

Dominic Mircea KRISTÁLY

Anca VULPE



LIMBAJUL DE PROGRAMARE JAVA



Îndrumar de laborator | Braşov, 2009

Cuprins

1. Crearea și rularea programelor Java din linie de comandă	1
1.1. Crearea codului sursă	1
1.2. Compilarea programului	2
1.3. Lansarea în execuție a programului	4
2. Crearea și rularea programelor Java cu ajutorul platformei Eclipse.....	5
2.1. Interfața de lucru a platformei Eclipse.....	6
2.2. Crearea unui proiect Java.....	7
2.3. Crearea pachetelor.....	9
2.4. Crearea claselor.....	9
2.5. Aplicația Hello World.....	11
3. Structura unui program Java	13
3.1. Definiția clasei	13
3.2. Definiția metodelor	13
3.3. Instrucțiuni	13
4. Tipuri de date primitive.....	15
4.1. Tipuri întregi.....	15
4.2. Tipuri reale	16
4.3. Tipul caracter.....	16
4.4. Tipul logic	17
4.5. Conversii între tipurile de date primitive	18
4.6. Variabile.....	18
4.7. Constante	19
4.8. Operatori	19
4.9. Aplicație cu variabile și operatori.....	20
5. Citirea de la tastatură – clasa <code>Scanner</code>	21
5.1. Introducerea datelor de la tastatură și afișarea lor	22

6. Structuri de control	23
6.1. Instrucțiuni de decizie / selecție	23
6.1.1. Instrucțiunea <code>if</code>	23
6.1.1.1. Clauza <code>else</code>	24
6.1.1.2. Clauza <code>else if</code>	24
6.1.1.3. Instrucțiuni <code>if</code> imbricate	25
6.1.1.4. Utilizarea instrucțiunii <code>if</code>	26
6.1.2. Instrucțiunea <code>switch</code>	27
6.1.2.1. Instrucțiuni <code>switch</code> imbricate	28
6.2. Instrucțiuni iterative	29
6.2.1. Instrucțiunea <code>for</code>	29
6.2.1.1. Cicluri <code>for</code> imbricate	30
6.2.1.2. Utilizarea instrucțiunii <code>for</code>	31
6.2.2. Instrucțiunea <code>while</code>	35
6.2.2.1. Utilizarea instrucțiunii <code>while</code>	36
6.2.3. Instrucțiunea <code>do . . while</code>	37
6.3. Instrucțiuni de salt	38
6.3.1. Instrucțiunea <code>break</code>	38
6.3.2. Instrucțiunea <code>continue</code>	39
6.3.3. Instrucțiunea <code>return</code>	39
6.4. Utilizarea instrucțiunilor <code>do . . while</code> , <code>switch</code> , <code>break</code> și <code>continue</code>	40
7. Tablouri	43
7.1. Tablouri unidimensionale – Vectori	43
7.1.1. Declararea variabilei tablou	43
7.1.2. Instanțierea	43
7.1.3. Inițializarea	44
7.2. Tablouri multidimensionale – Matrice	45
7.2.1. Crearea unui tablou multidimensional	45
7.2.2. Atribuirea valorilor către elementele matricei	45
7.2.3. Proprietăți <code>length</code>	46

7.3. Clasa <code>Arrays</code>	46
7.3.1. Metoda <code>equals()</code>	46
7.3.2. Metoda <code>fill()</code>	47
7.3.3. Metoda <code>sort()</code>	48
7.3.4. Metoda <code>binarySearch()</code>	48
7.3.5. Metoda <code>arraycopy()</code>	49
7.4. Aplicație cu vectori	50
7.5. Aplicație cu vectori: Loteria.....	52
7.6. Aplicație cu o matrice bidimensională	53
8. Șiruri de caractere	55
8.1. Clasa <code>String</code>	55
8.2. Clasa <code>StringBuffer</code>	60
9. Clase și programare orientată obiect	63
9.1. Termeni utilizați în programarea orientată obiect	63
9.2. Definiția unei clase	63
9.2.1. Modificatori de acces	64
9.2.2. Alți modificatori.....	64
9.2.2.1. Modificatorul <code>final</code>	64
9.2.2.2. Modificatorul <code>static</code>	65
9.2.2.3. Modificatorul <code>synchronized</code>	65
9.2.3. Proprietăți	65
9.2.4. Metode	65
9.2.5. Constructori.....	67
9.2.6. Declararea unei instanțe a unei clase	68
9.2.7. Accesul la membrii unei clase	68
9.2.8. Supraîncărcarea metodelor.....	69
9.2.9. Cuvântul cheie <code>this</code>	70
9.3. Crearea și utilizare claselor (1)	72
9.4. Crearea și utilizarea claselor (2)	74
9.5. Clase interne.....	75

10. Moștenirea	81
10.1. Accesul la membrii unei clase moștenite.....	82
10.2. Apelarea constructorilor	83
10.2.1. Folosirea cuvântului cheie <code>super</code>	84
10.2.1.1. Folosirea cuvântului cheie <code>super</code>	84
10.2.2. Moștenirea pe mai multe niveluri	85
10.2.2.1. Moștenirea pe mai multe niveluri	85
10.2.3. Supradefinirea metodelor folosind moștenirea	87
10.2.3.1. Supradefinirea metodelor folosind moștenirea	88
10.2.4. Cuvântul cheie <code>final</code> și moștenirea	90
11. Tratarea excepțiilor.....	91
11.1. Proceduri de tratare a excepțiilor	91
11.1.1. Blocuri <code>try</code> imbricate.....	97
11.2. Lucrul cu excepții neinterceptate	98
11.3. Metode care nu tratează excepțiile	99
11.4. Excepții verificate și neverificate	100
11.5. Tratarea excepțiilor folosind superclasa <code>Exception</code>	101
11.6. Exemple.....	102
11.6.1. Excepții de I/O.....	102
11.6.2. Excepții: Depășirea indexului unui vector.....	103
11.6.3. Excepții: Vector cu dimensiune negativă.....	104
11.6.4. Excepții: <code>NullPointerException</code>	105
11.6.5. Excepții: <code>ArrayIndexOutOfBoundsException</code>	106
12. Interfețe	107
13. Fluxuri de intrare / ieșiere (fișiere)	112
13.1. Fișiere și sisteme de fișiere	112
13.2. Clasa <code>FILE</code>	112
13.2.1. Afișarea listei de fișiere dintr-un director	115

13.3. Fluxuri.....	116
13.3.1. Scrierea într-un fișier.....	117
13.3.2. Citirea dintr-un fișier.....	118
13.3.3. Adăugarea datelor într-un fișier.....	119
13.3.4. Citirea și scriere unui obiect într-un fișier	120
14. Interfețe grafice	124
14.1. Pachetul <code>javax.swing</code>	125
14.2. O aplicație cu interfață grafică simplă.....	125
14.3. Crearea unei casete de dialog	127
14.4. Concatenarea a două șiruri de caractere	128

LABORATORUL 1

1. Crearea și rularea programelor Java din linie de comandă

Pașii ce trebuie urmați pentru crearea unui program Java sunt prezentați schematic în figura de mai jos:

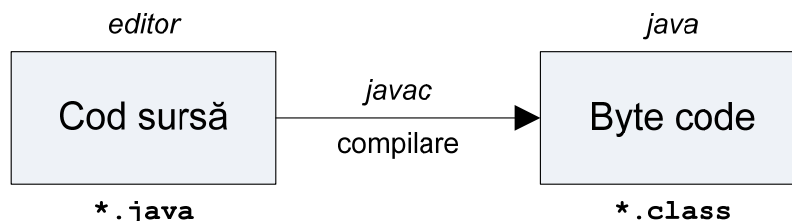


Fig. 1. Etapele necesare creării unui program Java

1.1. Crearea codului sursă

Codul sursă este scris în limbajul Java și rezidă într-unul sau mai multe fișiere text având extensia “.java”. Pentru scrierea programului se poate utiliza orice editor de texte. Dacă se lucrează sub sistemul de operare *Microsoft Windows*, se poate utiliza, de exemplu, aplicația *Notepad*. Dacă se lucrează sub Unix/Linux programul *vi* poate fi folosit pentru scrierea codului sursă.

Figura 2 prezintă codul sursă al unui program Java care afișează pe ecran textul „Hello World din Java!”, așa cum arată el în aplicația *Notepad*.

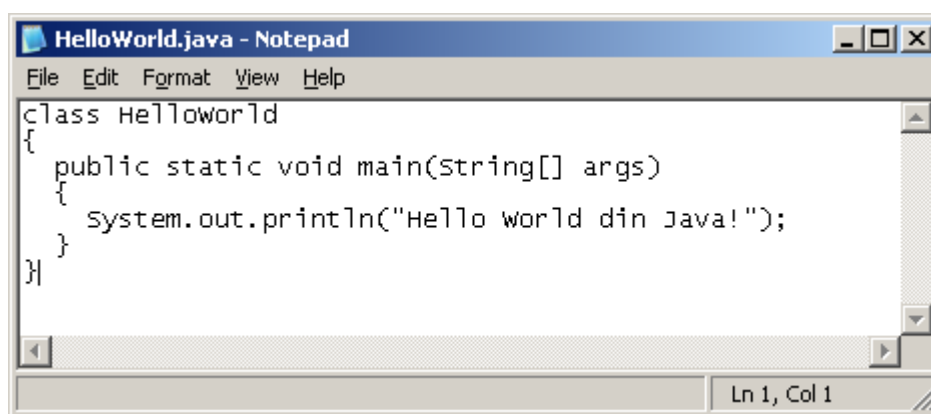


Fig. 2. Editarea programelor Java cu ajutorul aplicației *Notepad* din Windows



Numele fișierului care conține codul sursă al programului trebuie să aibă numele identic cu numele clasei ce conține metoda `main()`.

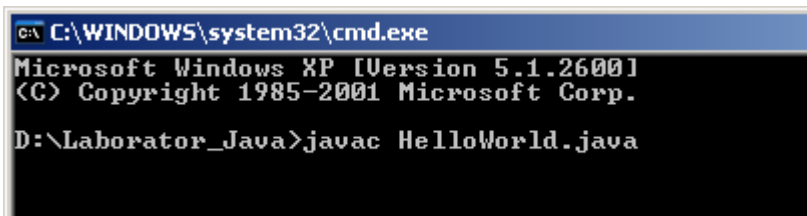
1.2. Compilarea programului

Transformarea codului sursă în codul de octeți (*byte code*) înțeles de JVM (*Java Virtual Machine*) se realizează prin compilarea programului. Pe sistemele Windows acest lucru este realizat de executabilul `javac.exe`, ce poate fi apelat dintr-o fereastră sistem.

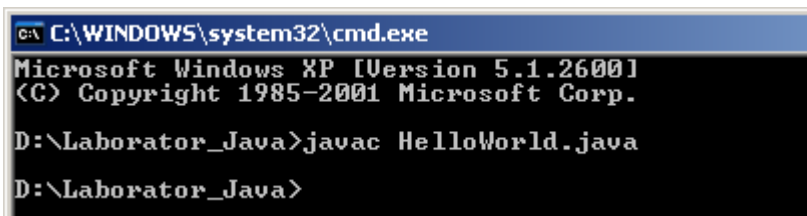
Compilerul Java face parte din pachetul JDK (*Java Development Kit*) care trebuie să fie instalat pe calculatorul pe care se dezvoltă programe Java. Acest pachet poate fi descărcat, gratuit, de pe site-ul companiei *Sun microsystems*. Programele prezentate în continuare au fost scrise și testate folosindu-se versiunea 6 a pachetului JDK. Acest pachet poate fi descărcat de la adresa: <http://java.sun.com/javase/downloads/index.jsp>.

Pentru a compila programul „HelloWorld”, prezentat în figura 2, se deschide o fereastră sistem (*Command Prompt*), în care se scrie următoarea comandă, urmată de tasta *CR* (*Enter*):

```
javac HelloWorld.java
```



Dacă programul a fost compilat cu succes, pe ecran apare din nou „*command prompt*”-ul:



În directorul de lucru apare un nou fișier, numit `HelloWorld.class`, ce conține codul de octeți al programului (așa cum arată figura 3).

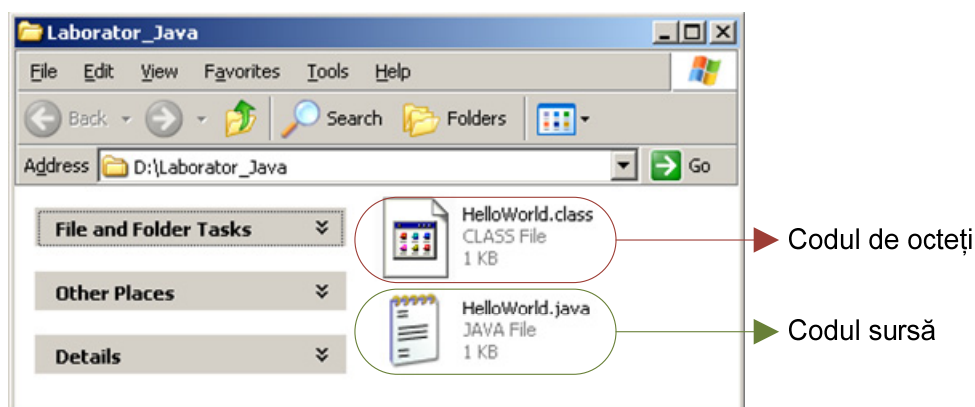


Fig. 3. Codul sursă și fișierul obținut în urma compilării

Dacă programul nu a fost compilat cu succes, în fereastra sistem pot apare diferite mesaje de eroare, precum:

- **'javac' is not recognized as an internal or external command, operable program or batch file.**

Cauze posibile:

1. nu este instalat pachetul JDK.
 - Soluție: instalarea pachetului JDK;
2. este instalat pachetul JDK, dar calea directorului `bin` nu a fost inclus în variabila de sistem `PATH`.
 - Soluție: adăugarea la variabila de sistem `PATH` a directorului `bin`. Acest lucru se poate realiza, în Windows, din fereastra *System properties*, accesibilă prin clic dreapta pe iconița *My Computer* și selecția opțiunii *Properties*, sau din Control Panel (accesibil din *Start Menu* → *Settings*)
În secțiunea *Advanced* se efectuează clic pe butonul *Environment Variables*. În fereastra care se deschide, din lista *System variables*, se selectează variabila `PATH`, după care se apasă butonul *Edit*.
La sfârșitul valorii variabilei se adaugă caracterul `' ; '`, dacă nu există, după care se trece calea completă către directorul `bin` al pachetului JDK.

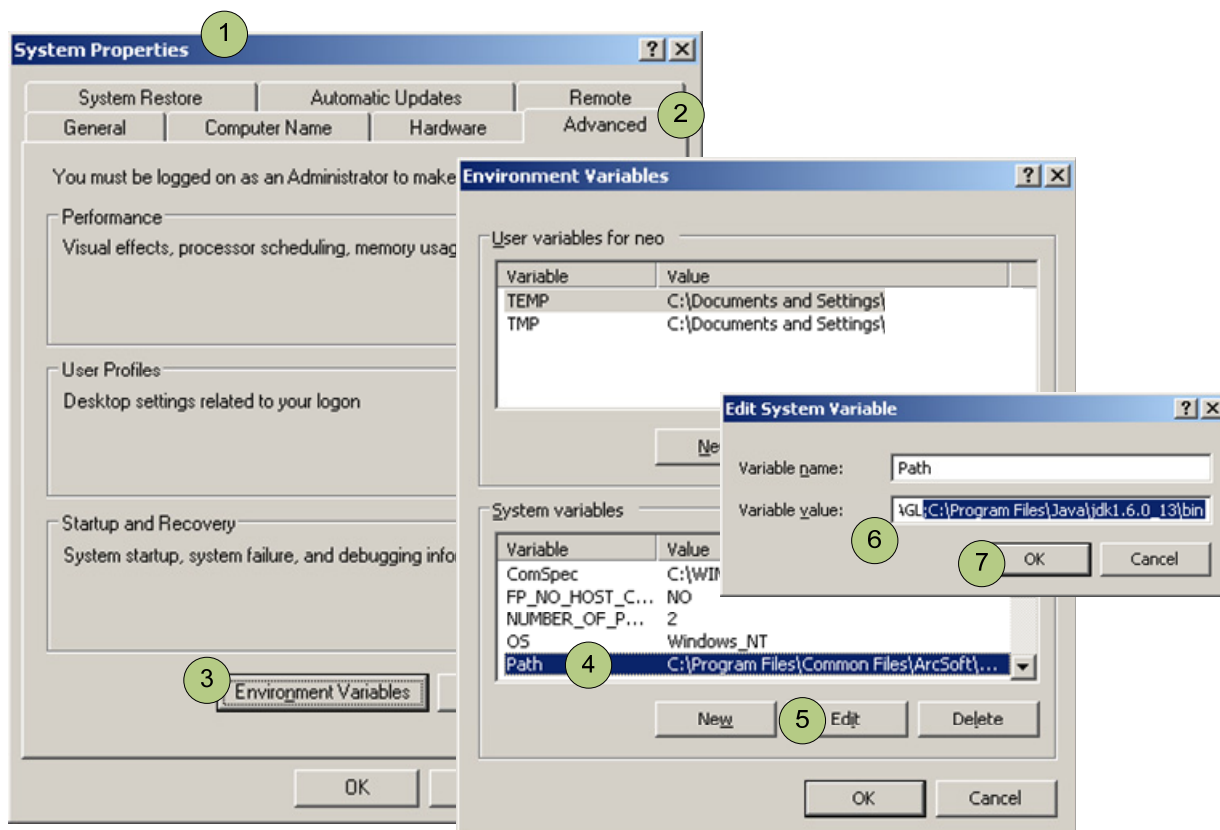


Fig. 4. Includerea directorului `bin` al pachetului JDK în variabila de sistem `PATH`

2. "...should be declared in a file named...."

Cauze posibile:

1. numele fișierului nu este salvat cu același nume cu cel al clasei conținute.
 - Soluție: redenumirea fișierului sau a clasei, astfel încât cele două nume să coincidă.
- **javac:invalid flag:** dacă apare acest mesaj, urmat de o listă lungă de cuvinte care încep cu '-' cauzele posibile sunt:
 1. compilatorul nu poate găsi fișierul sursă Java. Se va afișa directorul curent.
 2. scrierea greșită a numelui fișierului la compilare.
 3. omiterea extensiei fișierului (.java).
- **Hello java:6:';expected**

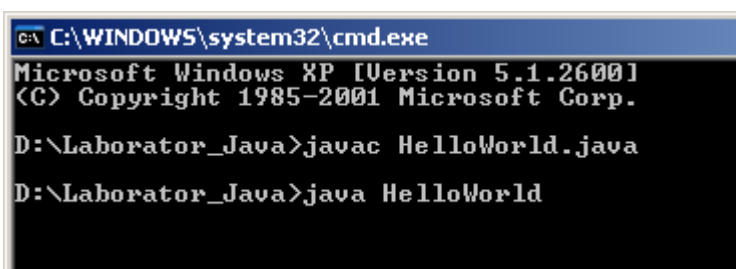
Când o eroare conține cuvântul *expected* urmat de un fragment de program înseamnă că fragmentul respectiv conține o eroare de sintaxă. În acest exemplu, linia 6 este cea care a generat eroarea.

1.3. Lansarea în execuție a programului

Pentru a putea lansa în execuție un program Java, calculatorul gazdă trebuie să aibă instalată mașina virtuală Java (JVM). Aceasta este reprezentată de pachetul JRE (*Java Runtime Environment*), care este inclus și în pachetul JDK.

Lansarea în execuție a programului „HelloWorld” se realizează prin comanda:

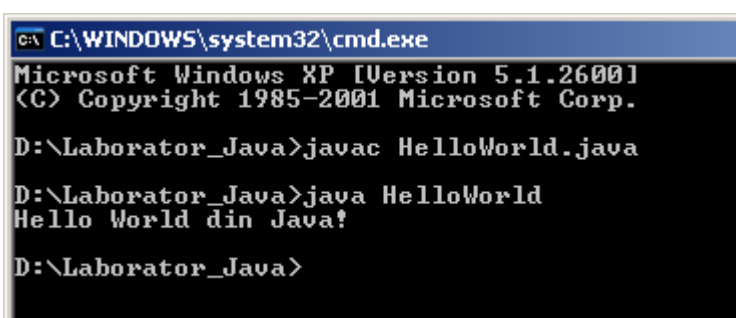
```
java HelloWorld
```



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\Laborator_Java>javac HelloWorld.java
D:\Laborator_Java>java HelloWorld
```

În fereastra *Command Prompt* apare mesajul „Hello World din Java!”:



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\Laborator_Java>javac HelloWorld.java
D:\Laborator_Java>java HelloWorld
Hello World din Java!

D:\Laborator_Java>
```



La lansarea în execuție a unui program Java din linie de comandă este utilizat fișierul cu extensia „.class”, dar această extensie nu se menționează, ci numai numele fișierului (care trebuie să coincidă exact cu numele clasei ce conține metoda `main()`).

2. Crearea și rularea programelor Java cu ajutorul platformei Eclipse

Eclipse este o platformă multi-scop pentru dezvoltarea de software, scrisă, în mare parte, în limbajul Java, astfel putând rula pe orice sistem de operare actual. Oferă un mediu integrat de dezvoltare (*IDE – Integrated Development Environment*) pentru diverse limbaje de programare (*Java, C/C++, PHP, Python, Perl, Cobol*). Baza codului sursă provine din platforma *VisualAge* dezvoltată de *IBM*, astfel se explică suportul primit din partea acestei companii.

Platforma *Eclipse* se încadrează în categoria programelor gratuite și *open source*. Cea mai recentă versiune, *Eclipse Ganymede*, poate fi descărcată de pe site-ul oficial www.eclipse.org, de la adresa: <http://www.eclipse.org/downloads/>, link-ul *Eclipse IDE for Java Developers*.

Printre facilitățile platformei *Eclipse* merită a fi menționate:

- crearea și gestiunea de proiecte;
- *debuging*;
- completarea automată a codului (*code completion*);
- automatizări pentru operații des utilizate (redenumire, creare de set-ere și get-ere, completarea automată a secțiunii de import).



Toate programele Java prezentate în continuare sunt dezvoltate folosind platforma *Eclipse Ganymede* și *JDK 6* și rulează pe *JRE 6*.

Versiunile platformei *Eclipse*, începând cu versiunea 3.2, sunt denumite după sateliții naturali și artificiali ai planetei *Jupiter* (*Callisto, Europa, Ganymede*). Următoarea versiune, programată a fi lansată oficial în 26 iunie 2009, se va numi *Eclipse Galileo*.

2.1. Interfața de lucru a platformei Eclipse

La pornirea mediului *Eclipse* este cerută calea unui director care va fi spațiul de lucru al platformei. Aici se vor salva proiectele și vor fi stocate datele necesare rulării proiectelor în lucru.

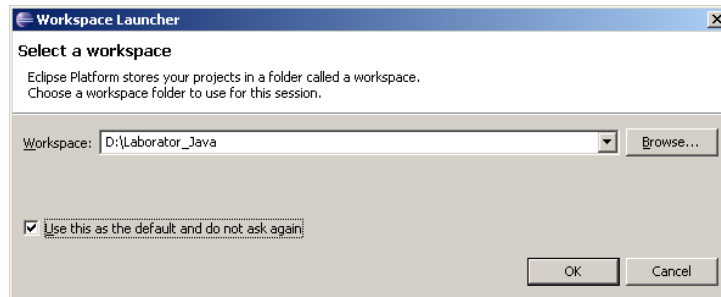


Fig. 5. Selecția spațiului de lucru

Dacă se utilizează același spațiu de lucru pentru toate proiectele, se poate bifa căsuța de validare *Use this as the default and do not ask again*, pentru a elimina această întrebare la fiecare pornire a platformei.

Ferestrele vizibile din interfața *Eclipse* sunt grupate logic în *perspective*.

O perspectivă indică numărul de ferestre vizibile, poziția lor și opțiunile vizibile în aceste ferestre, în funcție de limbajul utilizat în proiectul deschis. În capturile de ecran prezentate în acest îndrumar se va folosi perspectiva Java. Perspectiva curentă se poate schimba de la butoanele dedicate situate în partea dreaptă-sus a *workbench*-ului *Eclipse*.

Figura 6 prezintă ferestrele vizibile din perspectiva Java și semnificația lor.

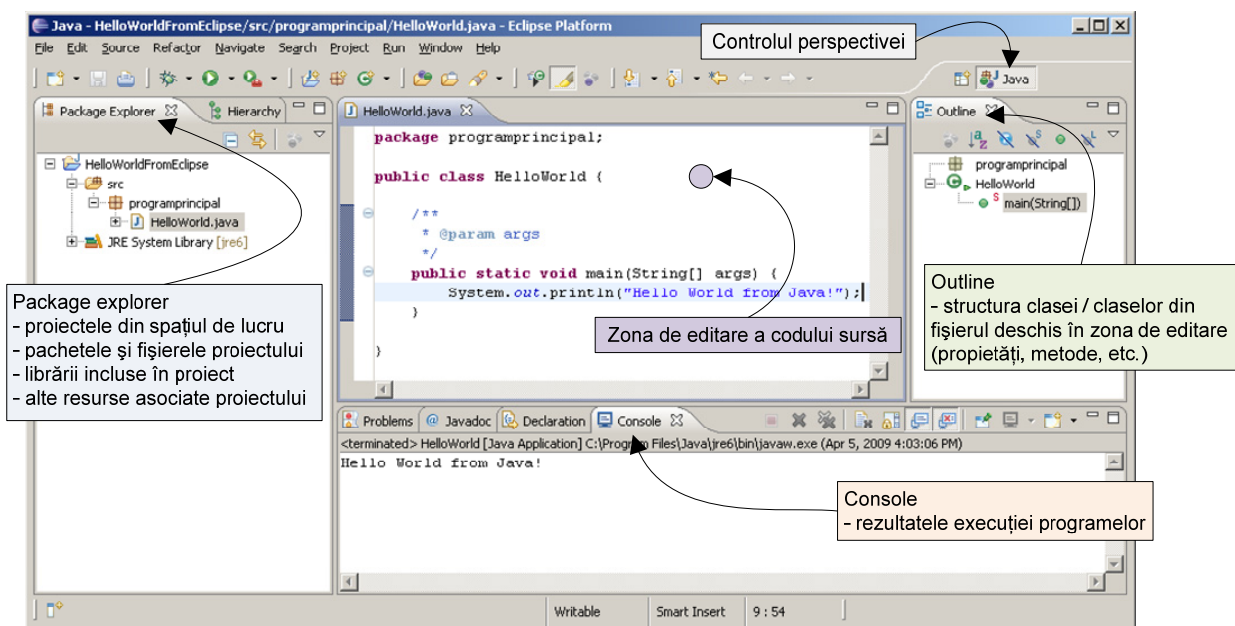


Fig. 6. Componenta perspectivei Java

2.2. Crearea unui proiect Java

Din meniul *File*→*New* se alege opțiunea *Java Project*.

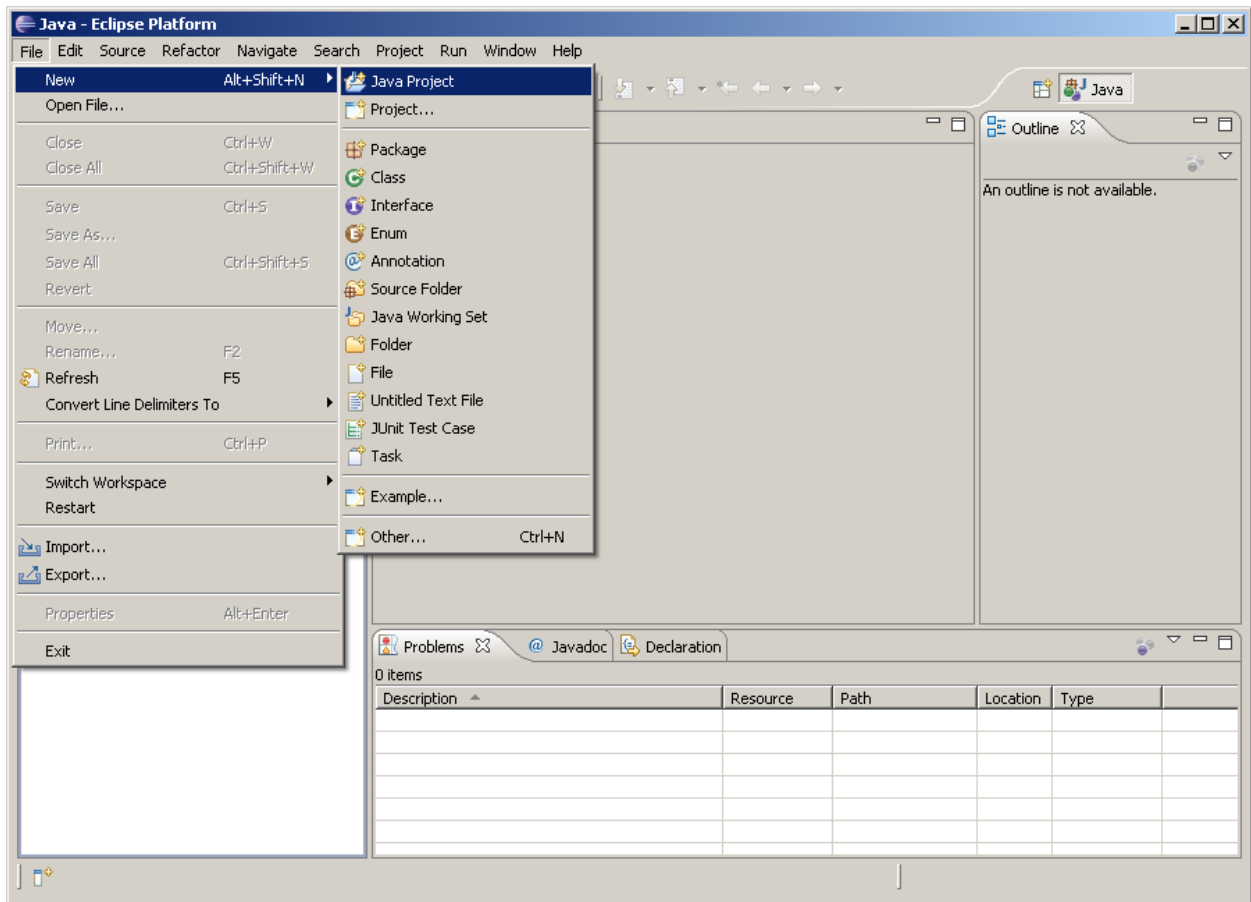


Fig. 7.a. Crearea unui proiect Java în Eclipse

În fereastra care se deschide trebuie specificat numele proiectului. Este indicat să fie selectată opțiunea *Create separate folders for sources and class files* din secțiunea *Project layout*, pentru a nu amesteca fișierele ce conțin codul sursă cu cele ce conțin codul de octeți, executabil. Pentru un proiect complet nou este bine să se aleagă opțiunea *Create new project in workspace* a secțiunii *Contents*. Apăsarea butonului *Next* va determina accesarea la mai multe opțiuni, cum ar fi importul unor librării suplimentare, modificarea directoarelor unde vor fi salvate sursele și codul de octeți al proiectului, etc.

Crearea și deschiderea în *workbench* a noului proiect se realizează prin apăsarea butonului *Finish*.

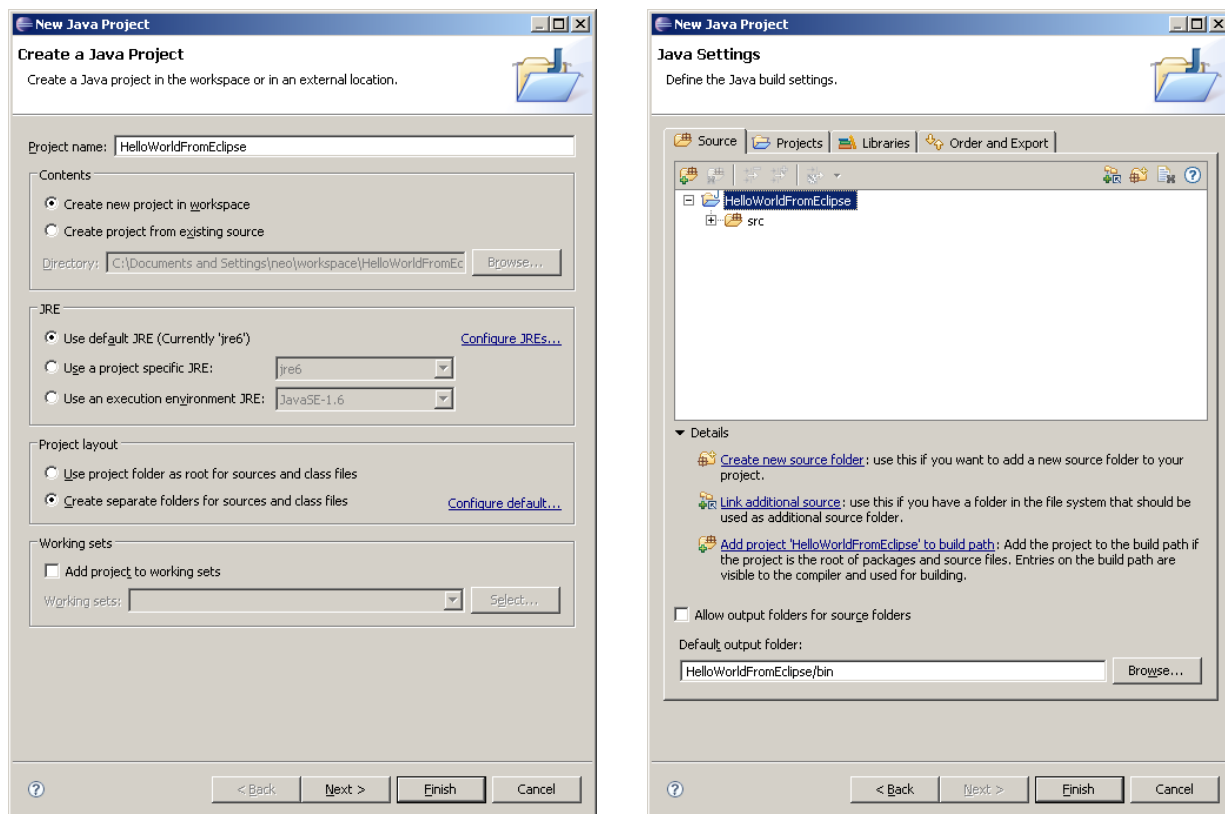


Fig. 7.b. Crearea unui proiect Java în Eclipse

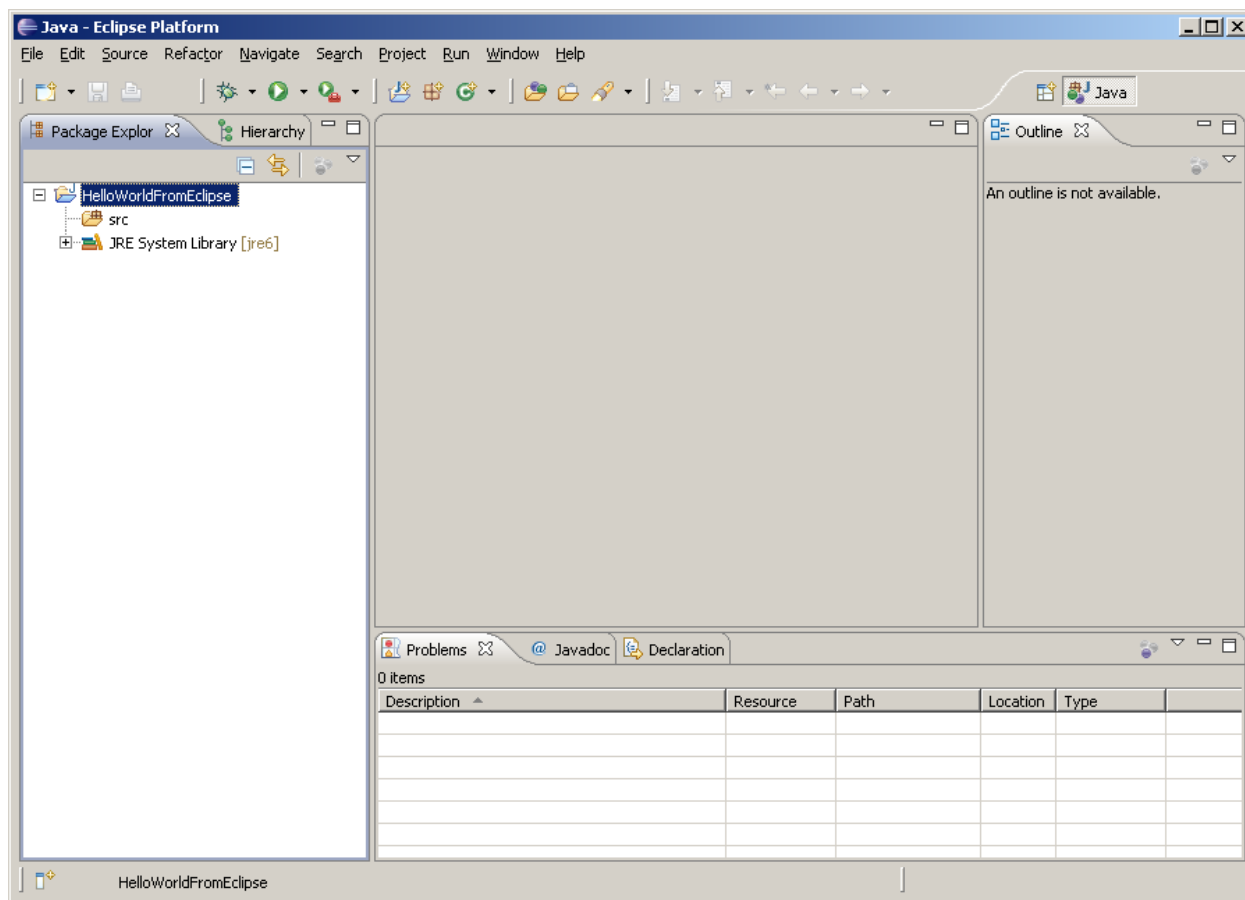


Fig. 8. Proiect nou deschis în workbench-ul platformei Eclipse

2.3. Crearea pachetelor

Clasele unui proiect Java sunt grupate, de regulă, în *pachete*. Criteriile de grupare țin de funcțiile pe care le îndeplinesc acele clase în proiect (lucrul cu fișiere, accesul la baze de date, comunicația prin rețea, etc.). De asemenea, pachetele asigură și controlul numelor și al vizibilității. Fizic, un pachet este un director al proiectului.

Accesul la clasele dintr-un pachet se face prin utilizarea instrucțiunii `import` la începutul codului sursă. Instrucțiunea `import` trebuie să conțină numele pachetelor și ale claselor care vor fi folosite în codul sursă.

Crearea unui pachet în *Eclipse* se realizează prin clic dreapta pe numele proiectului, iar din meniul care apare se alege *New→Package*. În fereastra ce se deschide se specifică numele noului pachetului, după care se apasă butonul *Finish*.

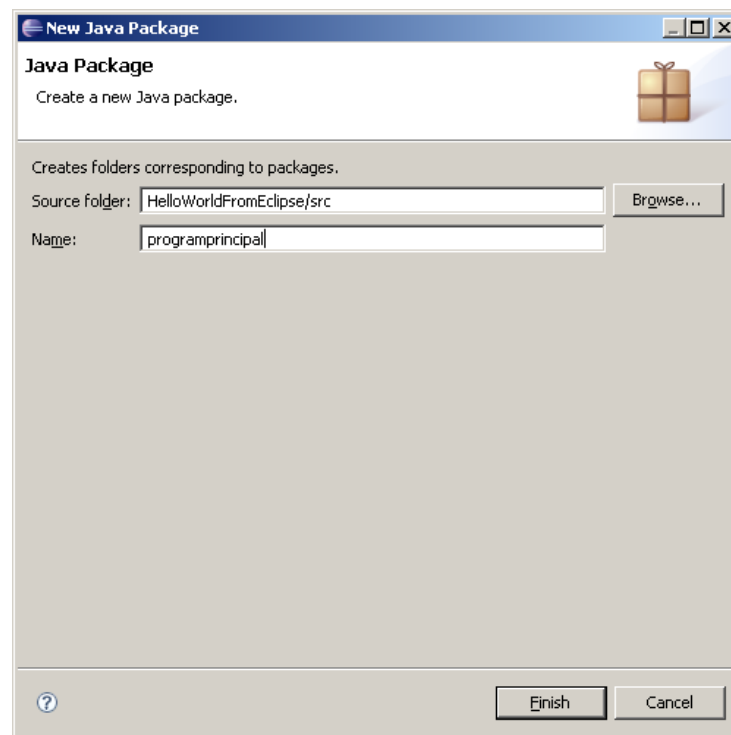


Fig. 9. Creare pachetului *programprincipal*

Un pachet poate conține alte pachete, care, la rândul lor, pot conține alte pachete. În cazul acesta se efectuează clic pe numele pachetului în care se face adăugarea, după care se parcurge aceleași etape prezentate mai sus.

2.4. Crearea claselor

Clasele aparțin pachetelor, prin urmare adăugarea se va face prin clic dreapta pe numele pachetului; se selectează *New→Class*. În fereastra care se deschide se completează numele clasei și se selectează opțiunile pentru aceasta.

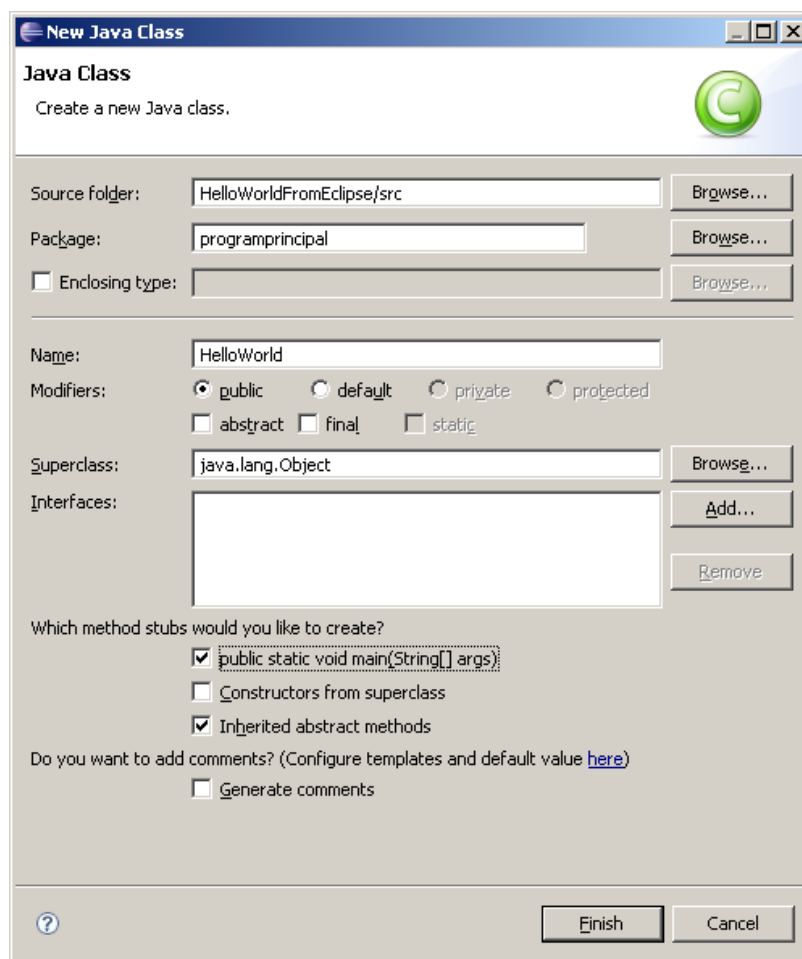


Fig. 10. Crearea unei clase

Dacă se dorește ca clasa creată să conțină metoda `main()`, atunci se bifează opțiunea *public static void main(String[] args)* din secțiunea *Which method stubs would you like to create?*. Se recomandă ca modificatorul de acces al claselor să fie cel `public` (secțiunea *Modifiers*).



Se recomandă respectarea următoarei convenții de notare: numele claselor încep întotdeauna cu literă mare. Dacă numele conține mai mulți atomi lexicali (cuvinte), fiecare dintre ei încep cu literă mare. Mai multe detalii despre convenția de notare Java a identificatorilor și beneficiile aduse de respectarea acestora se găsesc la adresa: [http://en.wikipedia.org/wiki/Naming_conventions_\(programming\)#Java_language](http://en.wikipedia.org/wiki/Naming_conventions_(programming)#Java_language)

După apăsarea butonului *Finish*, este deschis automat fișierul ce conține declarația noii clase, așa cum se vede în figura 11.

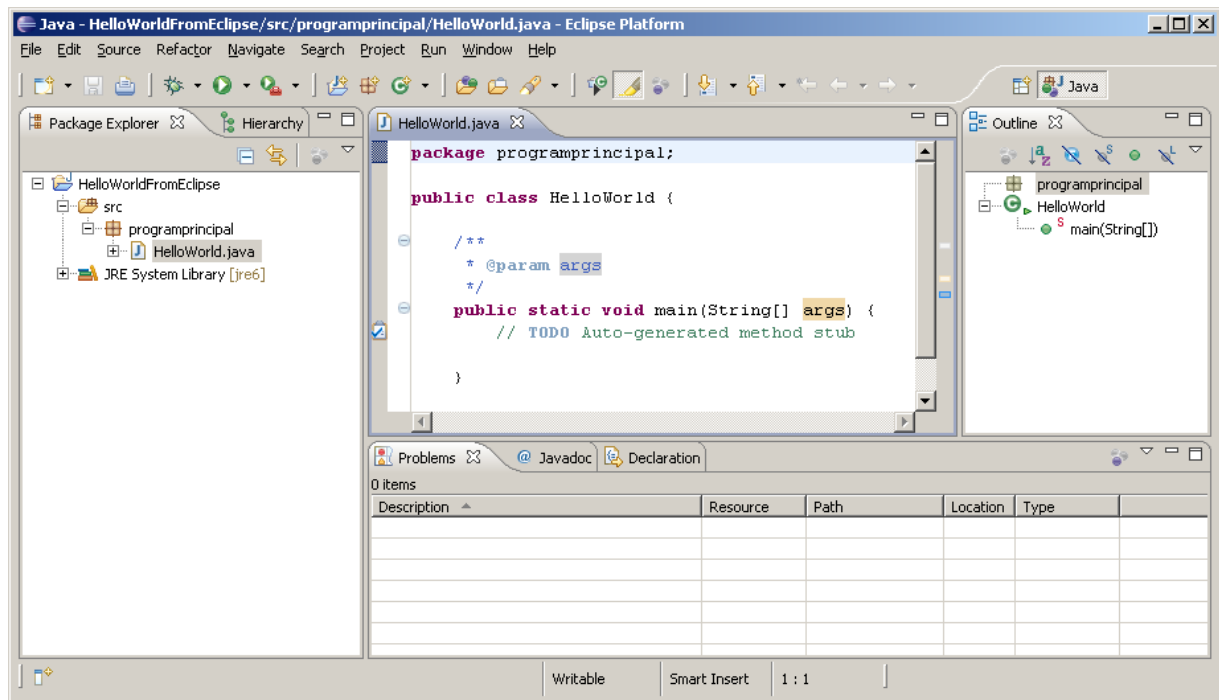


Fig. 11. Clasa generată automat de *Eclipse*

2.5. Aplicația Hello World

Pentru afișarea mesajului *“Hello World din Java!”* se folosește instrucțiunea:

```
System.out.println("Hello World din Java!");
```

Această instrucțiune trebuie plasată în interiorul metodei `main()` (care este echivalentul programului principal: execuția programului începe cu această metodă).

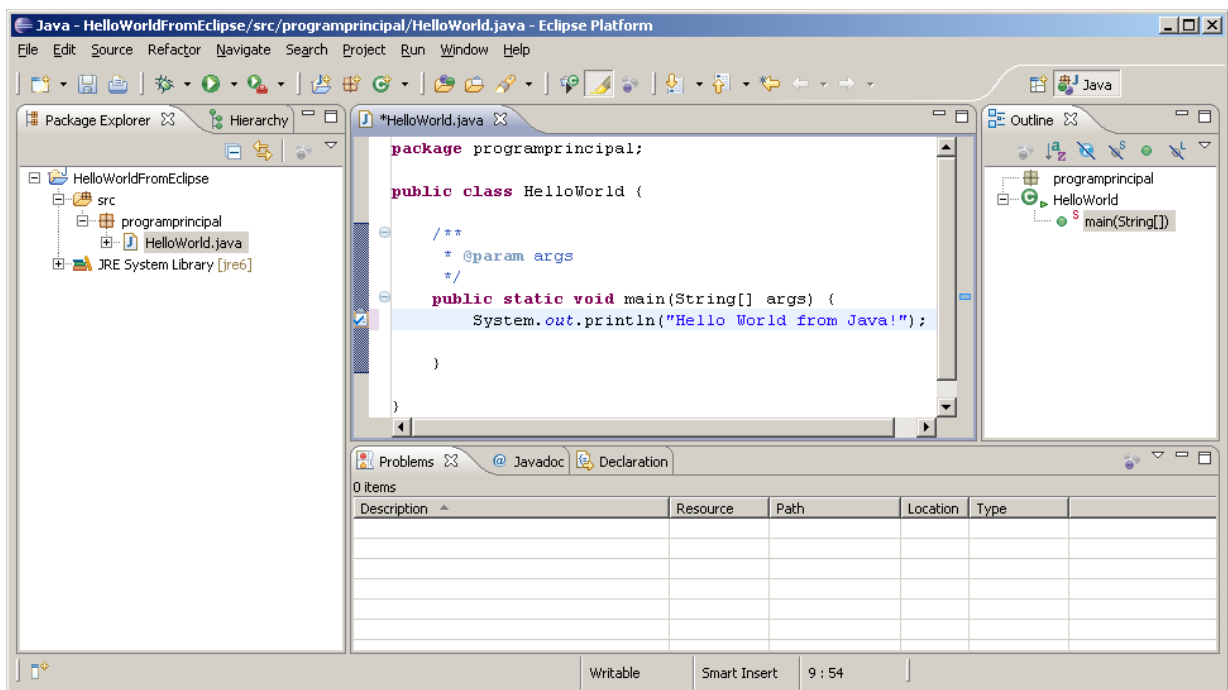



Fig. 12. Aplicația Hello World



Deoarece instrucțiunea `System.out.println()` este foarte uzitată, editorul platformei Eclipse oferă o scurtătură pentru scrierea ei. Tastarea textului `syso`, urmat de combinația de taste `CTRL+Space` determină introducerea instrucțiunii `System.out.println()`.

Lansarea în execuție a programului se poate face din meniul *Run*→*Run*, prin combinația de taste `CTRL+F11` sau prin apăsarea butonului  din bara de instrumente.

Înainte de lansarea în execuție a unui program este recomandabil să salvați fișierele pe care le-ați modificat. Oricum, mediul *Eclipse* vă va cere permisiunea de a salva aceste fișiere înainte de lansarea în execuție.

Rezultatul rulării programului este afișat în fereastra *Console*, aflată în parte de jos a workbench-ului.

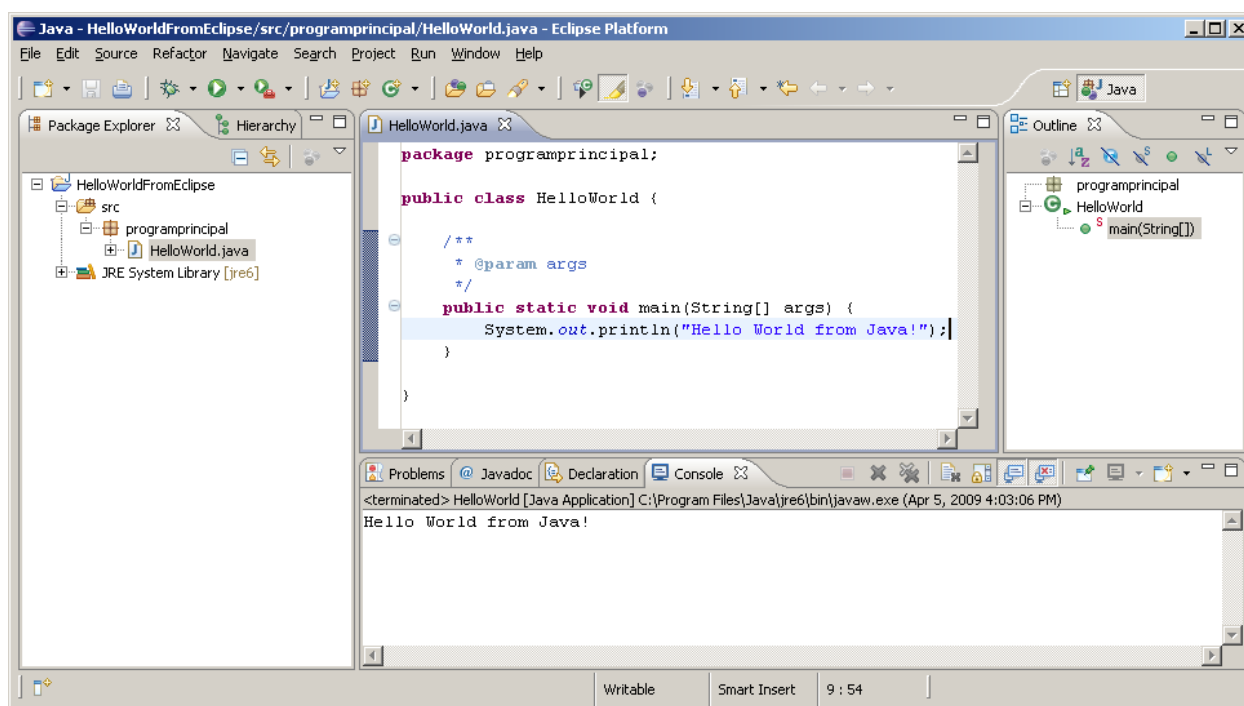
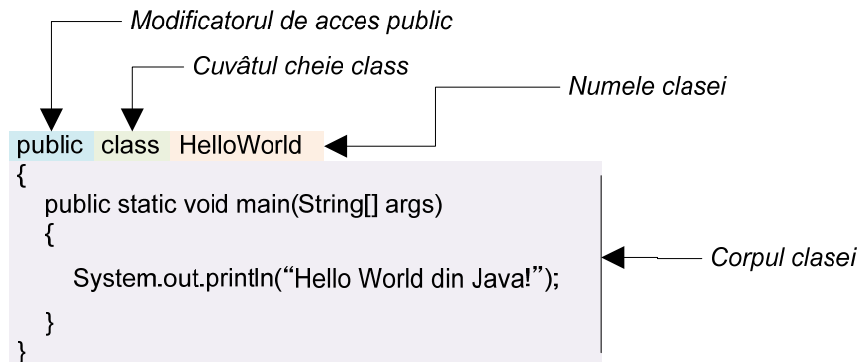


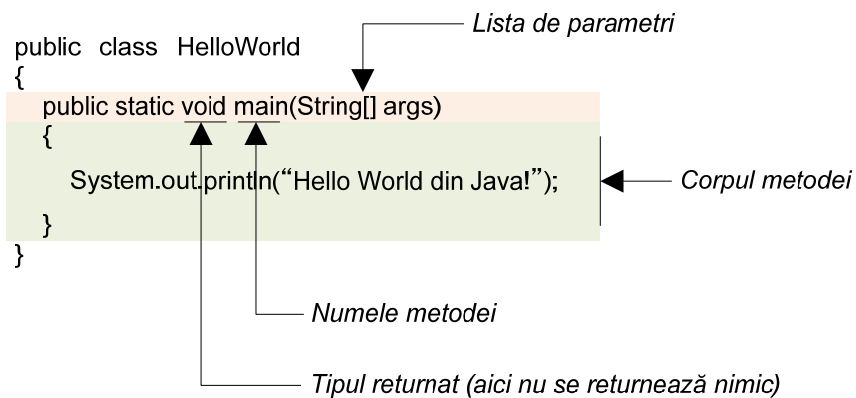
Fig. 13. Rezultatele rulării sunt afișate în fereastra *Console*

3. Structura unui program Java

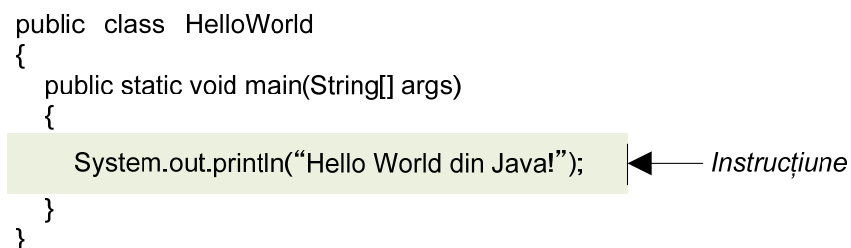
3.1. Definiția clasei



3.2. Definiția metodelor



3.3. Instrucțiuni



LABORATORUL 2

4. Tipuri de date primitive

În limbajul Java există două categorii de tipuri de date:

- tipuri primitive: tipurile numerice, tipul caracter și tipul logic;
- tipul referință: reține o referință (adresă) către o instanță a unei clase (obiect).

Limbajul Java definește opt tipuri de date primitive. Acestea se pot clasifica în:

- tipuri întregi: `byte`, `short`, `int` și `long`;
- tipuri reale: `float` și `double`;
- tipul caracter: `char`;
- tipul logic: `boolean`.

4.1. Tipuri întregi

Aceste tipuri sunt reprezentate în memorie ca numere cu semn (reprezentare în complement față de 2). Limbajul Java nu oferă tipuri întregi fără semn.

Tipul	Octeți alocați	Acoperirea
byte	1	-128 până la 127
short	2	-32768 până la 32767
int	4	-2147483648 până la 2147483647
long	8	-9223372036854755808L până la 922337203685475807L

Tipul **byte** este folosit, de obicei, pentru efectuarea unor operații la nivel de bit sau la prelucrarea la nivel scăzut (*low-level*) a fluxurilor (*stream* – fișiere, socket-uri, etc.).

Tipul **short** este deseori folosit în locul tipului `int`, mai ales din motive de optimizare a utilizării memoriei.

Tipul **int** este cel mai des folosit, având avantajul că are o acoperire suficient de mare, iar numărul de octeți folosiți pentru memorare este de două ori mai mic decât tipul următor (`long`).

Tipul **long** este mai rar folosit, înlocuind tipul `int` doar în cazul depășirii acoperirii. Literalii întregi de tip `long` vor avea sufixul `L`; de exemplu: `long l = 23568912457801112L`.

În mod implicit, literalii întregi sunt de tip `int`. Java suportă utilizarea literalilor întregi reprezentați și în bazele de numerație hexazecimal și octal.

Utilizarea literalilor în bazele de numerație hexazecimal și octal

```
// numărul 10 scris în hexazecimal

int example_hexa = 0xA;

// numărul 8 scris în sistemul octal (folosirea acestei baze de numerație
// nu este recomandată dată fiind forma confuză de declarare)

int example_octa = 010;
```

4.2. Tipuri reale

Numerele reale sunt reprezentate în limbajul Java ca numere cu semn (reprezentare în complement față de 2, virgulă mobilă simplă sau dublă precizie). Limbajul Java nu oferă tipuri reale fără semn.

Există două tipuri de numere reale reprezentate în virgulă mobilă: `float` (simplă precizie) și `double` (dublă precizie). Numerele iraționale (cum ar fi π sau $\sqrt{2}$) sunt trunchiate, putându-se lucra doar cu o aproximare a lor.

Este recomandată folosirea tipului `float` numai când viteza la care trebuie să ruleze aplicația trebuie să fie foarte mare (sau memoria folosită de aceasta trebuie să fie foarte mică).

În mod implicit literalii numere reale sunt reprezentați ca valori `double`. Literalii de tip `float` poartă sufixul „F”. De exemplu:

```
// declarare de float
float example_float = 5.505F;
```

Caracteristicile celor două tipuri este prezentat în tabelul următor:

Tipul	Octeți alocați	Acoperirea
<code>float</code>	4	Aproximativ: $\pm 3.40282347\text{E}+38\text{F}$ (6-7 zecimale)
<code>double</code>	8	Aproximativ: $\pm 1.79769313486231570\text{E}+308$ (13-14 zecimale)

4.3. Tipul caracter

Utilizarea caracterelor în Java se face cu ajutorul tipului `char`.

Tipul caracter utilizează pentru stocare doi octeți ce rețin codul *Unicode* al caracterului.

Avantajul folosirii standardului *Unicode* este acela că acoperirea este de 65535 de caractere (valorile limită fiind `'\u0000'` și `'\uFFFF'`), față de codul *ASCII* (*American Standard Codes for Information Interchange*) care are doar 128 de caractere sau de *ASCII extins* cu 256 de caractere. Deși se poate utiliza orice caracter, afișarea acestuia depinde de sistemul de operare (caracterele *Unicode* necunoscute sistemului de operare nu vor putea fi afișate).

Literalii de tip caracter sunt scriși între apostrofuri și pot conține secvențe *escape* (secvențe de caractere care încep cu `'\'`).

Declararea unei variabile de tip caracter cu inițializare:

```
// variabila ch va contine caracterul avand codul 1111
// (codul este un numar hexazecimal)

char ch = '\u1111';
```

Următorul tabel prezintă cele mai folosite secvențe escape:

Secvența escape	Valoarea Unicode	Denumire
<code>\b</code>	<code>\u0008</code>	Backspace
<code>\t</code>	<code>\u0009</code>	tab (TAB)
<code>\n</code>	<code>\u000A</code>	Linefeed (LF)
<code>\r</code>	<code>\u000D</code>	carriage return (CR)
<code>\"</code>	<code>\u0022</code>	ghilimele
<code>\'</code>	<code>\u0027</code>	apostrof
<code>\\</code>	<code>\u005C</code>	backslash

4.4. Tipul logic

Tipul logic permite memorarea doar a două valori: *adevărat* (`true`) și *fals* (`false`).

Desemnarea acestui tip, în limbajul Java, se face cu ajutorul cuvântului cheie `boolean`.

Tipul `boolean` este folosit, în principiu, pentru evaluarea condițiilor logice. Introducerea sa a eliminat neclaritățile din limbajele C și C++, unde evaluarea condițiilor se făcea folosind întregi.



Nu se poate face conversie între tipurile `boolean` și cele întregi.

4.5. Conversii între tipurile de date primitive

Tabelul următor sintetizează conversiile permise între tipurile de date primitive:

	byte	short	int	long	char	float	double
byte	=	*	*	*	!	≈	*
short	!	=	*	*	!	≈	*
int	!	!	=	*	!	≈	*
long	!	!	!	=	!	≈	≈
char	!	!	*	*	=	≈	*
float	!	!	!	!	!	=	*
double	!	!	!	!	!	!	=

= - același tip;

! - conversia nu se poate face;

*

≈ - conversia se poate face, dar cu pierdere de precizie.

4.6. Variabile

Ca în orice limbaj de programare, tipurile de date sunt utilizate, în principal, pentru descrierea *variabilelor* și stabilesc felul datelor pe care le pot memora, cantitatea de memorie necesară și valorile ce le pot fi asociate.

Mai concret, o variabilă este un spațiu de memorie destinat stocării unei valori într-un format corespunzător tipului declarat și careia i se atașează o etichetă – numele variabilei. Variabilele își pot schimba valoarea oricând pe parcursul programului.

Pentru *declararea* variabilelor, limbajul Java utilizează sintaxa:

Sintaxă utilizată: `id_tip id_var1[, id_var2 [, ...]];`

De exemplu, se poate scrie:

```
int variabla;
boolean bool;
```

Se mai pot face și declarații de forma:

```
int v1, v2;
```

dar această modalitate nu este indicată.

Deseori este recomandat ca variabila declarată să conțină și o valoare inițială:

```
int v1 = 10;
```

Astfel, se poate spune, pe scurt, că *definirea* unei variabile reprezintă *declararea* ei, la care se adaugă operația de *inițializare*.

Declararea variabilelor se poate face oriunde pe parcursul programului. De asemenea, variabilele se pot redeclara, cu restricția ca redeclararea să nu se facă în același bloc cu prima declarație (deci să nu se afle în același domeniu de declarație – *scope*).

Referitor la numele variabilelor se recomandă denumirea lor în concordanță cu semnificația datelor pe care le reprezintă. Această măsură conferă o claritate sporită sursei și minimizează timpul necesar efectuării unor corecturi sau modificări.

De asemenea, numele unei variabile nu poate începe cu o cifră; ele pot începe cu orice literă sau caracterul *underscore* ('_').

4.7. Constante

Definirea constantelor în Java se face prefixând definirea unei variabile cu **final**. Folosirea constantelor este indicată când programatorul vrea ca valoarea unei variabile să nu poată fi schimbată sau când se dorește stabilirea unor aliasuri. Imaginați-vă că într-un program este necesară folosirea de mai multe ori a constantei matematice π . Acest lucru se poate face elegant făcând o declarație de forma (exemplul este pur didactic; în practică, clasa `java.lang.Math` poate oferi valoarea acestei constante):

```
final double PI = 3.14159;
```

4.8. Operatori

În tabelul următor sunt prezentați toți operatorii, în ordinea precedenței lor.

Operatorul	Tipul de asociativitate
[] . () (apel de metodă)	de la stânga la dreapta
! ~ ++ -- +(unar) -(unar) () (cast) new	de la dreapta la stânga
* / %	de la stânga la dreapta
+ -	de la stânga la dreapta
<< >> >>>	de la stânga la dreapta
< <= > >= instanceof	de la stânga la dreapta
== !=	de la stânga la dreapta
&	de la stânga la dreapta
^	de la stânga la dreapta
	de la stânga la dreapta
&&	de la stânga la dreapta

Operatorul	Tipul de asociativitate
	de la stânga la dreapta
?:	de la stânga la dreapta
= += -= *= /= %= &= = ^= <<= >>= >>>=	de la dreapta la stânga

4.9. Aplicație cu variabile și operatori

VariabileOperatori.java

```

public class VariabileOperatori
{
    public static void main(String[] args)
    {
        // declararea variabilelor
        int a = 1;
        int b = 1;

        // operatori aritmetici
        int c = a + b;
        System.out.println("a+b=" + c);
        c = a - b;
        System.out.println("b-a=" + c);
        System.out.println("b%a=" + b % a);

        // incrementare decrementare
        b = a++;/*
            * i se transmite valoarea variabilei a(3) variabilei b(b=3)
            * si apoi variabila a este incrementata(a=4)
            */
        System.out.println("a=" + a + "\tb=" + b);

        b = ++a;/*
            * se incrementeza valoarea variabilei a(++a=5)si apoi i se
            * transmite noua valoare variabilei b(b=5)
            */
        System.out.println("a=" + a + "\tb=" + b);
        b = a--;
        System.out.println("a=" + a + "\tb=" + b);
        b = --a;
        System.out.println("a=" + a + "\tb=" + b);

        // operatori relationali
        if (a > b)
            System.out.println("a este mai mare de decat b");
        else
            if (a < b)
                System.out.println("a este mai mic decat b");
            else
                System.out.println("a si b sunt egale");
    }
}

```

```
// operatorul ternar
double x = 1;
double y = 4;
char d = x < y ? 'x' : 'y';
System.out.println("dintre x si y mai mic este: " + d);

// Math
double z = Math.sqrt(y);
System.out.println("radical din x este:" + z);
double u = Math.pow(y, y);
System.out.println("x la puterea x este" + u);

// casts
double t = 9.9999;
int tt = (int) t;
System.out.println("t=" + t + "\ttt=" + tt);
double r = Math.round(t);
System.out.println("t rotunjit este " + r);
}
}
```

5. Citirea de la tastatură – clasa **Scanner**

Permite formatarea unor date primite pe un flux de intrare.

Pentru a avea acces la clasa `Scanner`, aceasta trebuie *importată* în proiect folosind instrucțiunea `import`:

```
import java.util.Scanner;
```

Pentru a citi date de la tastatură trebuie construit un obiect de tip `Scanner` ce are ca argument al constructorului fluxul `System.in`:

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

În continuare se pot utiliza metodele definite în clasa `Scanner` pentru preluarea datelor de la tastatură. De exemplu:

- date ce vor fi memorate într-o variabilă `String`

```
String nume = tastatura.nextLine();
```

- date ce vor fi memorate într-o variabilă de tip întreg

```
int ani = tastatura.nextInt();
```

Mai multe detalii: <http://java.sun.com/javase/6/docs/api/java/util/Scanner.html>.

5.1. Introducerea datelor de la tastatură și afișarea lor

Intrarilesiri.java

```
import java.util.Scanner;

public class IntrariIesiri
{
    public static void main(String[] args)
    {
        /*
         * pentru a citi date de la consola trebuie construit un obiect de
         * tip Scanner
         */
        Scanner tastatura = new Scanner(System.in);
        System.out.print("Introduceti numele si prenumele: ");
        /*
         * utilizarea metodei nextLine(). nextLine() parcurge linia si
         * returneaza intrarea care a fost parcursa
         */
        String nume = tastatura.nextLine();
        System.out.print("Cati ani aveti?");
        // metoda next.Int() scaneaza intrarea intr-o variabila de tip int.
        int ani = tastatura.nextInt();
        System.out.println("Buna " + nume + " anul viitor veti avea " +
                           (ani + 1) + " ani.");

        tastatura.nextLine();
        System.out.print("str=");
        String str = tastatura.nextLine();
    }
}
```

LABORATORUL 3

6. Structuri de control

6.1. Instrucțiuni de decizie / selecție

6.1.1. Instrucțiunea `if`

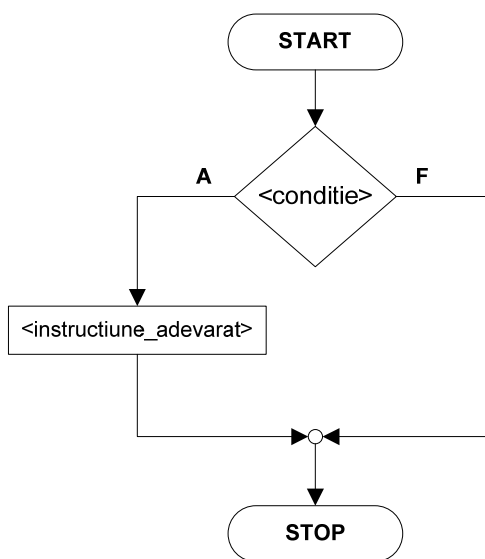
Instrucțiunea `if` cere mașinii virtuale Java să evalueze o expresie (condiție) la o valoare de adevăr. Dacă expresia este adevărată, atunci sunt executate una sau mai multe instrucțiuni aflate în blocul `if`. Dacă expresia nu este adevărată, instrucțiunile respective sunt omise.

Expresia evaluată dintr-o instrucțiune `if` se numește *expresie condițională*.

Format general:

```
if (<conditie>)  
    [<instructiune_adevarat>;
```

Schemă logică:



Exemplu:

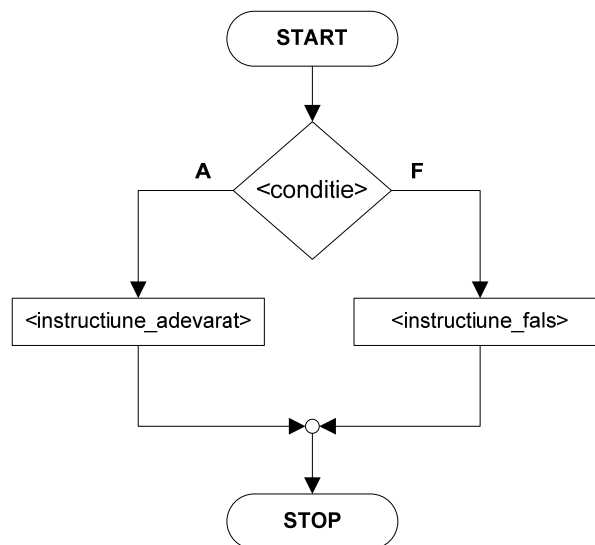
```
int m=3;  
int n=4;  
if(m>n)  
    System.out.println("m");  
if(n>=m)  
    System.out.println("n");
```

6.1.1.1. Clauza else

Această clauză aparține instrucțiunii `if` și conține una sau mai multe instrucțiuni care sunt executate atunci când expresia condițională din instrucțiunea `if` este falsă.

Format general:

```
if (<conditie>)
    [<instructiune_adevarat>];
else
    [<instructiune_fals>];
```

Schemă logică:**Exemplu:**

```
int m=3;
int n=4;
if(m>n)
    System.out.println("m");
else
    System.out.println("n");
```

6.1.1.2. Clauza else if**Format general:**

```
if (<conditie_1>)
    <instructiune_adevarat_1>;
else if (<conditie_2>)
    <instructiune_adevarat_2>;
else
    <instructiune_fals>;
```


Exemplu:

```
int m=3;
int n=4;
if(m>n)
    System.out.println("m");
else if(m==n)
    System.out.println("m=n");
else
    System.out.println("n");
```

6.1.1.3.Instrucțiuni if imbricate

Există situații când trebuie luată o nouă decizie dacă prima condiție este adevărată.

Format general:

```
if (<conditie>)
    if (<conditie_i>)
        <instructiune_adevarat_i>;
    else
        <instructiune_fals_i>;
else
    <instructiune_fals>;
```

Exemplu:

```
int m=3;
int n=4;
if(m>0 && n>0)
    if(m==n)
        System.out.println("m=n");
    else
        System.out.println("m!=n");
else
    System.out.println("m si n sunt negative");
```



1. după condiție nu se pune ;
2. dacă blocurile `if` sau `else` conțin mai mult de o instrucțiune, atunci este necesar ca instrucțiunile să fie încadrate de acolade. Se recomandă utilizarea acoladelor întotdeauna, indiferent de numărul de instrucțiuni din blocurile `if` și `else`.

6.1.1.4. Utilizarea instrucțiunii if**EcuatiaDeGradulDoi.java**

```
import java.util.*;

public class EcuatiaDeGradulDoi
{
    public static void main(String[] args)
    {
        /*
         * construirea unui obiect de tip Scanner pentru citirea de la
         * tastatura a coeficientilor ecuatiei de gradul 2
         */
        Scanner valoare = new Scanner(System.in);

        System.out.print("coeficientul lui x^2:");
        // memorarea coeficientului lui x^2 in variabila a
        double a = valoare.nextDouble();

        System.out.print("coeficientul lui x:");
        // memorarea coeficientului lui x in variabila b
        double b = valoare.nextDouble();

        System.out.print("termenul liber:");
        // memorarea termenului liber in variabila c
        double c = valoare.nextDouble();

        if (a == 0 && b == 0 && c == 0)
        {
            System.out.println("Ecuatie nedeterminata !");
        }
        else
        {
            if (a == 0 && b == 0)
            {
                System.out.println("Ecuatie imposibila !");
            }
            else
            {
                if (a == 0)
                {
                    System.out.println("Ecuatie de grad I");
                    System.out.println("Solutia este:" + -c / b);
                }
                else
                {
                    double delta = b * b - 4 * a * c;
                    if (delta < 0)
                    {
                        System.out.println("Radacini complexe");
                    }
                }
            }
        }
    }
}
```

```

        System.out.println("x1= " + (-b / (2 * a)) + "+ i*" +
                           (Math.sqrt(-delta) / (2 * a)));
        System.out.println("x2= " + (-b / (2 * a)) + "- i*" +
                           (Math.sqrt(-delta) / (2 * a)));
        // afisarea cu printf
        // System.out.printf("x1=%.2f +i*%.2f", -b / (2 * a),
        // Math.sqrt(-delta) / (2 * a));
        // System.out.printf("\nx2=%.2f -i*%.2f", -b / (2 * a),
        // Math.sqrt(-delta) / (2 * a));
    }
    else
    {
        if (delta == 0)
        {
            System.out.println("Radacini reale si egale:");
            System.out.println("x1=x2=" + (-b / (2 * a)));
        }
        else
        {
            if (delta > 0)
            {
                System.out.println("Radacini reale si distincte:");
                System.out.println("x1=" + ((-b + Math.sqrt(delta)) /
                                           (2 * a)));
                System.out.println("x2=" + ((-b - Math.sqrt(delta)) /
                                           (2 * a)));

                // afisarea cu printf
                // System.out.printf("x1=", -b + Math.sqrt(delta) /
                //                      (2 * a));
                // System.out.printf("\nx2=", -b - Math.sqrt(delta) /
                //                      (2 * a));
            }
        }
    }
}

```

6.1.2. Instrucțiunea *switch*

O soluție alternativă la folosirea unei lungi serii de clauze `if...else` o reprezintă folosirea unei instrucțiuni de control `switch`, cunoscută și sub numele de instrucțiunea `switch...case`.

Instrucțiunea `switch` cere mașinii Java să compare o expresie cu mai multe valori date, conținute într-o etichetă `case`. Dacă valorile se potrivesc, se execută instrucțiunile din blocul `case`.

Format general:

```

switch(<expresie>)
{
    case(<valoare_1>): <instructiuni_1>;
                      [break;]
    case(<valoare_2>): <instructiuni_2>;
                      [break;]

    :
    :

    case(<valoare_n>): <instructiuni_n>;
                      [break;]
    default: <instructiuni>;
             [break;]
}

```

Exemplu:

```

int optiune=10;
switch (optiune)
{
    case5: System.out.println("5");
           break;
    case10: System.out.println("10");
            break;
    default: System.out.println("Nici o potrivire");
}

```



1. În cazul în care în clauza `case` a cărei constantă se potrivește cu valoarea expresiei din `switch` și comanda `break` nu este folosită, atunci se execută și instrucțiunile din clauzele `case` următoare, până când întâlnește primul `break` sau se termină instrucțiunea `switch`.
2. O clauză `else` este folosită într-o instrucțiune `if` pentru a defini instrucțiunile executate atunci când condiția instrucțiunii nu este adevărată. Instrucțiunea `switch` are o funcție similară, numită clauză implicită (*default statement*).
3. Clauza implicită seamănă cu o clauză `case` și este plasată întotdeauna după ultima clauză `case`. Aceasta nu are atașată o constantă și nici o instrucțiune `break`, deoarece este ultima instrucțiune din blocul `switch`. Instrucțiunile conținute în clauza implicită sunt executate atunci când Java ajunge la aceasta.

6.1.2.1. Instrucțiuni switch imbricate

O instrucțiune `switch` imbricată este o instrucțiune `switch` inserată într-o clauză `case` sau `default` a unei alte instrucțiuni `switch`.

6.2. Instrucțiuni iterative

Acest tip de instrucțiuni cer mașinii virtuale Java să execute una sau mai multe instrucțiuni, în mod repetat, atât timp cât este îndeplinită o condiție.

În Java există 3 tipuri de instrucțiuni iterative:

- `for`: instrucțiune iterativă cu test inițial, cu număr cunoscut de pași;
- `while`: instrucțiune iterativă cu test inițial, cu număr necunoscut de pași;
- `do...while`: instrucțiune iterativă cu test final, cu număr necunoscut de pași.

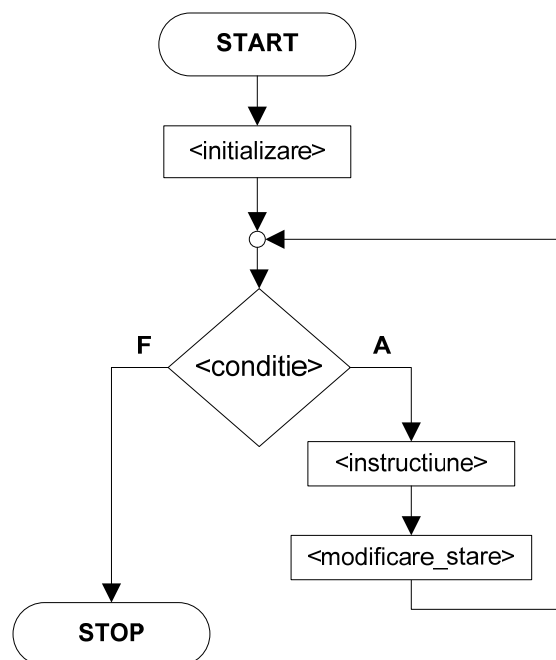
6.2.1. Instrucțiunea `for`

Instrucțiunea `for` se utilizează atunci când se cunoaște sau se poate ante-calcula numărul de repetări a instrucțiunii / blocului de instrucțiuni și corpul instrucțiunii `for`.

Format general:

```
for ([<initializare>]; [<conditie_continuare>]; [<modificare_stare>])
    [<instrucțiune>];
```

Schemă logică:



Exemplu:

```
for (int i=0 ; i<=2 ; i++)
{
    System.out.println("i="+i);
}
```

Modul de lucru:

Ciclu	Instrucțiune / operație
1	i=0 test 0<=2 (A) afișare „i=0” i=1
2	test 1<=2 (A) afișare „i=1” i=2
3	test 2<=2 (A) afișare „i=2” i=3
4	test 3<=2 (F) → se termină instrucțiunea for

Expresii de inițializare alternative:

```
int i;
for(i=0;i<=2;i++)
{
    System.out.println("i="+i);
}
```

```
int i=0;
for(;i<=2;i++)
{
    System.out.println("i="+i);
}
```



1. Instrucțiunea `for` are, în paranteze, trei expresii separate prin caracterul ‘;’ – toate expresiile sunt opționale, dar caracterele punct și virgulă și parantezele sunt obligatorii.
2. După parantezele instrucțiunii `for` nu se pune ‘;’. Caracterul ‘;’ în această poziție are semnificația instrucțiunii vide, adică instrucțiunea care nu face nimic. Practic această instrucțiune vidă se va executa în ciclu.

6.2.1.1. Cicluri for imbricate

Se poate de vorbi despre cicluri `for` imbricate atunci când se plasează unul sau mai multe cicluri `for` în interiorul altui ciclu `for`.

Ciclul `for` imbricat este numit *ciclu intern* și este plasat în interiorul unui *ciclu extern*.

De fiecare dată când se parcurge ciclul `for` extern, se execută, complet, și ciclul `for` intern.

Exemplu:

```

for(int i=0;i<3;i++)
{
    System.out.print("\ni="+i);
    for(int j=0;j<=2;j++)
    {
        System.out.print("\tj="+j);
    }
}

```

Rezultatul execuției:

```

i=0    j=0    j=1    j=2
i=1    j=0    j=1    j=2
i=2    j=0    j=1    j=2

```

6.2.1.2. Utilizarea instrucțiunii for**ExempluFor.java**

```

public class ExempluFor
{
    public static void main(String[] args)
    {
        Scanner tastatura = new Scanner(System.in);
        System.out.print("Introduceti un numar:");
        /*
         * memorarea numarului introdus de la tastatura intr-o variabila "a"
         */
        int a = tastatura.nextInt();

        System.out.print("Introduceti un divizor: ");
        int b = tastatura.nextInt();

        // declararea unei variabile de tip primitiv (int)
        int x;

        {
            int suma = 0;
            for (x = 1; x <= a; x++)
                suma += x; // acelasi lucru cu: suma=suma+x;
            System.out.println("Suma celor " + a + " numere este " + suma);
        }

        {
            int suma = 0;
            for (x = 1; x <= a; x += 2)
                suma += x;
        }
    }
}

```

```
        System.out.println("Suma numerelor impare pana la " + a +  
                            " este " + suma);  
    }  
  
    {  
        int suma = 0;  
        for (x = 0; x <= a; x += 2)  
            suma += x;  
        System.out.println("Suma numerelor pare pana la " + a +  
                            " este " + suma);  
    }  
  
    // numere prime varianta 1  
    {  
        System.out.print("Numerele prime de la 1 la " + a +  
                            " (v1) sunt:");  
  
        for (int i = 1; i <= a; i++)  
        {  
            boolean numarPrim = true;  
  
            /*  
             * realizarea unui ciclu for imbricat pentru a verifica  
             * daca este indeplinita conditia ca numarul prim  
             * sa fie divizibil doar cu 1 si cu el insusi  
             */  
  
            for (int j = 2; j < i; j++)  
            {  
                if (i % j == 0)  
                {  
                    numarPrim = false;  
                    break; // fortarea iesirii din instructiunea for  
                }  
            }  
  
            // afisarea numarului daca este prim  
            if (numarPrim)  
            {  
                System.out.print("\t" + i);  
            }  
        }  
    }  
  
    // numere prime varianta 2  
    {  
        System.out.print("\nNumerele prime de la 1 la " + a +  
                            " (v2) sunt:");  
  
        for (int i = 1; i <= a; i++)  
        {  
            boolean numarPrim = true;
```



```
        for (int j = 2; j <= i/2; j++)
        {
            if (i % j == 0)
            {
                numarPrim = false;
                break;
            }
        }
        if (numarPrim)
        {
            System.out.print("\t" + i);
        }
    }
}

// varianta 3
{
    System.out.print("\nNumerele prime de la 1 la " + a +
        " (v3)sunt:");

    for (int i = 1; i <= a; i++)
    {
        boolean numarPrim = true;
        for (int j = 2; j <= Math.sqrt((double) i); j++)
        {
            if (i % j == 0)
            {
                numarPrim = false;
                break;
            }
        }

        if (numarPrim)
        {
            System.out.print("\t" + i);
        }
    }
}

{
    System.out.print("\nNumerele de la 1 la " + a +
        " divizibile cu " + b + " sunt: ");
    for (int i = 1; i <= a; i++)
    {
        if (i % b == 0)
            System.out.print("\t" + i);
    }
}
}
```


LABORATORUL 4

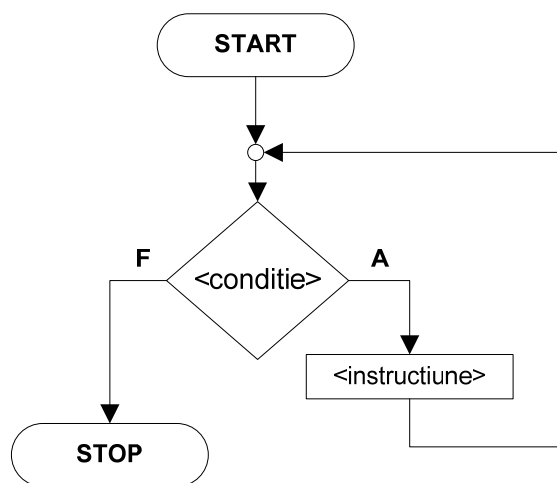
6.2.2. Instrucțiunea *while*

Ciclul `while` cere mașinii virtuale Java să execute una sau mai multe instrucțiuni, din corpul ciclului, atât timp cât o expresie condițională este adevărată.

Format general:

```
while (<conditie_continuare>)\n    [<instrucțiune>];
```

Schemă logică:



Exemplu:

```
int i=0;\nwhile (i<=2)\n{\n    System.out.println("i="+i);\n    i++;\n}
```

Atunci când întâlnește un ciclu `while` în program, Java evaluează expresia condițională. Dacă această condiție este evaluată ca falsă, Java sare peste corpul instrucțiunii `while` și continuă execuția cu prima instrucțiune care apare în program, după corpul instrucțiunii `while`.

Dacă una din instrucțiunile aflate în corpul instrucțiunii `while` este `break`, Java întrerupe execuția instrucțiunii `while` și continuă cu prima instrucțiune de după `while`.

6.2.2.1. Utilizarea instructiunii while**ExempluWhile.java**

```
import java.util.Scanner;

public class ExempluWhile
{
    public static void main(String[] args)
    {
        Scanner a = new Scanner(System.in);

        System.out.print("Cati bani doresti sa retragi? ");
        // memorarea valorii creditului in variabila retragere
        double retragere = a.nextDouble();

        System.out.print("Cu cati bani vrei sa contribui in fiecare an? ");
        // memorarea valorii ratei anuale in variabila rata
        double rata = a.nextDouble();

        System.out.print("Introduceti dobanda %: ");
        /*
         * declararea unei variabile (dobanda) pentru memorarea ratei
         * dobanzii (in procente)
         */
        double dobanda = a.nextDouble();

        /*
         * declararea unei variabile (returnat) pentru memorarea valorii
         * returnate in fiecare an
         */
        double returnat = 0;

        /*
         * declararea unei variabile (luni) pentru memorarea timpului scurs
         * de la acordarea creditului pana la lichidarea creditului
         */
        double luni = 0;
        while (returnat < retragere)
        {
            /*
             * declararea unei variabile pentru memorarea valorii dobanzii,
             * valoare ce este calculata in fiecare an in functie de
             * suma ramasa de rambursat
             */
            double dobandaP = ((retragere - returnat) * dobanda) / 100;

            // valoarea rambursata in fiecare an
            double x = rata - dobandaP;
```

```

    if ((retragere - returnat) < (x))
    {
        // calculul ultimelor luni cand suma de rambursat impreuna cu
        // dobanda este mai mica decat rata anuala
        luni += (((retragere - returnat) + dobandaP) * 12) / x;
    }
    else
    {
        luni += 12;
    }
    returnat += x;

}

int l = (int) luni % 12;
System.out.println("Poti plati in " + (int) luni / 12 + " ani si " +
                    l + " luni.");
}
}

```

6.2.3. Instrucțiunea *do..while*

Ciclul *do..while* cere mașinii virtuale Java să execute una sau mai multe instrucțiuni, din corpul ciclului, atât timp cât o expresie condițională este adevărată.

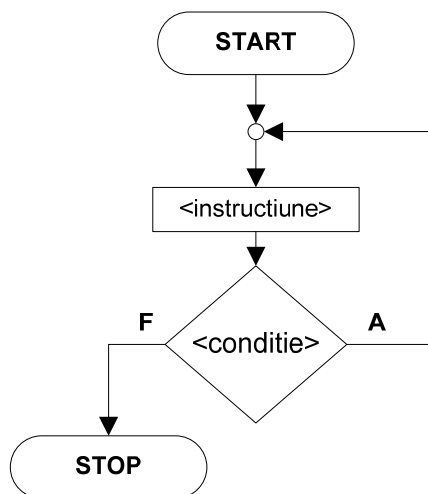
Format general:

```

do {
    [<secventa_de_instructiuni>;]
} while (<conditie_de_continuare>);

```

Schemă logică:



Ciclul *do..while* este format din șase componente: cuvântul cheie *do*, corpul ciclului *do..while*, definit de acolade de deschidere și închidere, una sau mai multe instrucțiuni din

corpul ciclului, cuvântul cheie `while`, expresia condițională, încadrată de paranteze, și un caracter punct și virgulă.

Exemplu:

```
int i=0;
do {
    System.out.println("i="+i);
    i++;
} while(i<=2);
```



Instrucțiunile dintr-un ciclu `do..while` (aceasta fiind o instrucțiune de ciclare cu test final) sunt executate cel puțin o dată, chiar dacă expresia condițională este falsă.

6.3. Instrucțiuni de salt

Aceste instrucțiuni transferă controlul execuției într-o altă parte a programului.

6.3.1. Instrucțiunea *break*

Această instrucțiune cere mașinii virtuale Java să iasă dintr-un bloc de cod definit de acolade de deschidere și închidere.

Exemplu:

```
for(int i=2;i<=5;i++)
{
    if(i==4)
        break;

    System.out.println("i="+i);
}
```

Rezultatul rulării:

```
i=2
i=3
```

În momentul în care condiția din `if` este îndeplinită ($i=4$), se execută instrucțiunea `break` și chiar dacă condiția din `for` ($i<=5$) este adevărată ciclul se încheie.

6.3.2. Instrucțiunea *continue*

Instrucțiunea *continue* este folosită în corpul unui ciclu pentru a cere mașinii virtuale Java să renunțe la execuția iterației curente și să treacă la următoarea.

Exemplu:

```
for(int i=2;i<=5;i++)
{
    System.out.println("Inainte de continue i="+i);

    if(i==4)
        continue;

    System.out.println("Dupa continue i="+i);
}
```

Rezultatul rulării:

```
Inainte de continue i=2
Dupa continue i=2
Inainte de continue i=3
Dupa continue i=3
Inainte de continue i=4
Inainte de continue i=5
Dupa continue i=5
```

În momentul în care condiția din *if* este îndeplinită (*i=4*) se execută instrucțiunea *continue*. După acest apel se ignoră toate instrucțiunile aflate după instrucțiunea *continue* și se trece la următoarea iterație.

6.3.3. Instrucțiunea *return*

Instrucțiunea *return* este folosită în metode pentru a transfera înapoi controlul execuției la instrucțiunea care a apelat metoda. Instrucțiunea poate returna o valoare, dar acest lucru nu este obligatoriu.

6.4. Utilizarea instrucțiunilor `do..while`, `switch`, `break` și `continue`

ExDoWhileSwitch.java

```
import java.util.Scanner;

public class ExDoWhileSwitch
{
    public static void main(String[] args)
    {
        Scanner n = new Scanner(System.in);

        int a, b, c;
        do {
            System.out.print("Introdu un numar natural: ");
            a = n.nextInt();
            System.out.print("Introdu un divizor: ");
            b = n.nextInt();
        } while (a <= 0 && b <= 0);

        do {
            // afisarea meniului

            System.out.println("\nMENIU:");
            System.out.println("1.Afisarea sumei primelor " + a
                               + "numere naturale");
            System.out.println("2.Afisarea sumei numerelor impare de la 1 la"
                               + a);
            System.out.println("3.Afisarea sumei numerelor pare de la 1 la"
                               + a);
            System.out.println("4.Afisarea primelor numere prime de la 1 la"
                               + a);
            System.out.println("5.Afisarea numerelor de la 1 la " + a
                               + "divizibile cu" + b);
            System.out.println("6.IESIRE DIN PROGRAM");

            // citirea optiunii utilizatorului
            System.out.print("Introduceti optiune dvs.:");
            c = n.nextInt();

            switch (c)
            {
                case 1:
                    int suma = 0;
                    for (int x = 1; x <= a; x++)
                        suma += x;
                    System.out.println("Suma primelor " + a +
                                       " numere naturale este: " + suma);

                    break;

                case 2:
```



```
        int sumaimp = 0;
        for (int x = 1; x <= a; x += 2)
            sumaimp += x;
        System.out.println("Suma numerelor impare de la 1 la " + a +
                           " este: " + sumaimp);

        break;

    case 3:
        int sumap = 0;
        for (int x = 0; x <= a; x += 2)
            sumap += x;
        System.out.println("Suma numerelor pare de la 1 la " + a +
                           " este: " + sumap);

        break;

    case 4:
        System.out.print("Numerele prime de la 1 la " + a + " sunt: ");
        for (int i = 1; i <= a; i++)
        {
            boolean numarPrim = true;
            for (int j = 2; j < i/2; j++)
            {
                if (i % j == 0)
                {
                    numarPrim = false;
                    break;
                }
            }
            if (numarPrim)
            {
                System.out.print("\t" + i);
            }
        }
        break;

    case 5:
        System.out.print("Numerele de la 1 la " + a +
                           " divizibile cu " + b + " sunt:\n");
        for (int i = 1; i <= a; i++)
            if (i % b == 0)
            {
                System.out.print("\t");
                System.out.print(i);
            }
        break;

    case 6:
        System.out.println("Sfarsit de program");
        break;

    default:
```

```
        System.out.println("Optiunea dumneavoastra nu este definita.");  
    }  
  
    } while (c != 6);  
}  
}
```

LABORATORUL 5

7. Tablouri

Rezervarea memoriei pentru stocarea unei date este o operație foarte simplă: se folosește un tip de date și un nume pentru a declara o variabilă.

Să presupunem că trebuie rezervată memorie pentru a stoca 100 de elemente de un anumit tip. La prima vedere nu pare complicat: trebuie declarate 100 de variabile. Acest lucru presupune găsirea a 100 de nume sugestive și unice, o sarcină nu tocmai ușoară, iar managementul variabilelor ar fi un coșmar. Acest lucru poate fi simplificat prin declararea unui *tablou*. Un tablou are nevoie de un singur nume unic și sugestiv, care poate fi apoi folosit pentru referirea tuturor celor 100 de elemente de date sau oricâte sunt necesare în program.

Un tablou este o colecție de variabile de același tip de date, căreia îi este asociat un nume. Fiecare variabilă din colecție este numită element al tabloului. Un element al tabloului este identificat printr-o combinație între numele tabloului și un index unic.

Un index este un întreg cuprins între zero și numărul maxim de elemente ale tabloului, minus unu. Indexul este specificat între paranteze drepte, în dreapta numelui variabilei tablou.

7.1. Tablouri unidimensionale – Vectori

7.1.1. Declararea variabilei tablou

Declararea unei variabile de tip tablou unidimensional se face utilizând sintaxa:

Sintaxă utilizată:	<code><tip_elemente> <identificator_tablou>[] ;</code> sau <code><tip_elemente>[] <identificator_tablou>;</code>
--------------------	--

unde:

- `<tip_elemente>` reprezintă tipul de date al elementelor vectorului
- `<identificator_tablou>` este numele vectorului, care trebuie să fie unic și sugestiv
- `[]` specifică faptul că este declarat un vector, și nu o simplă variabilă

7.1.2. Instanțierea

Instanțierea definește procesul de creare a unui obiect, respectiv alocarea memoriei pentru acesta și efectuarea tuturor inițializărilor necesare.



Tipul tablou este un tip de date referință, prin urmare trebuie utilizată instanțierea pentru crearea unui obiect de tip tablou.

Memoria este alocată dinamic, printr-un proces în trei etape:

1. Declararea unei variabile referință;
2. Alocarea memoriei;
3. Asocierea locației de memorie rezervată cu variabila referință.

Declararea și instanțierea unui vector pot fi făcute simultan:

```
int[] note = new int[6];
```

Operatorul `new` cere mașinii virtuale Java să rezerve memorie pentru stocarea a 6 valori întregi prin `new int[6]`.

Operatorul `new` returnează adresa primului element. Celelalte elemente sunt dispuse, în memorie, una lângă cealaltă, secvențial, după primul element.

Adresa returnată este memorată în variabila referință.



1. Când se declară un tablou se specifică întotdeauna numărul de elemente.
2. Primul index al tabloului este 0, și nu 1, iar ultimul element al tabloului are indexul *numărul de elemente-1*, și nu *numărul de elemente*.

7.1.3. Inițializarea

Reprezintă procesul prin care unei variabile `i` se dă o valoare.

La inițializarea unui vector valorile inițiale trebuie incluse între acolade și separate prin virgule.

```
int note[]={10,9,10,8,9};
```

Nu este nevoie să se specifice explicit dimensiunea tabloului. Numărul de valori inițiale este folosit pentru a stabili dimensiunea acestuia.



Nu se folosește operatorul `new` pentru declararea unui tablou dacă acesta se inițializează. Java alocă dinamic memorie suficientă pentru tablou, folosind numărul de valori din inițializare pentru a-i determina dimensiunea.

7.2. Tablouri multidimensionale – Matrice

Tablourile pot avea mai multe dimensiuni. Acestea sunt numite tablouri multidimensionale sau matrice.

7.2.1. Crearea unui tablou multidimensional

Declararea unei matrice se poate face folosind operatorul `new`, într-un mod asemănător cu cel folosit pentru tablouri unidimensionale. Exemplu următor ilustrează această tehnică.

```
int note[][] = new int [3][2];
```

Fiecare pereche de paranteze drepte din declararea unei matrice reprezintă o dimensiune.

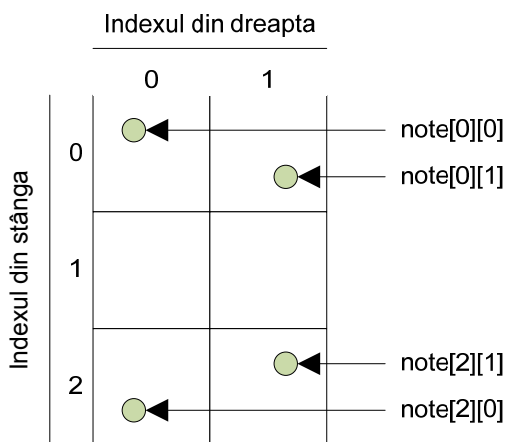


Fig. 14. Elementele tablourilor multidimensionale

7.2.2. Atribuirea valorilor către elementele matricei

Un element al unui tablou este folosit într-o expresie la fel ca și o variabilă. Singura diferență dintre o variabilă și un element al unui tablou este faptul că pentru elementul tabloului trebuie specificat atât numele tabloului cât și indexul elementului.

Se poate atribui o valoare unui element al tabloului folosind operatorul de atribuire.

```
note[0]=9;
```

Valoarea atribuită unui element al unui tablou poate fi folosită într-o expresie la fel ca și valoarea unei variabile.

```
note[1]=note[0];
```

Același format este folosit și pentru tablourile ce folosesc două sau mai multe dimensiuni:

```
note[1][1]=note[0][1];
```

Un tablou multidimensional este, de fapt, un tablou de tablouri. În exemplu anterior, tabloul `note` este un vector de 3 elemente de tip vector de 2 elemente.

7.2.3. Proprietăți *length*

Tipul tablou, fiind un tip referință, are proprietăți ce pot fi utilizate de programator. Cea mai utilă este proprietatea `length`, care returnează numărul de elemente al tabloului.

Exemplu:

```
int note[]={10,9,10,8,9};

System.out.print("Notele sunt:");

for(int i=0;i<note.length;i++)
{
    System.out.print(" " + note[i]);
}
```

7.3. Clasa `Arrays`

Clasa `Arrays` definește un număr mare de metode utile în lucrul cu tablouri. Cele mai des utilizate sunt: `equals()`, `fill()`, `sort()`, `binarySearch()`. Toate aceste metode sunt *statice*, deci nu depind de instanțe, putându-se folosi prin intermediul numelui clasei (`Arrays`).

Clasa `Arrays` se află în pachetul `java.util`, deci pentru a o putea folosi trebuie importat acest pachet.

7.3.1. Metoda *equals()*

Această metodă este folosită pentru a compara elementele unui tablou.

Două tablouri sunt transmise ca parametrii metodei `equals()`, care stabilește dacă tablourile sunt identice (elementele conținute sunt egale). În caz afirmativ, metoda `equals()` returnează valoarea booleana `true`, altfel returnează valoarea booleana `false`.

ExempluEquals.java

```
import java.util.*;

public class ExempluEquals
{
    public static void main(String[] args)
    {
        int note1[] = new int[3];
        note1[0] = 10;
        note1[1] = 9;
        note1[2] = 8;
```

```

    int note2[] = new int[3];
    note2[0] = 10;
    note2[1] = 9;
    note2[2] = 8;

    if (Arrays.equals(note1, note2))
        System.out.println("Notele sunt egale!");
    else
        System.out.println("Notele nu sunt egale!");
}
}

```

Metoda `equals()` este apelată prin referirea clasei `Arrays`. Numele clasei trebuie să preceadă numele metodei folosită în program, iar numele clasei și numele metodei trebuie să fie separate prin operatorul punct.

7.3.2. Metoda `fill()`

Se folosește când trebuie atribuite valori inițiale elementelor unui tablou.

În exemplul următor sunt prezentate două versiuni ale metodei:

- Prima versiune are nevoie de două argumente: numele tabloului și valoarea care va fi atribuită elementelor matricei. Este important ca tipul de date al acestei valori să fie compatibil cu cel al elementelor tabloului.
- A doua versiune are nevoie de patru argumente: numele tabloului, indexul primului element a cărui valoare va fi schimbată, indexul ultimului element al cărui valoare va fi schimbată și valoarea ce va fi atribuită elementelor în intervalul specificat.

ExempluFill.java

```

import java.util.*;

public class ExempluFill
{
    public static void main(String[] args)
    {
        // se utilizeaza un vector de 2000 de elemente de tip int
        int note[] = new int[2000];

        Arrays.fill(note, 0);

        for (int i = 0; i < note.length; i++)
            System.out.println("" + note[i]);

        Arrays.fill(note, 100, 1500, 3);
    }
}

```

```
        for (int i = 0; i < note.length; i++)
            System.out.println(i + " " + note[i]);
    }
}
```

7.3.3. Metoda *sort()*

Sortează crescător elementele unui tablou folosind algoritmul *Quick-Sort*. Dacă elementele tabloului sunt tipuri referință (clase), acestea trebuie să implementeze interfața *Comparable*.

Metoda primește un singur argument – variabila tablou.

ExempluSort.java

```
import java.util.*;

public class ExempluSort
{
    public static void main(String[] args)
    {
        int note[] = { 5, 2, 4, 1, 7, 3, 5, 7, 0 };

        Arrays.sort(note);

        for (int i = 0; i < note.length; i++)
            System.out.println(note[i]);
    }
}
```

7.3.4. Metoda *binarySearch()*

Această metodă caută un anumit element într-un tablou.

Pentru a folosi această metodă tabloul trebuie să fie sortat înainte de a se face căutarea.

Metoda `binarySearch()` are nevoie de două argumente: numele tabloului și criteriul de căutare. Criteriul de căutare trebuie să fie compatibil cu tipul de date al elementelor tabloului.

Metoda returnează o valoare întreagă. Acesta poate fi un număr pozitiv sau un număr negativ. Un număr pozitiv reprezintă indexul elementului care conține valoarea căutată. Un număr negativ înseamnă că valoarea căutată nu a fost găsită în tablou.

ExempluBinarySearch.java

```
import java.util.*;

public class ExempluBinarySearch
{
    public static void main(String[] args)
    {
        int note[] = { 5, 2, 4, 1, 7, 3, 5, 7, 0 };
        int index;

        Arrays.sort(note);

        index = Arrays.binarySearch(note, 2);
        System.out.println(index);

        index = Arrays.binarySearch(note, 99);
        System.out.println(index);
    }
}
```

Rezultatul rulării:

```
2
-10
```

7.3.5. Metoda `arraycopy()`

Metoda `arraycopy()` se utilizează pentru a copia elementele unui tablou în alt tablou sau pentru a copia o parte din elementele unei tablou în alt tablou.

Această metodă statică face parte din clasa `System` și are 5 argumente: numele tabloului sursă, indexul elementului din tabloul sursă de unde începe copierea, numele tabloului destinație, indexul elementului din tabloul destinație de unde începe copierea și numărul de elemente ce vor fi copiate.

ExempluArrayCopy.java

```
import java.util.*;

public class ExempluArrayCopy
{
    public static void main(String[] args)
    {
        int note1[] = { 5, 2, 4, 1, 7, 3, 5, 7, 0 };
        int note2[] = new int[9];

        Arrays.fill(note2, 0);
    }
}
```

```
System.arraycopy(note1, 3, note2, 1, 4);

for (int i = 0; i < note2.length; i++)
    System.out.print(":" + note2[i]);
}
```

Rezultatul rulării:

```
::0::1::7::3::5::0::0::0::0
```

7.4. Aplicație cu vectori

Vectori.java

```
import java.util.*;

public class Vectori
{
    public static void main(String[] args)
    {
        int[] nota = new int[4];

        System.out.println("Introduceti notele de la fiecare laborator: ");
        citire(nota);
        afisare(nota);
        System.out.println("Media este: " + media(nota));

        // schimbarea valorilor unor elemente
        Arrays.fill(nota, 0, 2, 9);
        afisare(nota);

        System.out.println("Laboratorul 1 si 2 au acum valorile: " +
                           nota[0] + " si " + nota[1]);

        System.out.println("Ordonare  notelor");

        // argumentul este de tip referinta iar metoda ii va modifica starea
        Arrays.sort(nota);
        afisare(nota);

        // copierea vectorilor
        int notaCopie1[] = nota;

        System.out.println("Afisarea primei copii");
        afisare(notaCopie1);
    }
}
```

```

int notaCopie2[] = new int[4];
System.arraycopy(nota, 0, notaCopie2, 0, nota.length);

System.out.println("Afisarea copiei numarul 2");
afisare(notaCopie2);

int[] mat1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] mat2 = { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
/*
 * se copieaza mat1 incepand cu indexul 1(2 elemente) in mat2
 * la pozitia index5 a tabloului
 * ( arraycopy(from,formindex,to,to index,count) )
 */
System.arraycopy(mat1, 1, mat2, 5, 2);
System.out.println("Noile elementele ale matricii mat2 sunt:");
afisare(mat2);
}

static void citire(int nota[])
{
    //atribuirea valorilor catre elementele tabloului
    Scanner valoare = new Scanner(System.in);
    for (int i = 0; i < nota.length; i++)
    {
        System.out.print("Laboratorul " + (i + 1) + ": ");
        nota[i] = valoare.nextInt();
    }
}

static void afisare(int[] nota)
{
    for (int i = 0; i < nota.length; i++)
        System.out.println("Laborator " + (i + 1) + ": " + nota[i]);
}

static double media(int nota[])
{
    double media = 0;
    for (int i = 0; i < nota.length; i++)
        media += nota[i];

    return media / nota.length;
}

static void afisareCopie1(int[] nota)
{
    for (int i = 0; i < nota.length; i++)
        System.out.println(nota[i]);
}
}

```

7.5. Aplicație cu vectori: Loteria

Loteria.java

```
import java.util.*;

public class Loteria
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Cate bile doresti sa extragi? ");
        int k = in.nextInt();
        System.out.print("Care este nuamarul maxim de bile? ");
        int n = in.nextInt();

        // construirea unei matrici numarul total de bile
        int[] bile = new int[n]; // maxim de bile
        for (int i = 0; i < bile.length; i++)
            bile[i] = i + 1;

        // construirea unei matrici pentru bilele extrase
        int[] extras = new int[k]; // bile extrase
        for (int i = 0; i < extras.length; i++)
        {
            // realizarea unei extrageri in mod aleatoru intre 0 si n-1
            int r = (int) (Math.random() * n);
            // extragerea unei bile
            extras[i] = bile[r];
            // mutarea ultimului element in locatia aleatoare
            bile[r] = bile[n - 1];
            n--;
        }

        // sortarea bilelor extrase in ordine crescatoare
        Arrays.sort(extras);
        System.out.println("Cea mai buna combinatie: ");
        // afisarea bilelor extrase
        // for(variabila:colectie)->afiseaza elementele colectiei

        for (int r : extras)
            System.out.println(r);

        // sau
        // for(int r=0;r<extras.length;r++)
        // System.out.println(extras[r]);
    }
}
```

7.6. Aplicație cu o matrice bidimensională

MatriceBi.java

```
public class MatriceBi
{
    public static void main(String[] args)
    {
        final int capTabel = 10;
        final int coloane = 6;
        final int linii = 10;

        // setarea ratei dobanzii 10 . . . 15%(dobandap=dobanda procent)
        double[] dobandap = new double[coloane];
        for (int j = 0; j < dobandap.length; j++)
        {
            dobandap[j] = (capTabel + j) / 100.0;
        }

        // crearea unei matrici bidimensionale pentru afisarea valorilor
        // depozitului pe 10 ani(coloane)cu diferite dobanzi(10-15%)
        // (6 coloane)
        double[][] balanta = new double[linii][coloane];

        // setarea valorii initiale cu 10000
        for (int j = 0; j < balanta[0].length; j++)
        {
            balanta[0][j] = 10000;
        }

        // calcularea ratei pentru anii viitori
        for (int i = 1; i < balanta.length; i++)
        {
            for (int j = 0; j < balanta[i].length; j++)
            {
                // valoarea anului trecut pentru calculul dobanzii
                double oldBalance = balanta[i - 1][j];
                // calcularea valorii dobanzii
                double dobanda = oldBalance * dobandap[j];
                // calcularea valorii corespunzatoare anului si dobanzii
                balanta[i][j] = oldBalance + dobanda;
            }
        }

        // afisarea randului cu valorile procentelor
        for (int j = 0; j < dobandap.length; j++)
        {
            System.out.printf("%9.0f%%", 100 * dobandap[j]);
        }

        System.out.println(); // trecere la randul urmator
    }
}
```

```
// afisarea valorilor in tabel
for (double[] row : balanta)
{
    // afisarea fiecarui rand din tabel
    for (double b : row)
        System.out.printf("%10.2f", b);
    System.out.println();
}
}
```

LABORATORUL 6

8. Șiruri de caractere

Șirurile de caractere pot fi reprezentate de tipurile `String` și `StringBuffer`.

Dacă se dorește ca un șir de caractere să rămână constant, atunci se folosește tipul `String`. Dacă se dorește modificarea șirului, atunci se va utiliza tipul `StringBuffer`, deoarece această clasă pune la dispoziție metode de prelucrare a șirului încapsulat (`append()`, `insert()`, `delete()`, `reverse()`).

Concatenarea șirurilor de caractere se realizează prin intermediul operatorului '+', în cazul șirurilor de tip `String`, și prin intermediul metodei `append()` în cazul șirurilor de tip `StringBuffer`.

8.1. Clasa `String`

Lista tuturor proprietăților și metodelor ale clasei `String` se găsește la adresa:

<http://java.sun.com/javase/6/docs/api/java/lang/String.html>.

Exemple de declarare și instanțiere a obiectelor de tip `String`

```
String sir1 = "sir";

String sir2 = new String("sir");

char sir[] = {'s','i','r'};
String sir3 = new String(sir);

System.out.println(sir1);
System.out.println(sir2);
System.out.println(sir3);
```

ExempluString.java

```
public class ExempluString
{
    public static void main(String[] args)
    {
        // declararea a patru siruri de caractere cu initializare
        String s1 = "Happy girl!";
        String s2 = "Happy boy!";
        String ss1 = "Happy!";
        String ss2 = ss1;
    }
}
```

```
// afisarea sirurilor s1 si s2
System.out.println(s1);
System.out.println(s2);

// compararea sirurilor
int comparare = s1.compareTo(s2);

if (comparare < 0)
{
    System.out.println("Sirul \"" + s1 + "\" este mai mic decat sirul"
        + " \"" + s2 + "\"");
}
else
{
    if (comparare > 0)
    {
        System.out.println("Sirul \"" + s2 + "\" este mai mare decat"
            + " sirul \"" + s2 + "\"");
    }
    else
    {
        System.out.println("Sirul \"" + s1 + "\" este egal cu sirul \""
            + s2 + "\"");
    }
}

if (s1 == s2)
    System.out.println("Sirurile sunt egale!");
else
    System.out.println("Sirurile sunt diferite!");

System.out.println(s1 == s2);

if (ss1.equals(ss2))
{
    System.out.println("Sirul \"" + ss1 + "\" este egal cu sirul \""
        + ss2 + "\"");
}
else
{
    System.out.println("Sirul \"" + ss1 + "\" nu este egal cu sirul"
        + " \"" + ss2 + "\"");
}

// concatenarea sirurilor
// sirul s1 este format din concatenarea valorii sale initiale cu
// cele ale sirului s2
s1 += s2;

System.out.println(s1);
```



```
String s3 = "opt zeci si trei";

// concatenarea a doua caractere la un sir
System.out.println(s3 + " este " + 8 + 3);

String s4 = "unsprazece";

// concatenarea rezultatului unei operatii aritmetice
System.out.println(8 + 3 + " este " + s4);

// utilizarea metodei valueOf(int i)pentru a reprezenta argumentul
// int ca si string
String myString1 = "opt zeci si trei este: " + String.valueOf(8)
    + String.valueOf(3);
String myString2 = String.valueOf(8) + String.valueOf(3)
    + " este: opt zeci si trei";
System.out.println(myString1);
System.out.println(myString2);

// declararea unui tablou de tip caracter
char[] carray = new char[30];

// utilizarea unui ciclu for pentru afisarea fiecarui caracter in
// parte a sirului
for (int index = 0; index < s2.length(); index++)
{
    // utilizarea metodei charAt(index)pentru returnarea caracterului
    // de la indexul specificat
    carray[index] = s2.charAt(index);
    System.out.print(carray[index]);
}
System.out.println("\n");

// utilizarea metodei replace pentru inlocuirea intr-un sir a unui
// caracter cu un alt caracter(carcterul 'H' este inlocuit
// cu caracterul 'X')
s1 = s1.replace('H', 'X');
System.out.println(s1);

// utilizarea metodei startsWith(String prefix)pentru a verifica
// daca sirul incepe cu prefixul specificat
if (ss2.startsWith("He"))
    System.out.println("Sirul \"" + ss2 + "\" incepe cu prefixul He");

// utilizarea metodei startsWith(String prefix,int toffset) pentru
// a verifica daca in subsirul ce incepe de la indexul specificat
// (int/ toffset)din sir este precedat de sufixul respectiv
if (ss2.startsWith("pp", 2))
{
    System.out.println("Sirul \"" + ss2
        + "\" include subsirul \"pp\" la pozitia 2");
}
```

```
// utilizarea metodei endsWith(String suffix)pentru a verifica daca
// sirul se termina cu sufixul specificat
if (ss2.endsWith("y!"))
{
    System.out.println("Sirul \"" + ss2
                        + "\" se termina cu subsirul y!");
}

// declararea unui sir de caractere (valoarea initiala este
// sirul vid "")
String m1 = new String();

// declararea si initializarea unui sir de caractere
String m2 = new String("Acesta este un sir");

// declararea celui de-al treilea sir si initializarea sa cu
// valoarea celui de-al doilea sir. aceasta valoare se va mentine
// chiar daca sirul s2 va suferi modificari
String m3 = m2;

// afisarea celor trei siruri
System.out.println(m1);
System.out.println(m2);
System.out.println(m3);

// cocatenarea sirului s2 cu sirul "modificat"
m2 += "modificat";

// afisarea noilor valori
System.out.println(m2);
System.out.println(m3);

char[] textchar = {'A', 'c', 'e', 's', 't', 'a', ' ', 'e', 's', 't',
                  'e', ' ', 'u', 'n', ' ', 's', 'i', 'r', '.' };

// metoda copyValueOf(char[]data) returneaza un sir format din
// caracterele specificate de vector (char[]textchar)
String n1 = String.copyValueOf(textchar);
System.out.print(n1);

String text = "Un sir declarat";
System.out.println("Sirul este:" + "\"" + text + "\"");

// utilizarea metodei substring(int beginIndex, int endIndex) pentru
// crea un un subsir dintr-un sir pornind de la pozitia beginIndex
// la pozitia endIndex a sirului
String subsir = text.substring(7, 15);
System.out.println("Subsirul de la pozitia 7 la 15: \"" + subsir
                  + "\"");
```

```
String propozitie = "Problema principala este de a gasi grupul de"
                    + " caractere din sirul curent si, cateodata, a"
                    + " verifica daca caracterele exista in sir sau"
                    + " nu. Atentie! grupul de caractere poate"
                    + " forma un cuvant din sir sau poate fi o"
                    + " parte a unui cuvant din sir.";

int siCount = 0;
int deCount = 0;
int index;

System.out.println("String-ul este:");
System.out.println(propozitie);

// gasirea lui "si"
index = propozitie.indexOf("si"); // gaseste primul "si"
while (index >= 0)
{
    ++siCount;
    index += "si".length(); // sare la pozitia dupa ultimului "si"
    index = propozitie.indexOf("si", index); // cauta "si" de la index
                                           // la sfarsitul textului
}

// gasirea lui "de"
index = propozitie.lastIndexOf("de"); // gaseste ultimul de
while (index >= 0)
{
    ++deCount;
    index -= "de".length(); // sare la pozitia de dinaintea
                           // ultimului "de"
    index = propozitie.lastIndexOf("de", --index);
}

System.out.println("Textul contine " + siCount + " \"si\"\\n"
                  + "Textul contine " + deCount + " \"de\"");
}
```

8.2. Clasa StringBuffer

Lista tuturor proprietăților și metodelor ale clasei `StringBuffer` se găsește la adresa:

<http://java.sun.com/javase/6/docs/api/java/lang/StringBuffer.html> .

ExempluStringBuffer.java

```
public class ExempluStringBuffer
{
    public static void main(String[] args)
    {
        // StringBuffer este un String care poate fi modificat fara crearea
        // unei noi instante a clasei
        StringBuffer sb1 = new StringBuffer(); // declararea unui
                                                // StringBuffer vid ce are
                                                // capacitatea initiala
                                                // de 16 caractere

        // crearea unei instante ce are ca valoare initiaial textul
        // "Acesta este un StringBuffer"
        StringBuffer sb2 = new StringBuffer("Acesta este un StringBuffer");

        // declararea unui StringBuffer fara caractere ce are
        // capaciatatea initiala de 20 de caractere
        StringBuffer sb3 = new StringBuffer(20);

        sb1 = new StringBuffer("Eu am mere");
        sb3 = new StringBuffer("Eu am pere");

        System.out.println(sb1);
        System.out.println(sb2);
        System.out.println(sb3);

        // utilizarea metodei insert(int offset,char c) pentru a insera
        // un caracter c la pozitia specificata offset
        sb1.insert(6, '3');
        System.out.println("Noul sir sb1 este: " + sb1);
        sb1.insert(6, '3');
        System.out.println("Noul sir sb1 este: " + sb1);

        // transformarea SringBuffer intr-un String
        String myString1 = sb2.toString();

        // declararea unui string si initializarea sa cu valoarea
        // StringBufferului declarat anterior
        String myString2 = new String(sb2);
        System.out.println("Sirul din StringBuffer este: " + myString1);
        System.out.println("Sirul din StringBuffer este: " + myString2);

        // declararea unui String Buffer
        StringBuffer sir = new StringBuffer("12345678901234567890");
```

```
// afisarea lui
System.out.println("Sirul este: " + sir);

// folosirea metode replace(int start,int end,String str)
// pentru inlocuirea in sirul specificat a unui subsir cu un alt
// subsir(str)incepand de la pozitia start pana la pozitia end
sir.replace(2, 15, "abcdefghijkl");

System.out.println("Noul sir este: " + sir);

// utilizarea metodei reverse() pentru inversarea sirului specificat
sir.reverse();

System.out.println("Noul sir inversat este: " + sir);
}
}
```


LABORATORUL 7

9. Clase și programare orientată obiect

Utilizarea *claselor* în scrierea de programe se încadrează în paradigma de *programare orientată obiect* (POO – OOP – *Object Oriented Programming*), care presupune crearea unor tipuri de date complexe care se aseamănă, în multe aspecte, cu obiecte reale. Astfel, aceste tipuri conțin atât datele utile, cât și metodele de operare asupra acestora.

O clasă va conține proprietăți (atribute) și metode (comportamente), și va descrie, într-un mod abstract, natura obiectului ce se dorește a fi transpus în program.

9.1. Termeni utilizați în programarea orientată obiect

O **clasă** este un tip de date ce **încapsulează** atât datele utile cât și modul de operare asupra acestora.

Instanțierea este procesul prin care se obține un **obiect** care are propria sa locație de memorie și propriul cod executabil. Un **obiect** este o **instanță** a unei clase, adică este construit pe baza declarațiilor din clasă. Utilizând o analogie, o clasă este un plan al unei case (practic un desen), iar obiectul, sau instanța, este casa construită.

Moștenirea este mecanismul prin care se pot obține noi clase din clase deja existente. Noile clase vor moșteni caracteristici (proprietăți și metode) ale clasei de bază (*superclasei*).

Operația prin care se obține o clasă nouă din una existentă poartă denumirea de **derivare**.

Polimorfismul reprezintă mecanismul care permite accesarea proprietăților și metodelor unui obiect de un anumit tip prin intermediul unor tipuri de date din care acesta a fost obținut prin derivare.

9.2. Definiția unei clase

Cea mai simplă definiție a unei clase, în Java, este formată din trei componente:

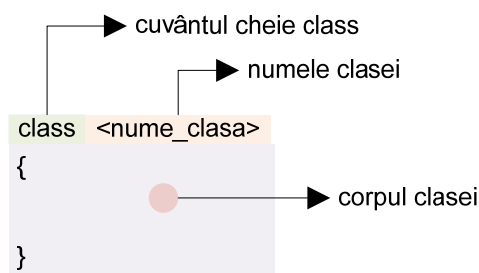


Fig. 15. Componentele esențiale ale unei clase



1. Prin convenție, numele claselor Java trebuie să înceapă cu literă mare. Dacă numele conține mai mulți atomi lexicali, fiecare dintre ei trebuie să înceapă cu literă mare. De exemplu:

```
class ClasaAceastaRespectaConventiaDeNotare
{
}

```

2. Este foarte indicat ca fiecare clasă să fie scrisă în propriul fișier. Numele fișierului, în mod **obligatoriu**, trebuie să aibă exact același nume cu cel al clasei conținute.

Corpul clasei conține declarațiile de *proprietăți* și *metode*.

9.2.1. Modificatori de acces

Programarea orientată obiect impune existența unui mecanism de control asupra accesului la clase și proprietățile și metodele unui obiect. În Java, acest mecanism este implementat prin **modificatori de acces**.

Există 4 modificatori de acces: `private`, `protected`, `public` și *implicit* (atunci când nici unul din cei anteriori nu sunt menționați). De asemenea, se pot evidenția 4 domenii de vizibilitate a metodelor și proprietăților: clasa care conține metodele și proprietățile, clasele derivate din aceasta (subclase), pachetul în care se află clasa și lumea exterioară (zona publică).

Tabelul următor prezintă semnificația modificatorilor de acces din punct de vedere a vizibilității proprietăților și metodelor.

Specificator	Clasă	Subclasă	Pachet	Zona publică
<code>private</code>	X			
<code>protected</code>	X	X	X	
<code>public</code>	X	X	X	X
<i>implicit</i>	X		X	



Modificatorii `private` și `protected` nu pot fi aplicați claselor.

9.2.2. Alți modificatori

9.2.2.1. Modificatorul `final`

Pentru declarațiile de proprietăți are semnificația faptului că acea proprietate este constantă (nu poate fi modificată).

Metodele purtând acest modificador nu mai pot fi *supradefinite* în subclase.

În cazul claselor, `final` precizează că acea clasă nu mai poate fi derivată (nu are moștenitori).

9.2.2.2. Modificadorul `static`

Acest modificador poate fi utilizat numai pentru proprietăți și metode.

Proprietățile și metodele declarate cu modificadorul `static` poartă denumire de proprietăți, respectiv metode **de clasă**. Dacă o proprietate sau o metodă nu conține modificadorul `static` în declarație, atunci ea se numește proprietate, respectiv metodă **de instanță**.

9.2.2.3. Modificadorul `synchronized`

Acest modificador este utilizat în aplicații care rulează mai multe fire de execuție în paralel.

Modificadorul asigură exclusivitate accesului la execuție, astfel încât să nu apară erori de violare a accesului.

9.2.3. Proprietăți

O proprietate este un element de date asociat clasei.

Definirea / declarare proprietăților se face la fel ca în cazul variabilelor. Proprietățile se declară în interiorul clasei, dar în afara oricărei metode. Declarațiile pot fi precedate de modificali de acces `private`, `protected` sau `public` și pot fi inițializate în momentul declarării.

Proprietățile pot fi de *instanță*, prin urmare pot fi accesate doar prin numele variabilei care face referire la obiect, și de *clasă*, care sunt comune pentru toate instanțele și pot fi accesate direct prin numele clasei.



Modificările făcute asupra unei proprietăți de instanță a unui obiect nu are efect asupra proprietăților de instanță a unui alt obiect.

9.2.4. Metode

O metodă este o parte a unui program care conține logica pentru executarea unei sarcini. De fiecare dată când trebuie executată sarcina respectivă trebuie apelată metoda.

Principalul motiv pentru folosirea metodelor în programe este reducerea numărului de instrucțiuni duplicate în program. De asemenea, folosirea metodelor duce la simplificarea întreținerii programului. Modificările sunt făcute într-un singur loc – în codul metodei.

În mod obligatoriu metodele trebuie declarate în interiorul claselor.

Declararea metodelor:

Format general:

```
<modificatori> <tip_returnat> <id_metoda>(<listă_parametri>)  
{  
    <declaratii_proprietati_si_metode>  
}
```

Tipul returnat poate fi un tip de date primitiv sau o referință către un obiect. Dacă metoda nu returnează date se utilizează cuvântul cheie `void` pentru tipul returnat.



A nu se confunda cuvântul cheie `void` cu `0`. Zero este o valoare, pe când `void` semnifică absența unei valori.

Numele metodei este date de programator. Numele trebuie să fie sugestiv pentru ceea ce face metoda și să respecte convenția de denumire Java.

Lista de parametri este formată din datele de care metoda are nevoie pentru a-și duce la îndeplinire sarcina. Lista este formată prin specificarea tipului de date și a numelui pentru fiecare element de date transmis metodei. Tipul de date al parametrului este același cu tipul de date al unei variabile. Acesta spune mașinii virtuale Java câtă memorie să rezerve pentru parametru și ce fel de date vor fi stocate. Numele parametrului este similar cu numele unei variabile; este o etichetă pentru o adresă de memorie care conține datele transmise de instrucțiunea care apelează metoda. În cadrul unei metode, numele parametrului se folosește la fel ca o variabilă.

Numele unui parametru trebuie să reflecte natura datelor stocate de acesta. Și acest nume trebuie să respecte convenția de denumire Java.

Lista de parametri poate conține oricâți parametri, separați prin virgule, iar numele lor să fie unice.

Lista de parametri este opțională. Numai o metodă care are nevoie de date transmise de instrucțiunea care o apelează trebuie să aibă o listă de parametri. Totuși, definiția metodei trebuie să includă parantezele, chiar dacă nu există o listă de parametri.

Transferul parametrilor se face în Java prin valoare.

Modificatorii pot fi:

- un specificator de acces:
 - `public`
 - `private`
 - `protected`
- unul din cuvintele rezervate:
 - `static` (metoda este de clasă, nu de instanță)
 - `abstract` (metodă abstractă ce trebuie să facă parte dintr-o clasă abstractă; această metodă nu are implementare)
 - `final` (metoda nu poate fi supradefinită)
 - `native`
 - `synchronized`

Numele metodei, împreună cu tipul parametrilor pe care aceasta îi are formează **semnătura** sa.

Semnătura trebuie să fie unică în cadrul aceleiași clase. Pot fi definite metode cu același nume, dar având semnături diferite, mecanismul folosit numindu-se supraîncărcare (*overloading*).

9.2.5. Constructori

Un constructor este o metodă specială care este apelată automat la crearea unei instanțe a unei clase.

Din punct de vedere tehnic, constructorul este apelat înainte ca operatorul `new` să-și termine execuția. Constructorul se folosește atât pentru inițializarea proprietăților de instanță cât și pentru executarea unor operații la crearea unui obiect.

Un constructor este definit ca orice metodă însă:

- numele constructorului trebuie să fie același cu numele clasei;
- constructorul nu trebuie să conțină instrucțiunea `return`;

Se pot defini mai mulți constructori pentru o clasă, fiecare având semnături diferite (supraîncărcarea constructorului).

Dacă programatorul nu scrie nici un constructor, Java utilizează un constructor implicit (creat automat de Java). Constructorul implicit nu are listă de parametri. Un constructor se supraîncarcă prin definirea unui constructor care are o listă de parametri. Acesta se numește *constructor parametrizat* (cu parametri).



Dacă într-o clasă este definit un constructor, indiferent de numărul de argumente pe care le acceptă, Java nu mai creează constructorul implicit. Prin urmare, dacă se dorește un constructor fără argumente după ce s-a definit unul cu argumente trebuie creat constructorul fără argumente în mod explicit.

Constructorul unei clase poate avea modificator de acces.

- Dacă modificatorul de acces este `public`, atunci se poate crea o instanță a clasei respective în orice altă clasă.
- Dacă modificatorul de acces este `protected`, se pot crea instanțe ale clasei respective doar în clasele care moștenesc clasa respectivă.
- Dacă modificatorul de acces este `private`, nu se pot crea instanțe ale clasei în alte clase.
- Dacă modificatorul de acces este implicit, atunci nu se pot crea instanțe ale clasei decât în clasele din același pachet în care este clasa pentru care se creează instanța.

9.2.6. Declararea unei instanțe a unei clase

La crearea unui obiect (instanțe) Java alocă memorie pentru toate proprietățile de instanță, apoi apelează constructorul.

```
myClass x = new myClass();
```

Instrucțiunea de mai sus implică trei operații:

- Operatorul `new` cere mașinii Java să rezerve un bloc de memorie pentru instanță. Acest bloc de memorie este suficient de mare pentru stocarea proprietăților de instanță. Operatorul `new` returnează adresa blocului de memorie rezervat.
- Este declarată o variabilă referință a unei instanțe a clasei `myClass`, numită `x`. Referința este declarată folosind numele clasei.
- A treia operație constă în a atribui referinței prima adresă din blocul de memorie al instanței. Referința (`x`) este folosită apoi în program ori de câte ori se face referire la instanța creată.

În exemplul anterior s-a declarat o referință și o instanță a clasei într-o singură instrucțiune. Cele două declarații pot fi separate în două instrucțiuni:

```
myClass x;  
x=new myClass();
```

Referințelor declarate finale nu le pot fi atribuite alte valori.

9.2.7. Accesul la membrii unei clase

După ce este declarată o instanță a unei clase, accesul la membrii clasei se face prin intermediul unei referințe a clasei (variabila).

Pentru a obține accesul la un membru al unei clase, se folosește numele referinței, urmat de operatorul punct și de numele membrului.

Exemplul următor definește clasa `myClass`, care are 3 membri. Aceștia sunt o variabilă de instanță numită `student`, un constructor și o metodă numită `print()`. Variabila de instanță este inițializată cu numărul de identificare al unui student, de către constructor, la declararea instanței. Instanța este apoi folosită pentru a apela metoda `print()`, care afișează numele de identificare al studentului.

MyClass.java

```
class MyClass
{
    int student;

    MyClass()
    {
        student = 1234;
    }

    public void print()
    {
        System.out.println("'Neata, student: " + student);
    }
}
```

Demonstratie.java

```
class Demonstratie
{
    public static void main(String args[])
    {
        //declararea unei instante a clasei MyClass
        MyClass x = new MyClass();

        //apelarea functiei print() folosind instant clasei MyClasss
        x.print();
    }
}
```

9.2.8. Supraîncărcarea metodelor

O metodă se identifică după semnătura sa, care reprezintă combinația între numele metodei și lista tipurilor parametrilor. Acesta înseamnă că două metode pot avea același nume, dar parametri diferiți.

O clasă poate avea mai multe metode cu același nume, dar semnătura fiecărei metode să fie unică în definiția clasei.

Atunci când într-o clasă există două metode cu același nume se spune că a doua metodă supraîncarcă prima metodă.



1. Două metode ale unei clase nu pot avea aceeași semnătură.
2. Metode a două clase diferite pot avea aceeași semnătură.

Această idee este ilustrată în următorul exemplu, în care există două metode numite `print()`. Prima versiune nu are o listă de argumente și afișează un mesaj de întâmpinare prestabilit de fiecare dată când este apelată. A doua versiune conține un text care va fi încorporat în mesajul afișat. Astfel se poate opta între afișarea unui mesaj de întâmpinare generic sau un mesaj personalizat.

MyClass.java

```
class MyClass
{
    //functia print() fara parametrii
    public void print()
    {
        System.out.println("'Neata!");
    }

    //functia print cu parametru de tip String
    public void print(String str)
    {
        System.out.println("'Neata " + str);
    }
}
```

Demonstratie.java

```
class Demonstratie
{
    public static void main(String args[])
    {
        //declararea unei instante a clasei MyClass
        MyClass x = new myClass();

        /* apelarea functiei print(), functie fara parametrii,
           folosind instant clasei MyClassss */
        x.print();

        /* apelarea functiei print(), functie cu parametru de tip String,
           folosind instant clasei myClassss */
        x.print("Student");
    }
}
```

9.2.9. Cuvântul cheie *this*

O metodă are acces, în mod implicit, la variabilele de instanță și la celelalte metode ale clasei în care este definită.

Există situații în care se declară într-o metodă o variabilă care are același nume ca și o proprietate de instanță. O variabilă declarată în interiorul unei metode se numește *variabilă locală*. Dacă o instrucțiune dintr-o metodă referă un nume de variabilă, Java folosește variabila locală în locul variabilei de instanță cu același nume.

Se poate cere mașinii virtuale Java să folosească o variabilă de instanță în locul variabilei locale cu același nume, folosind cuvântul cheie `this`. Cuvântul cheie `this` permite referirea la obiectul curent al clasei.

MyClass.java

```
class MyClass
{
    //functia print() fara parametrii
    public void print()
    {
        System.out.println("'Neata!");
    }

    public void print()
    {
        int student = 12;

        // afiseaza variabila locala
        System.out.println("'Neata ,student: " + student);

        // se afiseaza variabila de instanta utilizand cuvantul this
        System.out.println("'Neata ,student: " + this.student);
    }
}
```

Demonstratie.java

```
class Demonstratie
{
    public static void main(String args[])
    {
        MyClass x = new myClass();

        x.print();
    }
}
```

Următorul exemplu prezintă modul cum se poate utiliza cuvântul cheie `this` pentru apelul unui constructor din interiorul altui constructor.



Dacă se dorește apelarea unui constructor în mod explicit, această instrucțiune trebuie să fie prima în constructor.

MyClass.java

```
class MyClass
{
    int studentID;
    String studentNume;
    double media;

    MyClass(int x, String y, double z)
    {
        studentID = x;
        studentNume = y;
        media = z;
    }

    MyClass(int x, String y)
    {
        // Apelam constructorul cu 3 argumente
        this(x, y, 0);
    }

    MyClass()
    {
        // Apelam constructorul cu 2 argumente
        this(0, "");
    }
}
```

9.3. Crearea și utilizare claselor (1)

Student.java

```
public class Student
{
    private int student; // variabila de instanta

    public Student()
    {
        // initializarea variabilei de instanta prin constructor
        student = 1234;
    }
}
```



```
public void print()
{
    int student = 12;

    // afiseaza variabila locala
    System.out.println("Student: " + student);

    // se afiseaza variabila de instanta utilizand cuvantul this
    System.out.println("Student: " + this.student);
}

public void print(String str)
{
    System.out.println("Student: " + str);
}
}
```

Clase.java

```
public class Clase
{
    public static void main(String[] args)
    {
        // declararea unor instante ale unor clase
        Student x, y, temp; // declararea referintelor

        x = new Student(); // atribuie instanta referintei
        /*
         * operatorul new alocă dinamic memorie pentru instanta clasei (java
         * rezerva memorie în timpul rularii, nu în timpul compilării)
         */

        y = new Student();

        // accesul la membrii aceleiasi instante
        temp = x;
        temp = y;

        /*
         * apelarea metodei print() (metoda a clasei Student) folosind
         * instanta clasei Student
         */

        temp.print();

        // metoda ce suprîncarca prima metoda print();
        temp.print("Ana");
    }
}
```

9.4. Crearea și utilizarea claselor (2)

Angajat.java

```
class Angajat
{
    /*
     * definirea celor 3 variabile de tip private adica sunt accesibile
     * doar membrilor clasei in care sunt declarate.
     */

    private String Nume;
    private double salariu;
    private Date aniversare;

    /*
     * definirea constructorului clasei (parametrii sunt cei definiti la
     * initializarea obiectelor)
     */
    public Angajat(String n, double s, int an, int luna, int zi)
    {
        Nume = n;
        salariu = s;
        /*
         * crearea unui obiect de tip GregorianCalendar pentru afisarea
         * timpului (datei)
         */
        GregorianCalendar calendar = new GregorianCalendar(an, luna-1, zi);
        // GregorianCalendar utilizeaza 0 pentru Ianuarie
        aniversare = calendar.getTime();
    }

    // definirea metodei care preia parametrul nume
    public String getNume()
    {
        return Nume;
    }

    // definirea metodei care preia parametrul salariu
    public double getSalariu()
    {
        return salariu;
    }

    // definirea parametrului care preia data
    public Date getaniversare()
    {
        return aniversare;
    }
}
```

```
// definirea metodei pentru calculul salariului marit cu 5 %
public double Salariu(double Procent)
{
    return salariu += salariu * Procent / 100;
}
}
```

Clase.java

```
public class Clase
{
    public static void main(String[] args)
    {
        // declararea unei instante a unei clase
        Angajat[] obiect = new Angajat[3];

        /*
         * initializarea celor trei obiecte a clasei Angajat (initializarea
         * tine cont de constructorul clasei Angajat)
         */
        obiect[0] = new Angajat("Popescu Aurel", 750, 1987, 12, 15);
        obiect[1] = new Angajat("Pintilie Mihai", 1000, 1989, 10, 1);
        obiect[2] = new Angajat("Anghel Laurentiu", 1200, 1990, 3, 15);

        // afisarea informatiilor despre toate obiectele Angajat
        for (Angajat i : obiect)
        {
            System.out.println("Nume: " + i.getNume() + "\t|salariu: "
                               + i.Salariu(5) + "\t|aniversare="
                               + i.getaniversare());
        }
    }
}
```

9.5. Clase interne

Definiția unei clase poate conține definiția unei alte clase. Acestea se numesc *clase imbricate*. Există două tipuri de clase imbricate: *statice* și *nestatice*.

O clasă imbricată statică este definită cu modificator de acces static și nu are acces direct la membrii clasei în cadrul căreia este definită. Trebuie ca, mai întâi, să se declare o instanță a clasei externe și apoi să se folosească instanța creată pentru accesul la membrii clasei externe.

O clasă imbricată nestatică este denumită clasă internă. Ea este definită în interiorul unei alte clase denumită clasă externă. Clasa internă are acces direct la membrii clasei externe fără să se declare o instanță a clasei externe.

Clasa internă poate conține proprietăți de instanță și metode membre. Totuși, numai clasa internă are acces direct la acești membri. Clasa externă poate avea acces la membrii unei clase interne declarând o instanță a clasei interne și folosind această instanță pentru accesul la membrii clasei interne. De asemenea, clasa internă nu este accesibilă din exteriorul clasei externe.



O instanță a unei clase interne poate fi declarată numai în interiorul clasei externe.

Figura.java

```
import java.util.Scanner;

class Figura
{
    // definirea unei metode membre a clasei
    void metoda()
    {
        int alegere;

        // definirea unui meniu pentru alegerea unei optiuni
        do
        {
            System.out.println("Alegeti o figura geometrica");
            System.out.println("cub: 1\tcon: 2\tcilindru: 3");
            System.out.println("Pentru terminarea programului apasati" +
                               + " tasta 4");

            Scanner figura = new Scanner(System.in);
            alegere = figura.nextInt();

            if (alegere == 1)
            {
                // definirea unui obiect calcul de tip Cub
                Cub calcul = new Cub();
                // apelarea metodei membre a clasei Cub
                calcul.calculare();
            }

            if (alegere == 2)
            {
                Con calcul = new Con();
                calcul.calculare();
            }
        }
    }
}
```

```
        if (alegere == 3)
        {
            Cilindru calcul = new Cilindru();
            calcul.calculare();
        }
    } while (alegere != 4);
}

// definirea clasei Cub ce este clasa inbricata in clasa Figura
class Cub
{
    void calculare()
    {
        // float latura;
        System.out.print("Introdu latura cubului: ");
        Scanner Cub = new Scanner(System.in);
        float latura = Cub.nextFloat();

        System.out.println("Aria bazei cubului este: "
            + Math.pow(latura, 2));
        System.out.println("Aria laterala a cubului este: "
            + 4 * Math.pow(latura, 2));
        System.out.println("Aria totala a cubului este: "
            + 6 * Math.pow(latura, 2));
        System.out.println("Volumul cubului este: "
            + Math.pow(latura, 3));
        System.out.println("Diagonala bazei este: "
            + latura * Math.sqrt(2));
        System.out.println("Diagonala cubului este: "
            + latura * Math.sqrt(3) + "\n");
    }
}

// definirea clasei Con ce este clasa imbricata in clasa Figura
class Con
{
    void calculare()
    {
        float raza;
        float generatoarea;
        float inaltimea;

        System.out.print("Introdu raza conului: ");
        Scanner con = new Scanner(System.in);
        raza = con.nextFloat();

        System.out.print("Introdu generatoarea conului: ");
        generatoarea = con.nextFloat();

        System.out.print("Introdu inaltimea conului: ");
        inaltimea = con.nextFloat();
    }
}
```

```

        System.out.println("Aria bazei conului este: "
            + Math.PI * Math.pow(raza, 2));
        System.out.println("Aria laterala a conului este: "
            + Math.PI * raza * generatoarea);
        System.out.println("Aria totala a cubului este: "
            + Math.PI * raza * (raza + generatoarea));
        System.out.println("Volumul conului este: "
            + Math.PI * Math.pow(raza, 2) * inaltimea / 3);
    }
}

// definirea clasei Cilindru ce este clasa inbricata in clasa Figura
class Cilindru
{
    void calculare()
    {
        float raza;
        float generatoarea;
        float inaltimea;

        System.out.print("Introdu raza cilindrului: ");
        Scanner cilindru = new Scanner(System.in);
        raza = cilindru.nextFloat();

        System.out.print("Introdu generatoarea cilindrului: ");
        generatoarea = cilindru.nextFloat();

        System.out.print("Introdu inaltimea cilindrului: ");
        inaltimea = cilindru.nextFloat();

        System.out.println("Aria bazei cilindrului este: "
            + Math.PI * Math.pow(raza, 2));
        System.out.println("Aria laterala a cilindrului este: "
            + 2 * Math.PI * raza * generatoarea);
        System.out.println("Aria totala a cilindrului este: "
            + Math.PI * raza * (raza + 2 * generatoarea));
        System.out.println("Volumul conului este: "
            + Math.PI * Math.pow(raza, 2) * inaltimea);
    }
}
}

```

Imbricate.java

```

import java.util.Scanner;

public class Imbricate
{
    public static void main(String[] args)
    {

```

```
// definirea unui obiect de tip Figura
Figura extern = new Figura();

/*
 * apelarea metodei metoda() ce apartine clasei Figura prin
 * intermediul obiectului extern
 */
extern.metoda();
}
}
```


LABORATORUL 8

10. Moștenirea

Proprietățile de instanță și metodele comune mai multor obiecte se declară într-o singură clasă (ex. clasa `A`). Pentru obiectele înrudite se poate declara o nouă clasă (ex. clasa `B`) ce conține atât membrii clasei `A` cât și membrii particulari clasei `B`.

Este inutil să se declare și definească aceeași membri atât timp cât există o soluție mai bună și anume **moștenirea**.

O clasă (`B` – copil) care moștenește o altă clasă (`A` – părinte) are acces la toți membrii clasei părinte, ca și cum aceștia ar fi fost declarați și definiți pentru clasa copil.

Regula pentru a stabili dacă moștenirea este potrivită pentru un anumit program se numește “este-un” și impune ca un obiect să aibă o relație cu un alt obiect înainte de a-l moșteni.

Conform regulii “este-un”, punem următoarea întrebare: “obiectul `A` este un obiect `B`?”. Dacă răspunsul este afirmativ, obiectul `A` poate să moștenească obiectul `B`. Din punct de vedere tehnic nu există nimic care să împiedice o clasă să moștenească o altă clasă, dar orice relație de moștenire ar trebui să respecte regula “este-un”.

De exemplu, trăsătura comună între un student masterand și un student în primii ani este aceea că ambii sunt studenți. Cu alte cuvinte, ambii au atribute și comportamente comune tuturor studenților. Totuși, un student masterand are atribute și comportamente diferite de ale altor studenți.

Ar trebui definite trei clase:

- clasa `Student` care definește atributele și comportamentele diferite tuturor studenților;
- clasa `StudentMasterand` pentru studenții masteranzi;
- clasa `StudentNelicentiat` pentru studenții din primii ani de facultate.

Aplicând regula “este-un” obiectului care reprezintă studentul masterand, punem următoarea întrebare: “Studentul masterand este un student?”. Răspunsul este afirmativ deci clasa `StudentMasterand` poate să moștenească clasa `Student`.

Aplicând aceeași regulă clasei `StudentNelicentiat` (“Studentul nelicențiat este un student?”) observăm că `StudentNelicentiat` poate moșteni clasa `Student`.

O clasă moștenește o altă clasă folosind cuvântul cheie `extends` în definiția clasei.

Clasa moștenită este numită **superclasa** (clasa părinte), iar clasa care moștenește se numește **subclasa** (clasa copil).

Moștenirea:

```
public class A
{

}

class B extends A
{

}
```

10.1. Accesul la membrii unei clase moștenite

Accesul la membrii unei clase se face în funcție de specificatorii de acces folosiți.

Un membru declarat ca `public` este accesibil pentru orice alt membru, din orice altă clasă. Un membru declarat ca `private` este accesibil pentru orice metodă membră din propria clasă. Un membru declarat ca `protected` este accesibil pentru orice metodă membră din propria clasă și pentru metodele membre din subclasele care moștenesc propria clasă.

Student.java

```
class Student
{
    private int studentID;

    public Student()
    {
        studentID = 1234;
    }

    protected void afisare()
    {
        System.out.println("studentul nr:" + studentID);
    }
}
```

StudentMasterand.java

```
class StudentMasterand extends Student
{
}
```

Demonstratie.java

```
class Demonstratie
{
    public static void main(String args[])
    {
        StudentMasterand ob = new StudentMasterand();
        ob.afisare();
    }
}
```

În exemplul de mai sus, clasa `Student`, pe lângă constructor, are doi membri: o proprietate de instanță numită `studentID` și o metodă membră numită `afisare()` ce afișează pe ecran valoarea proprietății `studentID`. Constructorul este folosit pentru inițializarea proprietății de instanță.

Clasa `StudentMasterand` nu are nici un membru, dar are acces la membrii publici și protejați ai clasei `Student`. Acesta înseamnă că o instanță a clasei `StudentMasterand` poate folosi metoda `afisare()` din clasa `Student`, ca și cum metoda ar fi membră a clasei `StudentMasterand`.

Această idee este ilustrată de instrucțiunile din metoda `main()` a aplicației. Prima instrucțiune declară o instanță a clasei `StudentMasterand`. A doua instrucțiune folosește instanța creată pentru a apela metoda `afisare()`.

La prima vedere metoda `afisare()` s-ar putea crede ca este membră clasei `StudentMasterand`, deși este, de fapt, membră a clasei `Student`.

Moștenirea este o relație *unidirecțională*. Cu alte cuvinte, o subclasă are acces la membrii publici și protejați ai superclasei, dar superclasa nu are acces la membrii publici și protejați ai subclaselor.

10.2. Apelarea constructorilor

Atunci când o subclasă moștenește o superclasă, la instanțierea subclasei sunt implicați cel puțin doi constructori.

Atunci când se declara o instanță a subclasei, Java apelează atât constructorul subclasei cât și cel al superclasei.

10.2.1. Folosirea cuvântului cheie *super*

Cuvântul cheie `super` este utilizat pentru referirea la membrii unei superclase în metodele unei subclase.

Cuvântul cheie `super` se folosește la fel cum se folosește o referință a unei instanțe pentru a referi un membru al unei clase din program.

10.2.1.1. Folosirea cuvântului cheie *super***Student.java**

```
class Student
{
    private int studentID;

    Student()
    {
        studentID = 1234;
    }

    Student(int ID)
    {
        studentID = ID;
    }

    protected void afisare()
    {
        System.out.println("Studentul nr: " + studentID);
    }
}
```

StudentMasterand.java

```
class StudentMasterand extends Student
{
    StudentiMasteranzi()
    {
        super(222);
    }

    public void afisare()
    {
        super.afisare();
    }
}
```

Demonstratie.java

```
public class Demonstratie
{
    public static void main(String args[])
    {
        StudentMasterand ob = new StudentMasterand();
        ob.afisare();
    }
}
```

În exemplul de mai sus, în clasa `Student` sunt definiți doi constructori ambii fiind folosiți pentru inițializarea proprietății de instanță. Primul constructor folosește pentru inițializare o valoare prestabilită, iar al doilea folosește pentru inițializare o valoare primită ca parametru. Se spune că al doilea constructor *supraîncarcă* primul constructor.

Constructorul clasei `StudentMasterand` conține o singură instrucțiune ce folosește cuvântul cheie `super` pentru a apela constructorul supraîncărcat al clasei `Student`. Metoda `afisare()` din clasa `StudentMasterand` conține o singură instrucțiune ce folosește cuvântul cheie `super` pentru a apela metoda membra `afisare()` a clasei `Student`.

10.2.2. Moștenirea pe mai multe niveluri

În programele Java moștenirea poate fi pe mai multe niveluri. Acesta permite unei clase să moștenească membri din mai multe superclase. Se folosește moștenirea pe mai multe niveluri pentru a grupa obiecte mai simple în obiecte mai complexe.

Unele limbaje de programare, cum ar fi C++, permit folosirea mai multor tipuri de moștenire multiplă. În Java există un singur tip de moștenire multiplă, numită **moștenire pe mai multe niveluri** (*multilevel inheritance*). Moștenirea pe mai multe niveluri permite unei subclase să moștenească direct o singură superclasă. Totuși, superclasa respectivă poate fi, la rândul său, o subclasă a unei superclase.

10.2.2.1. Moștenirea pe mai multe niveluri

Persoana.java

```
class Persoana
{
    private String nume;

    Persoana()
    {
        nume = "Popescu";
    }
}
```

```
protected void metodaNume()  
{  
    System.out.println("Nume student: " + nume);  
  
}  
}
```

Student.java

```
//definirea unei clase ce mosteneste superclasa Persoane  
  
class Student extends Persoana  
{  
    private int student;  
  
    public Student()  
    {  
        student = 12345;  
    }  
  
    protected void metodaStudent()  
    {  
        System.out.println("ID Student: " + student);  
    }  
}
```

StudentMasterand.java

```
// definirea unei clase ce mosteneste clasa Studenti,  
// clasa ce mosteneste calasa Persoane  
  
class StudentMasterand extends Student  
{  
    protected void metoda()  
    {  
        // apelarea metodei din clasa mostenita Persoane  
        metodaNume();  
  
        // apelarea metodei din clasa mostenita Studenti  
        metodaStudent();  
    }  
}
```

Mostenire.java

```
public class Mostenire  
{  
    public static void main(String[] args)  
    {
```

```
StudentMasterand a = new StudentMasterand();  
a.metoda();  
}  
}
```

În exemplul anterior s-au definit trei clase: `Persoana`, `Student`, `StudentMasterand`. Fiecare clasă trece prin testul “este un” (studentul este o persoană, iar un student masterand este un student). Aceasta înseamnă că se pot lega cele trei clase într-o relație, folosind moștenirea.

Moștenirea pe mai multe niveluri permite unei subclase să moștenească o singură superclasă. Totuși în aplicație se dorește ca `StudentMasterand` să moștenească atât clasa `Student`, cât și clasa `Persoana`. Se poate ocoli restricția impusă de moștenirea simplă creând mai întâi o relație de moștenire între clasa `Persoana` și clasa `Student`. Clasa `Student` are acces la toți membrii protejați și publici ai clasei `Persoana`. Apoi se poate crea o relație de moștenire între clasa `Student` și clasa `StudentMasterand`. Clasa `StudentMasterand` moștenește toți membrii publici și protejați ai clasei `Student`, ceea ce include și accesul la membrii publici și protejați ai clasei `Persoana`, deoarece clasa `Student` a moștenit deja accesul la acei membrii prin moștenirea clasei `Persoana`.

Acest exemplu creează o moștenire pe mai multe niveluri. Primul nivel este format din clasele `Persoana` și `Student`. Clasa `Persoana` este superclasă, iar clasa `Student` este subclasă. Al doilea nivel este format din clasele `Student` și `StudentMasterand`: clasa `Student` este superclasă, iar clasa `StudentMasterand` este subclasă.

Se poate realiza orice număr de niveluri de moștenire într-un program cu condiția ca fiecare clasă să treacă de testul “este-un”. Totuși este de preferat să nu existe un număr mai mare de trei niveluri deoarece acesta duce la creșterea complexității și face ca programul să fie mai dificil de întreținut și actualizat.

10.2.3. Supradefinirea metodelor folosind moștenirea

O metodă permite unei instanțe a unei clase să implementeze un anumit tip de comportament, cum ar fi afișarea unei variabile de instanță. O subclasă moștenește comportamentul unei superclase, atunci când are acces la metodele superclasei.

Uneori comportamentul unei metode membre a superclasei nu îndeplinește cerințele unei subclase. De exemplu, maniera în care metoda membră a superclasei afișează o proprietate de instanță nu este exact modul în care trebuie să afișeze acea proprietate. În acest caz, se definește în subclasa o nouă versiune a metodei superclasei, incluzând instrucțiuni care îmbunătățesc comportamentul metodei din superclasă. Acesta tehnică este numită **supradefinire** (*method overriding*).



A nu se confunda *supraîncărcarea* (*overloading*) unei metode cu *supradefinirea* (*overriding*) unei metode.

Supraîncărcarea reprezintă definirea unei metode care are același nume cu o altă metodă, dar cu o listă de parametri diferită.

Supradefinirea reprezintă definirea unei metode care are același nume cu o alta metodă și aceeași listă de paramteri, însă are comportament diferit.

10.2.3.1. Supradefinirea metodelor folosind moștenirea

Salarii.java

```
class Salarii
{
    private String nume;
    private double salariu;

    // definirea constructorului cu lista de parametrii
    public Salarii(String n, double s)
    {
        nume = n;
        salariu = s;
    }

    public String getNume()
    {
        return nume;
    }

    public double getSalariu()
    {
        return salariu;
    }

    public double Salariu(double procent)
    {
        return salariu += salariu * procent / 100;
    }
}
```

Manager.java

```
class Manager extends Salarii
{
    private double bonus;
```



```

/*
 * clasa Manager fiind o subclasa a clasei Salarii are acces la
 * membrii acesteia publici si protejati
 */

public Manager(String n, double s)
{
    /*
     * apelarea constructorului din superclasa ce contine lista de
     * parametri(n,s)
     */
    super(n, s);
    bonus = 0;
}

public double getSalariu()
{
    // super.getSalariu apelarea metodei getSalariu metoda a
    // superclasei(Salarii)
    return (super.getSalariu() + bonus);
}

public double setBonus(double b)
{
    return bonus = b;
}
}

```

Mostenire.java

```

import java.util.*;

public class Mostenire
{
    public static void main(String[] args)
    {
        // construirea unui obiect de tip Manager
        Manager sef = new Manager("Popescu Aurel", 4000);
        sef.setBonus(500);

        Manager secretar = new Manager("Anghel Razvan", 2500);
        secretar.setBonus(200);

        Salarii[] obiect = new Salarii[3];

        // ocuparea matricii obiect cu instante de tip Manager si Salarii

        obiect[0] = sef;
        obiect[1] = secretar;
        obiect[2] = new Salarii("Grigore Mihai", 2000);
    }
}

```

```
// cresterea salariilor cu 5%
for (Salarii i : obiect)
{
    // apelarea metodei Salariu
    i.Salariu(5);

    // afisarea informatiilor despre toate obiectele de tip Salarii
    System.out.println("Nume=" + i.getNume() + "\tSalariu="
        + i.getSalariu());
}
}
```

10.2.4. Cuvântul cheie *final* și moștenirea

Cuvântul cheie `final` are doua utilizări în moștenire. În primul rând, poate fi folosit împreună cu o metodă membră a superclasei pentru a împiedica o subclasă să suprascrie o metodă:

Persoane.java

```
class Persoane
{
    final void MesajAvertizare()
    {
        System.out.println("Aceasta metoda nu poate fi supradefinita.");
    }
}
```

În al doilea rând, poate fi folosit împreună cu o clasă pentru a împiedica folosirea clasei respective ca superclasă (împiedică alte clase să moștenească clasa respectivă).

Persoane.java

```
final class Persoane
{
    void MesajAvertizare()
    {
        System.out.println("Aceasta clasa nu poate fi mostenita.");
    }
}
```

LABORATORUL 9

11. Tratarea excepțiilor

Într-un program pot apărea două tipuri de erori: *erori de compilare* și *erori de execuție*. O eroare de compilare apare, de obicei, atunci când programatorul a făcut o greșeală de sintaxă la scrierea codului (de ex. omiterea caracterului punct și virgule la sfârșitul unei instrucțiuni). Aceste erori sunt ușor descoperite de compilator.

O eroare de execuție apare în timpul rulării programului și poate avea diverse cauze: introducerea de către utilizator a unor date de altă natură decât cele așteptate de program, indisponibilitatea unei resurse (de ex. un fișier), etc.

Erorile de execuție sunt mai grave decât erorile de compilare deoarece acestea apar întotdeauna, pe când eroarele de execuție apar numai în anumite situații. Din acest motiv, aceste erori trebuie anticipate și trebuie incluse în program modalitățile lor de dezvoltare.

11.1. Proceduri de tratare a excepțiilor

O procedură de tratare a excepțiilor este o porțiune a programului care conține instrucțiuni ce sunt executate când apare o anumită eroare de execuție.

Includerea unei proceduri de tratare a excepțiilor în program se numește *tratarea excepțiilor*.

Anumite instrucțiuni dintr-un program sunt susceptibile la generarea erorilor de execuție. De obicei, aceste instrucțiuni depind de surse din afara programului, cum ar fi date introduse de utilizatorul programului sau prelucrări care ar putea cauza erori de execuție.

În loc să se verifice datele provenite din surse externe sau din alte prelucrări se monitorizează aceste instrucțiuni și se *aruncă* o excepție dacă apare o eroare de execuție. De asemenea, se include în program instrucțiuni care sunt executate dacă este aruncată o excepție. Această procedură se numește *prinderea* unei excepții.

Instrucțiunile ce trebuie monitorizate trebuie încadrate într-un bloc **try**.

```
try
{
    <instrucțiuni>;
}
```

Instrucțiunile executate atunci când apare o excepție sunt plasate într-un bloc **catch**.

```
catch (<tip_exceptie> <id_var_exceptie>)
{
    <instructiuni>;
}
```

Mai există o secțiune – `finally` – ce conține instrucțiuni care se execută indiferent dacă a apărut o excepție sau nu. De obicei, în blocul `finally` se plasează instrucțiuni care eliberează resursele rezervate de program.

```
finally
{
    <instructiuni>;
}
```

Fiecărui bloc `try` trebuie să-i corespundă cel puțin un bloc `catch` sau un bloc `finally`. Blocul `catch` trebuie să fie imediat după blocul `try` pentru a nu se genera o eroare de compilare.

Format general:

```
try
{
    <instructiuni_test>;
}
[catch (<tip_exceptie1> <id_var_exceptie1>)
{
    <instructiuni_tratare_exceptie1>;
}]
[catch (<tip_exceptie2> <id_var_exceptie2>)
{
    <instructiuni_tratare_exceptie2>;
}]
[...]
[finally
{
    <instructiuni_care_se_executa_totdeauna>;
}]
```

Presupunem exemplul:

ImpartireLaZero.java

```
public class ImpartireLaZero
{
    public static void main(String args[])
    {
        int x = 3;
        int y = 0;
```

```
try
{
    int z = impartire(x, y);
}
catch (ArithmeticException e)
{
    System.out.println(e);
    System.out.println("Exceptia Aritmetica: impartirea la 0");
}

public static int impartire(int a, int b)
{
    int z = a / b;
    return z;
}
}
```

La rulare va afișa:

```
java.lang.ArithmeticException: / by zero
Exceptia Aritmetica: impartirea la 0
```

deoarece la sfârșitul blocului `try` se afla un bloc `catch`, care prinde excepțiile de tip `ArithmeticException`, reprezentate prin variabila `e`. Codul monitorizat de blocul `try` generează o eroare aritmetică atunci când se încearcă o împărțire la zero.

Când se încearcă o împărțire la zero, instrucțiunea în cauză aruncă o excepție de împărțire la zero, care este prinsă de variabila `e`.

O serie de instrucțiuni pot genera mai multe tipuri de erori de execuție. Prin urmare se folosesc blocuri `catch` multiple pentru a intercepta fiecare tip de excepție.

Blocurile `catch` multiple trebuie plasate imediat după blocul `try` în care ar putea fi aruncată excepția. De asemenea, blocurile `catch` trebuie să fie plasate unul după altul.

Atunci când apare o excepție, Java o aruncă către blocurile `catch` care urmează după blocul `try`, în ordinea în care apar acestea în program. Dacă există două blocuri `catch` și primul bloc prinde excepția, aceasta nu mai ajunge la al doilea bloc `catch`, deoarece este tratată în primul bloc.

Demo1.java

```
public class Demo1
{
    public static void main(String[] args)
    {
        try
        {
            int a[] = new int[4];
            a[0] = 0;
            a[1] = 10;
            a[2] = 5;
            a[3] = a[0] / a[4];

        }
        catch (ArithmeticException e)
        {
            System.out.println("Eroare: nu se poate imparti la 0");
            System.out.println(e);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Eroare: s-a depasit indexul vectorului a);
            System.out.println(e);
        }
    }
}
```

Rezultatul rulării:

```
Eroare: s-a depasit indexul vectorului a
java.lang.ArrayIndexOutOfBoundsException: 4
```

Excepția prinsă este de depășire a indexului: `ArrayIndexOutOfBoundsException`.

Demo2.java

```
public class Demo2
{
    public static void main(String[] args)
    {
        try
        {
            int a[] = new int[4];
            a[0] = 0;
            a[1] = 10;
            a[2] = 5;
            a[3] = a[1] / a[0];
        }
    }
}
```

```
catch (ArithmeticException e)
{
    System.out.println("Eroare: nu se poate imparti la 0");
    System.out.println(e);
}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Eroare: s-a depasit indexul vectorului a");
    System.out.println(e);
}
}
```

Rezultatul rulării:

```
Eroare: nu se poate imparti la 0
java.lang.ArithmeticException: / by zero
```

Excepția prinsă este o excepție aritmetica de împărțire a unui număr la zero: `ArithmeticException`.

Demo3.java

```
public class Demo3
{
    public static void main(String[] args)
    {
        try
        {
            int a[] = new int[4];
            a[0] = 0;
            a[1] = 10;
            a[2] = 5;
            a[3] = a[1] / a[0];
            a[3] = a[0] / a[4];

        }
        catch (ArithmeticException e)
        {
            System.out.println("Eroare: nu se poate imparti la 0");
            System.out.println(e);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Eroare: s-a depasit indexul vectorului a");
            System.out.println(e);
        }
    }
}
```

Rezultatul rulării:

Eroare: nu se poate imparti la 0
java.lang.ArithmeticException: / by zero

Prima eroare interceptată este de tip `ArithmeticException`.

Exemplul următor ilustrează folosirea blocului `finally` într-un program Java:

DemoFinally.java

```
public class DemoFinally
{
    public static void main(String[] args)
    {
        try
        {
            int a[] = new int[4];
            a[0] = 0;
            a[1] = 10;
            a[2] = 5;
            a[3] = a[0] / a[4];
        }
        catch (ArithmeticException e)
        {
            System.out.println("Eroare: nu se poate imparti la 0");
            System.out.println(e);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Eroare: s-a depasit indexul vectorului a");
            System.out.println(e);
        }
        finally
        {
            System.out.println("Executarea blocului finally");
        }
    }
}
```

Rezultatul rulării:

Eroare: s-a depasit indexul vectorului a
java.lang.ArrayIndexOutOfBoundsException: 4
Executarea blocului finally

11.1.1. Blocuri `try` imbricate

Există situații când se combină două sau mai multe blocuri `try`, plasând un bloc `try` în interiorul unui alt bloc `try`. Acest procedeu se numește imbricarea blocurilor `try`, iar blocul intern se numește bloc `try` imbricat. Colectiv, acestea se numesc perechi imbricate de blocuri `try`.

Fiecare bloc `try` dintr-o pereche imbricată trebuie să aibă unul sau mai multe blocuri `catch` sau un bloc `finally`, care prind excepțiile. Blocurile `catch` sunt plasate imediat după blocul `try` corespunzător, la fel ca atunci când se scriu blocuri `try` simple.

Atunci când o instrucțiune din blocul `try` intern determină aruncarea unei excepții, Java verifică dacă blocurile `catch` asociate cu blocul `try` intern pot să prindă excepția. Dacă nici unul din aceste blocuri nu poate să prindă excepția aruncată, sunt verificate blocurile `catch` asociate blocului `try` extern. Dacă nici în acest fel excepția nu este tratată atunci excepția este tratată de procedura implicită.

Demo.java

```
public class Demo
{
    public static void main(String[] args)
    {
        try
        {
            try
            {
                int a[] = new int[3];
                a[0] = 10;
                a[1] = 0;
                a[2] = a[3] / a[1];
            }
            catch (ArithmeticException e)
            {
                System.out.println("S-a im partit la 0!!!!");
                System.out.println("Eroare:" + e);
            }
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("S-a depasit indexul!!!!");
            System.out.println("Eroare:" + e);
        }
    }
}
```

Rezultatul rulării:

```
S-a depasit indexul!!!!  
Eroare: java.lang.ArrayIndexOutOfBoundsException: 3
```

11.2. Lucrul cu excepții neinterceptate

Deși în timpul execuției programului pot apărea foarte multe excepții, nu este nevoie să se scrie câte un bloc `catch` astfel încât să fie prinse toate. Orice excepție pentru care nu există un bloc `catch` este interceptată de blocul `catch` implicit din Java, numit *procedură implicită de tratare a excepțiilor (default handler)*.

Ori de câte ori prinde o excepție, procedura implicită afișează două tipuri de informații: un șir de caractere care descrie excepția apărută și o înregistrare a stivei. Înregistrarea stivei arată ce execută programul, începând cu punctul în care a apărut excepția și terminând cu punctul în care execuția programului s-a încheiat.

Demo.java

```
public class Demo  
{  
    public static void main(String[] args)  
    {  
        int a = 0, b = 2, c;  
        c = b / a;  
    }  
}
```

Rezultatul rulării:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Demo.main(Demo.java:6)
```

Prima linie este mesajul care descrie excepția. A doua linie este înregistrarea stivei, care spune că excepția a apărut în linia 6 din metoda `main()` a programului `Demo.java`. Aceasta este linia în care se face calculul împărțirii lui 2 la 0. Pe această linie se termină programul odată cu aruncarea excepției.

Dacă excepția este generată de o instrucțiune dintr-o metodă apelată de metoda `main()`, înregistrarea stivei include ambele metode, arătând calea până la instrucțiunea care a cauzat excepția.

Demo.java

```
public class Demo
{
    public static void main(String[] args)
    {
        metoda();
    }

    static void metoda()
    {
        int a = 0, b = 2, c;
        c = b / a;
    }
}
```

Rezultatul rulării:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Demo.metoda(Demo.java:9)
    at Demo.main(Demo.java:4)
```

Prima linie este mesajul care descrie excepția. Începând cu a doua linie începe înregistrarea stivei. Citind înregistrarea stivei începând cu ultima linie: mai întâi este apelată funcția `main()` a programului `Demo.java`, apoi în linia 4 a fost apelată metoda `metoda()`. Linia 9 a programului, care face parte din definiția metodei `metoda()`, este punctul în care s-a încheiat execuția programului, din cauza unei excepții.

11.3. Metode care nu tratează excepțiile

Nu este obligatoriu ca o metodă care aruncă o excepție să o și trateze. Totuși, antetul metodei trebuie să specifice excepțiile care pot fi generate de metoda respectivă. Pentru a apela metoda trebuie să se includă în program un bloc `try..catch` care tratează excepțiile ce nu sunt interceptate de metodă.

Se specifică excepțiile lansate folosind cuvântul cheie `throws` în antetul metodei, după lista de parametri. Dacă sun mai multe tipuri de excepții, ele vor fi separate prin virgule.

Exemplul următor ilustrează specificarea excepțiilor care sunt aruncate, dar nu și interceptarea de către metodă.

Metoda `myMethod()` poate arunca două excepții: `ArithmeticException` și `ArrayIndexOutOfBoundsException`. Totuși metoda `myMethod()` nu prinde nici una dintre aceste excepții. Metoda `main()` prinde excepția `ArithmeticException`, dar lasă excepția `ArrayIndexOutOfBoundsException` în seama procedurii de tratare implicită din Java.

Demo.java

```
public class Demo
{
    static void myMethod() throws ArithmeticException,
                               ArrayIndexOutOfBoundsException
    {
        int a[] = new int[3];
        a[0] = 10;
        a[1] = 0;
        a[2] = a[0] / a[1];
        System.out.println("myMethod");
    }

    public static void main(String[] args)
    {
        try
        {
            myMethod();
        }
        catch (ArithmeticException e)
        {
            System.out.println("S-a im partit la 0!!!!");
            System.out.println("Eroare: " + e);
        }
    }
}
```

11.4. Excepții verificate și neverificate

Clasele de excepții sunt împărțite în două grupuri: verificate și neverificate. Excepțiile verificate trebuie să fie incluse în lista de excepții aruncate de o metodă; dacă metoda respectivă poate să arunce excepția, dar nu o și tratează. Compilatorul verifică dacă există blocuri `catch` care tratează aceste excepții.

Cu alte cuvinte o excepție verificată trebuie tratată explicit, fie prin interceptarea într-un bloc `catch`, fie prin declararea ca excepție aruncată.

Excepțiile neverificate nu sunt obligatoriu să fie incluse în lista de excepții aruncate de o metodă, deoarece compilatorul nu verifică dacă există un bloc `catch` care tratează aceste excepții. Majoritatea excepțiilor, atât verificate, cât și neverificate, sunt definite în pachetul `java.lang`, care este importat implicit în toate programele Java.

Următorul tabel prezintă o parte din **excepțiile verificate**:

Excepție	Descriere
<code>ArithmeticException</code>	Eroare aritmetică, cum ar fi împărțirea la zero
<code>ArrayIndexOutOfBoundsException</code>	Indexul matricei a depășit limitele
<code>ArrayStoreException</code>	Atribuirea unui tip incompatibil unui element al matricei
<code>ClassCastException</code>	Conversie invalidă
<code>IllegalArgumentException</code>	Argument invalid folosit pentru apelarea unei metode
<code>IllegalMonitorStateException</code>	Operație de monitorizare ilegală, cum ar fi așteptarea unui fir de execuție blocat
<code>IllegalStateException</code>	Mediul sau aplicația se află într-o stare incorectă
<code>IllegalThreadStateException</code>	Operația solicitată nu este compatibilă cu starea curentă a firului de execuție
<code>IndexOutOfBoundsException</code>	Un tip oarecare de index a depășit limitele
<code>NegativeArraySizeException</code>	Matrice cu dimensiune negativă
<code>NullPointerException</code>	Utilizarea unei referințe nule
<code>NumberFormatException</code>	Conversie invalidă a unui șir de caractere într-un format numeric
<code>SecurityException</code>	Încercare de încălcare a securității
<code>StringIndexOutOfBoundsException</code>	Încercare de indexare dincolo de limitele unui șir de caractere
<code>UnsupportedOperationException</code>	A fost întâlnită o operație neacceptată

Următorul tabel prezintă o parte din **excepțiile neverificate**:

Excepție	Descriere
<code>ClassNotFoundException</code>	Clasa nu a fost găsită
<code>CloneNotSupportedException</code>	Încercarea de a clona un obiect care nu implementează interfața <code>Cloneable</code>
<code>IllegalAccessException</code>	Accesul la o clasă este interzis
<code>InstantiationException</code>	Încercarea de a crea o instanță a unei clase abstracte sau a unei interfețe
<code>InterruptedException</code>	Un fir de execuție a fost întrerupt de un alt fir de execuție
<code>NoSuchFieldException</code>	A fost solicitat un câmp care nu există
<code>(NoSuchMethodError</code>	A fost solicitată o metoda care nu există
<code>IOException</code>	A apărut o excepție în timpul unui proces de intrare/ieșire
<code>SQLException</code>	A apărut o excepție în timpul interconectării cu un sistem de gestionare a bazelor de date folosind SQL

11.5. Tratarea excepțiilor folosind superclasa `Exception`

Toate tipurile de excepții din Java sunt clase derivate din clasa `Exception`, astfel că, prin mecanismul de polimorfism, o variabilă de tip `Exception` poate prinde orice excepție.

Demo.java

```
public class Demo
{
    public static void main(String[] args)
    {
        try
        {
            int a[] = new int[4];
            a[0] = 0;
            a[1] = 10;
            a[2] = 5;
            a[3] = a[0] / a[4];
        }
        catch (Exception e)
        {
            System.out.println("A aparut o eroare!");
            System.out.println(e);
        }
        finally
        {
            System.out.println("Executarea blocului finally");
        }
    }
}
```

Rezultatul rulării:

```
A aparut o eroare!
java.lang.ArrayIndexOutOfBoundsException: 4
Executarea blocului finally
```

Acest tip de tratare are dezavantajul că nu se poate preciza tipul eroarii apărute.

11.6. Exemple

11.6.1. Excepții de I/O

ScanExceptii.java

```
import java.util.*;

public class ScanExceptii
{
    public static void main(String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Introduceti un numar: ");
        double d = 0;
        boolean numar = false;
```

```

while (!numar)
{
    try
    {
        String s = scanner.nextLine();
        /*
         * initializeza variabila d cu valoarea specificata de String in
         * cazul in care String poate fi reprezentat ca o valoare double
         */
        d = Double.parseDouble(s);
        numar = true;
    }
    catch (NumberFormatException e)
    {
        /*
         * NumberFormatException-valoarea String nu poate fi convertita
         * intr-o valoare double
         */
        System.out.println("Nu ai inserat un numar valid. "
                           + "Incearca din nou ");
        System.out.print("Introduceti un numar: ");
    }
}

System.out.println("Ai introdus: " + d);
// inchiderea Scanner
scanner.close();
}
}

```

11.6.2. Excepții: Depășirea indexului unui vector

DepasireIndex.java

```

public class DepasireIndex
{
    public static void main(String args[])
    {
        int[] Array = { 0, 1 };

        try
        {
            System.out.println("Intrare in blocul try");
            System.out.println(Array[2]);
            System.out.println("Iesire din blocul try");
        }
        catch (ArithmeticException e)
        {
            System.out.println("Exceptie aritmetica");
        }
    }
}

```

```
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Exceptie index depasit");
            // instructiunea return-iesirea din metoda return;
        }
    Finally
    {
        System.out.println("Blocul Finally este intodeaua executat");
    }

    System.out.println("Instructiune de dupa blocul try");
}
}
```

11.6.3. Exceptii: Vector cu dimensiune negativă

Negative.java

```
public class Negative
{
    public static void main(String args[])
    {
        int x = 0, y = 7, z;
        int vector[];

        try
        {
            System.out.println("Intrarea in blocul try.");
            // declararea unui vector cu dimensiune negativa
            vector = new int[x - y];
            System.out.println("Iesirea din blocul try.");
            // prinderea exceptiei
        }
        catch (NegativeArraySizeException e)
        {
            System.out.println("Exceptia consta intr-un tablou "
                               + "de dimensiune negativa.");

            /*
             * executarea blocului finally care se executa indiferent daca
             * apar sau nu exceptii
             */
        }
        finally
        {
            // redeclararea vectorului
            vector = new int[y - x];
            System.out.println("Blocul finally este intodeauna executat.");
        }
    }
}
```



```

System.out.println("Executia de dupa blocurile try-catch "
                  + "si finally.");
System.out.println("Depasirea indexului unui vector.");

try
{
    // initializarea vectorului
    for (int i = 0; i <= vector.length; i++)
        vector[i] = i;
    // prinderea exceptiei
}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Exceptie index depasit");
    try
    {
        // realizarea unei operatii de impartire a unui intreg la 0
        z = y / x;
        /*
        * prinderea exceptiei generate de impartirea unui intreg
        * la 0 si tratatrea ei
        */
    }
    catch (ArithmeticException i)
    {
        System.out.println("Exceptie aritmetica");
        z = x / y;
        System.out.println(z);
    }
}
}
}

```

11.6.4. Exceptii: *NullPointerException*

NullPointer.java

```

public class NullPointer
{
    public static void main(String args[])
    {
        Integer Val = new Integer(10);
        try
        {
            Val = null;
            int i = Val.intValue();
            System.out.println(i);
        }
    }
}

```

```
        catch (NullPointerException e)
        {
            System.out.println("Exceptia consta in faptul ca "
                               + "obiectul este este null.");
        }
    }
}
```

11.6.5. Excepții: *ArrayIndexOutOfBoundsException*

RandomIndex.java

```
public class RandomIndex
{
    public static void main(String args[])
    {
        // declararea unei matrici de 5 elemente cu initializare
        int[] Array = { 0, 1, 2, 3, 4 };
        // initializarea variabilei index cu 0
        int index = 0;
        // crearea unui ciclu: instructiunile sunt executate de 10 ori
        for (int i = 0; i < 10; i++)
        {
            /*
             * Math.random() genereaza un nr aleator intre 0.0-1.0 de tip
             * double. Se imulteste cu 10 pentru a genera un nr intre 0 si 10
             */
            index = (int) (Math.random() * 10);

            try
            {
                System.out.println("La indexul " + index + " valoarea este: "
                                   + Array[index]);
            }
            catch (ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Indexul " + index
                                   + " depaseste dimensiunea.");
            }
        }
    }
}
```

LABORATORUL 10

12. Interfețe

Interfețele implementează conceptul programării orientate obiect de separarea a modelului unui obiect de implementarea sa. O interfață definește metode, dar nu și implementarea lor.

O clasă care implementează o interfață trebuie să ofere implementarea efectivă a metodelor declarate în interfață.

De asemenea, o interfață poate conține și declarații de constante.

Practic, o interfață este un ansamblu de metode fără implementare și declarații de constante.



Toate metodele declarate într-o interfață au, implicit, modificatorul de acces public.

Format general:

```
[<modif_acces>] interface <Nume_Interfata>
    [extends <Super_Interfata1>[, <SuperInterfata2>[, ...]]]
{
    <declaratii_de_constane_si_metode>
}
```

O clasă implementează o interfață prin utilizarea cuvântului cheie `implements`.

Următoarele exemple prezintă modul în care se utilizează interfețele.

Masina.java

```
interface Masina
{
    String formattedText = "\nDetalii masina:"; // folosit pentru afisare

    String getMarca();
    String getCuloare();
    void setCuloare(String value);
    float getGreutate();
    void setGreutate(float value);
    int getPutereMotor();
    void setPutereMotor(int value);
    String getTipCombustibil(); // "benzina", "motorina"...
    void setTipCombustibil(String value);
}
```

```
int getPret();  
void setPret(int value);  
void afiseazaDetalii();  
}
```

Dacia.java

```
class Dacia implements Masina  
{  
    String culoare = "albastru"; // valori implicite  
    float greutate = (float) 1500.00;  
    int putereMotor = 70;  
    String tipCombustibil = "benzina";  
    int pret = 10000;  
  
    public String getMarca()  
    {  
        return "Dacia";  
    }  
  
    public String getCuloare()  
    {  
        return culoare;  
    }  
  
    public void setCuloare(String value)  
    {  
        culoare = value;  
    }  
  
    public float getGreutate()  
    {  
        return greutate;  
    }  
  
    public void setGreutate(float value)  
    {  
        greutate = value;  
    }  
  
    public int getPutereMotor()  
    {  
        return putereMotor;  
    }  
  
    public void setPutereMotor(int value)  
    {  
        putereMotor = value;  
    }  
}
```

```
public String getTipCombustibil()
{
    return tipCombustibil;
}

public void setTipCombustibil(String value)
{
    tipCombustibil = value;
}

public int getPret()
{
    return pret;
}

public void setPret(int value)
{
    pret = value;
}

public void afiseazaDetalii()
{
    System.out.println(formatedText);
    System.out.println("Marca = Dacia");
    System.out.println("Culoare = " + culoare);
    System.out.println("Greutate = " + String.valueOf(greutate));
    System.out.println("Putere motor = " + String.valueOf(putereMotor));
    System.out.println("Tip combustibil = " + tipCombustibil);
    System.out.println("Pret = " + String.valueOf(pret));
}
}
```

Interfete.java

```
class Interfete
{
    public static void main(String[] args)
    {
        Dacia masina1 = new Dacia();

        masina1.afiseazaDetalii();

        Dacia masina2 = new Dacia();
        masina2.setCuloare("rosu");
        masina2.setPret(34000);
        masina2.setGreutate((float) 1345.70);

        masina2.afiseazaDetalii();
    }
}
```

Instrument.java

```
interface Instrument
{
    // defineste o metoda fara implementare
    void play();
}
```

Pian.java

```
class Pian implements Instrument
{
    // clasa care implementeaza interfata
    // trebuie sa implementeze, obligatoriu, metoda play
    public void play()
    {
        System.out.println("Canta un Pian!");
    }
}
```

Vioara.java

```
class Vioara implements Instrument
{
    public void play()
    {
        System.out.println("Canta o Vioara!");
    }
}
```

Muzica.java

```
public class Muzica
{
    // clasa principala
    static void play(Instrument i)
    {
        // metoda statica care porneste un instrument generic
        // ce implementeaza interfata Instrument
        i.play();
    }

    static void playAll(Instrument[] e)
    {
        for (int i = 0; i < e.length; i++)
            play(e[i]);
    }
}
```

```
public static void main(String[] args)
{
    Instrument[] orchestra = new Instrument[2];
    int i = 0;
    orchestra[i++] = new Pian();
    orchestra[i++] = new Vioara();
    playAll(orchestra);
}
```

LABORATORUL 11

13. Fluxuri de intrare / ieșiere (fișiere)

Datorită faptului că informațiile prelucrate de majoritatea programelor se stochează pe suporturi externe, este necesară existența unor clase pentru operațiile de lucru cu aceste suporturi.

Pentru ca un program Java să aducă date din surse externe trebuie să deschidă un **canal de comunicație** (*flux*) către sursa externă și să *citească* datele respective. În mod similar, pentru a stoca datele dintr-un program pe o sursă externă trebuie să se deschidă un flux către sursa externă și să se scrie datele pe ea.

Java include un număr mare de clase pentru lucrul cu fișiere și fluxuri, concepute pentru operațiile de stocare și transfer ale datelor.

13.1. Fișiere și sisteme de fișiere

Un **fișier** reprezintă un grup de octeți înrudiți. Un sistem de fișiere este un produs software folosit pentru organizarea și păstrarea fișierelor pe un dispozitiv secundar de stocare.

Fiecare fișier are proprietăți care îl descriu. Sistemul de fișiere determină tipul proprietăților care descriu fișierul. În mod obișnuit, acesta înseamnă numele fișierului, permisiunile de acces, data și ora ultimei modificări a fișierului. În general, cele trei permisiuni de acces folosite sunt: citire / scriere (read / write), numai citire (read-only) și execuție.

Fișierele sunt organizate în directoare și subdirectoare. Directorul aflat pe poziția cea mai înaltă a ierarhiei se numește **director rădăcina**. De aici începe întreaga ierarhie de directoare și subdirectoare. Colectiv, directoarele și subdirectoarele formează **structura de directoare**.

13.2. Clasa **FILE**

Este inclusă în pachetul `java.io` și definește metode utilizate pentru interacțiunile cu sistemul de fișiere și navigarea în structura de directoare.

Când se creează o instanță a clasei `File` se folosește unul din cei trei constructori ai clasei:

```
File obj = new File(String directory);  
File obj = new File(String directory, String fileName);  
File obj = new File(String directoryObject, String fileName);
```

Primul constructor necesită transmiterea căii de acces la fișier. De multe ori calea de acces este o ierarhie de directoare terminate cu directorul de care are nevoie programul. De exemplu,

D:\Laborator_Java \test este o cale de acces care ajunge la subdirectorul **test** din directorul **aplicatii** de pe partiția **D**.

Al doilea constructor are nevoie de doi parametri. Primul parametru este calea de acces la director, iar al doilea parametru este numele unui fișier conținut în ultimul subdirector din calea de acces specificată.

Al treilea constructor este foarte asemănător ce cel de-al doilea, dar calea de acces la director este transmisă ca o instanță clasei `File`, nu ca un șir de caractere.



Acești constructori nu creează un director sau un fișier.

După ce programul are la dispoziție o cale de acces la un director sau un fișier, pot fi apelate metode definite în clasa `File`.

Metode definite în clasa `File` sunt prezentate sintetic în tabelul următor:

Metodă	Descriere
<code>isFile()</code>	Returnează valoarea booleană <code>true</code> dacă obiectul este un fișier. Metoda returnează valoare <code>false</code> când obiectul nu este fișier sau instanța clasei <code>File</code> referă un director, un subdirector, un canal sau un driver de dispozitiv.
<code>isAbsolute()</code>	Returnează valoarea booleană <code>true</code> dacă fișierul conține o cale de acces absolută altfel returnează valoarea <code>false</code> .
<code>boolean renameTo(File newName)</code>	Redenumeste un director, subdirector sau un fișier, folosind numele transmis ca argument.
<code>delete()</code>	Șterge un fișier de pe sursa externă.
<code>void deleteOnExit()</code>	Șterge fișierul după ce mașina virtuală Java termină execuția.
<code>boolean isHidden()</code>	Returnează valoarea booleană <code>true</code> dacă fișierul sau calea de acces este ascunsă; altfel returnează valoarea booleană <code>false</code> .
<code>boolean setLastModified(long ms)</code>	Modifică marca temporală a fișierului. Marca temporală furnizată ca argument trebuie să specifice data și ora în milisecunde, începând cu 1 ianuarie 1970 și terminând cu data și ora curentă.
<code>boolean setReadOnly()</code>	Stabilește permisiunea de acces numai pentru citire.
<code>compareTo()</code>	Compară două fișiere.
<code>length()</code>	Returnează lungimea fișierului în octeți.
<code>isDirectory()</code>	Returnează valoarea booleană <code>true</code> dacă obiectul este un director; altfel returnează valoarea booleană <code>false</code> .

<code>canRead()</code>	Returnează valoarea booleană <code>true</code> dacă directorul sau fișierul are permisiune de citire; altfel returnează valoarea booleană <code>false</code> .
<code>canWrite()</code>	Returnează valoarea booleană <code>true</code> dacă directorul sau fișierul are permisiune de scriere; altfel returnează valoarea booleană <code>false</code> .
<code>exists()</code>	Returnează valoarea booleană <code>true</code> dacă directorul sau fișierul există; altfel returnează valoarea booleană <code>false</code> .
<code>getParent()</code>	Returnează calea de acces la directorul părinte al subdirectorului.
<code>getAbsolutePath()</code>	Returnează calea de acces absolută a subdirectorului.
<code>getPath()</code>	Returnează calea de acces a subdirectorului.
<code>getName()</code>	Returnează numele directorului sau a fișierului.

Următoarea aplicație ilustrează utilizarea acestora.

DemoFile.java

```
import java.io.File;

public class DemoFile
{
    public static void main(String[] args)
    {
        File obj = new File("\\Laborator_Java\\DemoFile");
        System.out.println("Nume:" + obj.getName());
        System.out.println("Cale:" + obj.getPath());
        System.out.println("Despre cale:" + obj.getAbsolutePath());
        System.out.println("Parinte:" + obj.getParent());
        System.out.println("Existenta:" + obj.exists());
        System.out.println("Scriere:" + obj.canWrite());
        System.out.println("Citire:" + obj.canRead());
        System.out.println("Director:" + obj.isDirectory());
        System.out.println("File:" + obj.isFile());
        System.out.println("Cale absoluta:" + obj.isAbsolute());
        System.out.println("Length:" + obj.length());
    }
}
```

Rezultatul rulării:

```
Nume: DemoFile
Cale: \Laborator_Java\DemoFile
Despre cale: D:\Laborator_Java\DemoFile
Parinte: \Laborator_Java
Existenta: true
Scriere: true
Citire: true
```

```
Director: true
File: false
Cale absoluta: false
Length: 0
```

13.2.1. Afișarea listei de fișiere dintr-un director

Se poate returna conținutul unui director apelând metoda `list()`.

Metoda `list()` returnează un tablou de șiruri de caractere conținând numele fișierelor stocate într-un director. Metoda `list()` poate fi folosită numai pentru directoare, de aceea este recomandat să se apeleze metoda `isDirectory()` pentru a verifica dacă obiectul este sau nu director, înainte de apelarea metodei.

Exemplul următor ilustrează folosirea metodelor `list()` și `isDirectory()` pentru obținerea și afișarea conținutului unui director.

DemoFileDirector.java

```
import java.io.File;

public class DemoFileDirector
{
    public static void main(String args[])
    {
        // atribuirea numelui directorului unui sir de caractere "dir"
        String dir = "/acest_fisier";
        // transmiterea sirului "dir" constructorului clasei File
        File file1 = new File(dir);
        // verificarea daca exista director
        if (file1.isDirectory())
        {
            // afisarea numelui directorului
            System.out.println("director: " + dir);
            // apelarea metodei list() pentru obtinerea continutului
            // directorului, care este stocat intr-o matrice de caractere,
            // numita str1[]
            String str1[] = file1.list();
            // parcurgerea elementelor matricii
            for (int i = 0; i < str1.length; i++)
            {
                // transmiterea fiecarui element constructorului clasei File
                File file2 = new File(dir + "/" + str1[i]);
                // stabilirea daca elementul matricii este un director
                if (file2.isDirectory())
                {
                    System.out.println("Director: " + str1[i]);
                    String str2[] = file2.list();
                }
            }
        }
    }
}
```

```

        for (int j = 0; j < str2.length; j++)
        {
            File file3 = new File(dir + "/" + str1[i] + "/" + str2[j]);
            if (file3.isDirectory())
                System.out.println("Director: " + str2[j]);
            else
                continue;
        }
    }
    else
        continue;
}
}
else
    System.out.println(dir + " nu este un director");
}
}

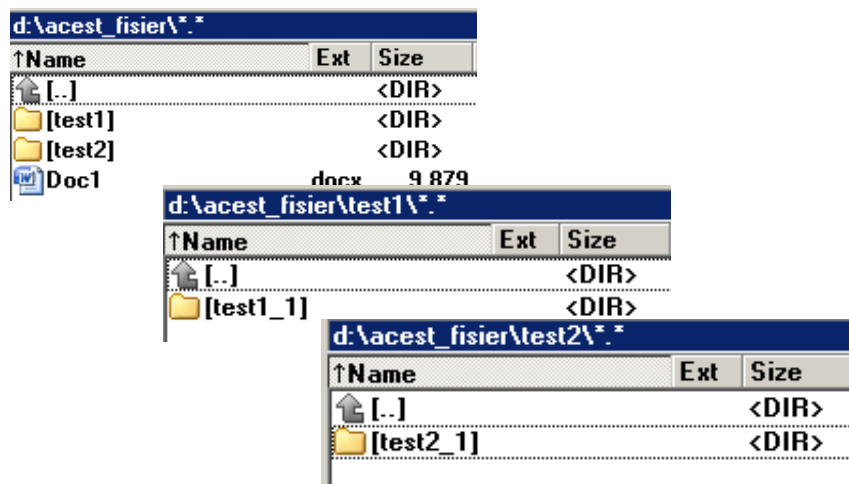
```

Rezultatul rulării:

```

director: /acest_fisier
Director: test1
Director: test1_1
Director: test2
Director: test2_1

```

Structura directorului:**13.3. Fluxuri**

Java include mai multe clase pentru fluxuri, construite pe baza a patru clase abstracte: `InputStream`, `OutputStream`, `Reader` și `Writer`.

13.3.1. Scrierea într-un fișier

Pentru a putea scrie date într-un fișier trebuie ca, mai întâi, să se creeze un flux de ieșire pentru fișier. Un flux de ieșire de fișiere deschide fișierul, dacă există, sau creează un nou fișier dacă nu există. După ce se deschide un flux de fișiere, se poate scrie date în fișier folosind un obiect de scriere `PrintWriter`.

Se deschide un flux de ieșire pentru fișiere folosind constructorul clasei `FileOutputStream`, căruia i se transmite numele fișierului pe care dorim să-l deschidem. Dacă fișierul nu se află în directorul curent se poate include și calea de acces, ca parte a numelui fișierului. Constructorul returnează o referință către fluxul de ieșire pentru fișiere.

Se creează un obiect de scriere apelând constructorul clasei `PrintWriter` și transmițându-i o referință către un flux de ieșire pentru fișiere. Constructorul returnează o referință a unui obiect `PrintWriter`, care poate fi folosită pentru scriere în fișier. După ce datele sunt scrise în fișier, trebuie apelată metoda `close()` pentru a închide fișierul.

Exemplul următor ilustrează deschiderea unui flux de ieșire pentru fișiere, crearea unui obiect de scriere `PrintWriter` și scrierea de date în fișier.

ScriereaInFisier.java

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

public class ScriereaInFisier
{
    public static void main(String[] args)
    {
        String studentnume = "bob";
        String prenume = "pen";
        String grade = "a";
        try
        {
            PrintWriter out = new PrintWriter(new FileOutputStream(
                "\\acest_fisier\\student.docx"));
            out.print(studentnume);
            out.print(prenume);
            out.println(grade);
            out.close();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

Exemplul începe cu declararea și inițializarea a trei șiruri de caractere, care vor fi scrise într-un fișier. În continuare, programul deschide un flux de ieșire pentru fișiere și crează un obiect de scriere. Fișierul folosit în acest exemplu se numește *Student.doc* și este stocat în directorul curent. Programul apelează apoi metoda `print()`, definită în clasa `PrintWriter`, pentru a scrie datele în fișierul *Student.doc*. De fiecare dată când este apelată metoda `print()`, îi sunt trimise datele care trebuie scrise în fișier. Fișierul este închis după scrierea ultimului element de date.

Observați că toate instrucțiunile implicate în deschiderea fluxului de ieșire și scrierea în fișier sunt incluse într-un bloc `try`. Dacă apare o problemă, cum ar fi spațiu insuficient pe disc, este aruncată o excepție, prinsă de un bloc `catch`.

13.3.2. Citirea dintr-un fișier

Există mai multe modalități de citire a datelor dintr-un fișier. Se pot citi octeți individuali sau blocuri de date format dintr-un număr specificat de octeți. De asemenea, se pot citi, printr-o singură operație, o linie de octeți. O linie este formată dintr-o serie de octeți care se termină cu un octet corespunzător caracterului linie nouă. Acest caracter este similar cu apăsarea tastei `ENTER` la sfârșitul unei propoziții. De obicei, tasta `ENTER` determină programul să insereze un caracter linie nouă. După citirea liniei programul trebuie să împartă linia în segmente semnificative, cum ar fi numele și prenumele unui student, folosind metode definite în clasa `String`. De asemenea, programul poate să afișeze direct pe ecran linia citită. O altă soluție frecvent folosită la citirea dintr-un fișier este să se citească datele ca șiruri de caractere, nu octeți.

Pentru citirea datelor trebuie să se deschidă fișierul. Se crează un obiect de citire din fișiere, folosind constructorul clasei `FileReader`, căruia i se transmite numele (și, eventual, calea) fișierului. Dacă fișierul nu se află în directorul curent, trebuie să se includă în nume și calea de acces la fișier.

Operația de citire a octeților de pe unitatea de disc se numește suprasarcină (*necessary overhead*), adică costul citirii unui fișier. Pentru a reduce suprasarcina, se citește printr-o singură operație un bloc de octeți, care se stochează într-o zonă de memorie numită *buffer* (zonă tampon). Programul citește apoi octeții din buffer, nu direct de pe disc.

Se poate folosi această tehnică în programe creând un cititor cu buffer (*buffered reader*), care creează un buffer pentru stocarea datelor citite pe unitatea de disc. Se creează un cititor cu buffer folosind constructorul clasei `BufferedReader`, căruia i se transmite ca argument o referință a obiectului `FileReader`, folosit pentru deschiderea fișierului. Clasa `BufferedReader` definește metodele folosite pentru citirea datelor din buffer.

Exemplul următor ilustrează crearea obiectelor pentru deschiderea fișierelor și citirea prin buffer. Acestea sunt apoi folosite pentru a citi date din fișierul creat în exemplu anterior. Programul deschide fișierul `Student.doc`, care conține date despre un student. Datele sunt copiate din fișier în buffer. Apoi este apelată metoda `readLine()`, care citește datele linie cu linie din buffer și le atribuie unei variabile de tip `String`.

Se observă că toate aceste operații sunt executate în expresia într-un ciclu `while`. Facem acest lucru pentru a stabili momentul în care ajungem la sfârșitul fișierului. Metoda `readLine()` returnează valoarea `null` atunci când ajunge la sfârșitul fișierului. Când se ajunge la sfârșitul fișierului, programul iese din ciclul `while` și apelează metoda `close()`, care închide fișierul.

CitireaDinFisier.java

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

public class CitireaDinFisier
{
    public static void main(String[] args)
    {
        String linie;
        try {
            BufferedReader in = new BufferedReader(new FileReader(
                "\\acest_fisier\\test2\\student.doc"));
            while ((linie = in.readLine()) != null)
            {
                System.out.println(linie);
            }
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

13.3.3. Adăugarea datelor într-un fișier

Se adaugă date într-un fișier folosind valoarea `true` pentru al doilea parametru al constructorului clasei `FileOutputStream`. Versiunea constructorului folosită în secțiunea *Scrierea într-un fișier* acceptă un singur argument, care este numele fișierului. În mod prestabilit, octeții sunt scriși de la începutul fișierului.

Există o altă versiune a constructorului `FileOutputStream`, care acceptă două argumente. Primul argument este, și pentru acest constructor, numele fișierului, iar al doilea argument este

valoarea booleana `true`. În acest caz, octeții sunt scriși la sfârșitul fișierului (*append*). Dacă nu există, fișierul este creat.

Exemplul următor ilustrează tehnica de adăugare a datelor în fișier.

AdaugareaDatelor.java

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

public class AdaugareaDatelor
{
    public static void main(String[] args)
    {
        String nume = "Mihaila";
        String prenume = "Dan";
        String grade = "9";

        try
        {
            PrintWriter out = new PrintWriter(new FileOutputStream(
                "\\acest_fisier\\test2\\student.doc", true));
            out.println(nume);
            out.println(prenume);
            out.println(grade);
            out.close();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

13.3.4. Citirea și scriere unui obiect într-un fișier

Într-un program real, multe din elementele de date pe care vreți să le stocați într-un fișier sunt date membre ale unei clase.

Multe attribute ale unei clase sunt stocate în variabile de instanță. O variabilă de instanță reprezintă o locație de memorie folosită pentru stocarea datelor specifice unei instanțe a clasei. Să presupunem că în definiția clasei sunt definite două variabile de instanță și sunt create cinci obiecte ale clasei. Java rezervă cinci seturi de variabile de instanță, independente unul față de celălalt. Totuși, fiecare dintre cele cinci obiecte folosește același set de metode – cele asociate clasei.

Atunci când se dorește păstrarea instanței unei clase, aceasta se salvează într-un fișier. Când se face acest lucru, în fișier sunt salvate numai variabilele de instanță, nu și metodele clasei.

Etapele scrierii și citirii unui obiect în/din fișier.

1. trebuie implementată interfața `Serializable` în clasa a cărei obiecte urmează să le scrieți în fișier. Interfața `Serializable` permite ca instanțele clasei respective să fie transformate într-un flux de octeți care poate fi scris pe disc sau trimis pe rețea. Această operație se numește *serializare*. La citirea obiectului, fluxul de octeți este *deserializat* și este reconstituită instanța clasei.
2. crearea unei instanțe a clasei și atribuirea valorii variabilelor de instanță. Apoi se deschide fișierul apelând constructorul clasei `FileOutputStream`, căruia i se transmite numele fișierului. Operatorul `new` returnează o referință a unui obiect `FileOutputStream`. Această referință este transmisă constructorului clasei `ObjectOutputStream`, care definește metodele folosite pentru scrierea unui obiect într-un fișier.
3. după crearea unei instanțe `ObjectOutputStream` se scrie un obiect în fișier. Pentru acest lucru se apelează metoda `writeObject()` și se transmite o referință a obiectului dorit a fi scris în fișier. Variabilele statice ale obiectului nu sunt salvate. Ultimul pas constă în apelarea metodei `close()`, pentru a închide fișierul.

Citirea unui obiect dintr-un fișier este la fel de simplă ca și scrierea unui obiect într-un fișier. Totuși, în locul clasei `ObjectOutputStream` se folosește clasa `ObjectInputStream`, iar în locul clasei `FileOutputStream` se folosește clasa `FileInputStream`. Se citește un obiect din fișier apelând metoda `readObject()`, definită în clasa `ObjectInputStream`. Metoda `readObject()` returnează o referință de tip `Object`. Această referință trebuie convertită la tipul specific al clasei obiectului citit.

În exemplul următor salvăm în fișier instanțe ale clasei `Student`. Atunci când citim o instanță din fișier o convertim la tipul `Student`. Obiectul returnat de metoda `readObject()` este apoi atribuit unei referințe a obiectului respectiv și este folosit ca orice alt obiect din program.

Exemplu definește o clasă `Student` care conține trei variabile de instanță, reprezentând numele, prenumele și nota studentului. Clasa implementează interfața `Serializable`, astfel încât să se poată transforma instanțele clasei în fluxuri de octeți. Sunt declarate trei instanțe ale clasei, pentru fiecare fiind transmise date de inițializare constructorului clasei. Datele transmise sunt atribuite variabilelor de instanță. În acest scop s-a folosit un tablou de obiecte, astfel încât să putem apela la un ciclu `for` pentru a scrie fiecare obiect în fișier. Folosirea ciclului `for` reduce numărul de linii pe care trebuie să le scriem în program.

Se deschide fișierul `Student.doc` apelând constructorul `FileOutputStream` și transmitându-i două argumente: primul este numele fișierului, iar al doilea argument este valoarea logică `true`, ceea ce înseamnă că vrem să adăugăm date în fișier.

Apelăm, apoi, metoda `writeObject()` în primul ciclu `for`, fiindu-i transmisă o referință a fiecărei instanței a clasei `Student` declarată în program. După ce este scrisă în fișier ultima instanță a clasei, fișierul este închis.

Fișierul este deschis din nou, dar de această dată pentru a citi obiectele stocate în fișier. În al doilea ciclu `for` este apelată metoda `readObject()`, care citește din fișier fiecare obiect. Se observă că valoarea returnată de metoda `readObject()` este convertită la tipul `Student`. Se realizează acest lucru deoarece metoda `readObject()` returnează o referință pentru clasa `Object`, care este prea generală pentru a fi folosită în acest program. Clasa `Object` este superclasa tuturor obiectelor. Tabloul `readStudentInfo` este folosit pentru a stoca fiecare obiect citit din fișier.

După ce ultimul obiect este citit din fișier, fișierul este închis, apoi sunt afișate pe ecran variabilele de instanță ale fiecărui obiect.

Student.java

```
import java.io.Serializable;

class Student implements Serializable
{
    String studentnume, studentprenume, studentnota;

    public Student()
    {
    }

    public Student(String nume, String prenume, String nota)
    {
        studentnume = nume;
        studentprenume = prenume;
        studentnota = nota;
    }
}
```

CitScrOb.java

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

public class CitScrOb
{
    public static void main(String[] args)
    {
        Student[] writeStudentInfo = new Student[3];
        Student[] readStudentInfo = new Student[3];
    }
}
```

```
writeStudentInfo[0] = new Student("Georgescu", "Catalin", "9");
writeStudentInfo[1] = new Student("Diaconu", "Maria", "8");
writeStudentInfo[2] = new Student("Radu", "Marius", "10");
try
{
    ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream(
            "\\acest_fisier\\test1\\studenti.doc", true));

    for (int y = 0; y < 3; y++)
        out.writeObject(writeStudentInfo[y]);

    out.close();

    ObjectInputStream in = new ObjectInputStream(new FileInputStream(
        "\\acest_fisier\\test1\\studenti.doc"));

    for (int y = 0; y < 3; y++)
        readStudentInfo[y] = (Student) in.readObject();

    in.close();

    for (int i = 0; i < 3; i++)
    {
        System.out.println(readStudentInfo[i].studentnume);
        System.out.println(readStudentInfo[i].studentprenume);
        System.out.println(readStudentInfo[i].studentnota);
    }

}
catch (Exception e)
{
    System.out.println(e);
}
}
```

LABORATORUL 12

14. Interfețe grafice

Interfața grafică sau, mai bine zis, *interfața grafică cu utilizatorul (GUI)*, este un termen cu înțeles larg care se referă la toate tipurile de comunicare vizuală între un program și utilizatorii săi. Aceasta este o particularizare a interfeței cu utilizatorul (UI), prin care vom înțelege conceptul generic de interacțiune între un program și utilizatorii săi. Așadar, UI se referă nu numai la ceea ce utilizatorul vede pe ecran ci la toate mecanismele de comunicare între acesta și program.

Limbajul Java pune la dispoziție numeroase clase pentru implementarea diverselor funcționalități UI, însă ne vom ocupa în continuare de acelea care permit realizarea unei interfețe grafice cu utilizatorul (GUI).

Pachetul care oferă servicii grafice se numește `java.awt`, AWT fiind prescurtarea pentru *Abstract Window Toolkit* și este pachetul care a suferit cele mai multe modificări în trecerea de la o versiune JDK la alta.

În principiu, crearea unei aplicații grafice presupune următoarele lucruri:

- Crearea unei suprafețe de afișare (cum ar fi o fereastră) pe care vor fi așezate obiectele grafice care servesc la comunicarea cu utilizatorul (butoane, controale de editare, texte, etc.);
- Crearea și așezarea obiectelor grafice pe suprafața de afișare în pozițiile corespunzătoare;
- Definirea unor acțiuni care trebuie să se execute în momentul când utilizatorul interacționează cu obiectele grafice ale aplicației;
- "Ascultarea" evenimentelor generate de obiecte în momentul interacțiunii cu utilizatorul și executarea acțiunilor corespunzătoare așa cum au fost ele definite.

Majoritatea obiectelor grafice sunt subclase ale clasei `Component`, clasa care definește generic o componentă grafică care poate interacționa cu utilizatorul. Singura excepție o constituie meniurile care moștenesc clasa `MenuItem`.

Așadar, printr-o componentă sau componentă grafică vom înțelege în continuare orice obiect care are o reprezentare grafică ce poate fi afișată pe ecran și care poate interacționa cu utilizatorul. Exemple de componente sunt ferestrele, butoanele, bare de defilare, etc. În general, toate componentele sunt definite de clase proprii ce se găsesc în pachetul `java.awt`, clasa `Component` fiind superclasa abstractă a tuturor acestor clase.

Crearea obiectelor grafice nu realizează automat și afișarea lor pe ecran. Mai întâi ele trebuie așezate pe o suprafață de afișare, care poate fi o fereastră sau suprafața unui *applet*, și vor deveni vizibile în momentul în care suprafața pe care sunt afișate va fi vizibilă. O astfel de suprafață pe care se așează obiectele grafice reprezintă o instanță a unei clase obținută prin extensia clasei `Container`; din acest motiv suprafețele de afișare vor mai fi numite și *containere*. Clasa `Container` este o subclasă specială a clasei `Component`, fiind la rândul ei superclasa tuturor suprafețelor de afișare Java (ferestre, applet-uri, etc).

Așa cum am văzut, interfața grafică servește interacțiunii cu utilizatorul. De cele mai multe ori programul trebuie să facă o anumită prelucrare în momentul în care utilizatorul a efectuat o acțiune și, prin urmare, obiectele grafice trebuie să genereze evenimente, în funcție de acțiunea pe care au suferit-o (acțiune transmisă de la tastatură, mouse, etc.). Începând cu versiunea 1.1 a limbajului Java evenimentele se implementează ca obiecte instanță ale clasei `AWTEvent` sau ale subclaselor ei.

Un *eveniment* este produs de o acțiune a utilizatorului asupra unui obiect grafic, deci evenimentele nu trebuie generate de programator. În schimb, într-un program trebuie specificat codul care se execută la apariția unui eveniment. Interceptarea evenimentelor se realizează prin intermediul unor clase de tip *listener* (ascultător, consumator de evenimente), clase care sunt definite în pachetul `java.awt.event`. În Java, orice componentă poate "consuma" evenimentele generate de o altă componentă grafică.

14.1. Pachetul `javax.swing`

Deoarece lucrul cu obiectele definite în pachetul `java.awt` este greoi și, pe alocuri, complicat, Java oferă un pachet care se suprapune peste AWT, oferind un API mai „prietenos” programatorului. Denumirea claselor derivă din cele AWT, cărora li s-a adăugat prefixul „J”. De exemplu, clasa `Button` a devenit `JButton`.

În continuare vom prezenta trei exemple care folosesc pachetul `javax.swing` pentru crearea aplicațiilor cu interfață grafică.

14.2. O aplicație cu interfață grafică simplă

CasetaDialog.java

```
import javax.swing.*;

public class CasetaDialog
{
    public static void main(String[] args)
    {
```

```
String str;

/*
 * crearea unei casete apeland metoda showMessageDialog ce accepta 4
 * argumente: referinta catre fereastra parinte, mesajul ce va fi
 * afisat, textul din bara de titlu, constanta pentru indicarea
 * tipului casetei acesta metoda este folosita si pentru a afisa
 * butonul OK pe care utilizatorul il selecteaza pentru a confirma
 * citirea mesajului. atunci cand este apasat mesajul determina
 * inchiderea casetei dedialog
 */

JOptionPane.showMessageDialog(null, "Mesaj", "Casetă",
    JOptionPane.PLAIN_MESSAGE);

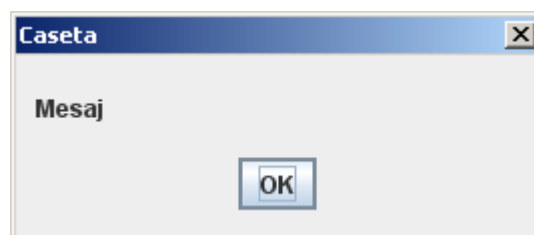
/*
 * o caseta de dialog pentru introducerea datelor este afisata prin
 * apelarea metodei showInputDialog(). argumentul metodei il
 * reprezinta mesajul prin care utilizatorul este invitat sa
 * introduca informatiile in caseta de dialog ce vor fi memorate in
 * variabila str
 */

str = JOptionPane.showInputDialog("Introduceti Studentul");

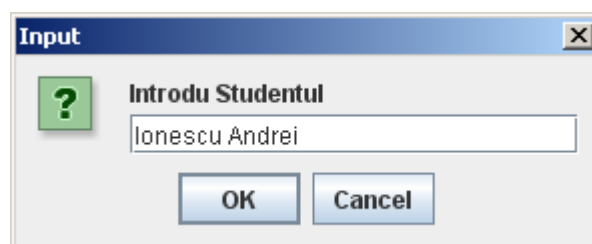
System.out.println(str);
System.exit(0);
}
}
```

Rezultatul rulării:

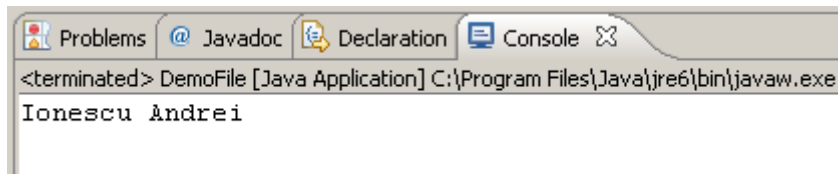
1. O casetă de dialog prin apelarea metodei showMessageDialog():



2. O casetă de dialog pentru introducerea datelor prin apelarea metodei showInputDialog():



3. Continutul șirului str (datele introduse în fereastra de dialog) în consolă:



14.3. Crearea unei casete de dialog

CasetaDialog.java

```
import javax.swing.*;

public class CasetaDialog
{
    public static void main(String[] args)
    {
        String str;

        /*
         * crearea unei casete apeland metoda showMessageDialog ce accepta 4
         * argumente: referinta catre fereastra parinte, mesajul ce va fi
         * afisat, textul din bara de titlu, constanta pentru indicarea
         * tipului casetei acesta metoda este folosita si pentru a afisa
         * butonul OK pe care utilizatorul il selecteaza pentru a confirma
         * citirea mesajului. atunci cand este apasat mesajul determina
         * inchiderea casetei dedialog
         */

        JOptionPane.showMessageDialog(null, "Mesaj", "Caseta",
            JOptionPane.PLAIN_MESSAGE);

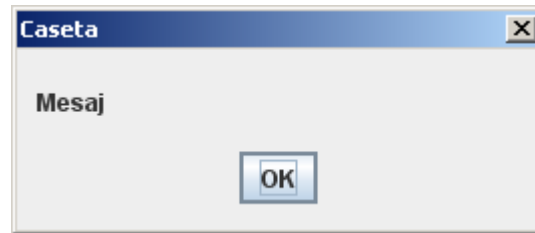
        /*
         * o caseta de dialog pentru introducerea datelor este afisata prin
         * apelarea metodei showInputDialog(). argumentul metodei il
         * reprezinta mesajul prin care utilizatorul este invitat sa
         * introduca informatiile in caseta de dialog ce vor fi memorate in
         * variabila str
         */

        str = JOptionPane.showInputDialog("Introduceti Studentul");

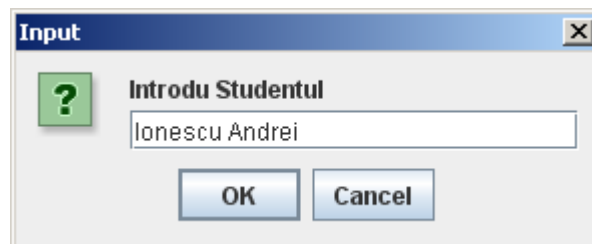
        System.out.println(str);
        System.exit(0);
    }
}
```

Rezultatul rulării:

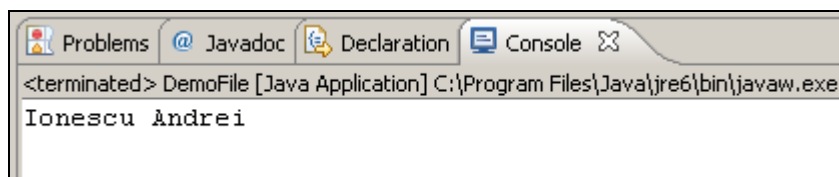
1. O casetă de dialog prin apelarea metodei `showMessageDialog()`:



2. O casetă de dialog pentru introducerea datelor prin apelarea metodei `showInputDialog()`:



3. Conținutul șirului `str` (datele introduse în fereastra de dialog) în consolă:



14.4. Concatenarea a două șiruri de caractere

Fereastra.java

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Fereastra extends JFrame
{
    JPanel contentPane; // container
    JLabel jl_first_text = new JLabel(); // control static de tip text
    JLabel jl_second_text = new JLabel();

    JTextField jtf_editare_text1 = new JTextField(); // control de editare
    JTextField jtf_editare_text2 = new JTextField();
    JTextField jtf_siruri_concatenate = new JTextField();
    JTextField jtf_lungime_sir = new JTextField();
```



```

JButton jb_concateneaza = new JButton(); // controale de tip buton

// JButton jb_reseteaza = new JButton(); (de adaugat in vers. 2)

Fereastra()
{
    initializare();
}

public void initializare()
{
    // stabileste titlul ferestrei
    setTitle("Fereastra Test");

    // se preia in obiectul "ContentPane" containerul principal
    // ferestrei
    contentPane = (JPanel) getContentPane();

    // stabilim tipul de container
    contentPane.setLayout(null);

    // se creeaza componentele ferestrei (obiecte)
    jl_first_text.setText("Primul text");
    jl_first_text.setBounds(10, 10, 130, 22);
    jl_first_text.setFont(new java.awt.Font("SansSerif",
                                             Font.PLAIN, 12));
    jl_first_text.setForeground(Color.black);

    jl_second_text.setText("Al doilea text");
    jl_second_text.setBounds(10, 40, 130, 22);
    jl_second_text.setFont(new java.awt.Font("SansSerif",
                                             Font.PLAIN, 12));
    jl_second_text.setForeground(Color.black);

    jtf_editare_text1.setText("");
    jtf_editare_text1.setBounds(130, 10, 240, 22);
    jtf_editare_text1.setFont(new java.awt.Font("SansSerif",
                                             Font.ITALIC, 12));
    jtf_editare_text1.setForeground(Color.black);

    jtf_editare_text2.setText("");
    jtf_editare_text2.setBounds(130, 40, 240, 22);
    jtf_editare_text2.setFont(new java.awt.Font("SansSerif",
                                             Font.PLAIN, 12));
    jtf_editare_text2.setForeground(Color.black);

    jb_concateneaza.setText("Concateneaza siruri");
    jb_concateneaza.setBounds(150, 70, 200, 22);
    jb_concateneaza.setFont(new java.awt.Font("SansSerif",
                                             Font.BOLD, 12));
}
```

```
jb_concateneaza.setForeground(Color.red);
jb_concateneaza.addActionListener(
    new java.awt.event.ActionListener() {
        // metoda apelata automat de fiecare data cand utilizatorul executa
        // un clic pe un buton
        public void actionPerformed(ActionEvent e)
        {
            concateneazaSiruri();
        }
    });

jtf_siruri_concatenate.setText("");
jtf_siruri_concatenate.setBounds(130, 100, 240, 22);
jtf_siruri_concatenate.setFont(new java.awt.Font("SansSerif",
                                                Font.PLAIN, 12));
jtf_siruri_concatenate.setForeground(Color.black);

jtf_lungime_sir.setText("");
jtf_lungime_sir.setBounds(380, 100, 20, 22);
jtf_lungime_sir.setFont(new java.awt.Font("SansSerif",
                                                Font.PLAIN, 12));
jtf_lungime_sir.setForeground(Color.blue);

// adaugam componentele in conatinerul principal
contentPane.add(jl_first_text, null);
contentPane.add(jl_second_text, null);
contentPane.add(jtf_editare_text1, null);
contentPane.add(jtf_editare_text2, null);
contentPane.add(jb_concateneaza, null);
contentPane.add(jtf_siruri_concatenate, null);
contentPane.add(jtf_lungime_sir, null);
}

public void concateneazaSiruri()
{
    // concatenez continutul celor 2 controale de editare
    String noulSir = jtf_editare_text1.getText() + " "
                    + jtf_editare_text2.getText();

    // setez valoarea in control de editare corespunzator
    jtf_siruri_concatenate.setText(noulSir);

    // setez lungimea sirului(ca text) in ultimul control
    jtf_lungime_sir.setText(String.valueOf(noulSir.length()));

    // resetez valorile din cele 2 controale
    jtf_editare_text1.setText("");
    jtf_editare_text2.setText("");
}
}
```

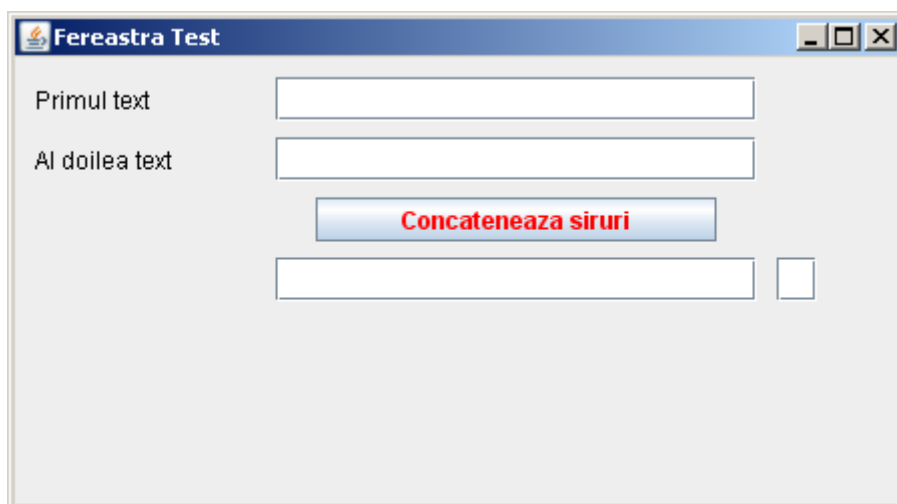
Main.java

```
class Main
{
    public static void main(String[] args)
    {
        // se creeaza un obiect de tip Fereastră
        Fereastră fereastră = new Fereastră();

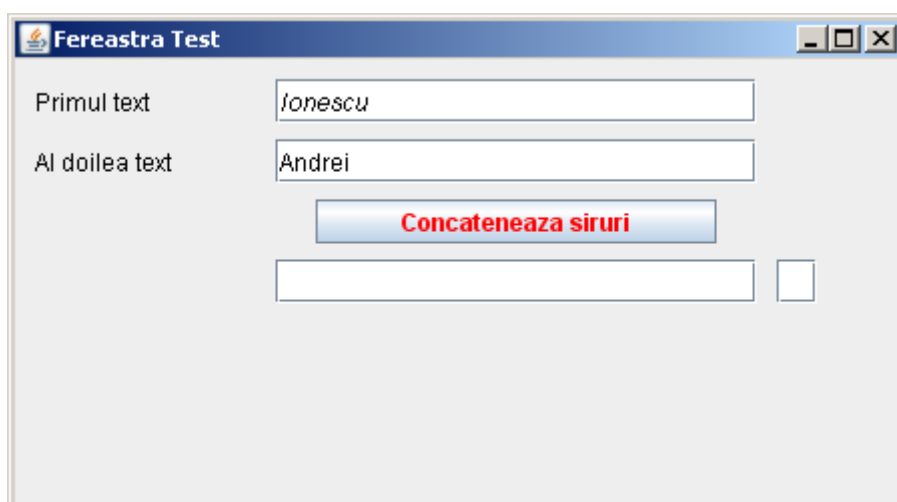
        // stabilește poziția ferestrei pe ecran (colțul stanga-sus)
        fereastră.setLocation(50, 50);

        // stabilește dimensiunea ferestrei
        fereastră.setSize(450, 250);

        // se face vizibilă fereastră creată
        fereastră.setVisible(true);
    }
}
```

Rezultatul rulării:

The screenshot shows a Java Swing window titled "Fereastră Test". Inside the window, there are two text input fields. The first is labeled "Primul text" and the second is labeled "Al doilea text". Below these fields is a button with the text "Concatenează siruri" in red. At the bottom of the window, there is another empty text input field and a small square checkbox.



This screenshot shows the same "Fereastră Test" window after some input. The "Primul text" field now contains the text "Ionescu" and the "Al doilea text" field contains the text "Andrei". The "Concatenează siruri" button remains visible. The bottom text field and checkbox are still empty.



Completați aplicația prin adăugarea unui buton de resetare a casetelor.