Lecţia 4

Câteva clase Java predefinite

În Lecţia 2 am văzut cum definim şi creăm un obiect într-un limbaj de programare obiectual, în particular Java. Pentru a fixa mai bine noţiunile independente de limbaj clasă şi obiect dar şi pentru a învăţa anumite particularităţi ale limbajului Java vom trece în revistă câteva clase predefinite.

4.1 Clasa String

4.1.1 Şirul de caractere în Java

În Java nu există tipul primitiv şir de caractere. Orice şir de caractere este un obiect instanţă a *String*, iar variabilele de tip *String* nu sunt altceva decât referinţe la obiecte şir de caractere. Conform cu modul de creare a obiectelor studiat în Lecţia 2, pentru a crea un şir de caractere putem proceda după cum urmează:

```
String sir = new String("Un sir de caractere");
```

Datorită faptului că şirurile de caractere sunt frecvent utilizate în cadrul programelor, compilatorul de Java vine în ajutorul programatorilor simplificând lucrul cu şirurile de caractere. Ca urmare, orice constantă şir de caractere poate fi folosită ca o referinţă la un obiect *String*, compilatorul fiind cel care se ocupă de crearea obiectului propriu-zis. Prin urmare, putem declara şi iniţializa o variabilă şir de caractere şi în modul prezentat mai jos.

```
String sir = "Un sir de caractere";
```

Trebuie amintit aici că cele două secvenţe de cod nu sunt perfect echivalente deoarece constanta “Un sir de caractere” este deja un obiect. Prin urmare, în al doilea exemplu variabila *sir* va referi obiectul creat implicit de compilator, pe când în primul exemplu variabila va referi un obiect *String* ce copiază conţinutul obiectului creat implicit

de compilator. De cele mai multe ori însă, acest comportament diferit nu e foarte important.

4.1.2 Operaţii cu şiruri de caractere

În Tabelul 4.1 prezentăm un subset de constructori şi metode definite de clasa *String* pentru lucrul cu şiruri de caractere. Mult mai multe pot fi consultate în documentaţia Java, la adresa web *http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html*. Una dintre metodele prezentate este *concat(String sir)* care crează un nou obiect *String* (adică un şir de caractere) obţinut prin concatenarea şirului apelat (adică a secvenţei de caractere corespunzătoare obiectului apelat) cu şirul dat ca parametru. Acelaşi efect se obţine şi prin utilizarea operatorului '+' cu operanzi şiruri de caractere. Prin urmare, următoarele porţiuni de cod conduc la crearea unui obiect *String* ce va fi referit de variabila *sir2*, obiect ce conţine secvenţa de caractere "Un sir de caractere".

```
String sir1 = "Un sir ";
String sir2 = sir1.concat("de caractere");
```

```
String sir1 = "Un sir ";
String sir2 = sir1 + "de caractere";
```

Atenţie Operaţia de concatenare produce un alt şir de caractere (alt obiect) şi nu-l modifică pe cel asupra căruia se aplică metoda *concat*. La fel se întâmplă şi în cazul altor metode care realizează anumite transformări ale şirurilor de caractere. Cu alte cuvinte, un şir de caractere odată creat nu mai poate fi modificat.

Atenţie Este corect să folosim operatorul + pentru a concatena două şiruri de caractere, dar nu este corect să utilizăm operatorul == pentru a vedea dacă două şiruri conţin aceeaşi secvenţă de caractere.

O altă metodă importantă e *equals(Object o)*. Ea testează dacă şirul de caractere apelat şi cel dat ca parametru conţin aceeaşi secvenţă de caractere. În acest context este foarte important să înţelegem rolul operatorului ==. El testează dacă valoarea celor doi operanzi este identică. Variabilele de tip referinţă (cum sunt şi variabilele de tip *String*) au ca valoare o referinţă la un obiect, iar egalitatea a două astfel de variabile denotă că ele referă acelaşi obiect! Prin urmare, dacă două variabile *String* sunt egale înseamnă că ele referă acelaşi obiect şi implicit aceeaşi secvenţă de caractere. Dar două obiecte diferite pot şi ele conţine aceeaşi secvenţă de caractere, lucru ce nu poate fi testat prin ==. Pentru a testa acest aspect trebuie folosită metoda *equals* prezentată în Tabelul 4.1.

Prototipul	Descriere
String()	Crează un obiect *String* corespunzător şirului de caractere vid
String(String str)	Crează un obiect *String* corespunzător aceluiaşi şir de caractere ca şi şirul corespunzător obiectului argument
char charAt(int index)	Returnează caracterul aflat pe poziţia *index* în şirul de caractere apelat
String concat(String str)	Returnează un nou obiect *String* reprezentând şirul de caractere obţinut prin concatenarea şirului dat ca argument la sfârşitul şirului apelat. Dacă *str* este şirul vid atunci se returnează obiectul apelat
boolean endsWith(String suffix)	Testează dacă şirul apelat se termină cu şirul *suffix*
boolean equals(Object str)	Testează dacă şirul apelat reprezintă aceeaşi secvenţă de caractere ca şi şirul dat ca parametru (nu acelaşi obiect şir de caractere !!!)
int indexOf(int ch)	Returnează cea mai din stânga poziţie din şirul apelat în care apare caracterul *ch*. Dacă ch nu apare în şir atunci se returnează -1
int indexOf(String str)	Returnează cea mai din stânga poziţie din şirul apelat de la care apare subşirul dat ca argument. Dacă *str* nu apare ca subşir se returnează -1
String intern()	Returnează referinţa la şirul de caractere care conţine aceeaşi secvenţă de caractere ca şi obiectul apelat (sunt egale din punctul de vedere a lui equals) şi care a fost primul astfel de şir de caractere pentru care s-a apelat metoda *intern*
int lastIndexOf(int ch)	Returnează cea mai din dreapta poziţie din şirul apelat în care apare caracterul *ch*. Dacă ch nu apare în şir atunci se returnează -1
int lastIndexOf(String str)	Returnează cea mai din dreapta poziţie din şirul apelat de la care apare subşirul dat ca argument. Dacă *str* nu apare ca subşir se returnează -1
int length()	Returnează lungimea şirului de caractere apelat
boolean startsWith(String prefix)	Testează dacă şirul apelat începe cu şirul *prefix*
String substring(int beginIndex)	Returnează un nou şir de caractere reprezentând subşirul de caractere din şirul apelat care începe la poziţia dată ca argument
String toUpperCase()	Returnează un nou şir de caractere conţinând acelaşi şir de caractere ca şi cel apelat dar în care toate literele mici sunt convertite în litere mari

Tabelul 4.1: Constructori şi metode pentru obiecte de tip String.

```
String sir1 = "Un sir";
String sir2 = "Un sir";

if(sir1 == sir2) {
    System.out.println("sir1 este egal cu sir2");
}
```

Explicaţia de mai sus ar putea fi suspectă ca eronată la rularea porţiunii de cod prezentate mai sus. La rulare, mesajul "sir1 este egal cu sir2" este tipărit pe ecran, deşi la prima vedere *sir1* şi *sir2* nu referă acelaşi obiect. Prin urmare s-ar putea crede că operatorul == testează egalitatea secvenţei de caractere dintr-un obiect şir de caractere. Ei bine, este total eronat. Mesajul este afişat pentru că *sir1* şi *sir2* referă acelaşi obiect. În general, orice expresie constantă care are ca valoare un şir de caractere este "internalizată" automat de compilator. Mai exact, pe obiectul *String* ce reprezintă valoarea sa este apelată metoda *intern* iar rezultatul dat de ea este returnat ca valoare a expresiei. Urmărind descrierea metodei *intern* din Tabelul 4.1 este clar de ce *sir1* şi *sir2* referă acelaşi obiect.

Ca să ne convingem că orice şir de caractere dintr-un program Java este un obiect încercaţi următorul cod. Surpriză! Este chiar corect.

```
class Surpriza {

    public static void main(String argv[]) {
        System.out.println("Acest sir are lungimea 25".length());
    }
}
```

4.1.3 Alte metode definite de clasa String

În plus faţă de metodele prezentate în Tabelul 4.1, clasa *String* mai defineşte un set de metode statice. Reamintim că metodele statice nu sunt executate pe un obiect, ele reprezentând operaţii ce caracterizează clasa căreia aparţin şi nu obiectele definite de respectiva clasă. În principiu, după cum se poate observa din Tabelul 4.2, aceste metode sunt utile pentru obţinerea reprezentării sub formă de şir de caractere a valorilor corespunzătoare tipurilor primitive.

Prototipul	Descriere
String valueOf(boolean b)	Returnează un şir de caractere corespunzător valorii logice date de argumentul *b*
String valueOf(char c)	Returnează un şir de caractere format numai din caracterul *c*
String valueOf(double d)	Returnează un şir de caractere corespunzător valorii argumentului *d*
String valueOf(float f)	Returnează un şir de caractere corespunzător valorii argumentului *f*
String valueOf(int i)	Returnează un şir de caractere corespunzător valorii argumentului *i*
String valueOf(long l)	Returnează un şir de caractere corespunzător valorii argumentului *l*

Tabelul 4.2: Un subset de metode statice definite de clasa String.

4.2 Clase înfăşurătoare

4.2.1 Orice ar putea fi obiect

Într-un limbaj de programare pur obiectual, totul e considerat a fi obiect şi tratat ca atare. Ca urmare, inclusiv valorile numerice sau logice corespunzătoare tipurilor primitive sunt considerate a fi obiecte. Java nu merge chiar atât de departe, dar predefineşte câte o clasă pentru fiecare tip primitiv, denumită clasă înfăşurătoare pentru acel tip primitiv. O clasă înfăşurătoare poate fi folosită de programatori dacă se doreşte tratarea unei valori corespunzătoare tipului primitiv asociat ca obiect. Fiecare clasă înfăşurătoare are o denumire sugestivă care spune aproape totul despre ea: *Boolean*, *Character*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*.

În general, toate clasele înfăşurătoare prezintă constructori şi metode similare. Din acest motiv noi ne vom rezuma aici doar asupra constructorilor şi metodelor definite de clasa *Integer* amintind punctual eventuale particularităţi ale celorlalte clase acolo unde este necesar (vezi Tabelul 4.3). Detalii despre metodele definite de clasele înfăşurătoare pot fi găsite în documentaţia Java la adresa web *http://java.sun.com/j2se/1.5.0/docs/api/*. Pe lângă aceste metode, clasa *Integer* defineşte şi metode statice pe care le prezentăm în Tabelul 4.4.

4.2.2 Mecanismul de autoboxing şi unboxing

După cum am văzut în secţiunea anterioară, utilizarea claselor înfăşurătoare permite tratarea valorilor tipurilor primitive ca şi obiecte. Spre deosebire însă de şirurile de caractere care sunt întotdeauna obiecte, tratarea valorilor tipurilor primitive ca şi obiecte trebuie realizată explicit de către programator. Să considerăm un exemplu concret.

Prototipul	Descriere	În alte clase înfăşurătoare
Integer(int value)	Crează un obiect de tip *Integer* pentru valoarea *int* dată ca parametru	Fiecare clasă are un constructor asemănător ce ia ca parametru o valoare de tipul asociat clasei respective (de exemplu, *Double* are un construtor *Double(double value)*)
Integer(String s)	Crează un obiect de tip *Integer* pentru valoarea *int* obţinută prin conversia şirului de caractere dat ca parametru la *int*	Cu excepţia clasei *Character*, toate celelalte au un constructor asemănător convertind şirul de caractere la o valoare de tipul asociat clasei respective
String toString()	Returnează un şir de caractere reprezentând valoarea int conţinută de obiect	Toate clasele definesc o astfel de metodă returnând şirul de caractere ce reprezintă valoarea înfăşurată
int intValue()	Returnează valoarea de tip *int* înfăşurată	Toate clasele definesc o metodă ce returnează valoarea înăşurată (de exemplu, *Character* defineşte *char charValue()*). Pentru tipurile numerice se defineşte câte o metodă pentru a returna aceeaşi valoare convertită la un alt tip numeric (*Double* defineşte şi *int intValue()*)
boolean equals(Object o)	Testează dacă obiectul apelat şi cel dat ca parametru înfăşoară aceeaşi valoare *int*	Toate clasele definesc această metodă
int compareTo(Integer i)	Returnează o valoare negativă dacă obiectul apelat înfăşoară o valoare int mai mică decât cel dat ca parametru, 0 dacă sunt egale, şi o valoare pozitivă în caz contrar	Toate clasele definesc o astfel de metodă care primeşte ca parametru o instanţă a aceleaşi clase. Citiţi documentaţia Java pentru a vedea ce se întâmplă în cazul clasei *Boolean*

Tabelul 4.3: Constructori şi metode pentru obiecte de tip Integer.

Prototipul	Descriere	În alte clase înfăşurătoare
Integer valueOf(int value)	Construieşte un obiect *Integer* ce înfăşoară valoarea dată ca parametru. Este bine să utilizăm această metodă atunci când nu trebuie să creăm efectiv obiectul deoarece metoda foloseşte un cache pentru a nu mai crea noi obiecte dacă o aceeaşi valoare *int* este înfăşurată de mai multe ori	Cu excepţia clasei *Character*, toate clasele înfăşurătoare definesc o astfel de metodă statică, având ca parametru o valoare corespunzătoare tipului înfăşurat (de exemplu, *Double* defineşte *Double valueOf(double value)*)
int parseInt(String s)	Returnează valoarea întreagă reprezentată în şirul de caractere dat ca parametru	Cu excepţia clasei *Character*, toate clasele înfăşurătoare definesc o astfel de metodă statică ce converteşte şirul de caractere într-o valoare corespunzătoare tipului asociat clasei respective (de exemplu *Boolean* defineşte *Boolean parseBoolean(String s)*)

Tabelul 4.4: Metode statice definite de clasa Integer.

```
class Autoboxing {

    public static void tiparesteIntreg(Integer x) {
        System.out.println("Intregul este:" + x);
    }

    public static void main(String argv[]) {
        tiparesteIntreg(5);
    }
}
```

În secvenţa de cod de mai sus există o problemă. Parametrul metodei *tiparesteIntreg* trebuie să fie o referinţă la un obiect de tip *Integer* şi nu o valoare de tip *int*! Prin urmare, compilarea acestui cod va genera o eroare [1]. Pentru a rezolva problema fără a modifica metoda *tiparesteIntreg*, va trebui să creăm un obiect de tip *Integer* care să înfăşoare valoarea 5 şi pe care să-l trimitem ca parametru metodei. Prin urmare, apelul corect al metodei este:

```
tiparesteIntreg(new Integer(5));
```

[1] Dacă se utilizează un compilator anterior versiunii 1.5.

Generalizând, este posibil ca la un moment dat să deţinem o referinţă la un obiect ce înfăşoară o valoare de tip primitiv iar noi să avem nevoie în acel moment de valoarea propiu-zisă. Ca urmare, va trebui să apelăm explicit o metodă corespunzătoare pentru a determina această valoare (de exemplu pentru un obiect *Integer* valoarea înfăşurată e dată de metoda *intValue()*). Pe de altă parte este posibil ca la un moment dat să deţinem o valoare de un anumit tip primitiv, iar noi să avem nevoie de tratarea respectivei valori ca un obiect, adică avem nevoie de o referinţă la un obiect ce înfăşoară respectiva valoare. Ca urmare, va trebui să creăm explicit obiectul ca în exemplul de mai sus.

Toate aceste operaţii care trebuie realizate explicit de către programator *aglomerează* artificial codul sursă al programului. Pentru a veni în ajutorul programatorilor, începând cu versiunea 1.5 a limbajului Java, compilatorul de Java furnizează aşa numitele mecanisme de *autoboxing* şi *unboxing*. Aceste mecanisme vin să rezolve problema amintită mai sus, permiţându-ne să ignorăm diferenţele dintre un tip primitiv şi tipul definit de clasa înfăşurătoare asociată respectivului tip primitiv. Astfel, începând cu versiunea Java 1.5, exemplul dat la începutul acestei secţiuni compilează fără eroare! Acest lucru se întâmplă pentru că mecanismul de *autoboxing* crează automat un obiect înfăşurător pentru valoarea 5 care va fi dat ca argument metodei *tiparesteIntreg*. Cu alte cuvinte compilatorul Java 1.5 crează implicit obiectul pe care programatorul era obligat să-l creeze explicit dacă folosea un compilator anterior versiunii 1.5.

Mecanismul de *unboxing* este inversul mecanismului de *autoboxing*. Să condiderăm secvenţa de mai jos.

```
class Unboxing {

    public static void tiparesteIntreg(Integer x) {
        System.out.println(5 + x);
    }

    public static void main(String argv[]) {
        tiparesteIntreg(new Integer(5));
    }
}
```

Pentru un compilator de Java anterior versiunii 1.5 exemplul de mai sus conţine o eroare. Nu se poate aduna o referinţă la un obiect, în acest caz de tip *Integer*, cu valoarea 5 pentru că pur şi simplu nu are sens. Pentru compilatorul de Java 1.5 codul este corect datorită mecanismului de *unboxing*. Astfel, compilatorul îşi dă seama că trebuie să adune o valoare întreagă cu valoarea înfăşurată de obiectul referit de parametrul x pentru că obiectul respectiv este instanţă a clasei *Integer*. Ca urmare, compilatorul apelează în mod implicit metoda *intValue()* pentru obiectul referit de x, iar valoarea returnată este apoi adunată la 5. La execuţie, acest program va afişa valoarea 10.

Concluzionând, mecanismele de autoboxing şi unboxing introduse în Java 1.5 simplifică sursa programelor permiţând tratarea valorilor corespunzătoare tipurilor primitive ca şi instanţe ale claselor înfăşurătoare corespunzătoare, respectiv tratarea instanţelor claselor înfăşurătoare ca şi valori primitive corespunzătoare.

4.3 Clase destinate operaţiilor de intrare-ieşire

Există un număr mare de clase predefinite în Java destinate realizării operaţiilor de intrare-ieşire. Explicarea exactă a rolului fiecărei clase şi a posibilităţilor de combinare a instanţelor lor este destul de dificilă în acest moment, când încă nu cunoaştem toate mecanismele specifice programării orientate pe obiecte. Din acest motiv ne vom limita la exemple de citire a şirurilor de caractere (conversia lor la valori corespunzătoare tipurilor primitive a fost descrisă mai devreme în această lucrare) şi la exemple de realizare a operaţiilor de ieşire.

Abstracţiunea de bază utilizată în cadrul operaţiilor de intrare şi ieşire este fluxul de intrare, respectiv fluxul de ieşire. Un flux de intrare poate fi văzut ca o secvenţă de "entităţi" care "vin" sau care "curg" către un program din exteriorul programului respectiv. Analog, un flux de ieşire poate fi văzut ca o secvenţă de "entităţi" care "pleacă" sau care "curg" dinspre un program spre exteriorul său. La un moment dat, pot exista mai multe fluxuri de intrare şi de ieşire pentru un program.

La cel mai primitiv nivel o astfel de "entitate" este octetul, vorbindu-se astfel de flux de intrare de octeţi respectiv de flux de ieşire de octeţi. În Java aceste abstracţiuni se numesc *InputStream* respectiv *OutputStream*. În general, un obiect corespunzător acestor abstracţiuni ştie să citească următorul octet din flux (metoda *read()*), respectiv ştie să scrie următorul octet în flux (metoda *write(int b)*). În ambele cazuri obiectul ştie să închidă fluxul prin metoda *close()*.

4.3.1 Citirea liniilor de text

La iniţializarea unui program Java, se crează implicit un obiect corespunzător abstracţiunii *InputStream* care ştie să ne furnizeze prin metoda *read()* un octet provenit de la intrarea standard a programului reprezentată de cele mai multe ori de tastatură. Acest obiect poate fi accesat prin referinţa *System.in*.

Dacă se doreşte citirea dintr-un fişier vom crea un obiect corespunzător abstracţiunii *InputStream* prin instanţierea clasei *FileInputStream*. Acest obiect ştie să ne furnizeze următorul octet disponibil din fişier prin metoda *read()*.

```
FileInputStream file_stream = new FileInputStream("fisierul_meu.txt");
```

Pentru unii ar putea fi greu de înţeles cum atât obiectul *System.in* cât şi un obiect instaţă a clasei *FileInputStream* poate corespunde abstracţiunii *InputStream*. Imaginaţi-vă că pe cineva îl interesează cât este ora la un moment dat. Pentru a rezolva această problemă are nevoie de un obiect corespunzător abstracţiunii Ceas care ştie să-i spună cât e ora. Pentru a afla efectiv ora persoana respectivă poate să se uite la un ceas de mână dar poate să se uite şi la telefonul său mobil. Prin urmare, ambelor obiecte le corespunde aceeaşi abstracţiune din perspectiva întrebării "Cât este ora?". Asemănător, din perspectiva întrebării "Care e următorul octet din fluxul de intrare?" atât obiectului *System.in* cât şi oricărei instanţe a clasei *FileInputStream* le poate corespunde aceeaşi abstracţiune, în cazul nostru *InputStream*.

Utilizând obiectele descrise mai sus, am putea să citim informaţii de la tastatură sau dintr-un fişier octet cu octet. Totuşi, acest lucru nu ne prea ajută dacă vrem să citim un caracter sau, mai rău, un şir de caractere.

Atenţie

În Java un caracter nu e reprezentat printr-un octet care conţine codul ASCII al caracterului respectiv !!!

Prin urmare, ar fi foarte util un alt obiect care să convertească un flux de intrare de octeţi într-un flux de intrare de caractere. Un astfel de obiect se poate obţine prin instanţierea clasei *InputStreamReader*. Constructorul acestei clase primeşte ca parametru fluxul de intrare de octeţi care va fi convertit.

```
InputStreamReader keybord_char_stream = new InputStreamReader(System.in);

InputStreamReader file_char_stream =
    new InputStreamReader(new FileInputStream("fisierul_meu.txt"));
```

Interfaţa acestui obiect declară metoda *read()* prin intermediul căreia se poate citi următorul caracter disponibil din fluxul de intrare. Mai rămâne de rezolvat o singură problemă: eficienţa citirilor. Citirea unui caracter din fluxul de intrare de caractere implică citirea unui sau mai multor octeţi din fluxul de intrare de octeţi. Din motive de organizare a discului, citirea octet cu octet a unui fişier este ineficientă ca timp. Pentru a creşte eficienţa citirilor este de dorit ca la un moment dat să se citească mai mulţi octeţi într-un singur acces la disc. În cazul nostru acest lucru implică citirea mai multor caractere şi păstrarea lor în memorie. Acest lucru se poate realiza prin crearea unui obiect al clasei BufferedReader aşa cum se arată mai jos.

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

BufferedReader file_char_stream =
    new BufferedReader(new InputStreamReader(
        new FileInputStream("fisierul_meu.txt")));
```

Sfat

Documentaţia Java sfătuieşte programatorii să utilizeze un obiect *BufferedReader* atunci când se realizează citiri din fişiere dar şi pentru a evita lucrul direct cu obiecte *InputStreamReader*, care prezintă o implementare destul de ineficientă a metodei *read()*.

Gândiţi-vă că o persoană trebuie să prelucreze într-un anumit fel boabele de grâu dintr-un hambar situat la 1 kilometru distanţă de locul unde trebuie realizată prelucrarea. Ar fi cam ineficient ca pentru fiecare bob în parte persoana să se deplaseze la hambar pentru a lua bobul după care să-l prelucreze. Pentru a creşte eficienţa activităţii sale persoana va aduce la locul prelucrării un sac de boabe de grâu. Ei bine, clasa *BufferedReader* defineşte un obiect care conţine un astfel de "sac" de caractere. Când sacul se goleşte obiectul îl umple la loc, eliberându-l pe clientul său de sarcina reîncărcării "sacului" şi lăsându-l să se ocupe doar de prelucrarea efectivă a caracterelor.

Odată ce am creat un obiect *BufferedReader* aşa cum am arătat mai sus, putem utiliza metodele *read()* şi *readLine()* pentru a citi eficient următorul caracter din flux-ul de intrare respectiv pentru a citi o linie de text din acelaşi flux. Aceste metode returnează *0* respectiv *null* când nu mai sunt caractere disponibile în fluxul de intrare (a apărut sfârşitul de fişier).

4.3.2 Scrierea liniilor de text

Discuţia din secţiunea anterioară este în esenţă aplicabilă şi în cazul operaţiilor de ieşire. Singura diferenţă este că lucrurile trebuie privite în sens invers, dinspre program spre exteriorul său. Vorbim astfel de fluxuri de ieşire de caractere şi fluxuri de ieşire de octeţi. Clasa *OutputStreamWriter* defineşte un obiect care transformă un flux de caractere într-un flux de octeţi. Aceşti octeţi sunt trimişi apoi spre un obiect *OutputStream* dat ca parametru constructorului clasei *OutputStreamWriter*. După cum se poate observa, lucrurile sunt foarte asemănătoare cu cele prezentate în secţiunea anterioară.

La iniţializarea unui program Java se crează implicit un obiect corespunzător abstracţiunii *OutputStream* care ştie să scrie prin metoda *write(int b)* un octet la ieşirea standard a programului, reprezentată de cele mai multe ori de monitor. Acest obiect poate fi accesat prin referinţa *System.out*.

Dacă se doreşte scrierea într-un fişier vom crea un obiect corespunzător abstracţiunii *OutputStream* prin instanţierea clasei *FileOutputStream*. În continuare putem folosi un obiect *OutputStreamWriter* care ne permite să lucrăm cu caractere şi nu cu octeţi. Metoda *write(int b)* a acestui obiect permite scrierea unui caracter în fluxul de ieşire.

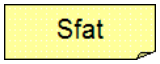
```
FileOutputStream file_stream = new FileOutputStream("fisierul_meu.txt");
OutputStreamWriter file_char_stream = new OutputStreamWriter(file_stream);
```

Totuşi, utilizând un astfel de obiect, tipărirea de rezultate de către program ar fi cam dificilă (totul ar trebui convertit în caractere). Pentru a rezolva această situaţie, este mult mai bine să se utilizeze un obiect instanţă a clasei *PrintStream* (în realitate *System.out* este un obiect de acest tip).

```
PrintStream file_complex_stream =
    new PrintStream(new FileOutputStream("fisierul_meu.txt"));
```

Un astfel de obiect are în principiu două metode: *print* şi *println*. Efectul lor e asemănător. Fiecare metodă converteşte valoarea unicului său parametru într-un şir de caractere care va fi convertit apoi mai departe într-o secvenţă de octeţi corespunzătoare. Aceşti octeţi sunt apoi scrişi în fluxul de ieşire conţinut de obiect. Singura deosebire este că a doua metodă trece la linie nouă după scriere. Este important de ştiut că cele două metode sunt supraîncărcate, ele putând să ia ca parametru o valoare de tip primitiv, un obiect *String* sau un orice alt obiect.

Nu uitaţi să închideţi un flux în momentul în care nu mai este nevoie de el. Acest lucru se realizează prin apelarea metodei *close()* a obiectului flux. Atenţie însă la închiderea fluxurilor reprezentate de obiectele *System.in* şi *System.out*.



Consultaţi pagile de manual ale claselor discutate în acest paragraf. Ele pot fi găsite la adresa web *http://java.sun.com/j2se/1.5.0/docs/api/java/io/package-summary.html*. Acolo veţi putea vedea că începând cu versiunea 1.5 a limbajului Java, clasa *PrintStream* defineşte şi metode destinate scrierii formatate (asemănătoare funcţiei *printf* din limbajul C).

4.3.3 Exemplu

În continuare vom vedea un exemplu de utilizare a claselor descrise în această secţiune. Programul următor citeşte un număr de întregi de la tastatură şi calculează suma lor. Numerele citite vor fi memorate într-un fişier. În final, suma lor este scrisă la rândul ei în acelaşi fişier dar va fi şi afişată pe ecran.

```
import java.io.*;

class ExempluIO {

    public static void main(String argv[]) {
        int n,i,suma,temporar;
        try {
            BufferedReader in_stream_char =
                new BufferedReader(new InputStreamReader(System.in));
            PrintStream out_stream = new PrintStream(
                new FileOutputStream("out.txt"));

            System.out.print("Dati numarul de intregi:");
            n = Integer.parseInt(in_stream_char.readLine());

            suma = 0;
            for(i = 1; i <= n; i++) {
                System.out.print("Dati numarul " + i + ":");
                temporar = Integer.parseInt(in_stream_char.readLine());
                suma+= temporar;
                out_stream.println(temporar);
            }

            out_stream.println(suma);
            System.out.println("Suma este:" + suma);
            out_stream.close();

        } catch(IOException e) {
            System.out.println("Eroare la operatiile de intrare-iesire!");
            System.exit(1);
        }
    }
}
```

O particularitate a acestui program este utilizarea blocurilor *try-catch*. Orice operaţie de intrare-ieşire poate produce erori. De exemplu, accesul la fişierul în care dorim să scriem este restricţionat. În astfel de cazuri se generează un "mesaj" de eroare de tip *IOException*. În principiu este obligatoriu ca aceste "mesaje" să fie "prinse" utilizând blocuri *try-catch*. Blocul *try* cuprinde codul asociat execuţiei normale a programului. Dacă apare o eroare *IOException* se trece în mod automat la execuţia instrucţiunilor din blocul *catch* asociat erorii *IOException*.

4.4 Tablouri

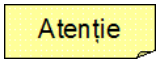
În Java tablourile sunt obiecte. În general, declararea unei referinţe la un tablou se face în felul următor.

```
tip_elemente[] nume_referinta_tablou;
```

În exemplul de mai sus doar am declarat o referinţă la un obiect tablou. Ca şi în cazul obiectelor obişnuite, tablourile trebuie create explicit folosind operatorul *new*. Mai mult, trebuie specificată şi dimensiunea tabloului care nu se mai poate schimba odată ce tabloul a fost creat. În exemplul de mai jos se declară şi se iniţializează o referinţă către un tablou care conţine zece valori de tip *int* şi o referinţă către un tablou care conţine zece referinţe la obiecte *Integer*.

```
//Declararea si crearea unui tablou de zece elemente intregi
int[] tablou_intregi = new int[10];

//Declararea si crearea unui tablou de zece referinte la obiecte Integer
Integer[] tablou_obiecte_intregi = new Integer[10];
```



La crearea celui de-al doilea tablou din exemplu, se alocă memorie pentru zece referinţe la obiecte *Integer*. Cu alte cuvinte NU se crează zece obiecte *Integer*. Sarcina creării lor revine programatorului.

Accesul la elementele unui tablou se realizează prin indexare. Cum numele tabloului este o referinţă la un obiect, indexarea poate fi privită ca apelarea unei metode speciale a obiectului referit. Mai mult, deoarece tabloul este un obiect, el ar putea avea şi câmpuri. Există un astfel de câmp special denumit *length* care conţine dimensiunea tabloului.

Numerotarea elementelor unui tablou începe de la 0.

```
//Initializarea tabloului de intregi
int i;
for(i = 0; i < tablou_intregi.length; i++) {
    tablou_intregi[i] = 0;
}

//Initializarea tabloului de referinte la obiecte Integer
int j;
for(j = 0; j < tablou_intregi.length; j++) {
    tablou_obiecte_intregi[j] = new Integer(0);
}
```

Atenţie

Încercarea de a accesa un tablou printr-o referinţă neiniţializată (care nu referă nici un tablou) constituie o eroare.

O metodă poate avea ca şi parametri referinţe la tablouri. În acelaşi timp o metodă poate să aibă ca valoare returnată o referinţă la un tablou. Un exemplu este prezentat mai jos: metoda primeşte o referinţă la un tablou de referinţ *Integer*, iniţializează toate elementele tabloului şi întoarce spre apelant referinţa primită prin parametru.

```
class Utilitare {

    public static Integer[] init(Integer[] tab) {
        int i;
        for(i = 0; i < tab.length; i++) {
            tab[i] = new Integer(0);
        }
        return tab;
    }
}
```

Discuţia de până acum a prezentat toate elementele necesare utilizării tablourilor în programe Java. Mai rămâne de discutat o singură problemă: cum anume putem lucra cu tablouri multi-dimensionale (de exemplu cu matrice). Răspunsul este foarte simplu. După cum am spus un tablou este un obiect. În particular putem avea un tablou de referinţe la un anumit tip de obiecte (în exemplele anterioare am avut un tablou de referinţe la obiecte Integer). Dar cum tablourile sunt obiecte, putem avea tablouri de tablouri (mai exact, tablouri de referinţe la tablouri). În exemplul următor arătăm modul de declarare şi creare a unei matrice.

```
//Declararea si crearea unei matrice de 10 X 5 elemente
int[][] matrice_intregi = new int[10][5];

//Instructiunea de mai sus e echivalenta cu urmatoarea
int i;
int[][] matrice_intregi = new int[10][];
for(i = 0; i < 10; i++) {
    matrice_intregi[i] = new int[5];
}
```

Este interesant de observat că în ultimul exemplu nimic nu ne-ar fi oprit să atribuim elementelor tabloului de tablouri referinţe spre tablouri de dimensiuni diferite. În exemplul următor vom crea o matrice triunghiulară.

```
//O matrice triunghiulara
int i;
int[][] matrice_speciala = new int[10][];
for(i = 0; i < 10; i++) {
    matrice_speciala[i] = new int[i + 1];
}

//Tiparirea elementelor ei
int a,b;
for(a = 0; a < matrice_speciala.length; a++) {
    for(b = 0; b < matrice_speciala[a].length; b++) {
        System.out.print(matrice_speciala[a][b] + " ");
    }
    System.out.println();
}
```

4.5 Exerciţii

1. Rulaţi şi studiaţi programul dat ca exemplu în Secţiunea 4.3.3.
2. Cum determinaţi dacă două obiecte *Boolean* înfăşoară aceeaşi valoare logică, fără a utiliza metoda *booleanValue()*? Verificaţi răspunsul printr-un program Java.
3. Scrieţi un program Java care citeşte de la tastatură o linie de text şi numele unui fişier. Programul trebuie să determine şi să afişeze pe ecran numărul de linii de text din fişierul indicat care sunt egale cu linia de text citită de la tastatură.
4. Să se scrie un program Java care citeşte de la tastatură două matrice de numere reale de dimensiune NxM, respectiv MxP, înmulţeşte cele două matrice şi scrie într-un fişier matricea rezultată. Toate matricele trebuie să conţină ca elemente obiecte *Double*.
5. Se dă un fişier "intervale.dat" care conţine perechi de întregi pozitivi (câte un întreg pe linie) reprezentând intervale numerice şi un număr oarecare de fişiere care conţin numere reale. Să se scrie un program Java care calculează pentru fiecare interval dat procentul de numere reale (din fişierele menţionate mai sus) conţinute.
   Programul trebuie să respecte următoarele cerinţe:
   - toate numerele reale citite din fişiere trebuie utilizate ca obiecte *Double* sau *Float* şi nu ca variabile de tip primitiv *double* sau *float*.
   - numele fişierelor ce conţin numerele reale se citesc de la tastatură unul câte unul.
   - numerele reale dintr-un fişier nu se prelucrează de mai multe ori; dacă utilizatorul furnizează acelaşi fişier de mai multe ori, programul atrage atenţia utilizatorului asupra erorii.

- după prelucrarea tuturor fişierelor cu numere reale, statistica trebuie scrisă într-un fişier al cărui nume este specificat ca argument al programului; dacă nu se specifică nici un argument, statistica va fi tipărită în fişierul standard de ieşire (ecran).
- fiecare interval va fi reprezentat printr-un obiect; clasa acestuia va conţine câmpuri private pentru limitele intervalului, pentru numărul de numere testate şi pentru numărul de numere testate conţinute de intervalul respectiv; clasa mai conţine: un constructor ce permite iniţializarea corespunzătoare a câmpurilor (cele care trebuie iniţializate), o metodă de testare ce preia ca parametru un obiect *Double*/*Float* şi care verifică apartenenţa parametrului la intervalul respectiv actualizând corespunzător câmpurile mai sus amintite şi o metodă care ia ca parametru un flux de ieşire în care scrie rezultatele obţinute (intervalul respectiv şi procentul obţinut).

Sfat Argumentele date unui program sunt disponibile prin parametrul metodei *main*. Acest parametru este un tabloul de şiruri de caractere.

Bibliografie

1. David Flanagan, *Java In A Nutshell. A Desktop Quick Reference*, Third Edition, O'Reilly, 1999.
2. Sun Microsystems Inc., *Online Java 1.5 Documentation, http://java.sun.com/j2se/1.5.0/docs/api/*, 2005.