new FileInputStream(f);
```

Cel mai uzual constructor al clasei File este:

```
public File( String fisier)
```

Metodele mai importante ale clasei File sunt:

boolean isDirectory( ) boolean isFile( )	Testeaza daca un obiect **File** reprezinta un fisier sau un director
String getName( ) String getPath( ) String getAbsolutePath( ) String getParent( )	Returneaza: numele (fara cale) calea fisierului sau directorului reprezentat de obiectul respectiv
boolean exists( ) boolean delete( ) boolean mkdir( ) boolean mkdirs( )	Testeaza daca exista un anumit fisier/director Sterge fisierul/directorul reprezentat de obiect Creeaza un director Creeaza o succesiune de directoare

boolean renameTo(File dest)	Redenumeste un fisier/director
String[] list( ) String[] list (FilenameFilter filter)	Returneaza o lista cu numele fisierelor dintr-un director Returneaza o lista cu numele fisierelor dintr-un director filtrate dupa un anumit criteriu specificat.
boolean canRead( ) boolean canWrite( )	Testeaza daca fisierul/directorul reprezentat de obiectul **File** poate fi folosit pentru citire, respectiv scriere
long length( ) long lastModified( )	Returneaza lungimea (in bytes) si data ultimei modificari.