Java - Cursuri

Contents

1 Introducere **4**

2 Caracteristicile limbajului java **6**

3 Limbajul Java și principiile programării orientate pe obiecte **9**

4 Mediul de programare Java **11**

5 Compilatorul Java **12**

6 Programul de execuție a applet-urilor **15**

7 Generatorul de documentație **16**

8 Limbajul Java **17**

9 Operatori **21**
9.1 Operatori pentru operații aritmetice 21
9.2 Operatori de atribuire 21
9.3 Operatori de incrementare/decrementare 21
9.4 Operatori de comparație 22
9.5 Operator binar de comparație prin inegalitate 22
9.6 Operatori pentru operații logice 22
9.7 Operatori pentru operații boolean 22
9.8 Operatorul ternar 23

10 Separatori **24**

11 Comentariile **25**

12 Variabile **26**

13 Expresii **28**

14 Codul sursă al programelor java **30**

15 Structuri pentru controlul fluxului **31**
15.1 Structuri condiționale 31
15.2 Structuri de ciclare 32

16 Clase, interfețe, pachete **35**

17 Crearea obiectelor în Java **41**
17.1 Clase interne 42
17.2 Interfețe 44
17.3 Pachete 47

18 Excepții **52**

19 Fire de execuție **56**

20 Operații de intrare, ieșire în Java **68**

21 Serializare **74**

22 Clase pentru interfața grafică **81**

1 Introducere

În anul 1969 au fost conectate pentru prima dată 4 calculatoare aflate la distanță și a fost creată prima rețea de calculatoare din nume, numită ARPANET. Rețeaua s-a dezvoltat în continuare, ajungând în 1972 să conțină 50 de calculatoare. Iar în 1980 conținea circa 80000 de calculatoare. Acestea erau conectate într-o serie de subrețele locale care la rândul lor erau interconectate. Rețeaua acestor rețele a devenit ceea ce numim astăzi internet.

Până în 1992 rețeaua, internetul a fost strict rezervat cercetării și educației. Firmele comerciale nu aveau voie să folosească rețeaue în reclame sau afaceri. În 1992, această interdicție a fost ridicată, ceea ce a dus la o dezvoltare foarte rapidă a internetului. Un strat foarte important al internetului este stratul web. WEB-ul este creația lui Tim Berners-Lee, pe vremea aceia, fizician la laboratorul european pentru fizica particulelor. Tim a propus în 1989 o nouă modalitate de comunicare între fizicienii din întreaga lume, bazată pe informații stocate în rețea sub forma unor documente de tip hypertext.

Pentru a realiza acest lucru era nevoie de un nou limbaj, care să definească format-ul documentelor hypertext și de un nou protocol care să asigure transferul acestor documente în rețea. Tim Berners-Lee a creat noul limbaj inspirat dintr-un limbaj mai vechi, numit SGML și la denumit HTML(hypertext markup language). Acest limbaj furnizează un set de reguli pentru formatarea textului îmbunătățit, reguli asemănătoare celor folosite în criptografii. Noul instrument creat, denumit HTTP (hypertext transfer protocol), este un protocol ce furnizează mijloace internet pentru transfer. Pentru utilizarea celor 2 au fost create niște aplicații client(BROWSER). Primul browser realizat a fost un program în mod text care funcționa pe sistemul de funcționare UNIX. Din cauză că era un browser în mod text, nu a avut succes. Apariția în 1993 a primului browser graphic numit MOSAIC, a condus la o dezvoltare exponențială a WEB-ului. Acest browser afișa imaginile din documentele html și permitea o navigare ușoară prin intermediul mouse-ului prin resursele web-ului. O perioadă, numărul de site-uri web se dubla la 6 luni. În 1995, erau circa 50000 de site-uri. Anul 1995 este un an de referință în revoluția web-ului. În acest an firma microsoft a introdus primele secvențe video în web. Firma Netscape a lansat o versiune îmbunătățită a browser-ului, Netscape Navigation, care era cel mai dezvoltat la vremea respectivă. Acesta permitea plugin-uri. Astfel a început războiul browser. Tot în 1995 au fost finalizate specificațiile limbajului de modelare a realității virtuale (VRML) și s-a luat în discuție implementarea acestuia în web. SUN NICROSYSTEMS a lansat oficial Java în 1995. În 1991 firma SUN intra în topul primelor producătoare din lume de stații UNIX și avea o cifră de afaceri de 2,5 miliarde de dolari. În 1986 cifra de afaceri a firmei era doar de 210 milioane de dolari. Ca să mențină acest ritm ascendent de dezvoltare, conducerea firmei SUN s-a orientat către piața produselor electronice comerciale. Pentru

a pătrunde pe această piață, firma SUN și-a dat seama că trebuie să producă dispozitive mai ieftine, dar la fel de performante ca și concurența. Fiecare astfel de dizpozitiv era controlat de un micro procesor care avea un soft specific. Aceste microprocesoare, aveau arhitecturi diferite. Ei și-au dat seama că pentru a reduce costurile și timpii de producție aveau nevoie de un sistem software unic care să funcționeze fără modificări pe toate arhitecturile de procesor. În acest scop a fost creat grupul de cercetare numit GREEN.

Inițial, programatorii de la GREEN au încercat să dezvolte proiectul în C++, dar au renunțat repede datorită dificultăților mari întâmpinate la modificarea compilatorului. Ei și-au dat seama că aveau nevoie de un limbaj nou construit de la început. James Gosling, programator la GREEN, a început să dezvolte noul limbaj pe care l-a numit OAK. Apoi a fost schimbat în Java. JAMES GOSLING este considerat părintele limbajului Java. Limbajul a fost dezvoltat ulterior în grup și grupul a fost integrat în companie ca grup autonom cu numele de JavaSoft. Numeroși programatori au fost dați ulterior afară de la JAVASOFT. Mediul multiplatformă distribuit, oferit de internet și de web era perfect pentru testarea noului limbaj. Prin urmare, limbajul a fost adaptat pentru web, prin construcția așa numitelor APPLET-uri (programe java ce rulează în web), iar testele au fost de succes. Drept urmare limbajul a fost lansat oficial în 1995. De atunci limbajul a evoluat continuu, fiind limbajul de programare cu cea mai lungă evoluție.

2 Caracteristicile limbajului java

Majoritatea limbajelor de programare actuale se încadrează în una din următoarele 2 categorii:

1. Interpretate
2. Compilate

Limbajele interpretate sunt cele mai ușor de învățat și de utilizat. După ce am scris un program într-un limbaj interpretat, este nevoie de un interpretor adegvat pentru a-l interpreta. Interpretorul parcurge codul sursă instrucțiune cu instrucțiune, le interpretează și le execută.

Prin urmare, un program interpretat, poate fi executat pe orice platformă care furnizează un interpretor adegvat. Deci limbajele interpretate sunt portabile. Faptul că, în timpul execuției, instrucțiunile sursei sunt interpretate, conduce la o latență de execuție. Din acest motiv, majoritatea programatorilor preferă limbajele compilate.

După ce am scris un program într-un limbaj compilat, codul sursă este trecut printr-un compilator, care interpretează instrucțiunile și le transformă în cod mașină. Codul mașină este alcătuit din biți și este scris în limbajul microprocesorului. Astfel, în timpul execuției, microprocesorul execută acest cod mașină, fără să mai aibă loc vreo interpretare, lucru care face ca programele compilate să ruleze mai rapid ca cele interpretate.

Codul mașină respectiv, depinde de arhitectura sistemului, prin urmare este dependent de platformă.

Concluzie: limbajele interpretate sunt portabile, dar lente, iar cele compilate sunt rapide, dar dependente de platformă.

Limbaje compilate:

1. Fortran
2. Cobol
3. Pascal
4. C
5. C++

Ultimele 2 sunt și cele mai folosite.

Limbajul C a apărut la începutul anului 70, și a atras rapid atenția programatorilor, fiind mult mai performant decât celelalte, drept urmare a fost cel mai folosit între 70 și 80. La începutul anului 80 își fac apariția pc-urile și lucrul acesta a dus la o cerere masivă de programe, dar firme de software erau puține,

drept urmare, firmele producătoare de soft își puneau tot mai des problema refolosirii codurilor. S-a constatat însă, că este dificil să adaptezi programele scrise în C. De multe ori este mai ușor și mai rapid de scris un program de la început. Soluția a venit de la o echipă de programatori condusă de BJARNE STROUSTRUP, care propunea o extensiune a limbajului C, numită C++. Noul limbaj, permitea crearea unor module de cod, distincte, care să execute funcții specifice și care puteau fi integrate în orice program. Astfel ia naștere POO. Limbajul C++ este printre cele mai folosite până în prezent.

Limbajul Java constituie o nouă etapă în evoluția limbajelor de programare. A fost construit de experți în C++, care știau toate deficiențele acestui limbaj. Principalele probleme din C++ sunt moștenite din C și sunt legate de folosirea pointerilor și de gestionarea memoriei. În Java, aceste probleme au fost eliminate. Java nu folosește pointeri în mod explicit, iar gestionarea memoriei se face automat, de către Garbage Collector.

Limbajul Java este standardizat. Spre deosebire de C++ care este secvențial (datorită limbajului C) și OOP, limbajul Java este complet orientat pe obiecte.

Limbajul Java este compilat și interpretat.

Compilatorul Java parcurge instrucțiunile codului sursă ale unui program Java, le interpretează și le transformă într-un cod intermediar numit cod de octeți. Codul de octeți mai are nevoie de încă o interpretare în timpul execuției. Întrucât codul de octeți este apropiat ca structură de codul mașină, latența în execuție este mică. Codul de octeți, rulând aproape la fel de rapid ca cel mașină. Problema latenței a fost rezolvată și prin utilizarea compilatoarelor JIT (Just in time) cu care sunt înzestrate majoritatea mediilor de programare Java actuale. Un compilator JIT compilează codul de octeți la cod mașină înaintea execuției, pentru platforma pe care lucrează și execută acel cod mașină prin sistemul de execuție al platformei. Ceea ce face ca programul Java să ruleze la fel de rapid ca unul compilat.

Orice compilator, inclusiv Java, compilează codul sursă pentru o anumită platformă. În cazul programelor Java, această platformă este însă virtuală, construită în memoria calculatorului. Un program special numit interpretor Java, construiește în memoria RAM un alt calculator, numit JVM. Și pot fi executate pe orice platformă reală care furnizează o mașină virtuală Java.

Concluzie: Java e portabil și neutru din punct de vedere arhitectural.

Întrucât Java folosește fire de execuție, spunem că este un limbaj cu execuție multifilară.

Diferența principală dintre mașina reală și mașina virtuală Java, constă în faptul că mașina reală e esențial bazată pe regiștrii, în timp ce, mașina virtuală Java e esențial bazată pe stivă.

Java e un limbaj distribuit. Programele Java pot încărca resurse atât din sistemul local de fișiere, cât și din rețea.

Limbajul Java recunoaște și lucrează cu principalele protocoale de rețea.

Deoarece în Java, gestionarea memoriei se face automat de JVM, accesul la stiva sistemului și la zona de memorie liberă este blocat, ceea ce face din Java un limbaj sigur.

În concluzie, principalele caracteristici ale limbajului Java sunt:

- Este un limbaj standardizat
- Este complet orientat pe obiecte
- Este un limbaj compilat și interpretat
- Este un limbaj neutru din punct de vedere arhitectural și portabil
- Gestionarea memoriei se face automat
- Este un limbaj cu execuție multifilară
- Este un limbaj distribuit
- Este compatibil cu operarea în rețea
- Este un limbaj sigur

3 Limbajul Java și principiile programării orientate pe obiecte

Principala informație adusă de limbajul C++ a fost conceptul de clasă. Clasa este un nou tip de date care generalizează tipul struct din C, și care permite gruparea după anumite criterii logice, a unui număr de variabile și a unui număr de funcții.

Este creat astfel un șablon care poate fi multiplicat într-un program, în oricâte exemplare (în funcție de resursele disponibile). Un exemplar al clasei respective se mai numește Obiect al clasei sau instanță. Variabilele declarate într-o clasă se mai numesc membrii sau atribute. Iar funcțiile declarate într-o closă se mai numesc funcții membre sau metode. Gruparea logică a acestora în clasă se mai numește incapsulare.

Despre o clasă trebuie să știm doar semnificația atributelor ei și acțiunile pe care clasa le poate efectua prin metodele ei. Nu este necesar să știm cum aceste metode au fost implementate în clasă. Știind aceste lucruri, o clasă poate fi utilizată în orice program, astfel realizându-se refolosirea codurilor.

Pentru a folosi un atribut sau o metodă dintr-un obiect al unei clase trebuie să trimitem un mesaj obiectului. În Java, aceste mesaje sunt de tipul obiect.atribut, respectiv obiect.Functie(parametrii).

Elementul limbajului Java este clasa. Clasele pot fi grupate la rândul lor după anumite criterii logice, în biblioteci de clase. În Java, bibliotecile de clase se numesc pachete.

O clasă Java poate fi asemănată cu un program C, în care variabilele programului sunt atributele clasei, funcțiile sunt metodele, iar funcția main, de lansare în execuție este substituită printr-o metodă specială numită constructor. În Java, constructorii nu au tip returnat și au numele identic cu cel al clasei. O clasă poate avea mai mulți constructorii, cu condiția ca aceștia să aibă liste diferite de parametrii. Acest lucru se mai numește supraîncărcarea constructorilor. Fiecare constructor va da un alt tip de inițializare pentru obiectele clasei respective.

Pentru a folosi într-un program un obiect al unei clase, acesta trebuie declarat și inițializat printr-un constructor. Întrucât Java operează numai cu obiecte, un program Java poate fi interpretat ca un sistem format din mai multe programe, care rulează secvențial sau în paralel și care comunică între ele.

Metodele dintr-o clasă pot fi supraîncărcate. Putem avea metode cu același nume, cu condiția ca aceștia să aibă liste diferite de parametrii. Programarea orientată pe obiect permite crearea de noi clase care să extindă funcționalitatea unei clase date, deja construite. Acest lucru se numește moștenire. Clasa dată se mai numește super-clasă directă sau clasă părinte, pentru clasa nou creată, iar clasa nouă se mai numește sub-clasă directă sau clasă derivată sau clasă copil a clasei părinte. Clasa părinte poate avea la rândul ei clase părinți și așa mai

departe. Toate aceste clase see vor numi superclase pentru clasa nou creată. Clasa nou creată va moșteni toate super-clasele ei, în sensul că, putem apela fără restricții atribute și metode din super-clase. Mai mult, aceste atribute și metode moștenite pot fi redeclarate în clasa nou creată, acest lucru numinduse suprascrierea atributelor și metodelor. Limbajul C++ folosește moștenirea multiplă, adică o clasă poate avea mai multe superclase directe. Acest lucru poate conduce însă la confuzie în ceea ce privește originea atributelor și metodelor și chiar la erori de compilare.

Să presupunem că avem A cu 2 super-clase directe B și C, iar B și C au ca super-clasă directă pe D. Presupunem că în D se găsește metoda x. Crearea unui obiect a, de tipul clasei A, va duce la crearea a 2 obiecte, un b de tip B și un c de tip C. Presupunem că executăm a.x(), mediul de execuție va căuta instrucțiunea în a. Nu o găsește, deci o va căuta metoda x în obiectele părinți b și c. Nu o găsește nici aici, deci merge mai sus, și o găsește de 2 ori în D.

Această problemă a fost rezolvată, întrucât Java folosește moștenirea simplă. În Java, fiecare clasă are o unică super-clasă directă. Se construiește astfel o ierarhie a claselor, în vârful căreia se găsește clasa java.lang.Object. Clasa Object este super-clasă pentru toate clasele Java. Și este singura clasă care nu are super-clasă directă. Dacă o clasă Java, nu are specificată super-clasă directă la declarare, atunci, în mod implicit, super-clasa ei directă este object. În programarea orientată pe obiecte, accesul la atribute și metode este controlat prin modificatorii de acces.

În Java avem 3 tipuri de acces:

public: atributele și metodele publice pot fi accesate din orice clasă sau metodă

protejat: atributele și metodele protejate pot fi accesate în clasa respective și în copii acesteia

privat: atributele și metodele private pot fi accesate doar în clasa în care au fost create

4 Mediul de programare Java

Mediul de programare Java se numește JDK (java development kit). Acesta este oferit gratuit (cu excepția utilizărilor comerciale), de către firma Oracle. Limbajul Java a fost lansat oficial în luna mai 1995 sub numele de Java Alfa. Iar până la sfârșitul anului 1995 au mai fost lansate 2 versiune, Java Beta și JDK 1.0. Limbajul a evoluat continuu, ajungând la versiunea 21. Evoluția limbajului s-a manifestat mai ales prin evoluția limbajului Java API (Application Programming Interface). Interfața Java API s-a dezvoltat prin adăugarea de noi clase și pachete, iar clasele deja scrise au fost modificate cu păstrarea compatibilității cu programele deja scrise. În versiunea Java Beta, prin posibilitatea încărcării claselor din arhive. Aceste arhive pot avea extensia zip sau jar (dar sunt zip). Începând cu versiunea JDK 1.0, întreaga interfață Java API a fost comprimată într-o singură arhivă.

Programe executabile la nivel de JDK: Aceste programe se găsesc în directorul JDK/bin. Găsim aici compilatorul, interpretorul, programul de execuție a appleturilor, depanatorul, dezasamblorul, generatorul de documentație.

5 Compilatorul Java

În Java, pot fi scrise 2 tipuri de programe:

- Aplicațiile Java
- Appleturile Java

Aplicațiile Java sunt programe ce vor fi executate prin intermediul mașinii virtuale construite de interpretor. Appleturile sunt acele programe Java care sunt lansate în execuție dintr-un fișier html și sunt executate pe o mașină virtuală Java, integrată într-un browser.

Codul sursă al unui program Java este conținut într-un fișier sursă. Fișierele sursă sunt fișiere text, cu extensia *.java*. Un fișier sursă poate conține o singură declarație de clasă sau mai multe declarații de clasă cu blocuri de cod disjuncte.

Dacă fișierul sursă conține o singură declarație de clasă, numele fișierului trebuie să fie IDENTIC cu cel al clasei declarate în interior (java face distincție între litere mici și mari). Dacă fișierul sursă are mai multe declarații de clasă, obligatoriu una singură trebuie să fie publică, iar numele fișierului va fi identic cu cel al clasei publice. Un program Java, este lansat în execuție prin intermediul unei clase speciale, numită clasă primară. La aplicațiile java, clasa primară este clasa ce conține metoda:

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

La appleturi, clasa primară este clasa care extinde pe java.applet.Applet.

Compilatorul Java, parcurge instrucțiunile codului sursă, le interpretează și le transformă în cod de octeți, furnizând câte un fișier compilat, pentru fiecare clasă declarată în codul sursă. Un astfel de fișier compilat, are numele identic cu al clasei conținute, și extensia .class. Acestea se vor numi fișiere class sau clasă.

Compilatorul Java este programul javac. În windows este javac.exe. Acesta se lansează din linia de comandă, și privește ca parametru numele fișierului sursă cu tot cu extensia java. Sintaxa linie de comandă este:

javac [options] nume_sursă.java

Deși putem compila fiecare clasă separat, este indicat să lansăm compilarea cu clasa primară. Acest lucru va conduce recursiv la compilarea tuturor claselor, deci la compilarea întregului program. În mod prestabilit, compilatorul scrie fișierele clasă alături de cele sursă. Pentru a schimba acest lucru putem folosi opțiunea -D.

Exemplu:
javac -d c:\ clase\ NumeFișier.java

Dacă programul utilizează clase și pachete ce nu aparțin interfeței java API, trebuie specificat la compilare de unde să fie încărcate acestea. Acest lucru se face utilizând opțiunea classpath.

Exemplu:
javac -classpath c:\alteclase\;c:\alteclase1\;D:\class.java; NumeFișier.java

Opțiunea -O conduce la o optimizare a codului de octeți. Aceasta este recomandată doar la clasele foarte mari.

Optimizarea constă într-o aranjare optimă a intrărilor din tabela constantelor clasei respective.

-VERBOSE forțează compilatorul să furnizeze informații despre compilare: Când și de unde este încărcată o clasă, cât a durat compilarea ei etc.

În urma compilării compilatorul poate afișa și unele mesaje de avertisment, acestea nu reprezintă erori. Un mesaj de avertisment frecvent este cel de depreciere a metodelor.

-NOWARN elimină avertismentele

În mod prestabilit, compilatorul inserează în codul de octeți al metodelor, atributul LineNumberTable, acest atribut realizează o corespondență între instrucțiunile codului de octeți, care poate fi folosită în procesul de depanare a metodelor respective cu ajutorul depanatorului jdk. Informațiile respective nu au însă niciun rol în timpul execuției Opțiunea -g determină compilatorul să insereze în codul de octeți al metodelor și atributul LocalVariableTable. Acest atribut furnizează informații detaliate despre fiecare variabilă locală a metodei respective, exemplu: Nume, tip loc de declarare, domeniu de vizibilitate. Aceste informații folosesc tot în procesul de depanare și nu au niciun rol în timpul execuției. În plus, aceste informații pot fi utilizate de către decompilatoarele java, pentru a obține un cod sursă identic cu cel original. Din acest motiv și pentru a obține fișiere compilate mai mici este recomandat să facem compilarea finală a programului cu opțiunea -g.non, în urma căreia, compilatorul nu mai inserezează în codul de octeți cele 2 atribute.

Interpretorul este programul care creează în memoria ram mașina virtuală java, unde va fi încărcat și executat programul java. La lansarea în execuție a unui program java, i se asociază o mașină virtuală java. Fiecare program java are propria mașină virtuală. Interpretorul java este programul java.exe. Se lansează din linia de comandă și primește ca parametru numele clasei primare a programului (fără extensia .class). Linia de comandă pentru lansarea interpretorului este: -¿ java [opțiuni] NumeClasăPrimară [parametrii] Printre opțiuni putem folosi -classpath, care are aceeași semnificație ca la javac. Opțiunea -Xms setează dimensiunea inițială a zonei de memorie liberă.

Exemplu:
-Xms500m

Opțiunea -Xmx setează dimensiunea maximă a memoriei folosită de mașina virtuală. parametrii = o listă opțională de parametrii, de form: parametru1

parametru2 parametru2 unde fiecare parametru înseamnă o secvență de caractere. Acești parametrii vor fi preluați de către mașina virtuală java și vor fi introduși într-o matrice de tipul String, unde

args[0] = parametru1
args[1] = parametru2
args[2] = parametru3 etc.

Iar matricea args va fi dată ca parametru metodei main la lansarea în execuție a programului.

6 Programul de execuție a applet-urilor

Acesta este un program din directorul jdk/bin, utilizat pentru utilizarea APPLET-URILOR în lipsa unui browser. Programul de vizualizare a APPLET-urilor se numește appletviewer, se lansează din linia de comandă și primește ca parametru numele documentului html, care lansează în execuție applet-ul. Sintaxa lui de pe comandă este:

appletviewer [opțiuni] document.html

La fel, se poate folosi opțiunea -classpath cu aceeași semnificație.

În cazul applet-urilor, programul parcurge instrucțiunile codului html și execută doar etichetele APPLET, deschizând câte o fereastră de execuție pentru fiecare applet în parte.

Eticheta APPLET are următoarea structură:

```
<APPLET code="Clasa.class" [codebase="."] [archive="."] width=
   "" height="">
    <PARAM name="." value=".">
    <PARAM name="." value=".">
    <PARAM name="." value=".">
</APPLET>
```

Atributul code este obligatoriu și primește ca valoarea numele clasei primare a APPLET-ului cu extensia .class. Dacă această clasă primară nu se găsește în acelașă folder cu documentul html care lansează applet-ul, trebuie să specificăm calea către clasa primara, în atributul codebase. Valoarea acestui atribut poate fi o cale relativă la poziția documentului html sau poate fi o adresă URL.

În cazul în care clasa primară se găsește într-o arhivă (zip sau jar), numele arhivei va fi scris ca valoare a atributului archive.

Atributele width și height sunt obligatorii și reprezintă lățimea și înălțimea applet-ului.

Între cele 2 etichete applet se pot găsi 0 sau mai multe etichete PARAM. Eticheta PARAM este singulară și folosește la furnizarea parametrilor inițiali applet-ului.

Preluarea parametrilor din lista de PARAMS se face cu ajutorul metodei

```
public String getParameter(String name)
```

definită în clasa Applet. Metoda aceasta parcurge lista de etichete param din eticheta APPLET, identifică acel param pentru care name coincide cu numele parametrului și returnează ca String valoarea atributului value a param-ului respectiv.

7 Generatorul de documentație

Este un utilitar al pachetului jdk folosit la furnizarea de documentație asociată codului sursă. Se numește javadoc.exe. Se lansează din linia de comandă și primește ca parametru numele fișierului sursă cu tot cu extensia java.

javadoc [opțiuni] NumeSursă.java

Opțiuni: -d, -classpath cu aceleași semnificații ca la compilator.

Documentația generată este în formatul html. Programul generează câte un document html pentru fiecare clasă din codul sursă. Acest fișier va conține ierarhia de clase în care se află clasa respectivă, un index al atributelor și un index al metodelor, respective un tabel cu declarațiile tuturor metodelor. Aceste fișiere trebuiesc updatate de către programator.

Programul mai generează 3 fișiere suplimentare numite:

- packages.html (conine link-uri la toate pachetele)
- tree.html (conine ierarhia completă)
- AllNames.html (conine link-uri la toate clasele și metodele din program)

8 Limbajul Java

Elementele sintactice ale limbajului java se mai numesc atomi. Aceștia se împart în următoarele categorii:

- identificatorii
- cuvintele rezervate
- valorile literale
- operatorii
- separatorii
- comentariile

La începutul compilării unei clase, compilatorul elimină mai întâi spațiile care nu fac parte din șiruri de caractere. Extrage terminatorul de instrucțiune și procedează la extragerea atomilor și la interpretarea acestora.

Identificatorii sunt secvențe de litere mici sau mari, din cifre și alte simboluri ce folosesc pentru denumirea atributelor, metodelor, claselor, interfețelor și pachetelor în programele java. Un identificator poate să înceapă cu literă mică sau mare, cu simbolul dolar, sau cu liniuța de subliniere, dar nu poate începe cu o cifră.

Cuvintele cheie/rezervate sunt cuvinte speciale, integrate în limbajul java ce pot fi folosite doar în scopul pentru care au fost implementate. Utilizarea unui cuvânt rezervat va duce la o eroare de sintaxă. Cuvintele rezervate nu pot fi folosite ca identificatori.

Exemplu:
Cuvântul int

În java, toate cuvintele rezervate se scriu cu litere mici. Cuvintele rezervate ale limbajului java sunt:

- Cuvinte rezervate pentru tipuri de date:
  - byte
  - short
  - int
  - long
  - float

  - double
  - char
  - boolean
  - void
- Cuvinte rezervate pentru structuri:
  - class
  - interface
  - package
- Cuvinte rezervate pentru moșteniri:
  - extends
  - implements
- Cuvinte rezervate pentru intrucțiuni condiționale:
  - if
  - else
  - switch
  - case
  - default
  - break
- Cuvinte rezervate pentru structuri de ciclare:
  - while
  - do
  - for
  - continue
  - break
- Cuvinte rezervate pentru excepții:
  - try
  - catch
  - finally
  - throws
- Cuvinte rezervate pentru vizibilitate:

  - public
  - protected
  - private
- Cuvinte rezervate pentru modificatori de tip:
  - final
  - abstract
  - static
  - native
  - transient
  - synchronized
- Cuvinte rezervate pentru valori boolean:
  - true
  - false
- Cuvinte rezervate pentru instanțe predefinite:
  - this
  - super
  - null
- Altele:
  - new
  - return
  - import

Valorile literale sunt secvențe de caractere folosite pentru reprezentarea explicită a datelor în programele java. Acestea se împart în următoarele categorii:

- Numerice întregi
- Numerice cu virgulă mobilă
- Valori boolean
- Valori caracter
- Valori șir de caractere

Valorile literale numerice întregi reprezintă numere întregi stocate pe 32 de biți. Acestea pot fi zecimale, octale sau hexazecimale. Valorile zecimale cu excepția lui 0, încep cu cifre din mulțimea 1-9, urmate de cifre din mulțimea 0-9. Valorile întregi octale, încep întotdeauna cu 0 urmat de cifre octale din 0-7. Valorile întregi hexazecimale încep cu 0x sau 0X urmate de cifre hexazecimale 0-9 și A-F (sau a-f).

Valorile numerice cu virgulă mobilă reprezintă numere cu virgulă și pot fi de tip float (32 biți) sau double (64 biți). Diferența dintre tipuri se face cu sufixul f sau F pentru float și d sau D pentru double. În mod prestabilit, valoarea este de tipul double. Putem folosi și notația științifică:

```
1       3.193E2f
```

Valorile literale boolean sunt reprezentate de cuvintele rezervate true și false.

Valorile literale de tip caracter reprezintă simboluri încadrate de apostrofi.

Exemplu:
'a', 'A', '0'

Pe lângă caractere normale avem și caractere speciale: \', \", \; , \\

\b(backspace): mută cursorul un caracter înapoi

\t(tab orizontal): mută cusorul câteva caractere înainte

\r(carriage return): mută cursorul la începutul rândului

\n(line feed): mută cursorul la începutul rândului

\f(form feed): mută cursorul la începutul paginii

Valorile de tip șir de caractere reprezintă secvențe de simboluri încadrate între ghilimele, dacă o astfel de valoarea este prea lungă, ruperea ei pe mai multe rânduri se face doar prin concatenare. Concatenarea șirurilor de caractere se face cu +.

$$\text{"abc"} + \text{"abc"} \Rightarrow \text{"abcabc"}$$

Într-un șir de caractere putem insera și caractere speciale.

Exemplu:
"Acesta \n este \n exemplu"

Operatorii sunt simboluri speciale integrate în limbajul java, folosite pentru efectuarea operațiilor aritmetice, logice etc. În funcție de numărul de operanzi, operatorii pot fi:

1. unari
2. binari
3. ternari

9 Operatori

9.1 Operatori pentru operații aritmetice

Simbol	Semnificație	Utilizare	Explicație
+	Operator binar de adunare și de concatenare	a + b	Adună a și b și returnează rezultatul sau concatenează valorile de tip string
-	Operator binar de scădere	a - b	Scade a și b și returnează rezultatul
-	Operator unar de negare aritmetică	-a	Returnează opusul lui a
*	Operator binar de înmulțire	a * b	Înmulțește pe a cu b și returnează rezultatul
/	Operator binar de împărțire	a / b	Împarte pe a la b și returnează rezultatul
%	Operator binar de rest	a % b	Returnează restul împărțirii lui a la b

9.2 Operatori de atribuire

Simbol	Semnificație	Utilizare	Explicație
=		a = b	Stochează în a valoarea din b
+=	Adunare și atribuire	a += b	Realizează a + b, apoi stochează rezultatul în a
-=	Scădere și atribuire	a -= b	Realizează a - b, apoi stochează rezultatul în a
*=	Înmulțire și atribuire	a *= b	Realizează a * b, apoi stochează rezultatul în a
/=	Împărțire și atribuire	a /= b	Realizează a / b, apoi stochează rezultatul în a
%=	Împărțire și atribuire	a %= b	Realizează a % b, apoi stochează rezultatul în a

9.3 Operatori de incrementare/decrementare

1. ++
2. –

9.4 Operatori de comparație

1. ==
2. !=

9.5 Operator binar de comparație prin inegalitate

1. ¡
2. ¿
3. ¡=
4. ¿=

9.6 Operatori pentru operații logice

1. &
2. —
3. ˆ
4. ∼→ Unar, de complementariere pe biți
5. <<→ Șiftează pe a cu N poziții, unde N este numărul format din cei mai puțin semnficativi 5 biți ai lui b
6. >>→ Din stânga se completează cu cel mai semnificativ bit al valorii a! Șiftează pe a cu N poziții spre dreapta, unde N este numărul format din cei mai puțin semnficativi 5 biți ai lui b
7. >>>→ Deplasare pe biți spre dreapta fără păstrare de semn. $a >>> b$ Biții valorii a sunt deplasați spre dreapta cu N poziții (N este format din cei mai puțin semnificativi 6 biți ai lui b), din stânga completându-se cu 0.

9.7 Operatori pentru operații boolean

&&: Realizează ȘI boolean

||: Realizează SAU boolean

!: Operator unar de negare booleană

9.8 Operatorul ternar

expr ? v1 : v2;
Substituie structura if else.

10 Separatori

Sunt simboluri integrate în limbajul Java, folosite la segmentarea codului. Separatorii limbajului Java sunt:

{} → Delimitarea blocurilor de cod

() → Folosesc la declararea metodelor, la apelarea acestora și la schimbarea ordinii de evaluare a operatorilor în expresii

[] → Servesc doar la declararea matricilor și la operarea cu acestea

. → Este . zecimal în valori literale cu virgulă mobilă și separator în mesaje

, → Este folosită la declararea variabilelor și a metodelor și la invocarea metodelor

; → Terminator de instrucțiune

' → Definirea valorilor literale de tip caracter

" → Definirea valorilor literale de tip string

11 Comentariile

Sunt simboluri speciale în limbaj ce permit introducerea de text în codul sursă care nu va fi luat în considerare de către compilator:

→ // Comentariu pe o linie

→ /* Comentariu pe mai multe linii */

12 Variabile

O variabilă este o zonă de memorie folosită pentru stocarea temporară a datelor.

O variabilă este caracterizată de un identificator și de un tip de date.

Identificatorul reprezintă numele variabilei, prin care aceasta va fi apelată în program.

Tipul de date ne spune ce fel de date sunt stocate în zona respectivă de memorie.

Limbajul Java folosește 2 categorii de tipuri de date:

- Tipuri de date primitive
- Tipuri de date referință

Tipurile de date primitive reprezintă cea mai simplă formă de date integrată în Java. O variabilă de tip de date primitive va conține întotdeauna valoarea pe care o reprezintă.

Tipurile primitive sunt:

byte → Reprezintă numere întregi cu semn, stocate pe 8 biți, cu valori cuprinse între $-2^7 și 2^7 - 1$;

short → Tipul short reprezintă numere întregi cu semn, pe 16 biți, -2^15 la $2^15 - 1$;

int → Tipul int reprezintă numere întregi cu semn, pe 32 de biți -2^31 la $2^31 - 1$;

long → Tipul long reprezintă numere întregi cu semn, pe 64 de biți -2^63 la $2^63 - 1$

float → Tipul float, numere tip virgulă mobilă, 32 biți;

double → Tipul double, pe 64 biți;

char → Tipul char reprezintă numere întregi, fără semn, pe 16 biți, care sunt coduri de caractere UNICODE. Limbajul Java folosește pentru reprezentarea caracterelor, sistemul UNICODE. UNICODE este un sistem internațional, standardizat, care codifică caracterele pe 16 biți (peste 65 de mii de simboluri)

boolean → O variabilă de tip boolean poate fi încărcată cu cele 2 valori literale true și false. Spre deosebire de alte limbaje, în java true și false nu coincide cu 1 și 0. Acest lucru se întâmplă însă, la nivelul codului de octeți, unde true este 1 pe 32 biți și false e 0 pe 32 biți.

Observație:
Dacă într-o operație intervin 2 valori de tip numeric întreg de lungimi diferite, variabila cu lungimea mai mică va fi extinsă la lungimea celei mai mari. Dacă într-o operație algebrică în care avem o variabilă de tip numeric întreg este depășit domeniul tipul respectiv, nu se generează o eroare ci variabila va fi rulată în capătul celălalt. Contorizând aceste parcurgeri, putem face calcule cu numere foarte mari.

Putem înlocui true și false cu 1 și 0 prin 2 metode:

1. ```
   int boolean2int(boolean b) { return b ? 1 : 0; }
   ```

2. ```
   boolean int2boolean(int i) { return i != 0; }
   ```

Tipurile de date referință reprezintă referințe (adrese) la obiecte sau matrici.

Pentru a fi folosită, o variabilă trebuie declarată în program, declarația respectivă respectă următoarea sintaxă.

tip_date nume [= valoare];

Unde nume este identificatorul, tip_date poate fi un tip primitiv sau un tip referință.

Valoarea este folosită pentru inițializarea variabilei respective. Poate fi:

- Valoare literală, de același tip cu tip date, când tip date e primitiv;
- Expresie, care returnează o valoare de tip tip_date, această expresie poate conține și alte variabile cu condiția să fie declarate anterior;
- new NumeClasa([parametrii]), când tip_date este tip clasă;
- new tip_date[d1][d2]...[dn], unde d1,...dn sunt numere întregi ¿= 1, reprezentând dimensiunile matricei;

Exemple:

```
1  int i = 10;
2  long l1 = 3, l2, l4 = 1000;
3  l2 = 100;
4  byte b = 8;
5  i = b + i;
6
7  char c = 'a';
8  boolean b1 = false;
9
10 String s = "abc";
11 System.out.println(s.length());
12
13 java.awt.Color x = new java.awt.Color(1, 1, 1);
14
15 int[] m1 = new int[10];
```

Matricile de tip de date primitive vor fi inițializate implicit. Dacă tipul e numeric, toate elementele vor fi inițializate cu 0. O matrice poate fi inițializată și prin valoare, astfel:

```
float[] m2 = {3.1f, 3.14f};
System.out.println(m2.length);
java.awt.Color[] m3 = new java.awt.Color[3];
```

În această situație mașina virtuală nu alocă memorie decât pentru m3. Memoria va fi alocată la crearea fiecărui element în parte. System.out.println(m3[0]) va genera o eroare de sintaxă, căci elementul nu există.

```
m3[0] = new java.awt.Color(0, 0, 0);
m3[1] = new java.awt.Color(255, 0, 0);
m3[2] = new java.awt.Color(37, 56, 151);

java.awt.Color[] m4 = { new java.awt.Color(0, 0, 0), new java.awt.
   Color(37, 56, 151)};

int[][] m = new int[10][8];
```

O matrice n dimensională este defapt o matrice 1 dimensională, de matrici $n-1$ dimensionale. Din acest motiv, la declararea unei matrici n dimensionale, pot fi precizate doar primele m dimensiuni, unde $m <= n$;

```
int[][] m5 = new int[3][];
// System.out.println(m5[2][3]);  eroare
// m[0], m[1], m[2] trebuiesc initializate
m[0] = new int[100];
m[1] = new int[1000];

char[][][] c1 = {
        {{'a', 'b', 'c'}, {'b', 'd', 'e'}},
        {{'f', 'g', 'c'}, {'q', 's', 'r'}}
};
System.out.println(c1[1][1][1]);
```

13 Expresii

Expresiile constituie modalitatea de efectuare a calculelor în java, o expresie este o combinație de variabile cu valori literale, operatori, separatori și invocări de metode, care în urma evaluării returnează o valoare. Tipul valorii respective se numește și tipul expresie. O expresie care returnează o valoare boolean, se numește expresie booleană. O expresie ce returnează un obiect de tipul clasei String se numește expresie de tip string etc.

Evaluarea într-o expresie se face de la stânga spre dreapte, iar evaluarea operatorilor respectă ordinea de precedență:

1. [], ()

2. -(unar), ++, !, ∼
3. new, conversii
4. *, /, %
5. +, -
6. >>, <<, >>>
7. <, >, <=, >=
8. ==, !=
9. &
10. |
11. ˆ
12. &&
13. ||
14. ?
15. =, operator_=

Ordinea de evaluare a operatorilor în expresii poate fi schimbată cu ajutorul parantezelor rotunde.

14 Codul sursă al programelor java

Codul sursă al programelor java este alcătuit din instrucțiuni. Instrucțiunile pot fi grupate la rândul lor, în blocuri de cod.

Un bloc de cod este format din 0 sau mai multe instrucțiuni delimitate de acolade. Instrucțiunile unui bloc de cod, acționează ca o singură instrucțiune compusă.

Vom folosi următoarea convenție:

- instrucțiune = instructiune;
- Bloc de cod = {
  instrucțiune;
  ...
  instrucțiune;
  }
- instructiuni;

Prin instrucțiuni, înțelegem ori o singură instrucțiune, ori un bloc de cod. Locul unde se declară o variabilă în program, marchează începutul domeniului de vizibilitate al variabilei respective. Domeniul de vizibilitate al variabilei, este porțiunea de cod în care variabila este recunoscută Domeniul de vizibilitate începe cu instrucțiunea de declarare și se termină la sfârșitul blocului de cod în care a fost declarată variabila. Excepție de la această regulă o fac atributele (variabile claselor).

15 Structuri pentru controlul fluxului

Permit programelor să ia decizii și să execute instrucțiuni după anumite criterii date. Se împart în:

1. Conditionale
2. De ciclare

15.1 Structuri condiționale

Structurile condiționale permit programelor să execute instrucțiuni după o condiție:

- if
- else
- else if

```
if (conditie)
    instructiuni();
else
    instructiuni();
```

if else poate fi substituit cu operatorul ternar ?.

```
int v = (int)(100 * Math.random());
if (v % 2 == 0)
{
    System.out.println("PAR");
}
else
{
    System.out.println("IMPAR");
}

// SAU

System.out.println((v & 1) == 0 ? "PAR" : "IMPAR");
```

Structure if else permite și ramificare multiplă:

```
if (conditie1)
    instructiuni1();
else if (conditie2)
    instructiuni2();
...
else if (conditien)
    instructiunin();
else
    instructiuni();
```

De regulă ramificarea multiplă se face cu switch:

```
switch(expresie)
{
    case val1:
        instructiune();
        ...
        instructiune();
        [break;]
    ...
    case valn:
        instructiune();
        ...
        instructiune();
        [break;]
    [default:
        instructiune();
        ...
        instructiune();]
}
```

15.2 Structuri de ciclare

Execută o instrucțiune sau un block de instrucțiuni repetitiv până când o anumită condiție este îndeplinită. În Java, structurile de ciclare sunt:

1. while
2. do while
3. for

Structura while are următoarea sintaxă generală:

```
while(conditie)
    instructiuni;
*
```

Unde condiție e o expresie boolean.

așina virtuală Java evaluează expresia condiție. Dacă aceasta returnează true, execută instrucțiuni; iar procesul se repetă. Când condiție returnează false, fluxul continuă cu instrucțiunea *.

Structura do while are următoarea sintaxă:

```
do
    instructiuni;
while (conditie);
*
```

Mai întâi execută, apoi verifică condiție. Dacă este true, instrucțiunile sunt executate din nou. Când condiția e false, codul continuă de la *.

Diferența dintre do while și while este că la do while, instrucțiuni; se execută cel puțin odată. La while e posibil să se execute cel puțin odată.

For:

```
for (initializare; conditie; modificare)
    [instructiuni;]
*
```

În partea de inițializare se declară 0 sau mai multe variabile. Apoi, mașina virtuală evaluează expresia condiție, iar dacă aceasta e true, execută instrucțiuni; Apoi execută partea de modificare, unde variabilele implicate sunt modificate, apoi evaluează din nou condiție etc.

Când condiție e false, execuția continuă cu *.

Instrucțiunea break, determină ieșirea forțată dintr-un ciclu.

Exemplu:

```
while (conditie)
{
    ...
    if (test) break;
    ...
}
```

Când test returnează true, break va opri ciclul.

Instrucțiunea continue determină reluarea ciclului.

Exemplu:

```
while (conditie)
{
    *
    ...
    if (test) continue;
    ...
}
```

Dacă test e true, se continuă de la *(se reevaluează condiție).

```
et:
while (conditie)
{
    ...
    if (test) break et;
    ...
}
```

Dacă test returnează true, se sare la eticheta et.

Exemplu:

```
import java.awt.*;
import java.io.IOException;
```

```
3
4  public class Main {
5      public static void main(String[] args)
6      {
7          System.out.println(citeste_numar());
8
9      }
10
11     // Exemplu while
12     private static String citeste_numar()
13     {
14         String nr = "";
15         int c = 0;
16         boolean b1 = true;
17         boolean b2 = true;
18         while (true)
19         {
20             try
21             {
22                 c = System.in.read();
23             }
24             catch (IOException e)
25             {
26                 return "";
27             }
28
29             if (c == '\n')
30             {
31                 break;
32             }
33             if (c == '0' && b1)
34             {
35                 nr = "0";
36                 break;
37             }
38             if (c == '-' && b1)
39             {
40                 nr += (char)c;
41                 b1 = false;
42                 continue;
43             }
44             if (c == '0' && !b1 && b2)
45             {
46                 nr = "0";
47                 break;
48             }
49             b2 = false;
50             if (c < '0' || c > '9')
51             {
52                 continue;
```

```
53              }
54              nr += (char)c;
55          }
56          return nr;
57      }
58
59      // Exemplu for
60      private static double Arie(Point[] P)
61      {
62          double arie = 0.0;
63          int n = P.length;
64          for (int i = 0; i < n - 1; i++)
65          {
66              arie += (P[i].x + P[i + 1].x) * (P[i].y - P[i + 1].y);
67          }
68          arie += (P[n - 1].x + P[0].x) * (P[n - 1].y - P[0].y);
69          return arie / 2.0;
70      }
71  }
```

16 Clase, interfețe, pachete

O clasă este o structură de date formată din declarația clasei și blocul de cod al clasei.

Declararea unei clase respectă următoarea structură generală:

```
1  [modificatori] class NumeClasa [extends NumeSuperClasa]
2  {
3
4  }
```

Modificatori := modificator, modificator Modificator de acces(public sau acces prestabilit) și modificator de tip (abstract sau final).

Public declară o clasă publică. O clasă publică poate fi accesată din orice altă clasă, din orice pachet.

Dacă modificatorul public lipsește, accesul la clasă este cel prestabilit. În acest caz, clasa este accesibilă din subclasele ei și din clasele din același pachet.

Abstract declară o clasă abstractă. Servesc deobicei ca superclase. O clasă abstractă poate avea atribute și metode implementate, dar și metode abstracte. Metodele abstracte pot fi declarate doar în clase abstracte.

Modificatorul final declară o clasă finală. O clasă finală nu poate avea subclase. Deci o clasă nu poate fi simultan abstractă și finală. Cuvântul rezervat class este obligatoriu și el marchează declararea unei clase. Acesta este urmat obligatoriu de un identificator NumeClasă, care reprezintă numele clasei respective.

Numele clasei este urmat optional de cuvântul extends. Acesta marchează moștenirea în java. Cuvântul extends este urmat de un identificator(numele

superclasei respective). Clase NumeSuperClasă se va numi superclasa directă a clasei respective. Iar clasa NumeClasă se va numi subclasă direct sau clasă derivată a lui NumeSuperClasă. Dacă în declarația clasei extends lipsește superclasa directă a clasei respective va fi java.lang.object.

NumeSuperClasă are la rândul ei o superclasă directă.

Se construiește astfel o ierarhie a claselor în vârful căruia se găsește java.lang.object. Toate aceste clase se numesc superclase ale clasei declarate. Clasa declarată va moșteni toate superclasele ei. În sensul că, putem apela metode și atribute din superclase fără restricții. În plus, atributele și metodele moștenite din superclase pot fi suprascrise în noua clasă. La crearea unui obiect al unei clase, mașina virtuală va crea și un obiect al superclasei directe și așa mai departe. La apelarea unei metode din obiectul curent, mașina virtuală va căuta mai întâi metoda în obiectul respectiv. Dacă nu o găsește, va căuta metoda în superclasa direct, etc. Când este găsită va fi executată. Declarația unei clase este urmată obligatoriu de blocul de cod. Blocul de cod al unei clase poate conține attribute și/sau declarații de metode. Atributele unei clase sunt acele variabile din blocul de cod al clasei ce nu aparțin niciunui block de cod inclus în clasa respectivă. Atributele pot fi declarate oriunde în clasă.

Sintaxa general de declarare a unui atribut este:
[modificatori] tip_date nume [= valoare];
Modificatori de acces sau de tip.
Modificatori de acces:

public → declară un atribut public, acesta este accesibil din orice clasă, din orice pachet;

protected → declară un atribut protejat, acesta este accesibil în clasa respective și în subclasele ei;

private → declară un atribut privat, acesta este accesibil doar în clasa în care a fost declarat;

acces prestabilit → dacă modificatorul de acces lipsește, accesul la atribut este cel prestailit, adică poate fi accesat din clasa respectivă, din subclasele ei și din clasele din același pachet;

Modificatori de tip:

static → Declară un atribut static. Acesta ocupă o singură zonă de memorare comună zonei respective; Dacă un obiect modifică valoarea unui atribut static al clasei, un alt obiect va citi aceeași valoare. Întrucât atributele statice nu depind de obiecte, ci doar de clasă, acestea se apelează prin mesaje de forma: NumeClasă.nume_atribut_static Un atribut care nu este static, se numește atribut de instanță. Atributele de instanță vor avea câte o copie în fiecare obiect al

clasei respective. Același atribut de instanță poate avea valori diferite în obiecte diferite. Întrucât atributele de instanță depind de obiecte, acestea se apelează prin obiect.nume_atribut

final → Nemodificabil;

Atributele publice, statice și finale sunt constantele limbajului java.
tip_date:

- primitive
  - byte
  - short
  - int etc.
- referință
  - clasă
  - matrice

```
class A
{
    public static final double PI = Math.PI * 2;
    public double PI = 3.1415;
}

clasa B
{
    public static void main(String[] args)
    {
        System.out.println(A.PI2);
        System.out.println((new A()).PI);
    }
}
```

Dacă un atribut nu este inițializat explicit, va fi inițializat implicit. Dacă e număr, va fi inițializat cu 0, dacă e Boolean, cu false. Dacă e referință, va fi inițializat cu null.

O metodă în java are următoarea sintaxă generală:

```
[modificatori] tip_returnat nume([parametrii])
{
    ... // blocul de cod
}
```

Regulă de notare: De obicei clasele, interfețele se numesc cu litere mari, iar pachetele și membrii clasei se pun cu litere mici.

Modificatori: de acces sau de tip.

Cei de acces sunt ca la atribute.

Modificatori de tip:

static → O metodă statică ocupă o singură zonă de memorie, comună tuturor obiectelor clasei respective. Prin urmare o metodă statică se apelează prin numele clasei și numele metodei. O metodă care nu este statică, se numește metodă de instanță. Metodele de instanță au câte o copie în fiecare obiect al clasei respective. Prin urmare o metodă de instanță se apelează prin obiect.nume_metoda;

abstract → Modicatorul abstract declară o metodă abstractă. Conține doar declarația terminate cu ; și nu are bloc de cod. Metodele abstracte pot fi declarate doar în clase abstracte.

! Dacă o clasă are între superclasele ei o superclasă abstractă, atunci obligatoriu, trebuie să conțină toate declarațiile de metode abstracte din superclasa ei. Acestea nu vor avea modificatorul abstract și vor avea și bloc de cod, chiar dacă e vid.

final → declară o metodă finală. O metodă finală nu poate fi suprascrisă în subclase;

native → declară o metodă nativă.

Limbajul java, pe lângă metodele java standard, mai folosește și metode native, Acestea sunt scrise în c++, sunt compilate la cod mașină pentru o anumită platformă și sunt stocate în biblioteci cu legătură dinamică. Metodele java standard sunt executate prin mecanismul de execuție al mașinii virtuale java, bazată pe stiva de operanzi, pe când metodele native sunt executate prin mecanismul de execuție al platformei respective, bazat pe regiștrii. Prin urmare, metodele native vor avea o execuție mai rapidă decât cele standard, dar utilizarea lor, duce la pierderea portabilității programului.

Metodele native permit acces la funcțiile API ale sistemului de operare, permit acces la drivere, la dispositive periferice etc.

Pentru a utiliza într-o clasă metode native dintr-o bibliotecă cu legătură dinamică, trebuie să-I "spunem" mașinii virtuale de unde să le încarce.

Se face cu System.loadLibrary(path);

Unde path este un String ce reprezintă calea absolute a bibliotecii cu legătură dinamică în sistemul local de fișiere sau doar numele bibliotecii, atunci când aceasta se găsește în același director cu clasa respectivă.

În clasa respectivă, funcțiile ce urmează a fi utilizate, trebuiesc redeclarate. Declarațiile vor fi identice cu cele din biblioteca cu legătura dinamică, vor avea în față modificatorul native și se vor termina ; deci nu conțin bloc de cod.

Aceste metode native pot fi apelate în program la fel ca celelalte metode java.

Tipul returnat reprezintă tipul de date al valorii returnate de metodă.(tip_date/void)

Dacă funcția nu returnează nicio valoare, tipul returnat este void.

Returnarea dintr-o metodă se face cu ajutorul tipului returnat return;

Dacă funcția e void, utilizarea lui return este opțională. Aceasta poate fi folosită pentru ieșirea din metodă, în urma unui test;

if (test) return;

Dacă return lipsește, ieșirea din metodă se face la terminarea blocului de cod din metodă.

Dacă metoda returnează o valoare, utilizarea lui return este obligatorie și trebuie să acopere toate situațiile posibile.

return val; // Unde val poate fi valoare literală, variabilă, expresie de tip_date.

Exemplu:

```
double exemplu(int i)
{
    switch (i)
    {
        case 1:
        return Math.PI;
        case 2:
        return Math.E;
    }
    return 0;
}
```

Nume este identificatorul metodei prin care aceasta este apelată în program. Numele metodei este urmat obligatoriu de perechea de paranteze rotunde.

// parametrii = parametru, parametru, parametru

// parametru = tip_date nume_parametru

Numele parametrilor trebuie să fie unici.

În Java, metodele pot fi supraîncărcate, adică, în aceeași clasă putem defini mai multe metode cu același nume, cu condiția ca acestea să aibă liste diferite de parametrii.

În limbajul Java există 3 instanțe predefinite:

this → referința la obiectul curent

super → reprezintă referința la obiectul curent al super clasei directe

null → reprezintă referința la niciun obiect

Într-o metodă putem avea parametrii sau variabile locale cu aceleași nume cu ale unor atribute. În acest caz, în blocul de cod al metodei respective vor avea prioritate parametrii sau variabilele locale respective și nu atributele. Pentru apelarea atributului cu același nume putem folosi this.atribut

Dacă într-o clasă am suprascris atribute și metode din super-clasa directă. Pentru apelarea atributelor și metodelor respective definite în superclasă, acestea trebuie precedate de super.

super.atribut
super.metodă()

Referința null poate fi încărcată în orice variabilă de tip referință. Dacă un atribut de tip referință nu este inițializat explicit la declarare, va fi inițializat implicit cu null.

Exemplu:

```
1  class A
2  {
3      int i = 10;
4      void a()
5      {
6          System.out.println("a");
7      }
8
9      void a (String s)
10     {
11         System.out.println(s);
12     }
13 }
14
15 class B extends A
16 {
17     int i = 100;
18     void a()
19     {
20         System.out.println("b");
21     }
22
23     void a(int i)
24     {
25         if (i < 30)
26         {
27             System.out.println(i);
28         }
29         else if (i < 50)
30         {
31             System.out.println(this.i)
32         }
33         else
34         {
35             System.out.println(super.i);
36         }
37     }
38
39     public static void main(String[] args)
40     {
41         new B();
42     }
43
```

```
44      B()
45      {
46          a();
47          super.a();
48          a("10");
49          a(10);
50          a(40);
51          a((int)(Math.random() * 100));
52      }
53  }
```

// Afisare:
b
a
10
10
100
?

17 Crearea obiectelor în Java

În limbajele procedurale datele și codul reprezintă entități distincte. În Java, datele și codul reprezintă aceeași entitate, fiind încapsulate împreună în obiecte.

String s = "ABC";

Conține atât datele "ABC" cât și funcții de procesare.

System.out.println(s);

s.length(), s.startswith("...");

s.indexOf("B");

În programele procedural, memoria se alocă la începutul execuției și rămâne alocată până la finalul execuției.

În Java memoria se alocă doar când un obiect este creat și rămâne alocată doar cât timp obiectul este referit în program. În Java, obiectele sunt create cu new.

NumeClasă nume_variabila = new NumeClasă([parametrii]);

La execuția acestei instrucțiuni, mașina virtuală procedează astfel:

1. Alocă suficientă memorie, din zona de memorie liberă, pentru noul obiect;

2. Verifică dacă în memorie se mai găsește un obiect de tipul clasei respective;

3. Dacă în memorie nu găsește niciun obiect de tipul clasei respective, vor fi inițializate atributele statice ale clasei. Acestea pot fi inițializate prin valori literale, prin expresii ce conțin și alte atribute statice sau în cadrul unui bloc static.Blocul static servește la inițializările complexe ale atributelor

statice care presupun mai multe instrucțiuni. O clasă poate conține un singur bloc static;

```
static
{
    // bloc de cod;
}
```

La nivelul codului de octeți, blocul static este interpretat ca o metodă statică ce are numele ¡clint¿

4. Sunt inițializate atributele de instanță ale obiectului;
5. Este executat un constructor al clasei respective;
6. Este creat un obiect al superclasei directe și se reiau pașii 1-5 pentru aceștia;
7. Este alocată memoria cu numele nume_obiect și încărcată cu referința la obiectul nou creat

Constructorii sunt metode speciale ale clasei respective, folosite la inițializare.
Un constructor are numele identic cu cel al clasei și returnează void, dar tipul void este implicit și nu trebuie precizat în declarație.
La fel ca metodele, constructorii pot fi supraîncărcați. O clasă poate avea mai mulți constructori, doar să aibă liste diferite de parametrii.
Fiecare constructor va da un alt tip de inițializare pentru obiectele clasei respective. La nivelul codului de octeți, constructorii sunt niște metode ale clasei care au numele ¡init¿.
Dacă o clasă nu are declarat în mod explicit niciun constructor, putem crea totuși obiecte de tipul clasei respective. Se va apela constructorul implicit.
// LIPSURI DIN CURSUL 6

17.1 Clase interne

...
Exemplu: Fiserul C.java

```

class A
{
    String a = "a";
}

class B
{
    String b = "b";
}
```

```
11
12 public class C extends B
13 {
14     public static void main(String[] args)
15     {
16         new C();
17     }
18
19     C()
20     {
21         (new CA()).x();
22         System.out.println(a + b);
23     }
24
25     class CA extends A
26     {
27         void x()
28         {
29
30         }
31     }
32
33     class CD
34     {
35
36     }
37 }
```

A.class
B.class
C.class
C.$CA.class
C.$CD.class

17.2 Interfețe

Moștenirea multiplă este posibilă în java prin utilizarea interfețelor. Interfețele se aseamănă cu clasele abstracte, cu diferența că, Interfețele nu pot conține metode implementate. Prin urmare, interfețele nu pot avea constructori și deci nu pot fi instanțiate (nu putem avea obiecte de tip interfață).

Interfețele permit dezvoltarea modulară a programelor java, adică putem dezvolta doar o parte a unui program, în timp ce o altă parte rămâne la stadiul de schelet, construită cu ajutorul interfețelor.

Interfețele se declară cu ajutorul cuvântului rezervat "interface", după următoarea sintaxă generală:

[public] interface NumeInterfata [extends Interfata1, Interfata2,..., Interfatan] { // Bloc de cod }

Modificatorul public declară o interfață publică. O interfață publică este accesibilă din orice clasă/interfață din orice pachet. Dacă el lipsește, interfața va avea acces prestabilit. În acest caz, interfața este accesibilă sub-interfețelor ei și din clasele și interfețele din același pachet.

NumeInterfata este un identificator ce reprezintă numele prin care interfața este apelată în program, numele interfeței este urmat opțional de extends, prin care se declară moștenirea. Cuvântul extends este urmat obligatoriu de o listă de nume de interfețe separate prin virgule, cu cel puțin un element.

Interfata1, Interfata2, ... Interfatan se vor numi super-interfețe directe ale interfeței declarate. Dacă extends lipsește, interfața NU are nicio super-interfață directă în mod implicit.

Interfata1, Interfata2, ... Interfatan pot avea la rândul lor, super interfețe directe. Toate aceste interfețe se mai numesc super-interfețe pentru interfața declarată. Nu se construiește o ierarhie a interfețelor ca în cazul claselor.

Interfața declarată moștenește toate super-interfețele ei.

Blocul de cod al interfeței poate conține declarații de atribute și de metode. Un atribut declarat într-o interfață este în mod implicit public, static și final. (deci este o constantă Java).

Modificatorii public, static și final nu trebuie scriși în declararea atributului.

Un atribut dintr-o interfață poate fi declarat printr-o valoare literală sau printr-o expresie(aceasta poate conține și alte atribute din interfață, declarate înainte sau atribute moștenite din super-interfețe).

O metodă declarată într-o interfață este în mod implicit publică și abstractă. Dar aceștia nu trebuie precizați explicit în declararea metodelor.

Pentru a utiliza o interfață într-o clasă, aceasta trebuie implementată, cu cuvântul "implements", după următoarea sintaxă generală:

[modificatori] class NumeClasa [extends SuperClasa] [implements Interfata1, Interfata2, ... Interfatan]

{

}

Cuvântul implements trebuie urmat obligatoriu de o listă de interfețe separate prin virgulă.

În acest caz, în blocul de cod al clasei declarate, trebuie să declarăm obligatoriu toate metodele din Interfata1, Interfata2, ... Interfatan și din toate superinterfețele lor. Acestea vor avea și bloc de cod, chiar dacă acesta este vid.

Exemplu: Exemplu.java

```
1
2  interface A
3  {
4      int i = 100;
5      int[] j = {110, 1110, 2 * i};
6
7      void a(String s);
8      String a();
9  }
10
11 interface B
12 {
13     int k = 1000;
14
15     int b();
16
17     String b(int i);
18 }
19
20 interface C : extends A, B
21 {
22     int d = j[1] + k * 10;
23     void c(String s);
24 }
25
26 interface D
27 {
28     float d();
29
30     char d(char c);
31 }
32
33 class S
34 {
35     void s(String a)
36     {
37         System.out.println(a);
38     }
39 }
40
41 public class Exemplu extends S implements C, D
42 {
```

```
43      // Interfata C
44      void c(String s)
45      {
46
47      }
48
49      // Interfata A
50      void a(String s)
51      {
52
53      }
54
55      String a()
56      {
57          return "abc";
58      }
59
60      // Interfata B
61      int b()
62      {
63          return 0;
64      }
65
66      String b(int i)
67      {
68          return null;
69      }
70
71      // Interfata D
72      float d()
73      {
74          return 0.0f;
75      }
76
77      char d(char c)
78      {
79          return '0';
80      }
81  }
```

În urma compilării lui Exemplu.java, compilatorul creează fișierele
A.class
B.class
C.class
D.class
S.class
Exemplu.class

Interfețele pot fi date ca parametru în metode, urmând ca la apelarea metodelor respective, să dăm ca parametru pe poziția respectivă un obiect de tipul unei

clase care implementează interfața.

Exemplu: Exemplu.java

```
1  interface A
2  {
3      float pi = 3.14f;
4      void a(String s);
5  }
6
7  class B implements A
8  {
9      void a(String s)
10     {
11         System.out.println(s);
12     }
13 }
14
15 public class Exemplu
16 {
17     void ex(A a)
18     {
19         a.a("a");
20     }
21
22     void ex()
23     {
24         ex(new B());
25     }
26
27     Exemplu()
28     {
29         ex();
30     }
31
32     public static void main(String[] args)
33     {
34         new Exemplu();
35     }
36 }
```

17.3 Pachete

Bibliotecile limbajului Java se numesc pachete.

Pachetul reprezintă cea mai mare unitate logică în Java.

Pachetul e format prin gruparea după anumite criterii logice, a unui număr de clase și/sau interfețe.

Pachetele se folosesc pentru o gestionare mai bună a claselor și interfețelor și a accesului la acestuia. Pachetele se declară cu ajutorul cuvântului rezervat

“package”, după următoarea sintaxă generală:

package nume;

Unde nume este un identificator ce reprezintă numele pachetului și se scrie, de obicei, cu litere mici. Această instrucțiune trebuie să fie prima instrucțiune dintr-un fișier sursă și poate fi precedată doar de comentariu.

Dacă fișierul sursă conține mai multe declarații de clasă, instrucțiunea package este valabilă pentru toate. Clasele care conține această instrucțiune trebuie plasate într-un director cu denumirea ”nume”.

Limbajul Java suportă și ierarhia de pachete. Aceasta se declară cu următoarea sintaxă:

package p1.p2. ... .p(n-1).p(n)

Clasele ce conțin această instrucțiune se găsesc în pachetul p(n), care este subpachet al lui p(n - 1), etc.

Pentru a utiliza într-o clasă clase și interfețe dintr-un anumit pachet, trebuie să-i spunem mașinii virtuale unde se găsesc acestea. Acest lucru se face cu instrucțiunea “import” cu sintaxa:

1. import p.NumeClasa;

   Importă o singură clasă.

2. import p.*;

   Importă întreg pachetul p.

3. import p1.p2. ... .p(n).NumeClasa;

4. import p1.p2. ... .p(n).*;

Instrucțiunile de import, trebuie să preceadă declarația de clasă și pot fi precedate doar de package.

Exemplu:

joc/Joc.java

joc/a/A.java

joc/b/J1.java

joc/b/J2.java

```
1
2  // Joc.java
3  package joc;
4  import a.A;
5  import a.b.*;
6
7  public class Joc
8  {
9      public A a;
10     public J1 j1;
```

```
11      public J2 j2;
12
13      public static void main(String[] args)
14      {
15          new Joc();
16      }
17
18      public Joc()
19      {
20          a = new A(this);
21          j1 = new J1(this);
22          j2 = new J2(this);
23
24          a.pornesteJocul();
25      }
26  }
27
28  // A.java
29  package a;
30  import joc.Joc;
31
32  public class A
33  {
34      Joc joc;
35
36      public A(Joc joc)
37      {
38          this.joc = joc;
39      }
40
41      void lanseazaJocul()
42      {
43          if ((int)(Math.random() * 2)) == 0)
44          {
45              joc.j2.arunca();
46          }
47          else
48          {
49              joc.j1.arunca();
50          }
51      }
52  }
53
54
55  // J1.java
56  package a.b;
57  import joc.Joc;
58
59  public class J1
60  {
```

```
61      Joc joc;
62      int scor;
63
64      public J1(Joc joc)
65      {
66          this.joc = joc;
67      }
68
69      public void arunca()
70      {
71          while ((int)(Math.random() * 2) == 0)
72          {
73              scor++;
74
75              System.out.println("Scorul lui j1: " + scor);
76              try
77              {
78                  Thread.sleep(1000);
79              }
80              catch (Exception e){}
81          }
82
83          joc.j2.arunca();
84      }
85  }
86
87
88  // J2.java
89  package a.b;
90  import joc.Joc;
91
92  public class J2
93  {
94      Joc joc;
95      int scor;
96
97      public J2(Joc joc)
98      {
99          this.joc = joc;
100     }
101
102     public void arunca()
103     {
104         while ((int)(Math.random() * 2) == 0)
105         {
106             scor++;
107
108             System.out.println("Scorul lui j2: " + scor);
109             try
110             {
```

```
111                     Thread.sleep(1000);
112                 }
113                 catch (Exception e){}
114             }
115             joc.j1.arunca();
116         }
117 }
```

18 Excepții

În execuția programelor Java, pot să apară 2 categorii de probleme. Din prima categorie fac parte acele probleme grave, care nu pot fi anticipate de către programator și pe care programatorul nu le poate rezolva din cod. Acestea se numesc erori. Exemplu:

- Epuizarea memoriei
- Întreruperea conexiunii la rețea

Din a doua categorie fac parte problemele mai puțin grave, care pot fi anticipate și rezolvate din cod. Acestea se numesc excepții. De exemplu:

- Încercarea de citire dintr-un fișier care nu există.
- Încercarea de citire dintr-o matrice a unui element care nu există.
- Încercarea de conversie la un număr, a unui String ce nu reprezintă un număr.

Toate acestea constituie excepții.

În momentul în care sunt încălcate constrângerile semantice ale JDK-ului, acesta procedează la lansarea unei excepții, astfel: execuția programului se întrerupe în punctul respectiv, numit și punct de lansare și va continua dintr-un punct precizat de programator, numit punct de prindere. Dacă nu există niciun punct de prindere, firul de execuție va fi distrus. Din acest motiv, instrucțiunile sau blocurile de cod, care pot lansa excepții trebuiesc marcate în program. Acest lucru se face cu ajutorul lui try:

```
try
    instructiuni;
catch (Exception1 e)
    instructiuni1;
catch (Exception2 e)
    instructiuni2;
...
catch (Exceptionn e)
    instructiunin;
finally
    instructiunif;
*
```

Unde, instructiuni, reprezintă un bloc de cod ce poate lansa excepții. Să presupunem că în timpul execuției, blocul instructiuni lansează o excepție. În acest caz, mașina virtuală construiește un obiect care descrie excepția respectivă, iar execuția continuă din zona instrucțiunilor catch(toate instrucțiunile de după cea ce lansează excepția, nu mai sunt executate). Dacă excepția lansată aparține

clasei excepție 1, vor fi executate instructiuni1, după care vor fi executate instructiunif de la finally, dacă există, apoi se continuă de la *.

Exemplu:

```
1
2  public class Euclid
3  {
4      public static void main(String[] args)
5      {
6          int m = 0, n = 0;
7
8          try
9          {
10             m = Integer.parseInt(args[0]);
11         }
12         catch (ArrayIndexOutOfBoundsException e)
13         {
14             System.out.println("Nu a fost introdus niciun
                   parametru!");
15             return;
16         }
17         catch (NumberFormatException e)
18         {
19             System.out.println("Primul parametru este gresit!");
20             return;
21         }
22
23         try
24         {
25             n = Integer.parseInt(args[1]);
26         }
27         catch (ArrayIndexOutOfBoundsException e)
28         {
29             System.out.println("Ati introdus doar un parametru!");
30             return;
31         }
32         catch (NumberFormatException e)
33         {
34             System.out.println("Al doilea parametru este gresit!")
                   ;
35             return;
36         }
37
38         System.out.println("cmmdc(" + m + ", " + n + ") = " +
               cmmdc(m, n));
39         System.out.println("cmmmc(" + m + ", " + n + ") = " +
               cmmmc(m, n));
40     }
41
42     public static int cmmdc(int m, int n)
43     {
```

```
44          if (m == n)
45              return m;
46          int k = m;
47          if (n < m)
48          {
49              m = n;
50              n = k;
51          }
52
53          while (true)
54          {
55              k = m;
56              m = m % n;
57              n = k;
58              if (m == 0)
59                  return n;
60          }
61      }
62
63      public static int cmmmc(int m, int n)
64      {
65          return (m * n) / cmmdc(m, n);
66      }
67  }
```

Toate clasele care reprezintă excepții, vor avea ca super-clasă, nu neapărat directă, pe java.lang.Exception. Pentru a lansa o excepție, se folosește cuvântul rezervat throw. Să presupunem că Exceptie este o clasă ce definește o excepție. Lansarea unei excepții de tipul acestei clase se poate face astfel:

```
1  Exception c = new Exceptie();
2  throw c;
3
4  // SAU
5  throw new Exceptie();
```

Dacă o metodă lansează excepții, toate acele excepții trebuiesc precizate în declarația metodei, cu următoare sintaxă:

```
1  [modificatori] tip_returnat nume([parametrii]) throws Exceptie1,
       Exceptie2, ..., ExceptieN
2  {
3      if (test1)
4          throw new Exceptie1();
5
6      ...
7
8      if (testN)
9          throw new ExceptieN();
10 }
```

Exemplu:

```
class Exceptie1 extends Exception
{
    public Exceptie1(String message)
    {
        super(message);
    }
}

class Exceptie2 extends Exception
{
    public Exceptie2(String message)
    {
        java.util.Date d = new java.util.Date();

        super(message + "/" + g.getHours() + ":" + d.getMinutes()+
            ":" + d.getSeconds());
    }
}

public class Exemplu
{
    void exemplu(int i) thrwos Exceptie1, Exceptie2
    {
        if (i < 50)
            throw new Exceptie1("A fost lansata prima exceptie");
        else
            throw new Exceptie2("A fost lansata a doua exceptie");
    }

    public static void main(String[] args)
    {
        new Exemplu();
    }

    public Exemplu()
    {
        while (true)
        {
            int i = (int)(Math.random() * 100 + 1);

            try
            {
                exemplu(i);
            }
            catch (Exceptie1 e)
            {
                System.out.println(e.getMessage());
                return;
            }
            catch(Exceptie2 e)
```

```
50              {
51                  System.out.println(e.getMessage());
52              }
53
54              try
55              {
56                  Thread.sleep(1000);
57              }
58              catch (InterruptedException e)
59              {
60                  continue;
61              }
62          }
63      }
64  }
```

Clasele care implementează excepții din java.lang:

```
object
    Throwable
        Error (Super-clasa tuturor erorilor)
            LinkingError
                ClassFormatError
                IncompatibleClassChasisError
                    VirtualMethodError
            VirtualMachineError
                OutOfMemoryError
        Exception (Super-clasa tuturor exceptiilor)
            RuntimeException
                ArithmeticException
                IndexOutOfBoundsException
                    ArrayIndexOutOfBoundsException
                IllegalAccessException
                IllegalArgumentException
                    NumberFormatException
            IOException
                FileNotFoundException
                MalformatURLException
                    EOfException
            InterruptedException
```

19 Fire de execuție

Capacitatea unui sistem de operare de a executa mai multe programe în paralel se numește multi-tasking. Aceste programe se mai numesc task-uri sau procese.

Observație: Majoritatea sistemelor de operare actuale suportă multi-tasking.

Execuția simultană a mai multor procese este însă aparentă. Întrucât procesorul execută o singură operație în unitatea de timp. În timpul în care un proces

folosește procesorul, celelalte task-uri așteaptă să intre la procesor, într-o serie de cozi de așteptare. Mecanismul acestor cozi este gestionat de sistemul de operare. Datorită frecvențelor foarte mari de operare a procesorului, task-urile primesc dreptul la execuție foarte rapid și foarte des, ceea ce face ca programele să ruleze aparent în paralel. Pe micro-procesoarele actuale se realizează calcule paralele reale, pe core-uri(nuclee).

Capacitatea unui program de a executa mai multe subprograme în paralel, se numește multi-threading. Aceste sub-programe se mai numesc thread-uri sau fire de execuție.

Limbajul Java suportă multi-threading.

Ca și în cazul proceselor execuția în paralel a mai multor fire de execuție este doar aparentă, întrucât, la un moment dat, un singur fir de thread folosește mecanismul de execuție al mașinii virtuale Java. În acest timp, celelalte thread-uri așteaptă să intre la execuție într-o serie de cozi de așteptare. Mecanismul acestor cozi este gestionat de mașina virtuală Java și este format din 10 cozi.

În Java, pentru o tratare unitară, firele de execuție pot fi organizate în grupuri de fire de execuție. Acestea pot avea la rândul lor subgrupuri.

În momentul lansării unui program Java, interpretorul construiește în memoria RAM mașina virtuală Java. După alocarea zonelor de memorie, sunt construite o serie de fire de execuție, numite fire damon, care oferă suport pentru crearea și execuția celorlalte fire de execuție. Apoi, este creat grupul de fire de execuție numit System. Grupul System are un subgrup numit Main. Grupul Main conține un fir de execuție numit Main. Este lansat în execuție firul Main, ceea ce conduce la apelarea metodei run() a firului respectiv. Aceasta va apela metoda statică main, a clasei primare. În continuare, se execută codul programatorului, scris în metoda main().

Prin urmare, metoda main() funcționează în cadrul firului de execuție Main.

Limbajul Java oferă suport pentru operarea cu fire de execuție prin clasele Thread, ThreadGroup și interfața Runnable din pachetul java.lang .

Clasa Thread este super-clasa firelor de execuție. Aceasta are următorii constructori:

```
1 Thread(ThreadGroup grup, Runnable tinta, String nume)
```

Dacă grup este null, firul respectiv va fi inclus în grupul de fire al firului părinte. Dacă grup nu este null, firul creat va fi inclus în grupul de fire numit grup.

Ținta va fi un obiect de tipul unei clase ce implementează Runnable. Dacă tinta este null, la lansarea în execuție a firului va fi executată metoda run a firului. Dacă ținta nu este null, la lansarea în execuție a firului se va executa run() a obiectului tinta.

Nume este un String ce reprezintă numele firului în sistemul de gestiune al firelor de execuție. Dacă este null, atunci va fi atribuit automat astfel: Thread-nr

Alți constructori:

```
Thread() // Thread(null, null, null);

Thread(ThreadGroup grup) // Thread(grup, null, null)

Thread(Runnable tinta) // Thread(null, tinta, null)

Thread(String nume) // Thread(null, null, nume)

Thread(ThreadGroup grup, Runnable tinta) // Thread(grup, tinta,
   null)

Thread(ThreadGroup grup, String nume) // Thread(grup, null, nume)

Thread(Runnable tinta, String nume) // Thread(null, tinta, nume)
```

Frecvența prin care un fir primește execuția, este controlată prin prioritatea firului.

Prioritatea este dată de un int al firului de execuție ce poate lua valori între 1 și 10.

1 prioritate minimă
5 prioritate normală
10 prioritate maximă

În clasa Thread mai găsim constantele:

```
public static final int MIN_PRIORITY = 1;
public static final int NORMAL_PRIORITY = 5;
public static final int MAX_PRIORITY = 10;

public int getPriority() // returneaza prioritatea firului
public void setPriority(int p) // seteaza prioritatea firului la p

public String getName() // returneaza numele firului
public void start()
public void stop()
```

La apelul lui start(), firul respectiv este înregistrat în sistemul de gestiune a firelor mașinii virtuale și apoi este apelată metoda run().

Metoda stop() exclude firul din sistemul de gestiune al firelor de execuție.

Metoda run() este metoda prin care firul se manifestă. Codul pe care vrem să-l execute firul respectiv, trebuie plasat în metoda run().

Există 2 variante de creare a firelor de execuție:

1. Prin extinderea clasei Thread

2. Prin implementarea interfeței Runnable

Prima metodă se desfășoară astfel:

1. Este creată o clasă ce extinde clasa Thread.
2. Este suprascrisă în clasa de la punctul 1, metoda run() a clasei Thread.
3. Este creat în program un obiect de tipul clasei de la punctul 1.
4. Este lansat în execuție firul de la punctul 3, prin apelarea metodei start() a acestui obiect.

Metoda descrisă mai sus ne limitează doar la sub-clase ale clasei Thread. Există însă situații în care dorim ca o clasă ce nu este derivată din Thread să fie un fir de execuție.

În acest caz, vom folosi a doua modalitate de creare a firelor de execuție:

1. Este creată o clasă care implementează Runnable.
2. Este implementată în clasa de la punctul 1, metoda run(), declarată în Runnable.
3. Este creat în program, un obiect de tipul clasei de la punctul 1.
4. Este creat în program, un fir de execuție, inițializat în obiectul de la punctul de 3, ca țintă.
5. Este lansat în execuție, firul de la punctul 4, prin apelarea metodei start() a firului respectiv.

Exemplu:

```
1  class MyThread extends Thread
2  {
3      int i;
4      MyThread(int p, String name)
5      {
6          setName(name);
7          setPriority(p);
8      }
9
10     public void run()
11     {
12         while (true)
13         {
14             System.out.println(name + ": " + (i++));
15
16             try
17             {
18                 Thread.sleep(500);
19             }
20             catch (Exception e)
```

```
21              {
22
23              }
24          }
25      }
26  }
27
28  public class Exemplu
29  {
30      public static void main(String[] args)
31      {
32          MyThread[] mt = new MyThread[10];
33          for (int i = 0; i < mt.length; i++)
34          {
35              mt[i] = new MyThread((int)(Math.random() * 10 + 1), "
                    mt[" + (i + 1) + "]");
36              mt[i].start();
37          }
38      }
39  }
```

La un moment dat, un fir de execuție se poate afla în una din următoarele 4 stări:

1. starea nou creat
2. starea în execuție
3. starea nou blocat
4. starea terminat

Când un fir de execuție e creat cu ajutorul lui new, acesta se află în starea nou creat. În această stare, firul există ca obiect, dar nu se manifestă în niciun fel.

Prin apelarea metodei start a firului, acesta trece în starea în execuție. Starea în execuție are 2 substări:

- execuția propriu-zisă, în care firul folosește sistemul de execuție al mașinii virtuale;
- starea de așteptare într-una din cele 10 cozi de așteptare(indexate de la 1 la 10, în funcție de prioritate);

Un fir poate fi scos forțat de la execuție și trimis în coada sa de așteptare, prin apelarea metodei yield() a firului. Din starea în execuție, firul poate fi trecut într-o stare de blocaj temporar prin apelarea metodei suspend(). În această stare, firul își încetează activitatea, dar continuă să existe în sistemul de gestiune a firelor de execuție. Prin apelarea metodei resume(), firul trece din nou în starea de execuție și își reia execuția de unde a rămas.

Un fir mai poate fi trimis în starea blocat temporizat prin apelarea metodelor statice sleep() ale clasei Thread.

```
1 sleep(long m) // blocheaza firul de executie pe durata a m
    milisecunde
2 sleep(long m, int n) // blocheaza firul de executie pe durata a m
    milisecunde + n nano-secunde
```

Din starea de execuție sau starea blocat, firul poate fi trecut în starea terminat, prin funcție stop() a firului. Aceasta va exclude firul din sistemul de execuție a firelor, dar firul c ontinuă să existe ca obiect.

Metodele suspend() și sleep() produc un blocaj al firului din interiorul acestuia. Există însă situații când este necesară blocarea unui fir de execuție din interiorul lui. De exemplu: presupunem că avem un program cu 2 fire de execuție concurente, unul producător, unul consumator. Firul producător produce date pe care le duce la consumator pentru procesare. Dacă firul consumator nu reușește să proceseze datele îndeajuns de repede, trebuie să-i spunem să se oprească. Blocajul firului producător trebuie făcut din exteriorul acestuia. Acest lucru se poate face prin apelarea metodei wait() a unui obiect. Metoda wait() se găsește implementată în clasa Object și poate fi apelată în orice obiect java. Dacă un fir de execuție apelează wait(), firul va fi blocat și va rămâne astfel, până la apelarea metodei notify() a aceluiași obiect. Dacă mai multe fire de execuție au fost blocate prin apelarea metodei wait() a aceluiași obiect, putem să le deblocăm pe toate simultan prin notifyAll() a obiectului respectiv. notify() și notifyAll() provin din clasa Object.

Pentru un control al timpului blocării:

```
1 wait(long m)
2 wait(long m, long n)
```

Metodele care produc blocaj: suspend, sleep, wait, pot lansa excepția InterruptedException.

O metodă a unui obiect care este partajată de mai multe fire de execuție concurente, se mai numește secțiune critică. Firele de execuție respective vor executa metoda într-o manieră necontrolabilă. Ca să avem un control al execuției, secțiunea critică trebuie sincronizată. Acest lucru se face cu ajutorul cuvântului rezervat synchronized.

Dacă un fir execuție apelează o metodă sincronizată și aceasta nu e apelată de un alt fir, firul respectiv va executa metoda. Dacă în acest timp un alt fir apelează metoda, acesta nu primește dreptul la execuție și va fi pus să aștepte într-o coadă. Când primul fir termină execuția, va primii execuția al doilea fir ș.a.

O metodă sincronizată poate fi apelată doar din altă metodă sincronizată.

Metodele wait, notify, notifyAll sunt sincronizate.

Exemplu:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Cafe extends Applet implements ActionListener
{
    private Button burger, fries, cola, cooked;
    private Order order, complete;

    public void init()
    {
        Graphics g = getGraphics();

        burger = new Button("Burger");
        add(burger);
        burger.addActionListener(this);

        fries = new Button("Fries");
        add(fries);
        fries.addActionListener(this);

        cola = new Button("Cola");
        add(cola);
        cola.addActionListener(this);

        cooked = new Button("Cooked");
        add(cooked);
        cooked.addActionListener(this);

        order = new Order();
        complete = new Order();

        Queue queue = new Queue(g);

        Waiter waiter = new Waiter(g, order, queue);
        waiter.start();

        Cheif cheif = new Cheif(g, complete, queue);
    }

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == burger)
        {
            order.notifyEvent("burger");
        }
        else if (e.getSource() == fries)
        {
            order.notifyEvent("fries");
        }
```

```
51          else if (e.getSource() == cola)
52          {
53              order.notifyEvent("cola");
54          }
55          else if (e.getSource() == cooked)
56          {
57              complete.notifyEvent("cooked");
58          }
59      }
60  }
61
62  class Order
63  {
64      private String order = "";
65
66      public synchronized void notifyEvent(String comanda)
67      {
68          order = comanda;
69          notify();
70      }
71
72      public synchronized void waitForEvent()
73      {
74          while (order.equals(""))
75          {
76              try
77              {
78                  wait();
79              }
80              catch (InterruptedException e)
81              {
82
83              }
84          }
85
86          String comanda = order;
87          order = "";
88          return comanda;
89      }
90  }
91
92  class Queue
93  {
94      private Graphics g;
95      private String[] queue = new String[20];
96
97      private int caut;
98
99      public Queue(Graphics g)
100     {
```

```
101             this.g = g;
102         }
103 
104         public synchronized void enter(String item)
105         {
106             q[caut] = item;
107             caut++;
108             display();
109             notify();
110         }
111 
112         public synchronized String remove()
113         {
114             while (caut == 0)
115             {
116                 try
117                 {
118                     wait();
119                 }
120                 catch(InterruptedException e)
121                 {
122 
123                 }
124             }
125             String item = queue[0];
126             caut--;
127             for (int i = 0; i < caut; i++)
128             {
129                 queue[i] = queue[i + 1];
130             }
131 
132             display();
133             return item;
134         }
135 
136         public synchronized boolean isFull()
137         {
138             return caut == queue.length;
139         }
140 
141         private void display()
142         {
143             g.drawString("Comenzi", 100, 50);
144             g.drawRectangle(120, 50, 50, 220);
145 
146             for (int i = 0; i < caut; i++)
147             {
148                 g.drawString(queue[i], 120, 70 + i * 20);
149             }
150         }
```

```
151 }
152
153 class Waiter extends Thread
154 {
155     private Order order;
156     private Graphics g;
157     private Queue queue;
158
159     public Waiter(Graphics g, Order order, Queue queue)
160     {
161         this.g = g;
162         this.order = order;
163         this.queue = queue;
164     }
165
166     @Override
167     public void run()
168     {
169         while (true)
170         {
171             String comanda = order.waitForEvent();
172
173             g.clearRect(10, 50, 50, 25);
174
175             g.drawString(comanda, 10, 70);
176
177             try
178             {
179                 Thread.sleep(5000);
180             }
181             catch (InterruptedException e)
182             {
183
184             }
185
186             if (!queue.isFull())
187                 queue.enter(comanda);
188         }
189     }
190 }
191
192 class Cheif extends Thread
193 {
194     private Order complete;
195     private Graphics g;
196     private Queue queue;
197
198     public Cheif(Graphics g, Order complete, Queue queue)
199     {
200         this.g = g;
```

```
201             this.complete = complete;
202             this.queue = queue;
203         }
204 
205         public void run()
206         {
207             while (true)
208             {
209                 String order = queue.remove();
210                 g.clearRect(200, 55, 50, 25);
211                 g.drawString("cooking: " + order, 270, 70);
212 
213                 String cookedInfo = complete.waitForEvent();
214             }
215         }
216 }
```

Pentru o tratare unitară, firele de execuție pot fi grupate în grupuri de fire. Un grup poate avea subgrupuri și se creează astfel o ierarhie a grupurilor de fire, în vârful căreia se găsește grupul sistem.

Grupurile de fire de execuție sunt obiecte de tipul clasei ThreadGroup. Această clasă are 2 constructori:

```
1 ThreadGroup(String name) // creeaza un grup de fire cu numele name
      si care va fi subgrup al grupului de fire din care face parte
     firul parinte
2 ThreadGroup(ThreadGroup group, String name) // creeaza un grup de
     fire cu numele name si care va fi subgrup al grupului numit
     group
```

Clasa ThreadGroup are un argument maxPriority, care reprezintă prioritatea maximă a grupului. Toate firele dintr-un grup nu pot avea prioritatea mai mare decât maximul grupului.

```
1 getMaxPriority()
2 setMaxPriority(int p)
3 
4 void enumerate(Thread[] list) // va incarca parametrul list cu
     toate firele din acest grup si din toate subgrupurile sale
5 void enumerate(ThreadGroup[] list) // va incarca parametrul list
     cu toate subgrupurile din acest grup
```

Exemplu:

```
1 
2 class T extends Thread
3 {
4     int k;
5     boolean b = true;
6 
7     T(ThreadGroup grup, String nume)
8     {
```

```
9           super(grup, nume);
10      }
11
12      public void run()
13      {
14          while (b)
15          {
16              System.out.println(getName() + " " + k++);
17              try
18              {
19                  Thread.sleep(500);
20              }
21              catch (InterruptedException e)
22              {
23
24              }
25          }
26      }
27  }
28
29  public class Example
30  {
31      public static void main(String[] args)
32      {
33          ThreadGroup grup0 = new ThreadGroup("Grup0"); // va fi
               supgrup al grupului main
34          ThreadGroup grup1 = new ThreadGroup(grup0, "Grup1"); // va
               fi supgrup al grupului grup0
35          ThreadGroup grup2 = new ThreadGroup(grup0, "Grup2"); // va
               fi supgrup al grupului grup0
36
37          T t1 = new T(grup0, "T1");
38          T t2 = new T(grup0, "T2");
39
40          T t3 = new T(grup1, "T3");
41          T t4 = new T(Grup2, "T4");
42
43          grup0.setMaxPriority(8);
44          grup1.setMaxPriority(6);
45
46          t1.setPriority(8);
47          t2.setPriority(6);
48          t3.setPriority(4);
49
50          t1.start();
51          t2.start();
52          t3.start();
53          t4.start();
54
55          Thread[] list = null;
```

```
56          grup0.enumerate(list);
57
58          for (int i = 0; i < list.lengthl i++)
59          {
60              if (list[i].getName().startswith("T3"))
61              {
62                  list[i].b = false;
63              }
64          }
65      }
66  }
```

20 Operații de intrare, ieșire în Java

În timpul execuției programelor Java, datele procesate sunt stocate în memoria mașinii virtuale Java și se vor pierde la terminarea execuției.

Ca să le păstrăm, datele pot fi scrise în fișier.

Operațiile prin care un program scrie date într-un fișier, le tipărește pe display sau la imprimantă, le trimite în alt program sau în orice dispozitiv periferic, se numesc operații de ieșire sau de scriere.

Operațiile prin care un program citește date dintr-un fișier sau de la tastatură sau din alt program sau din orice dispozitiv periferic, se numesc operații de intrare sau de ieșire.

Limbajul Java furnizează suport pentru operațiile IO, prin clasele și interfețele din pachetul java.io. O clasă importantă este File. Aceasta oferă metode pentru lucrul cu sistemul de fișier, de exemplu: crearea de directoare, ștergerea fișierelor etc, dar nu furnizează metode pentru scrierea sau citirea fișierelor.

```
1  File(String path) // construieste un fisier avand calea relativa
       sau absoluta ‘‘path’’.
2  File(String dir, String name) // construieste un obiect File
       asociat fisierului cu numele name din directorul dir
3  File(File dir, String name) // construieste un obiect File asociat
        fisierului cu numele name din directorul cu file-ul asociat
       dir
4
5  canRead() // true daca fisierul poate fi citit, false in caz
       contrar
6  canWrite() // true daca fisierul poate fi scris, false in caz
       contrar
7  delete() // sterge fisierul
8  exists() // true daca fisierul exista, false in caz contrar
9  getAbsolutPath() // returneaza un String ce reprezinta calea
       absoluta a fisierului in sistemul local de fisiere
10 getName() // returneaza un String ce reprezinta numele fisierului
11 getParent() // returneaza un String ce reprezinta numele
       directorului parinte
```

```
getPath() // returneaza calea relativa sau absoluta a fisierului
isDirectory() // returneaza true daca obiectul e asociat unui director, false in caz contrar
isFile() // returneasza true daca obiectul este asociat unui fisier, false in caz contrar
lastModified // data ultimei modificari a fisierului
length() // dimensiunea in octeti a fisierului
String list() returneaza o matrice 1-dimensionala de tipul String care contine toate numele si subdirectoarele din directorul respectiv
File listFiles() returneaza o matrice 1-dimensionala de tipul File care contine toate obiectele de tipul File asociate fisierelor si subdirectoarelor din directorul respectiv
mkdir() // creaza un director cu numele si calea reprezentate de obiectul File
renameTo(File f) // redenumeste fisierul sau directorul asociat cu numele lui f
```

Exemplu.java

```
public class Exemplu
{
    public static void main(String[] args) throws Exception
    {
        if (args == null || args.length() == 0)
            return;

        File f = new File(args[0]);
        if (f.exists())
        {
            if (f.isDirectory())
            {
                File[] files = f.listFiles();
                for (int i = 0; i < files.length; i++)
                {
                    System.out.println(files[i]);
                }
            }
            else
            {
                System.out.println(f.getName());
                System.out.println(g.getAbsolutPath());
                System.out.println(f.length());
            }
        }
    }
}
```

În Java, scrierea, respectiv citirea fișierelor se face cu ajutorul obiectelor RandomAccessFile sauu prin intermediul fluxurilor.

Obiectele RandomAccessFile ne permit poziționarea pointerului de fișier la o

anumită poziție, de unde putem citi date, respectiv putem scrie date.

```
1  RandomAccessFile(String name, String mod) // creeaza un obiect
      RandomAccessFile asociat unui fisier cu numele name, deschis in
       modul de operare mod. Fisierul trebuie sa se gaseasca in
      acelasi fisier cu clasa respectiva, iar mod poate fi "r"(read,
      citire) sau "rw"(read-write, citire si scriere)
2  RandomAccessFile(File f, String mod) // creeaza un obiect
      RandomAccessFile asociat unui fisier reprezentat de obiectul f,
       deschis in modul de operare mod. Fisierul trebuie sa se
      gaseasca in acelasi fisier cu clasa respectiva, iar mod poate
      fi "r"(read, citire) sau "rw"(read-write, citire si scriere)
3
4  close() // inchide conexiunea cu fisierul
5  getFilePointer() // returneaza un long ce reprezinta pozitia
      curenta a pointerului de fisier
6  length() // lungimea in octeti
7  seek(long p) // muta pointerul de fisier la pozitia p
8
9  read() // citeste un byte
10 read(byte[] b, int start, int length) // Citeste o secventa de
      octeti de la pozitia curenta, pe care o incarca in matricea b
11 readBoolean() // citeste o valoare boolean
12 readByte() // citeste un byte
13 readChar() // citeste o valoare de tipul char
14 readDouble() // citeste o valoare de tipul double
15 readFloat()
16 readInt()
17 readLong()
18 readUTF() // citeste o valoare de tip UTF
19 readUnsignedByte() // citeste un byte fara semn
20 readUnsignedShort() // citeste un short fara semn
21
22 writeBoolean(boolean b) // scrie la pozitia cursorului valoarea b
23 writeByte(int b) // scrie la pozitia cursorului valoarea lui b
24 writeBytes(String s)
25 writeChar(int c) // scrie la pozitia curenta codul caracterului c
26 writeChars(String s)
27 writeFloat(float f)
28 writeDouble(double d)
29 writeInt(int i)
30 writeLong(long l)
31 writeShort(int s)
32 writeUTF(String s)
```

Exemplu.java

```
1  public class Exemplu
2  {
3      public static void main(String[] args) throws Exception
4      {
5          if (args == null || args.length() == 0)
```

```
6               return;
7
8           File f = new File(args[0]);
9
10          if (f.exists() && f.isFile())
11          {
12              RandomAccessFile raf = new RandomAccessFile(f, "rw");
13
14              raf.seek(100); // poate fi lansata exceptia
                   EOfException
15              System.out.println(raf.read());
16
17              byte[] b = {(byte)3, (byte)1, (byte)4};
18              raf.writeByte(b, 0, 3);
19
20              raf.seek(100);
21              System.out.println(raf.read() + "" + raf.read() + "" +
                    raf.read() + "" + raf.read());
22          }
23      }
24  }
```

Pe lângă obiectele de tipul RandomAccessFile, scrierea/citirea fișierelor se mai poate face cu ajutorul fluxurilor. Acestea permit o scriere/citire secvențială, de la începutul fișierului, până la sfârșit. Un flux prin care se citesc date dintr-un fișier se numește flux de intrare, iar un flux prin care se scriu date în fișier se mai numește flux de ieșire sau de scriere. Transferul datelor printr-un flux este uni-direcțional, adică, un flux de citire nu poate fi utilizat la scriere și nici invers.

2 fluxuri des utilizate sunt atributele in și out din clasa System. in este un flux de tipul InputStream, conectat la fișierul logic al tastaturii. Iar out este un flux de tipul output stream, conectat la fișierul logic al system-ului.

Fluxurile se împart în 2 categorii:

1. Fluxuri de nivel superior
2. Fluxuri de nivel inferior(Fluxuri nod)

Fluxurile nod sunt direct conectate cu fișierul și formează un sistem alcătuit din fișier, date și o conductă. Conducta este formată din cel puțin un fir de execuție care asigură transferul datelor prin flux. În funcție de datele transferate fluxurile nod pot fi orientate pe transfer de octeți sau pe transfer de caractere.

Principalele clase care implementează fluxuri nod sunt InputStream și OutputStream. InputStream este super-clasă pentru toate clasele care implementează fluxuri de citire orientate pe transfer de octeți.

Respectiv, OutputStream este super-clasă pentru toate clasele care implementează fluxuri de scriere orientate pe transfer de octeți.

Principalele clase care implementează fluxuri noi pe șiruri de caractere sunt Reader și Writer.

Reader este super-clasă pentru toate clasele care implementează fluxuri de citire orientate pe transfer de caractere.

Respectiv, Writer este super-clasă pentru toate clasele care implementează fluxuri de scriere orientate pe transfer de caractere.

Fluxurile de nivel superior nu se conectează direct la fișier.

DataInputStream este o clasă ce implementează fluxuri de nivel superior. Analog DataOutputStream.

```
1  // Secretariat.java
2  import java.util.Date;
3  import java.io.*;
4  import java.text.*;
5
6  class Student
7  {
8      private String name;
9      private int id;
10     private Date dob;
11
12     public static final int NAME_SIZE = 40;
13     public static final int REGISTER_SIZE = 2 * NAME_SIZE + 16;
14
15     public Student(String name, int id, Date dob)
16     {
17         this.name = name;
18         this.id = id;
19         this.dob = dob;
20     }
21
22     public void writeFixedString(String s, int size,
          DataOutputStream out) throws IOException
23     {
24         for (int i = 0; i < size; i++)
25         {
26             char c = '#';
27             if (i < c.length)
28             {
29                 c = s.charAt(i);
30                 out.writeChar(c);
31             }
32         }
33     }
34
35     public void write(DataOutputStream out) throws IOException
36     {
37         writeFixedString(name, NAME_SIZE, out);
38         out.writeInt(id);
```

```
39          out.writeInt(dop.getYear());
40          out.writeInt(dop.getMonth());
41          out.writeInt(dop.getDate());
42      }
43
44      public String readPixelString(int size, DataInputStream in)
           throws IOException
45      {
46          StringBuffer sb = new StringBuffer(size);
47          int i = 0;
48          boolean b = true;
49          while (b == true && i < size)
50          {
51              char c = in.readChar();
52              i++;
53              if (c == '#')
54                  b = false;
55              else
56              {
57                  sb.append('c');
58              }
59
60              in.skip(2 * (size - i));
61              return sb.toString();
62          }
63      }
64
65      public void read(DataInputStream in) throws IOException
66      {
67          name = readFixedString(NAME_SIZE, in);
68          id = in.readInt();
69          int y = in.readInt();
70          int m = in.readInt();
71          int d = in.readInt();
72
73          dob = new Date(y, m, d);
74      }
75
76      public void print()
77      {
78          System.out.println("\n\n");
79          System.out.println("Student: " + name);
80          System.out.println("Id: " + id);
81          System.out.println("Data nasterii: " + dob);
82      }
83  }
84
85  public class Secretariat
86  {
87      public static void main(String[] args)
```

```
88      {
89          Student[] student = new Student[500];
90
91          SimpleDateFormat df = new SimpleDateFormat("d/M/yy");
92
93          student[0] = new Student("Ionescu Ioana", 120, df.parse("
               1/4/2002"));
94          student[1] = new Student("Popescu Dan", 124, df.parse("
               12/8/2002"));
95
96          try
97          {
98              FileOutputStream fos = new FileOutputStream("studenti.
                   txt");
99              DataOutputStream out = new DataOutputStream(fos);
100             for(int i = 0; i < student.length; i++)
101             {
102                 if (student[i] != null)
103                 {
104                     student[i].write(out);
105                 }
106             }
107         }
108         catch (Exception e)
109         {
110
111         }
112     }
113 }
```

21 Serializare

Obiectele care pot fi salvate prin acest proces de serializare se numesc serializabile. Un obiect este serializabil daca apartine unei clase serializabile. O clasa se numeste serializabilă, dacă implementează interfața Serializable. Această interfață nu conține nicio declarare și este folosită doar pentru marcarea claselor serializabile. În momentul în care un obiect este serializat, vor fi serializate de asemenea toate atributele sale serializabile, prin urmare, va fi salvat în fișierul respectiv, o întreagă structură arborescentă. Dacă dorim ca un atribut să nu fie serializat, acesta se declară transcient.

Pentru a avea un control mai bun asupra serializării, se poate folosi interfața java.io.Externalizable. Aceasta este o subinterfață a lui Serializable.

Interfața Externalizable are 2 metode:

```
1 public void readExternal(ObjectInputString in);
2 public void writeExternal(ObjectOutputString out);
```

În momentul în care un obiect al clasei respective este serializat, mașina virtuală va executa metoda writeExternal.

```
out.writeObject();
out.writeInt();
...
```

De asemenea, în momentul deserializării va apela automat metoda readExternal. Prin urmare, în implementarea acestei metode, putem folosi metode de tipul:

```
in.readObject();
in.readInt();
...
```

Exemplu: Secretariat.java

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

class An implements Serializable
{
    private String nume;
    private Grupa[] grupe;

    public An(String nume, Grupa[] grupe)
    {
        this.nume = nume;
        this.grupe = grupe;
    }

    public void SetNume(String nume)
    {
        this.nume = nume;
    }

    public void SetGrupe(Grupa[] grupe)
    {
        this.grupe = grupe;
    }

    public String GetName()
    {
        return nume;
    }

    public Grupa[] GetGrupe()
    {
        return grupe;
    }

    @Override
```

```
37      public String toString()
38      {
39          String info = "Anul: " + nume + "\n";
40          for (int i = 0; i < grupe.length && grupe[i] != null; i++)
41          {
42              info += grupe[i].toString();
43          }
44          return info;
45      }
46  }
47
48  class Grupa implements Serializable
49  {
50      private String name;
51      private Vector students;
52
53      public Grupa(String name, Vector students)
54      {
55          this.name = name;
56          this.students = students;
57      }
58
59      public void SetName(String name)
60      {
61          this.name = name;
62      }
63
64      public void SetStudents(Vector students)
65      {
66          this.students = students;
67      }
68
69      public String GetName()
70      {
71          return name;
72      }
73
74      public Vector GetStudents()
75      {
76          return students;
77      }
78
79      public void AddStudent(Student student)
80      {
81          students.addElement(student);
82      }
83
84      public void RemoveStudent(Student student)
85      {
86          students.remove(student);
```

```
87      }
88
89      public void RemoveAllStudents()
90      {
91          students.removeAllElements();
92      }
93
94      public Student GetStudent(int index)
95      {
96          return (Student)students.elementAt(index);
97      }
98
99      public int GetStudentsCount()
100     {
101         return students.size();
102     }
103
104     @Override
105     public String toString()
106     {
107         String info = "Grupa: " + name + "\n\n";
108         for (int i = 0; i < students.size(); i++)
109         {
110             info += GetStudent(i).toString();
111         }
112
113         return info;
114     }
115 }
116
117 class Student implements Serializable
118 {
119     public static final String[] M = {"Ianuarie", "Februarie", "
        Martie", "Aprilie", "Mai", "Iunie", "Iulie", "August", "
        Septembrie", "Octombrie", "Noiembrie", "Decembrie"};
120     public static final String[] Q = {"student", "sef de grupa", "
        sef de an", "responsabil studenti"};
121
122     private String name;
123     private int id;
124     private Date dob;
125     private boolean status = true;
126     private int quality;
127     private Grupa grupa;
128     private Grupa[] grupe;
129     private An[] ani;
130
131     public Student(String name, int id, String dob, int quality,
        boolean status) throws ParseException
132     {
```

```
133             // "d/M/yy"
134 
135             this.name = name;
136             this.id = id;
137             this.dob = (new SimpleDateFormat("d/M/yy")).parse(dob);
138             this.quality = quality;
139             this.status = status;
140         }
141 
142         public void SetName(String name)
143         {
144             this.name = name;
145         }
146 
147         public void SetID(int id)
148         {
149             this.id = id;
150         }
151 
152         public void SetDOB(String dob) throws ParseException
153         {
154             this.dob = (new SimpleDateFormat("d/M/yy").parse(dob));
155         }
156 
157         public void SetQuality(int quality)
158         {
159             this.quality = quality;
160         }
161 
162         public void SetStatus(boolean status)
163         {
164             this.status = status;
165         }
166 
167         public void SetGrupa(Grupa grupa)
168         {
169             if (quality == 1)
170             {
171                 this.grupa = grupa;
172             }
173         }
174 
175         public void SetGrupe(Grupa[] grupe)
176         {
177             if (quality == 2)
178                 this.grupe = grupe;
179         }
180 
181         public void setAni(An[] ani)
182         {
```

```
183         if (quality == 3)
184         this.ani = ani;
185     }
186 
187     public String GetName()
188     {
189         return name;
190     }
191 
192     public int GetID()
193     {
194         return id;
195     }
196 
197     public Date GetDOB()
198     {
199         return dob;
200     }
201 
202     public int GetQuality()
203     {
204         return quality;
205     }
206 
207     public boolean GetStatus()
208     {
209         return status;
210     }
211 
212     public Grupa GetGrupa()
213     {
214         return grupa;
215     }
216 
217     public Grupa[] GetGrupe()
218     {
219         return grupe;
220     }
221 
222     public An[] getAni()
223     {
224         return ani;
225     }
226 
227     @Override
228     public String toString()
229     {
230         String info = name + " / ";
231         info += "DOB: " + dob.getDate() + " " +
232         Student.M[dob.getMonth()] + " " + dob.getYear() + " / ";
```

```
233
234             info += Student.Q[quality] + " / ";
235             info += (status? "integralist" : "restantier") + " / ";
236             info += "\n";
237             return info;
238         }
239     }
240
241     public class Secretariat
242     {
243         public static void main(String[] args) throws Exception
244         {
245             Grupa grupa = new Grupa("M534", new Vector<>());
246             Student student = new Student("Ionescu Paul", 101, "
                    22/3/2004", 1, true);
247
248             student.SetGrupa(grupa);
249
250             grupa.AddStudent(new Student("Evran Rebeca", 104, "
                    10/3/2004", 0, true));
251             grupa.AddStudent(new Student("Tandrea Ivan", 107, "
                    12/12/2004", 0, true));
252             grupa.AddStudent(new Student("Popescu Ana", 108, "
                    6/11/2004", 0, false));
253
254             System.out.println(student.GetGrupa());
255
256             ObjectOutputStream out = new ObjectOutputStream(new
                    FileOutputStream("studenti"));
257
258             out.writeObject(grupa);
259             out.close();
260
261             ObjectInputStream in = new ObjectInputStream(new
                    FileInputStream("studenti"));
262             grupa = (Grupa)in.readObject();
263             System.out.println(grupa);
264         }
265     }
```

Exemplu:

```
1   import java.util.*;
2   import java.io.*;
3
4
5   public class Account implements Externalizable
6   {
7       private String username;
8       private String password;
9
10      public Account(String username, String password)
```

```
11      {
12          this.username = username;
13          this.password = password;
14      }
15
16      @Override
17      public void readExternal(ObjectInput in) throws IOException,
           ClassNotFoundException
18      {
19          username = (String)in.readObject();
20          password = (String)in.readObject();
21      }
22
23      @Override
24      public void writeExternal(ObjectOutput out) throws IOException
25      {
26          out.writeObject(username);
27          out.writeObject(password);
28      }
29
30      public String toString()
31      {
32          return username + " " + password;
33      }
34
35      public static void main(String[] args) throws Exception
36      {
37          Account a = new Account("MyName", "MyPassword");
38
39          ObjectOutputStream out = new ObjectOutputStream(
40              new FileOutputStream("aaa")
41          );
42          out.writeObject(a);
43          out.close();
44
45          ObjectInputStream in = new ObjectInputStream(
46              new FileInputStream("aaa")
47          );
48          Account b = (Account)out.readObject();
49          System.out.println(b);
50      }
51  }
```

22 Clase pentru interfața grafică

Utilizatorii programelor Java interacționează cu acestea prin intermediul unei interfețe grafice. O interfață grafică este alcătuită din componente.

Componentele grafice aparțin unor clase care se găsesc în java.awt.

Super clasa claselor din acest pachet este java.awt.Component care definește o componentă abstractă. Super-clasa lui Component este Object.

```
action(Event e, Object o) // Este apelata automat cand se produce o actiune
bounce() // returneaza un obiect de tipul Rectangle.
createImage(int w, int h) // Returneaza un obiect de tipul Image cu dimensiunile w si h
diseable() // dezactiveaza componenta (nu va mai raspunde la evenimente)
enable() // activeaza componenta
enable(boolean b) // daca b e true activeaza componenta, altfel o dezactiveaza
getBackground() // returneaza un obiect Color ce reprezinta background-ul componentei
getForeground() // returneaza un obiect Color ce reprezinta culoarea celui de-al doilea strat al componentei
getGraphics() // returneaza un obiect de tipul Graphics ce reprezinta contextul grafic al componentei
getParent() // returneaza containerul parinte al componentei
getToolkit() // returneaza un obiect de tipul Toolkit
handleEvent(Event e) // este metoda universala de tratare a evenimentelor in Java
hide() // ascunde componenta respectiva
inside(int x, int y) // metoda boolean, care testeaza daca punctul (x, y) este in interiorul componentei
isEnable() // metoda boolean, care returneaza true pentru componenta activa, false in caz contrar
isVisible() // true daca e vizibila componenta, false in caz contrar
keyDown(Event e, int key) // este metoda prin care se tratateaza evenimentul de apasare a unei taste
keyUp(Event e, int key) // este metoda prin care se tratateaza evenimentul de eliberare a unei taste
mouseDown(Event e, int x, int y) // metoda care trateaza apasarea unui buton a mouse-ului
mouseUp(Event e, int x, int y) // metoda care trateaza eliberarea unui buton a mouse-ului
mouseEnter(Event e, int x, int y) // metoda care trateaza evenimentul de intrare cu mouse-ul in zona componentei
mouseExit(Event e, int x, int y) // metoda care trateaza evenimentul de iesire cu mouse-ul din zona componentei
mouseMove(Event e, int x, int y) // metoda care trateaza evenimentul de miscare a mouse-ului in interiorul componentei
mouseDrag(Event e, int x, int y) // metoda care trateaza evenimentul de miscare a mouse-ului in interiorul componentei, in timp ce un buton e apasat
move(int x, int y) // muta componenta la coltul din stanga sus dat de (x, y)
nextFocus() // transfera focalizarea urmatoarei componente
paint(Graphics g) // redeseneaza
```

```
repaint() // apeleaza paint, dandu-i ca parametru contextul grafic al componentei
repaint(int x, int y, int w, int h) // redeseneaza doar portiunea specificata
resize(int w, int h) // redimensioneaza componenta
setBackground(Color c) // seteaza culoarea de background la c
setForeground(Color c) // seteaza culoarea de foreground la c
setFont(Font f) // seteaza fontul pe care este folosit contextul grafic al componentei
show()
show(boolean b)
size() // returneaza Dimension
```

Clasa container (are deasupra pe Component). Obiectele de tipul acestei clase sunt componente speciale

```
add(Component c) // adauga c la container
add(Component c, int i) // plaseaza c in container, pe pozitia i
countComponents() // returneaza numarul de componente din container
getComponents() // returneaza componentele acestui container
getLayout() // returneaza managerul de pozitionare al containerului
paint() // redeseneaza toate componentele din container
remove(Component c) // elimina c din container
removeAll() // sterge toate componentele din container
setLayout(LayoutManager m) // seteaza managerul de pozitionare la obiectul m, unde m este un obiect de tipul unei clase ce implementeaza interfata LayoutManager
```

Window este subclasa a clasei container. Reprezinta ferestre fara bordura si bara de titlu.

Frame e subclasa a lui Window, defineste un container special, cu bordura si bara de titlu

```
Frame() // creeaza un Frame, care e initial invizibil
Frame(String titlu) // Creeaza un Frame, initial invizibil ce are in bara de titlu textul titlu

getCursorType() // returneaza tipul cursorului
getItemImage() // returneaza Image ce stocheaza imaginea din bara de titlu
getMenuBar() // returneaza un obiect de tip MenuBar
getTitle() // returneaza un String ce contine sirul de caractere din bara de titlu a ferestrei
isResizable()
remove(MenuLayout m) // inlatura bara cu meniuri din fereastra
setCursor(int c) //
setItemImage(Image image) // seteaza imaginea din bara de titlu a ferestrei
setMenuBar(MenuBar mb) // seteaza bara cu meniuri
setResizable(boolean b)
```

```
14 setTitle(String titlu)
```

java.awt.MenuComponent este o componenta de tip Menu, generala.

```
1 getFont()
2 getParent()
3 setFont(Font f)
```

```
1 MenuComponent
2     MenuBar
3
4 add(Menu m)
5 countMenus() // returneaza numarul de meniuri din bara de meniuri
6 getMenu(int i)
7 remove(int i)
8 remove(MenuComponent m)
9
10 MenuComponent
11     MenuItem
12         MenuItem(String s) // construieste un element de menu cu
               eticheta specificata
13
14 disable() // dezactiveaza meniul respectiv
15 enable()
16 enable(boolean b)
17 getLabel() // Returneaza un String cu eticheta acestui Menu
18 setLabel(String s)
19 isEnabled()
20
21 MenuItem
22     Menu
23         Menu(String s)
24
25 add(MenuItem m)
26 add(String s) // adauga un nou MenuItem cu eticheta s
27 addSeparator() // adauga o linie in Menu
28 countItems()
29 remove(MenuComponent m)
30 remove(int i)
```

Așezarea componentelor într-un container se poate face în 3 moduri:

1. Prin intermediul obiectelor de tipul Panel
2. Prin utilizarea unui Manager de poziționare
3. În mod manual

Clasa Panel este o subclasă a lui Container, iar obiectele de tipul Panel, reprezintă Containere simple fără bordură și fără facilități grafice suplimentare.

Interfața LayoutManager. Clasele care o implementează se numesc manageri de poziționare.

1. BorderLayout
2. FlowLayout
3. CardLayout
4. GridLayout
5. GridBagLayout

BorderLayout așează componentele de-alungul laturilor container-ului și în centru. Este prestabilit în apleturi și ferestre. FlowLayout așează componentele în container pe rânduri. Este managerul prestabilit în panouri.

Obiectele CardLayout așează componentele ca într-un pachet de cărți.

GridLayout așează elementele ca într-o grilă.

Iar GridBagLayout așează automat elementele într-o grilă cu constrângeri.

Exemple.java

```
1  import java.awt.*;
2
3  class Exemple extends Frame
4  {
5      public static void main(String[] args)
6      {
7          new Exemple();
8      }
9
10     public Exemple()
11     {
12         setTitle("Exemple de pozitionare automata");
13         addMenus();
14         addPanels();
15         resize(600, 600);
16
17         setVisible(true);
18     }
19
20     void addMenus()
21     {
22         MenuBar mb = new MenuBar();
23         Menu file = new Menu("File");
24         file.add("abc");
25         file.addSeparator();
26         file.add("Exit");
27         mb.add(file);
28
29         setMenuBar(mb);
30     }
31
```

```
32      void addPanels()
33      {
34          setLayout(new GridLayout(2, 2));
35
36          Panel flow = new Panel();
37          Panel border = new Panel();
38          Panel grid = new Panel();
39          Panel gridBag = new Panel();
40
41          // flow.setLayout(new FlowLayout());
42          border.setLayout(new BorderLayout());
43          grid.setLayout(new GridLayout(2, 2));
44          gridBag.setLayout(new GridBagLayout());
45          addButtons(flow);
46          addButtons(border);
47          addButtons(grid);
48          addButtons(gridBag);
49
50          add(flow);
51          add(border);
52          add(grid);
53          add(gridBag);
54      }
55
56      void addButtons(Panel p)
57      {
58          if (p.getLayout() instanceof BorderLayout)
59          {
60              p.add("North", new Button("North"));
61              p.add("East", new Button("East"));
62              p.add("West", new Button("West"));
63              p.add("South", new Button("South"));
64          }
65          else if (p.getLayout() instanceof GridBagLayout)
66          {
67              GridBagLayout layout = (GridBagLayout) p.getLayout();
68              GridBagConstraints c1 = new GridBagConstraints();
69              c1.fill = GridBagConstraints.BOTH; // Butonul asezat
                    in celula respectiva va umple toata celula
70              c1.gridwidth = 1;
71              c1.gridheight = 1;
72              c1.gridx = 0;
73              c1.gridy = 0;
74
75              GridBagConstraints c2 = new GridBagConstraints();
76              c2.fill = GridBagConstraints.BOTH;
77              c2.gridwidth = 2;
78              c2.gridheight = 1;
79              c2.gridx = 1;
80              c2.gridy = 0;
```

```
81
82              GridBagConstraints c3 = new GridBagConstraints();
83              c3.fill = GridBagConstraints.BOTH;
84              c3.gridwidth = 2;
85              c3.gridheight = 1;
86              c3.gridx = 0;
87              c3.gridy = 1;
88
89              GridBagConstraints c4 = new GridBagConstraints();
90              c4.fill = GridBagConstraints.BOTH;
91              c4.gridwidth = 1;
92              c4.gridheight = 1;
93              c4.gridx = 1;
94              c4.gridy = 1;
95
96              Button b1 = new Button("GB1");
97              Button b2 = new Button("GB2");
98              Button b3 = new Button("GB3");
99              Button b4 = new Button("GB4");
100
101             layout.setConstraints(b1, c1);
102             p.add(b1);
103             layout.setConstraints(b2, c2);
104             p.add(b2);
105             layout.setConstraints(b3, c3);
106             p.add(b3);
107             layout.setConstraints(b4, c4);
108             p.add(b4);
109         }
110         else
111         {
112             p.add(new Button("1"));
113             p.add(new Button("2"));
114             p.add(new Button("3"));
115             p.add(new Button("4"));
116         }
117     }
118
119     @Override
120     public boolean handleEvent(Event evt)
121     {
122         if (evt.id == Event.WINDOW_DESTROY)
123         {
124             System.exit(0);
125         }
126         else if (evt.id == Event.ACTION_EVENT && evt.target
               instanceof MenuItem)
127         {
128             if ("Exit" == evt.arg)
129             {
```

```
130                     System.exit(0);
131                 }
132             }
133 
134             return super.handleEvent(evt);
135         }
136 }
```