

INFORMATION ONLY

Cuprins

Cuvânt înainte

Introducere

PARTEA I-A: Fundamentele programării

Calculatorul este agenda, telefonul și televizorul secolului următor. "țiți să-l folosiți?"

Capitolul I: Arhitectura calculatoarelor

Modul în care programăm un calculator depinde direct de modul în care acesta este construit. Nu putem gândi reprezentarea și prelucrarea informațiilor fără să știm care sunt principiile constructive de bază ale calculatorului pentru care gândim. De aceea, înainte de toate, să încercăm să aflăm cum funcționează un calculator modern.

Capitolul II: Limbaje de programare

Pentru a putea programa, este necesar să avem cu calculatorul un limbaj comun, numit limbaj de programare, cu ajutorul căruia să descriem informațiile pe care dorim să le prelucrăm și modul în care acestea trebuie prelucrate. Limbajele de programare actuale, deși foarte diferite între ele, oferă în principiu aceleași tipuri de informație primitivă și aceleași tipuri de operații.

Capitolul III: Reprezentarea informațiilor cu obiecte

Programarea orientată pe obiecte este cea mai nouă tehnologie de proiectare a programelor de calculator. Aceasta ne permite, în esență, să modelăm informațiile cu care lucrează calculatorul într-un mod similar cu acela în care percepem obiectele reale: fiecare obiect este un uniune între proprietăți și comportament.

PARTEA A II-A: Descrierea limbajului Java

Despre un om care știe două limbi se spune că este de două ori mai deștept. Dar ce spunem despre un programator care știe două limbaje?

Capitolul IV: Structura lexicală Java

La începutul prezentării fiecărui limbaj de programare trebuie să definim exact setul de caractere cu care lucrează limbajul, modul de construcție a identificatorilor, comentariilor și literalilor specifici, care sunt operatorii și separatorii limbajului. În plus, trebuie să avem o listă completă a cuvintelor rezervate Java.

Capitolul V: Componente de bază ale programelor Java

Componentele de bază ale oricărui limbaj de programare sunt variabilele, expresiile și instrucțiunile. Ca orice alt limbaj de programare, Java își definește propriile tipuri de date, implementează principalele instrucțiuni ale programării structurate și își definește propriii operatori și modul în care pot fi folosiți aceștia la construcția expresiilor.

Capitolul VI: Obiecte Java

Pentru a fi un limbaj orientat obiect, Java trebuie să definească o cale de a crea noi clase de obiecte și de a le instanția. În plus, limbajul trebuie să ofere suport pentru principalele trăsături ale limbajelor orientate obiect: încapsulare și derivare.

PARTEA A III-A: Tehnici de programare în Java

A cunoaște o limbă nu este totul. Mai trebuie să ai și ceva de spus.

Capitolul VII: Modele de programare

Modul în care este lansat un program și modul în care evoluează execuția

sa ulterioară depinde de limbajul în care a fost scris programul și de mediul hardware și software în care rulează programul. Limbajul Java definește două moduri diferite pentru execuția programelor: aplicațiile Java și appleturile Java.

Capitolul VIII: Structura programelor

În mod normal, sursele unui program complex sunt împărțite în mai multe fișiere pentru o administrare mai ușoară. În Java, există în mod normal câte un fișier sursă pentru fiecare clasă de obiecte în parte. Limbajul Java definește o serie de structuri sintactice care permit conectarea codului rezultat în urma compilării diferitelor clase precum și organizarea acestor clase într-o structură ierarhică de pachete.

Capitolul IX: Fire de execuție și sincronizare

În contextul sistemelor de operare moderne, lucrul cu fire multiple de execuție este în același timp o necesitate și o modă. Din păcate, fiecare sistem de operare își definește propria sa bibliotecă de funcții pentru a suporta această facilități. Limbajul Java, pentru a fi portabil cu adevărat, este obligat să-și definească propriul său suport pentru firele multiple de execuție.

Bibliografie

[JavaRo](#)

[\(C\) IntegraSoft 1996-1998](#)

Cuvânt înainte

Deși trăim într-o societate în care rata de schimb a tehnologiei a ajuns să ne depășească, există domenii care se schimbă mult prea lent față de așteptările noastre. Să luăm de exemplu calculatoarele. Nu există zi în care să nu auzim de noutăți în ceea ce privește viteza, numărul de culori sau miniaturizarea. Nu există zi în care să nu auzim de noi aplicații și de noi domenii în care a fost introdusă informatica. Și totuși, nimic esențial nu s-a schimbat în partea de fundamente. Aceeași arhitectură numerică guvernează întreg spectrul de calculatoare aflate azi pe piață ca și acum jumătate de secol.

În ceea ce privește comunicarea om-calculator, lucrurile nu stau cu mult mai bine. Deși nu mai comunicăm folosindu-ne de cifre binare și nici în limbaje apropiate de mașină, comunicăm în continuare folosindu-ne de câteva primitive structurale de tip bucle sau condiții plus directive de calcul și transfer al informațiilor. Abstracții precum programarea logică, funcțională sau orientată obiect nu extind blocurile de bază cu care acționăm asupra mașinii, le fac doar mai accesibile pentru modul nostru de a gândi.

Într-un fel, programarea face exact aceeași greșală pe care a făcut-o și logica clasică statuând că orice enunț nu poate fi decât adevărat sau fals, fără nici o altă nuanțare. În pasul imediat următor s-au stabilit câteva enunțuri adevărate fără demonstrație și s-a considerat că toate celelalte enunțuri trebuie deduse din ele. Programarea a făcut aceleași presupuneri în ceea ce privește comunicarea om-calculator: există câteva primitive funcționale de bază și toată comunicarea trebuie să poată fi exprimată cu ajutorul acestora.

În aceste condiții, este normal ca apariția fiecărui nou limbaj de programare să trezească un interes major în lumea informatică. De fiecare dată, sperăm că noul limbaj ne va permite o exprimare mai ușoară, mai flexibilă, mai bogată. De aceea, la apariția fiecărui limbaj de programare care promite să iasă din anonimat, ne grăbim să aflăm care sunt principalele facilități care ni se oferă.

Apariția limbajului Java a fost însoțită de multă publicitate și de mult scepticism. Părerile au variat de la a spune că Java este o revoluție în programarea calculatoarelor și, mai ales, a rețelelor de calculatoare până la afirmații care neagă orice caracter novator al noului limbaj și care pun succesul Java în exclusivitate pe seama numelui de care se bucură firma Sun și a campaniei publicitare inteligent condusă de marketingul acestei firme.

Indiferent însă de motivul pentru care limbajul Java a ajuns la cota extraordinară de popularitate pe care o simțim cu toții, sentimentul general este acela că Java a devenit deja o realitate și va rămâne așa pentru suficient de mult timp. Suficient de mult pentru a fi interesanți să apreciem în cunoștință de cauză care este adevărul despre acest nou limbaj. Oricum, de la apariția limbajului C++, acum mai bine de un deceniu în urmă, nici un alt limbaj nu a înnegrit atâta hârtie și nu a adus atâtea beneficii vânzătorilor de servicii Internet.

Sper ca această carte să ofere suportul de care aveți nevoie pentru a vă putea forma propria părere: avem

în faþã o revoluþie sau un alt fapt comun?

Târgu Mureº, 24 aprilie 1996

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Introducere

[Scurt istoric](#)

[Ce este Java?](#)

[Despre această carte](#)

[Convenții utilizate în această carte](#)

[Sugestii și reclamații](#)

[Alte surse de informații](#)

[Mulțumiri](#)

Scurt istoric

Limbajul Java împreună cu mediul său de dezvoltare și execuție au fost proiectate pentru a rezolva o parte dintre problemele actuale ale programării. Proiectul Java a pornit cu scopul declarat de a dezvolta un software performant pentru aparatele electronice de larg consum. Aceste echipamente se definesc ca: mici, portabile, distribuite și lucrând în timp real. De la aceste aparate, ne-am obișnuit să cerem fiabilitate și ușurință în exploatare.

Limbajul luat inițial în considerare a fost C++. Din păcate, atunci când s-a încercat crearea unui mediu de execuție care să respecte toate aceste condiții s-a observat că o serie de trăsături ale C++ sunt incompatibile cu necesitățile declarate. În principal, problema vine din faptul că C++ este prea complicat, folosește mult prea multe convenții și are încă prea multe elemente de definiție lăsate la latitudinea celor care scriu compilatoare pentru o platformă sau alta.

În aceste condiții, firma Sun a pornit proiectarea unui nou limbaj de programare asemănător cu C++ dar mult mai flexibil, mai simplu și mai portabil. Așa s-a născut Java. Părintele noului limbaj a fost James Gosling care va este poate cunoscut ca autor al editorului **emacs** și al sistemului de ferestre grafice **NeWS**. Proiectul a început încă din 1990 dar Sun a făcut publică specificația noului limbaj abia în 1995 la SunWorld în San Francisco.

Numele inițial al limbajului a fost **Oak**, numele unui copac care crește în fața biroului lui James Gosling. Ulterior, s-a descoperit că numele fusese deja folosit în trecut pentru un alt limbaj de programare așa că a fost abandonat și înlocuit cu Java, spre deliciul programatorilor care iubesc cafelele și aromele exotice. 🏠

Ce este Java?

În primul rând, Java încearcă să rămână un limbaj *simplu* de folosit chiar și de către programatorii neprofesioniști, programatori care doresc să se concentreze asupra aplicațiilor în principal și abia apoi asupra tehnicilor de implementare a acestora. Această trăsătură poate fi considerată ca o reacție directă

la complexitatea considerabilă a limbajului C++.

Au fost îndepărtate din Java aspectele cele mai derutante din C++ precum supraîncărcarea operatorilor și menținerea multiplă. A fost introdus un colector automat de gunoaie care să rezolve problema dealocării memoriei în mod uniform, fără intervenția programatorului. Colectorul de gunoaie nu este o trăsătură nouă, dar implementarea acestuia în Java este făcută inteligent și eficient folosind un fir separat de execuție, pentru că Java are încorporate facilități de execuție pe mai multe fire de execuție. Astfel, colectarea gunoaielor se face de obicei în timp ce un alt fir așteaptă o operație de intrare-ieșire sau pe un semafor.

Limbajul Java este independent de arhitectura calculatorului pe care lucrează și foarte *portabil*. În loc să genereze cod nativ pentru o platformă sau alta, compilatorul Java generează o secvență de instrucțiuni ale unei mașini virtuale Java. Execuția aplicațiilor Java este *interpretată*. Singura parte din mediul de execuție Java care trebuie portată de pe o arhitectură pe alta este mediul de execuție cuprinzând interpretorul și o parte din bibliotecile standard care depind de sistem. În acest fel, aplicații Java compilate pe o arhitectură SPARC de exemplu, pot fi rulate fără recompilare pe un sistem bazat pe procesoare Intel.

Una dintre principalele probleme ale limbajelor interpretate este viteza de execuție, considerabil scăzută față de cea a limbajelor compilate. Dacă nu vă mulțumește viteza de execuție a unei astfel de aplicații, puteți cere mediului de execuție Java să genereze automat, plecând de la codul mașinii virtuale, codul specific mașinii pe care lucrați, obținându-se astfel un executabil nativ care poate rula la viteză maximă. De obicei însă, în Java se compilează doar acele părți ale programului mari consumatoare de timp, restul rămânând interpretate pentru a nu se pierde flexibilitatea. Mediul de execuție însuși este scris în C respectând standardele POSIX, ceea ce îl face extrem de portabil.

Interpretorul Java este gândit să lucreze pe *mașini mici*, precum ar fi procesoarele cu care sunt dotate aparatele casnice. Interpretorul plus bibliotecile standard cu legare dinamică nu depășesc 300 Kocteți. Chiar împreună cu interfața grafică totul rămâne mult sub 1 Moctet, exact ca-n vremurile bune.

Limbajul Java este *orientat obiect*. Cu el se pot crea clase de obiecte și instanțe ale acestora, se pot încapsula informațiile, se pot menține variabilele și metodele de la o clasă la alta, etc. Singura trăsătură specifică limbajelor orientate obiect care lipsește este menținerea multiplă, dar pentru a suplini această lipsă, Java oferă o facilități mai simplă, numită interfață, care permite definirea unui anumit comportament pentru o clasă de obiecte, altul decât cel definit de clasa de bază. În Java orice element este un obiect, în afară de datele primare. Din Java lipsesc funcțiile și variabilele globale. Ne rămân desigur metodele și variabilele statice ale claselor.


Java este *distribuit*, având implementate biblioteci pentru lucrul în rețea care ne oferă TCP/IP, URL și încărcarea resurselor din rețea. Aplicațiile Java pot accesa foarte ușor rețeaua, folosindu-se de apelurile către un set standard de clase.

Java este *robust*. În Java legarea funcțiilor se face în timpul execuției și informațiile de compilare sunt

disponibile până în momentul rulării aplicației. Acest mod de lucru face ca sistemul să poată determina în orice moment neconcordanța dintre tipul referit la compilare și cel referit în timpul execuției evitându-se astfel posibile intruziuni răuvoitoare în sistem prin intermediul unor referințe falsificate. În același timp, Java detectează referințele nule dacă acestea sunt folosite în operații de acces. Indicii în tablourile Java sunt verificați permanent în timpul execuției și tablourile nu se pot parcurge prin intermediul unor pointeri așa cum se întâmplă în C/C++. De altfel, pointerii lipsesc complet din limbajul Java, împreună cu întreaga lor aritmetică, eliminându-se astfel una din principalele surse de erori. În plus, eliberarea memoriei ocupate de obiecte și tablouri se face automat, prin mecanismul de colectare de gunoarie, evitându-se astfel încercările de eliberare multiplă a unei zone de memorie.

Java este un limbaj cu *securitate ridicată*. El verifică la fiecare încărcare codul prin mecanisme de CRC și prin verificarea operațiilor disponibile pentru fiecare set de obiecte. Robustețea este și ea o trăsătură de securitate. La un al doilea nivel, Java are incorporate facilități de protecție a obiectelor din sistem la scriere și/sau citire. Variabilele protejate într-un obiect Java nu pot fi accesate fără a avea drepturile necesare, verificarea fiind făcută în timpul execuției. În plus, mediul de execuție Java poate fi configurat pentru a proteja rețeaua locală, fișierele și celelalte resurse ale calculatorului pe care rulează o aplicație Java.

Limbajul Java are inclus suportul nativ pentru aplicații care lucrează cu *mai multe fire de execuție*, inclusiv primitive de sincronizare între firele de execuție. Acest suport este independent de sistemul de operare, dar poate fi conectat, pentru o performanță mai bună, la facilitățile sistemului dacă acestea există.

Java este *dinamic*. Bibliotecile de clase în Java pot fi reutilizate cu foarte mare ușurință. Cunoscuta problemă a fragilității superclasei este rezolvată mai bine decât în C++. Acolo, dacă o superclasă este modificată, trebuie recompile toate subclasele acesteia pentru că obiectele au o altă structură în memorie. În Java această problemă este rezolvată prin legarea târzie variabilelor, doar la execuție. Regăsirea variabilelor se face prin nume și nu printr-un deplasament fix. Dacă superclasa nu are o parte dintre vechile variabile și metode, ea va putea fi refolosită fără să fie necesară recompilarea subclaselor acesteia. Se elimină astfel necesitatea actualizării aplicațiilor, generată de apariția unei noi versiuni de bibliotecă așa cum se întâmplă, de exemplu, cu MFC-ul Microsoft (și toate celelalte ierarhii C++). 

Despre această carte

Această carte a fost scrisă pentru a veni în sprijinul acelor care doresc să programeze în limbajul Java. Spre deosebire de majoritatea cărților existente la ora actuală pe piața internațională, nu prea multe de altfel, cartea de față se focalizează asupra facilităților pe care le oferă noul limbaj, lăsând pe planul al doilea descrierea bibliotecilor standard, impresionante de altfel, definite de către Sun și colaboratorii acestuia. Alegerea a fost făcută din convingerea că bibliotecile vin și trec, se dezvoltă, în timp ce limbajul rămâne.

Cartea se adresează în același timp începătorilor și programatorilor profesioniști. Începătorii vor găsi în prima parte noțiuni fundamentale necesare oricărui programator. Profesioniștii, în schimb, vor găsi o referință completă a limbajului Java.

Prima parte a cărpii îi introduce cititorul pas cu pas în fundamentele funcționării și programării calculatoarelor tratând la un nivel accesibil noțiuni precum memoria și procesorul, datele și instrucțiunile, clasele de obiecte împreună cu trăsăturile fundamentale ale programării orientate obiect.

Partea a doua a cărpii prezintă limbajul Java în detaliu împreună cu o serie de exemple simple. Această parte este concepută pentru a fi în același timp o introducere în sintaxa Java pentru cei care au programat deja și o introducere în constrângerile sintactice și semantica unui limbaj de programare pentru începători.

Partea a treia a cărpii prezintă câteva dintre aspectele fundamentale ale programării în Java precum aplicațiile, apleturile, pachetele de clase, firele de execuție și tratarea excepțiilor. Această parte este concepută de sine stătătoare și poate servi ca exemplu de programare în Java. Fiecare secțiune din această parte conține exemple extinse de aplicații scrise în Java, comentate în sursă și în afara acesteia.

Cititorul începător în ale programării trebuie să parcurgă cartea de la început până la sfârșit pentru a putea intra treptat în tainele programării în general și ale limbajului Java.

Pentru cititorii avansați, prima parte nu constituie un interes major și poate fi sărită fără implicații majore în înțelegerea materialului din părțile următoare. Acești cititori pot să treacă direct la partea a treia pentru a urmări exemplele și abia după aceea să revină la partea a doua pentru informații detaliate asupra sintaxei și facilităților Java. Partea a doua poate fi folosită și pe post de referință.

O singură excepție: ultima secțiune din prima parte introduce noțiunea de interfață, puțin cunoscută programatorilor în alte limbaje decât Java. Este util ca această secțiune să fie consultată de către toți cititorii, indiferent de nivelul de pregătire în care se află. 🏠

Convenții utilizate în această carte

Pentru descrierea sintacticii limbajului Java, am apelat la următoarele convenții obișnuite de reprezentare a regulilor gramaticale:

- Cuvintele rezervate ale limbajului Java sunt scrise în caractere îngroșate, ca: **while**, **do**, **final**.
- Cuvintele care țin locul construcțiilor reale ale programului sunt scrise cu caractere oblice, ca în:

```
if(Condiție) Instrucțiune1 else Instrucțiune2
```

Condiție, *Instrucțiune1* și *Instrucțiune2*, care apar în descriere, trebuiesc înlocuite cu condițiile adevărate, respectiv cu instrucțiunile adevărate care trebuiesc executate în funcție de condiție. De exemplu, o construcție reală, care respectă descrierea de mai sus ar putea fi:

```

if( i < j )
min = i;
else
min = j;

```

- O listă de termeni separați prin caractere |, se poate înlocui cu oricare dintre termenii listei.

De exemplu:

public | **private** | **protected**

înseamnă că în locul acestui termen poate să apară oricare dintre cuvintele rezervate specificate în listă, **public**, **private** sau **protected**.

- Un termen pus între paranteze pătrate este opțional. De exemplu, în descrierea:

Tip NumeVariabilă [Inițializator] ;

Partea de *Inițializator* poate să lipsească.

- Dacă, după un termen apare caracterul *, acesta reprezintă faptul că termenul se poate repeta de un număr arbitrar de ori, eventual niciodată. De exemplu, în:

class *NumeClasă ClauzăImplements**

caracterul * reprezintă faptul că termenul notat prin *ClauzăImplements* se poate repeta de un număr arbitrar de ori, eventual niciodată.

Pe marginea din stânga a paginii, veți întâlni o serie de simboluri grafice, cu specificație precisă, care vă vor permite să navigați mai ușor în interiorul cărții. Iată semnificația acestora:


Semnul din stânga reprezintă faptul că în paragraful marcat introduce *definiția* unui nou termen. Termenul nou introdus este reprezentat în interiorul semnului și este scris cu caractere oblice în textul paragrafului. În cazul de față cuvântul nou introdus este cuvântul *definiția*. Atunci când găsiți o trimitere la un paragraf pentru explicații, căutați acest semn pe margine pentru a găsi mai repede definiția noțiunii dorite.

Semnul din stânga reprezintă o *trimitere înapoi* către o noțiune deja prezentată în partea de fundamente. Această trimitere conține deasupra simbolului grafic numele noțiunii pe care trebuie să îl căutați într-un simbol de definiție, și sub simbolul grafic numărul paragrafului unde se află definiția. Cititorul

Începător ca și cel avansat poate urma aceste trimiteri pentru a-și împărtăși cunoștințele sau pentru a se familiariza cu cuvintele sau expresiile folosite pentru a desemna noțiuni greu traducibile.

Semnul din stânga reprezintă o *referire în avans* a unei noțiuni care va fi introdusă mai târziu. Acest semn este destul de rar în interiorul cărții, dar este totuși nevoie de el pentru a face referire la anumite facilități care îți găsesc cel mai bine locul în paragraful în care apare semnul. Puteți sări fără probleme aceste precizări la prima citire a capitoului și să le luați în seamă doar atunci când folosiți cartea pe post de referință pentru limbajul Java.


Semnul din stânga reprezintă referirea unei noțiuni care nu este descrisă în carte. Acest semn apare atunci când se face o referință la una din clasele standard ale mediului Java. Puteți găsi informații exacte despre clasa respectivă la adresa www.javasoft.com, sau puteți încărca din Internet o documentație completă prin **ftp** de la adresa [ftp.javasoft.com](ftp://ftp.javasoft.com/docs), directorul [/docs](#).

Semnul din stânga reprezintă o trimitere către altă carte, în care noțiunile din paragraf sunt prezentate mai pe larg. Am folosit acest semn doar de două ori, pentru a trimite către o carte disponibilă la aceeași editură cu cartea prezentă, despre HTML, scrisă de Dumitru Rădoiu. O puteți găsi pe prima poziție în bibliografie. 

Sugestii și reclamații

Posibilele erori care s-au strecurat în această carte cad în sarcina exclusivă a autorului ei care îți cere pe această cale scuze în avans. Orice astfel de eroare sau neclaritate cu privire la conținutul cărții poate fi comunicată direct autorului pe adresa erotariu@agora.ro sau:

Eugen Rotariu
Computer Press Agora
Str. Tudor Vladimirescu, Nr. 63/1, Cod. 4300, Târgu Mureș
România

În Internet, puteți să contactați editura Computer Press Agora la adresa www.agora.ro. Pe server există o pagină separată dedicată acestei cărți. Puteți verifica periodic această pagină pentru eventuale corecturi, exemple suplimentare sau adăugiri la conținutul cărții. 

Alte surse de informații

Subiectul Java este departe de a fi încheiat în această carte așa că, dacă reacția dumneavoastră este pozitivă, voi încerca să construiesc o a doua carte dedicată claselor standard Java, mediilor de dezvoltare, din ce în ce mai numeroase, ierarhiilor de bază de date, obiecte distribuite și așa mai departe care și-au făcut apariția de la Sun sau din altă parte în ultimul timp. Nu ezitați să-mi dați de știre dacă veți considera utilă o asemenea continuare.

Până atunci însă, puteţi consulta cărpile listate în secţiunea de bibliografie de la sfârşitul acestei cărţi. Numai cărţi în limba engleză în acest domeniu deocamdată, dar mă aştept ca situaţia să se schimbe dramatic în perioada imediat următoare. Noi cărţi de Java dedicate cititorilor români nu vor întârzia să apară.

Internet-ul este de asemenea o sursă interminabilă de informaţii, situri Java există şi vor continua să apară în întreaga lume. Adresa de bază este probabil www.javasoft.com, adresă care vă pune în legătură directă cu firma care a creat şi întreţine în continuare dezvoltarea Java. În plus, puteţi consulta revista electronică JavaWorld de la adresa www.javaworld.com care conţine întotdeauna informaţii fierbinţi, cursuri, appleturi şi legături către alte adrese unde vă puteţi îmbogăţi cunoştinţele despre Java. Pentru documentaţii, exemple şi noutăţi în lumea Java puteţi consulta şi www.gamelan.com sau www.blackdown.org.

Pentru cei care doresc să dezvolte appleturi pe care să le insereze în propriile pagini HTML, recomand în plus citirea cărţii lui Dumitru Rădoiu, HTML - Publicaţii Web, editată de asemenea la Computer Press Agora.

În fine, nu încetaţi să cumpăraţi revistele PC Report şi Byte România ale aceleiaşi edituri, pentru că ele vor conţine ca de obicei informaţii de ultimă oră, cursuri şi reportaje de la cele mai noi evenimente din lumea calculatoarelor. 🏠

Mulţumiri

Mulţumesc tuturor celor care, voit sau nu, au făcut posibilă această carte. Mulţumesc celor de la editura Computer Press Agora şi managerului ei Romulus Maier pentru că mi-au facilitat publicarea acestei cărţi. Fără munca lor, cartea s-ar fi aflat în continuare în vitrina proprie cu visuri nerealizate. Mulţumesc celor care mi-au dăruit o parte din timpul lor preţios pentru a citi şi comenta primele versiuni ale cărţii: Iosif Fettich, Mircea şi Monica Cioată, Alexandru Horvath. Mulţumesc celor cu care am discutat de atâtea ori despre soarta calculatoarelor, programelor, României şi lumii în general. Ei au făcut din mine omul care sunt acum: Mircea Sârbu, Dumitru Rădoiu, Szabo Laszlo, Mircea Pantea. Mulţumesc celor care s-au ocupat de designul şi tehnoredactarea acestei cărţi în frunte cu Adrian Pop şi Octav Lipovan. Cel dintâi lucru care v-a atras la această carte este munca lor. Carmen, îţi mulţumesc că nu te-ai dat bătăută. Va veni şi vremea în care sărbătorim cu calculatoarele oprite. 🏠

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul I

Arhitectura calculatoarelor

[1.1](#) *Modelul Von Neumann de arhitectură a calculatoarelor*

[1.2](#) *Organizarea memoriei interne*

[1.3](#) *Reprezentarea informațiilor în memoria internă*

[1.4](#) *Modelul funcțional al calculatoarelor*

1.1 Modelul Von Neumann de arhitectură a calculatoarelor

Descrierea care urmează este o descriere a modelului Von Neumann de construcție a calculatoarelor. Se pot aduce destule critici acestui model care domină sever încă de la începuturile mașinilor de calcul electronic, dar el continuă să fie singurul model funcțional.

Modelul Von Neumann definește calculatorul ca pe un ansamblu format dintr-o unitate centrală și o memorie internă. *Unitatea centrală* sau *procesorul* este responsabilă cu administrarea și prelucrarea informațiilor în timp ce *memoria internă* servește la depozitarea acestora. În terminologia calculatoarelor, depozitarea informațiilor în memoria internă a calculatorului se numește *memorare*.

Acest ansamblu unitate centrală plus memorie internă comunică cu exteriorul prin intermediul unor *dispozitive periferice*. Dispozitivele periferice pot fi de *intrare* sau de *ieșire* în funcție de direcția în care se mișcă datele. Dacă datele sunt furnizate de dispozitivul periferic și transferate spre unitatea centrală, atunci dispozitivul este de intrare precum sunt tastatura sau mausul. Dacă datele sunt generate de unitatea centrală și transmise spre dispozitivul periferic atunci dispozitivul este de ieșire precum sunt ecranul sau imprimanta. Există și dispozitive mixte de intrare/ieșire precum sunt discurile pentru memorarea externă a informațiilor.

Tastatura calculatorului reprezintă un set de butoane (taste) inscripționate care ne permite transmiterea către unitatea centrală a unor litere, cifre, semne de punctuație, simboluri grafice sau comenzi funcționale. *Mausul* reprezintă un dispozitiv simplu, mobil, care ne permite indicarea unor regiuni ale ecranului cu ajutorul unui cursor.

În plus, mausul ne permite activarea regiunilor respective cu ajutorul celor 1-2-3 butoane ale sale.

Ecranul este un dispozitiv cu ajutorul căruia calculatorul comunică informații spre exterior. Aceste informații apar sub formă de litere, cifre, semne de punctuație, simboluri grafice sau desene oarecare într-o varietate mai mare sau mai mică de culori. Informația de pe ecran se pierde odată cu redesenarea acestuia. Pentru transferul acestor informații pe hârtie și îndosărierea lor s-au creat alte dispozitive periferice, numite *imprimante*.

Memoria internă pierde informațiile odată cu oprirea alimentării calculatorului. Pentru a salva informațiile utile, precum și programele de prelucrare ale acestora este nevoie de dispozitive de memorare permanente. Din această categorie fac parte *discurile* calculatorului. Există mai multe modele de discuri precum discurile fixe, discurile flexibile sau compact-discurile, fiecare dintre acestea având caracteristici, viteze de acces și capacități de memorare diferite. Informațiile salvate pe discuri pot fi încărcate din nou în memoria internă la o pornire ulterioară a calculatorului. Vă veți întreba desigur de ce este nevoie de două tipuri distincte de memorie: pentru că discurile au viteze de acces mult prea mici pentru a putea fi folosite direct de către unitatea centrală.

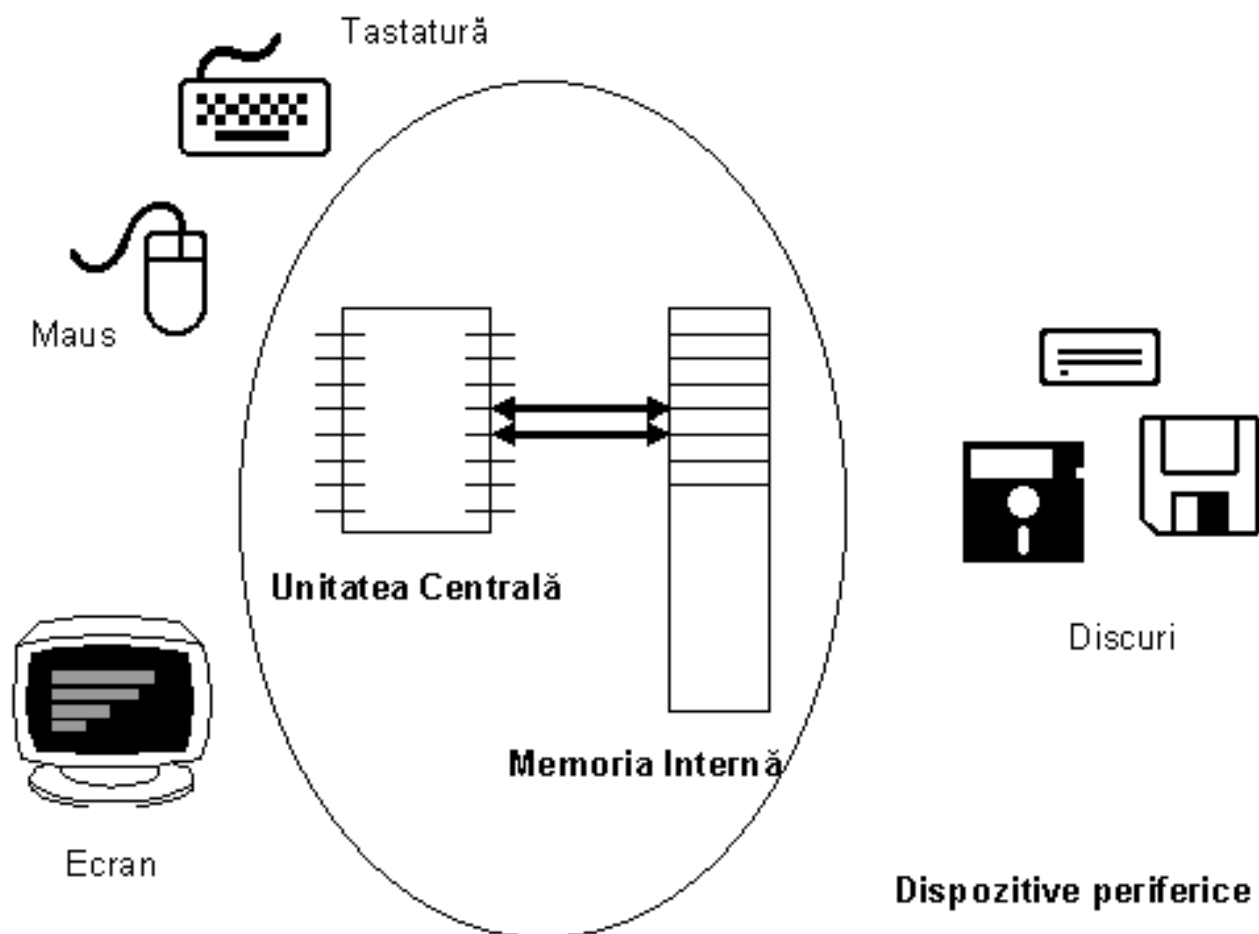


Figura 1.1 Modelul constructiv al calculatoarelor Von Neuman.

Deși modelul constructiv de bază al calculatoarelor nu a evoluat prea mult, componenta tehnologică a acestora s-a aflat într-o permanentă evoluție. Transformarea vizează la ora actuală viteza de lucru și setul de instrucțiuni ale unității centrale, capacitatea și viteza de stocare a memoriei interne precum și tipurile și calitatea dispozitivelor periferice. 🏠

1.2 Organizarea memoriei interne

Memoria calculatoarelor actuale este, din punct de vedere logic, o înțiruire de *cifre binare*, 0 sau 1. Alegerea bazei 2 de numerăție are în principal rațiuni constructive: este mult mai ușor și mult mai fiabil

să reprezintă un principiu binar ca absența/prezența sau plus/minus decât unul nuanțat. O cifră binară este numită, în termeni de calculatoare, *bit*.

Biții sunt grupați, opt câte opt, în unități de memorare numite *octeți*.

Iarăși, alegerea cifrei opt are rădăcini istorice: era nevoie de o putere a lui doi care să fie cât mai mare, pentru a putea permite transferuri rapide între diversele componente ale calculatorului (unitate centrală, memorie, dispozitive periferice), dar totodată suficient de mică pentru ca realizarea dispozitivelor implicate, cu tehnologia existentă, să fie posibilă. Cifra opt avea în plus avantajul că permitea reprezentarea tuturor caracterelor tipăribile necesare la ora respectivă precum: literele, cifrele sau semnele de punctuație. Într-un octet se pot reprezenta până la 256 (2^8) astfel de caractere. În prezent octetul este depășit datorită necesității de reprezentare a caracterelor tuturor limbilor scrise din lume.

Pentru a accesa o informație în memorie este nevoie de un mod de a referi poziția acesteia. Din acest motiv, octeții memoriei au fost numerotați unul câte unul începând de la 0 până la numărul maxim de octeți în memorie. Numărul de ordine al unui octet îl vom numi pentru moment *adresă*. Noțiunea de adresă și-a extins semnificația în ultimul timp dar, pentru înțelegerea acestui capitol, explicația de mai sus este suficientă.

Zona fizică de memorie rezervată unei anumite informații se numește *locuș* informației respective în memorie. În unele dintre locușurile din memorie putem păstra chiar adresa unor alte locușuri din memorie. Informația memorată în aceste locușuri se numește *referință*. Cu alte cuvinte, o referință este o informație memorată într-o locuș de memorie care ne trimite spre (se referă la) o altă locuș de memorie. O locuș de memorie se poate întinde pe mai mult decât un octet. 🏠

1.3 Reprezentarea informațiilor în memoria internă

După cum ați putut observa din cele prezentate până acum, în memoria calculatorului nu se pot înscrie decât numere naturale. Mai precis, fiecare octet de memorie poate memora un număr de la 0 la 2^8-1 , adică 255. Orice altă informație pe care dorim să o reprezentăm în memoria calculatorului trebuie redusă la unul sau mai multe numere naturale mici.

Această trăsătură a modelului actual de memorare introduce un pas suplimentar de abstractizare în procesul de proiectare de aplicații, și anume pasul în care se construiește un model de reprezentare în memorie a datelor, necesar aplicației.

Să presupunem, de exemplu, că o aplicație necesită reprezentarea în memoria calculatorului a unui set de culori. Pentru memorarea acestor culori este nevoie de o convenție care să stabilească o corespondență biunivocă între setul de culori și setul de numere naturale folosite la reprezentarea acestora. Corespondența este biunivocă pentru că ea trebuie să ofere posibilitatea de a regăsi în mod unic o culoare plecând de la un număr și, în același timp, să ofere o reprezentare unică, sub formă de număr, pentru fiecare culoare. De exemplu, putem pune în corespondență culoarea neagră cu numărul 0, culoarea roșie

cu numărul 1, culoarea albastră cu numărul 2 și așa mai departe. Ori de câte ori vom memora 2 vom memora albastru și ori de câte ori vom dori să memorăm roșu, vom memora 1.

Ca rezultat al folosirii acestei abstractizări, datele aplicației devin dependente de convenția de reprezentare utilizată. Presupunând că o aplicație construiește în memorie o imagine grafică folosindu-se de o anumită corespondență dintre culori și numere, oricare altă aplicație care vrea să utilizeze imaginea respectivă trebuie să folosească aceeași convenție.

O cale similară de rezolvare vom întâlni și la reprezentarea caracterelor. *Caracterele* sunt denumirea într-un singur cuvânt a literelor, cifrelor, semnelor de punctuație sau simbolurilor grafice reprezentate în memorie. Este nevoie de o convenție prin care atașăm fiecărui caracter câte un număr natural memorabil într-un octet.

În cazul reprezentării caracterelor, există chiar un standard internațional care definește numerele, reprezentabile pe un octet, corespunzătoare fiecărui caracter în parte, numit standardul *ASCII*. Alte standarde, cum ar fi standardul *Unicode*, reprezintă caracterele pe doi octeți, ceea ce le dă posibilitatea să ia în considerare o gamă mult mai largă de caractere.

Caracter	Reprezentare pe un octet (ASCII)
A-Z	65-90
a-z	97-122
0-9	48-57
Ã,ã	195,227
Î,î	206,238
Â,â	194,226
ª,º	170,186
Ð,ð	222,254

Tabelul 1.1 Un fragment din codurile ASCII și Unicode de reprezentare a caracterelor grafice în memoria calculatoarelor.

Desigur, este greu să ținem minte codul numeric asociat fiecărui caracter sau fiecărei culori. Este nevoie de pași suplimentari de codificare, care să pună informația în legătură cu simboluri mai ușor de ținut minte decât numerele. De exemplu, este mult mai ușor pentru noi să ținem minte cuvinte sau imagini. Dar să nu uităm niciodată că, pentru calculator, cel mai ușor este să memoreze și să lucreze cu numere. 🏠

1.4 Modelul funcțional al calculatoarelor

Fiecare calculator definește un număr de operații care pot fi executate de unitatea sa centrală. Aceste operații sunt în principal destinate memorării sau recuperării informațiilor din memoria internă, calculelor aritmetice sau logice și controlului dispozitivelor periferice. În plus, există un număr de instrucțiuni pentru controlul ordinii în care sunt executate operațiile.

O instrucțiune este o operație elementară executabilă de către unitatea centrală a unui calculator. O secvență de mai multe instrucțiuni executate una după cealaltă o vom numi program.

Instrucțiunile care compun un program trebuie să fie reprezentate în memorie, la fel ca orice altă informație, din cauza faptului că unitatea centrală nu are posibilitatea să-și păstreze programele în interior. Pentru memorarea acestor instrucțiuni este nevoie de o nouă convenție de reprezentare care să asocieze un număr sau o secvență de numere naturale fiecărei instrucțiuni a unității centrale.

Execuția unui program de către calculator presupune încărcarea instrucțiunilor în memoria internă și execuția acestora una câte una în unitatea centrală. Unitatea centrală citește din memorie câte o instrucțiune, o execută, după care trece la următoarea instrucțiune. Pentru păstrarea secvenței, unitatea centrală memorează în permanență o referință către următoarea instrucțiune într-o locație internă numită *indicator de instrucțiuni*.

Modelul de execuție liniară a instrucțiunilor, în ordinea în care acestea sunt așezate în memorie, este departe de a fi acceptabil. Pentru a fi util, un program trebuie să poată să ia decizii de schimbare a instrucțiunii următoare în funcție de informațiile pe care le prelucrează. Aceste decizii pot însemna uneori comutarea execuției de la o secvență de instrucțiuni la alta. Altfel, este necesar să putem executa o secvență de instrucțiuni în mod repetat până când este îndeplinită o anumită condiție exprimabilă cu ajutorul informațiilor din memorie. Numărul de repetări ale secvenței de instrucțiuni nu poate fi hotărât decât în momentul execuției. Aceste ramificări ale execuției se pot simula destul de ușor prin schimbarea valorii referinței memorate în indicatorul de instrucțiuni. 🏠

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul II

Limbaje de programare

[2.1](#) *Comunicația om-măcină*

[2.2](#) *Tipuri de numere reprezentabile în calculator*

[2.3](#) *Valori de adevăr*

[2.4](#) *Șiruri de caractere*

[2.5](#) *Tipuri primitive de valori ale unui limbaj de programare*

[2.6](#) *Tablouri de elemente*

[2.7](#) *Expresii de calcul*

[2.8](#) *Variabile*

[2.9](#) *Instrucțiuni*

2.1 Comunicația om-măcină

Pentru a executa un program de calculator este necesar să putem comunica cu unitatea centrală pentru a-i furniza instrucțiunile necesare. Cel mai simplu mod de a le comunica este înscrierea codului instrucțiunilor direct în memoria calculatorului de unde pot fi citite de către unitatea centrală. Această cale este însă extrem de anevoioasă pentru programator pentru că el trebuie să învețe să se exprime coerent într-un limbaj ale cărui componente de bază sunt coduri numerice.

O alternativă mai bună este folosirea unui limbaj de comunicație format dintr-un număr foarte mic de cuvinte și caractere speciale împreună cu un set de convenții care să ajute la descrierea numerelor și a operațiilor care trebuiesc executate cu aceste numere. Limbajul trebuie să fie atât de simplu încât calculatorul să poată traduce singur, prin intermediul unui program numit *compiler*, frazele acestui limbaj în instrucțiuni ale unității centrale. Ori de câte ori vom imagina un nou limbaj de comunicație cu calculatorul va trebui să creăm un nou program compilator care să traducă acest limbaj în instrucțiuni ale unității centrale, numite uneori și *instrucțiuni măcină*.

În realitate folosirea termenului de limbaj de comunicație nu este extrem de fericită pentru că de obicei noi doar instruiem calculatorul ce are de făcut și cum trebuie să facă acel lucru, fără să-i dăm vreo șansă acestuia să comenteze sarcinile primite. În continuare vom numi aceste limbaje simplificate *limbaje de programare* pentru a le deosebi de limbajele pe care le folosim pentru a comunica cu alți oameni și pe care le vom numi *limbaje naturale*.

Limbajele de programare trebuie să ne ofere o cale de descriere a modulului în care dorim să reprezentăm informațiile cu care lucrează programul, o cale de a specifica operațiile care trebuiesc executate cu aceste informații și o cale de a controla ordinea în care sunt executate aceste operații.

În plus, limbajele de programare trebuie să respecte următorul principiu fundamental: rezultatul

execuției unei comenzi dintr-un limbaj de programare trebuie să fie complet determinat. Asta înseamnă că în limbajele de programare nu este permisă nici o formă de ambiguitate a exprimării.

Informațiile reprezentate în memoria calculatorului și prelucrate de către un program scris într-un limbaj de programare se numesc date. Tipurile de date care se pot descrie direct cu un anumit limbaj de programare se numesc tipurile de date elementare ale limbajului. Operațiile elementare posibil de exprimat într-un limbaj de programare se numesc instrucțiuni ale limbajului.

Limbajele de programare au evoluat foarte mult de la începuturile calculatoarelor. Tendința generală este aceea de apropiere treptată de limbajele naturale și de modul în care acestea descriu și utilizează informațiile. Abstracțiile exprimabile cu ajutorul limbajelor de programare sunt din ce în ce mai multe și cuprind o gamă din ce în ce mai largă a noțiunilor fundamentale cu care operează mintea umană.

Un alt aspect în care se reflectă evoluția limbajelor de programare este reprezentat de creșterea profunzimii și calității controlului pe care compilatoarele îl fac în momentul traducerii din limbaj de programare în limbaj mașină, astfel încât șansa de a greși la descrierea programului să fie cât mai mică.

Cu toate că limbajele de programare au făcut pași esențiali în ultimul timp, ele au rămas totuși foarte departe de limbajele naturale. Pentru a programa un calculator, trebuie să poți gândi și să poți descrie problemele într-unul dintre limbajele pe care acesta le înțelege. Din acest motiv, scrierea programelor continuă să rămână o activitate rezervată unui grup de inițiați, chiar dacă acest grup este în continuă creștere. 🏰

2.2 Tipuri de numere reprezentabile în calculator

Deși toate informațiile reprezentabile direct în memoria calculatorului sunt doar numere naturale, constructorii calculatoarelor de astăzi au diversificat tipurile acestora prin stabilirea unor convenții de reprezentare pentru numerele negative, reale și pentru caractere. Aceste convenții folosesc numerele naturale pentru reprezentarea celorlalte tipuri de numere. Uneori, reprezentările ocupă mai mult de un octet în memoria calculatorului, dimensiunile obișnuite fiind 1, 2, 4 sau 8 octeți. Sau, echivalent, 8, 16, 32 sau 64 de biți.

Numerele naturale sunt întotdeauna pozitive. Pentru reprezentarea unui *număr întreg cu semn* pot fi folosite două numere naturale. Primul dintre acestea, având doar două valori posibile, reprezintă semnul numărului și se poate reprezenta folosind o singură cifră binară. Dacă valoarea acestei cifre binare este 0, numărul final este pozitiv, iar dacă valoarea cifrei este 1, numărul final este negativ. Al doilea număr natural folosit în reprezentarea numerelor întregi cu semn conține valoarea absolută a numărului final. Această convenție, deși are dezavantajul că oferă două reprezentări pentru numărul zero, un zero pozitiv și altul negativ, este foarte aproape de reprezentarea numerelor cu semn folosită de calculatoarele moderne.

În realitate, convenția care se folosește pentru reprezentarea numerelor întregi cu semn este așa numita

reprezentare în complement față de doi. Aceasta reprezintă numerele negative prin complementarea valorii lor absolute bit cu bit și apoi adunarea valorii 1 la numărul rezultat. Complementarea valorii unui bit se face înlocuind valoarea 1 cu 0 și valoarea 0 cu 1. Dacă avem de exemplu numărul 1, reprezentat pe un octet ca un șir de opt cifre binare 00000001, complementarea bitcu bit a acestui număr este numărul reprezentat pe un octet prin 11111110.

Pentru a reprezenta valoarea -1 nu ne mai rămâne altceva de făcut decât să adunăm la numărul rezultat în urma complementării un 1 și reprezentarea finală a numărului întreg negativ -1 pe un octet este 11111111.

În această reprezentare, numărul 00000000 binar reprezintă numărul 0 iar numărul 10000000, care mai înainte reprezenta numărul 0 negativ, acum reprezintă numărul -128. Într-adevăr, numărul 128 se poate reprezenta în binar prin 10000000. Complementat, acest număr devine 01111111 și după adunarea cu 1, 10000000. Observați că numerele 128 și -128 au aceeași reprezentare. Convenția este aceea că se păstrează reprezentarea pentru -128 și se elimină cea pentru 128. Alegerea este datorată faptului că toate numerele pozitive au în primul bit valoarea 0 și toate cele negative valoarea -1. Prin transformarea lui 10000000 în -128 se păstrează această regulă.

Folosind reprezentarea în complement față de doi, numerele întregi reprezentabile pe un octet sunt în intervalul -128 până la 127, adică -2^7 până la $2^7 - 1$. În general, dacă avem o configurație de n cifre binare, folosind reprezentarea în complement față de doi putem reprezenta numerele întregi din intervalul închis -2^{n-1} până la $2^{n-1} - 1$. În practică, se folosesc numere întregi cu semn reprezentate pe 1 octet, 2 octeți, 4 octeți și 8 octeți, respectiv 8, 16, 32 și 64 de biți.

Dacă dorim să reprezentăm un număr real în memoria calculatorului, o putem face memorând câteva cifre semnificative ale acestuia plus o informație legată de ordinul său de mărime. În acest fel, deși pierdem precizia numărului, putem reprezenta valori foarte aproape de zero sau foarte departe de această valoare.

Soluția de reprezentare este aceea de a păstra două numere cu semn care reprezintă cifrele semnificative ale numărului real respectiv un *exponent* care dă ordinul de mărime. Cifrele reprezentative ale numărului se numesc împreună *mantisă*. Numărul reprezentat în final este $0.mantisaE^{exponent}$. E are valoarea 256 spre deosebire de exponentul 10 pe care îl folosim în uzual. Dacă valoarea exponentului este foarte mare și pozitivă, numărul real reprezentat este foarte departe de 0, înspre plus sau înspre minus. Dacă exponentul este foarte mare în valoare absolută și negativ, numărul real reprezentat este foarte aproape de zero.

În plus, pentru a ne asigura de biunivocitatea corespondenței, avem nevoie de o convenție care să stabilească faptul că virgula este plasată imediat în fața cifrelor semnificative și că în fața acesteia se găsește o singură cifră 0. Numerele care respectă această convenție se numesc *numere normalizate*.

Această mutare a virgulei imediat în fața cifrelor semnificative poate să presupună modificarea exponentului care păstrează ordinul de mărime al numărului. Numerele fracționare se numesc în

limbajul calculatoarelor *numere în virgulă mobilă* sau *numere flotante* tocmai din cauza acestei eventuale ajustări a poziției virgulei.

Cu această convenție nu se poate reprezenta orice număr real, dar se poate obține o acoperire destul de bună a unui interval al axei numerelor reale cu valori. Atunci când încercăm să reprezentăm în memoria calculatorului un număr real, căutăm de fapt cel mai apropiat număr real reprezentabil în calculator și aproximăm numărul inițial cu acesta din urmă. Ca rezultat, putem efectua calcule complexe cu o precizie rezonabilă.

Descrierea convenției exacte de reprezentare a numerelor reale în calculator depășește cadrul acestei prezentări. Unele dintre detalii pot fi diferite de cele prezentate aici, dar principiul este exact acesta. Convenția de reprezentare a numerelor reale este standardizată de IEEE în specificația 754.

Desigur, în unele probleme, lipsa de precizie poate să altereze rezultatul final, mai ales că, uneori, erorile se cumulează. Există de altfel o teorie complexă, analiza numerică, concepută pentru a studia căile prin care putem ține sub control aceste erori de precizie. Putem crește precizia de reprezentare a numerelor reale prin mărirea spațiului rezervat mantisei. În acest fel mărim numărul de cifre semnificative pe care îl păstrăm. În general, unitățile centrale actuale lucrează cu două precizii: *numerele flotante simple*, reprezentate pe 4 octeți și *numerele flotante duble*, reprezentate pe 8 octeți. 🏠

2.3 Valori de adevăr

Uneori, avem nevoie să memorăm în calculator *valori de adevăr*. Există doar două valori de adevăr posibile: adevărat și fals. Uneori, aceste valori de adevăr se mai numesc și *valori booleene*. Deși pentru memorarea acestor valori este suficientă o singură cifră binară, în practică aceste valori sunt reprezentate pe un întreg octet pentru că operațiile la nivel de bit sunt de obicei prea lente. Valorile booleene se pot combina între ele prin operații logice: “și”, “sau”, “negare”. 🏠

2.4 Șiruri de caractere

După cum spuneam mai înainte, deși orice informație reprezentată în calculator este în final un număr, mintea umană este obișnuită să lucreze cu cuvinte și imagini mai mult decât cu numere. De aceea, calculatorul trebuie să aibă posibilitatea să simuleze memorarea informațiilor de acest fel.


În ceea ce privește cuvintele, ele pot fi reprezentate în memoria calculatorului prin caracterele care le formează. Înșiruirea acestor caractere în memorie duce la noțiunea de *șir de caractere*. Pe lângă caracterele propriu-zise care construiesc cuvântul, un șir de caractere trebuie să poată memora și numărul total de caractere din șir, cu alte cuvinte *lungimea* sa.

În realitate, un șir de caractere nu conține doar un singur cuvânt ci este o înșiruire oarecare de caractere printre care pot exista și caractere spațiu. De exemplu, următoarele secvențe sunt șiruri de caractere:

"Acesta este un °ir de caractere", "Eugen", "ABCD 0123", "HGkduI;.!".

Fiecare limbaj de programare trebuie sã ofere o convenþie de reprezentare a °irurilor de caractere în calculator precum °i o convenþie de scriere a acestora în program. De obicei, cea de-a doua convenþie este aceea cã °irul de caractere trebuie închis între apostroafe sau ghilimele. În paragraful anterior de exemplu, am folosit ghilimele pentru delimitarea °irurilor de caractere.


Convenþia de reprezentare în memorie diferã de la un limbaj de programare la altul prin modul în care este memoratã lungimea °irului, precum °i prin convenþia de reprezentare în memorie a caracterelor: ASCII, Unicode sau alta.

Împreună cu fiecare tip de datã, limbajele de programare trebuie sã defineascã °i operaþiile ce se pot executa cu datele de tipul respectiv. Pentru °irurile de caractere, principala operaþie este *concatenarea*. Prin concatenarea a douã °iruri de caractere se obþine un °ir de caractere care conþine caracterele celor douã °iruri puse în prelungire. De exemplu, prin concatenarea °irurilor de caractere "unu" °i "doi", rezultã °irul de caractere: "unu, doi". 

2.5 Tipuri primitive de valori ale unui limbaj de programare

Vom numi *tipuri primitive de valori ale unui limbaj* acele tipuri de valori care se pot reprezenta direct într-un anumit limbaj de programare. Pentru ca informaþia reprezentabilã sã fie independentã de calculatorul pe care ruleazã programele, un limbaj de programare trebuie sã-°i defineascã propriile sale tipuri primitive, eventual diferite de cele ale unitãþii centrale, tipuri care sã generalizeze tipurile primitive ale tuturor unitãþilor centrale.

Pentru fiecare dintre aceste tipuri primitive de valori, limbajul trebuie sã defineascã dimensiunea locaþiei de memorie ocupate, convenþia de reprezentare °i mulþimea valorilor care pot fi reprezentate în aceastã locaþie. În plus, limbajul trebuie sã defineascã operaþiile care se pot executa cu aceste tipuri °i comportarea acestor operaþii pe seturi de valori diferite.

Tipurile primitive ale unui limbaj sunt de obicei: numere de diverse tipuri, caractere, °iruri de caractere, valori de adevãr °i valori de tip referinþã. Totu°i, acest set de tipuri primitive, denumirea exactã a tipurilor °i operaþiile care se pot executa cu ele variazã mult de la un limbaj de programare la altul. 

2.6 Tablouri de elemente

Tipurile primitive împreună cu referinþele, adicã tipurile de date elementare, indivizibile ale unui limbaj, sunt insuficiente pentru necesitãþile unei aplicaþii reale. De obicei este nevoie de organizãri de date mai complicate în care avem structuri de date create prin asocierea mai multor tipuri de date elementare. Aceste structuri de organizare a informaþiilor pot conþine date de acela°i tip sau date de tipuri diferite.

În cazul în care dorim să reprezentăm o structură conținând date de același tip, va trebui să folosim o structură clasică de reprezentare a datelor numită *tablou de elemente*. Practic, un tablou de elemente este o alăturare de locașii de memorie de același fel. Alăturarea este continuă în sensul că zona de memorie alocată tabloului nu are găuri. De exemplu, putem gândi întreaga memorie internă a calculatorului ca fiind un tablou de cifre binare sau ca un tablou de octeți.

Informațiile de același fel reprezentate într-un tablou se pot prelucra în mod unitar. Referirea la un element din tablou se face prin precizarea locașiei de început a tabloului în memorie și a numărului de ordine al elementului pe care dorim să-l referim. Numărul de ordine al unui element se numește *indexul* elementului. De aceea, tablourile se numesc uneori și structuri de date indexate.

Să presupunem că, la adresa 1234 în memorie, deci începând cu octetul numărul 1234, avem un tablou de 200 de elemente de tip întregi cu semn reprezentați pe 4 octeți. Pentru a accesa elementul numărul 123 din tablou, trebuie să îi aflăm adresa în memorie. Pentru aceasta, trebuie să calculăm deplasamentul acestui element față de începutul tabloului, cu alte cuvinte, la câți octeți distanță față de începutul tabloului se găsește elementul numărul 123.

Pentru calculul deplasamentului, este nevoie să știm cu ce index începe numerotarea elementelor din tablou. De obicei aceasta începe cu 0 sau cu 1, primul element fiind elementul 0 sau elementul 1. Să presupunem, în cazul nostru, că numerotarea ar începe cu 0. În acest caz, elementul cu indexul 123 este de fapt al 124-lea element din tablou, deci mai are încă 123 de elemente înaintea lui. În aceste condiții, pentru a afla adresa elementului dat, este suficient să adăugăm la adresa de început a tabloului, deplasamentul calculat ca numărul de elemente din față înmulțit cu dimensiunea unui element din tablou. Adresa finală este $1234 + 123 * 4 = 1726$.

Pentru a putea lucra cu elementele unui tablou este deci suficient să memorăm adresa de început a tabloului și indexul elementelor pe care dorim să le accesăm. Alternativa ar fi fost să memorăm adresa locașiei fiecărui element din tablou.

Unul dintre marile avantaje ale utilizării tablourilor este acela că elementele dintr-un tablou se pot prelucra în mod repetitiv, apelându-se aceeași operație pentru un subset al elementelor din tablou. Astfel, într-un program putem formula instrucțiuni de forma: pentru elementele tabloului T începând de la al treilea până la al N -lea, să se execute operația O . Numărul N poate fi calculat dinamic în timpul execuției programului, în funcție de necesități.

Elementele unui tablou pot fi de tip primitiv, referință sau pot fi tipuri compuse, inclusiv alte tablouri.



2.7 Expresii de calcul

Sarcina principală a calculatoarelor este aceea de a efectua calcule. Pentru a putea efectua aceste calcule, calculatorul trebuie să primească o descriere a operațiilor de calcul pe care le are de executat. Calculele

simple sunt descrise cel mai bine prin expresii de calcul.

Expresiile sunt formate dintr-o serie de valori care intră în calcul, numite *operanzi* și din simboluri care specifică operațiile care trebuiesc efectuate cu aceste valori, numite *operatori*. Operatorii reprezintă operații de adunare, înmulțire, împărțire, concatenare a șirurilor de caractere, etc.

Operanzii unor expresii pot fi valori elementare precum numerele, șirurile de caractere sau pot fi referiri către locații de memorie în care sunt memorate aceste valori. Tot operanzi pot fi și valorile unor funcții predefinite precum sinus, cosinus sau valoarea absolută a unui număr. Calculul complex al valorii acestor funcții pentru argumentele de intrare este astfel ascuns sub un nume ușor de recunoscut.

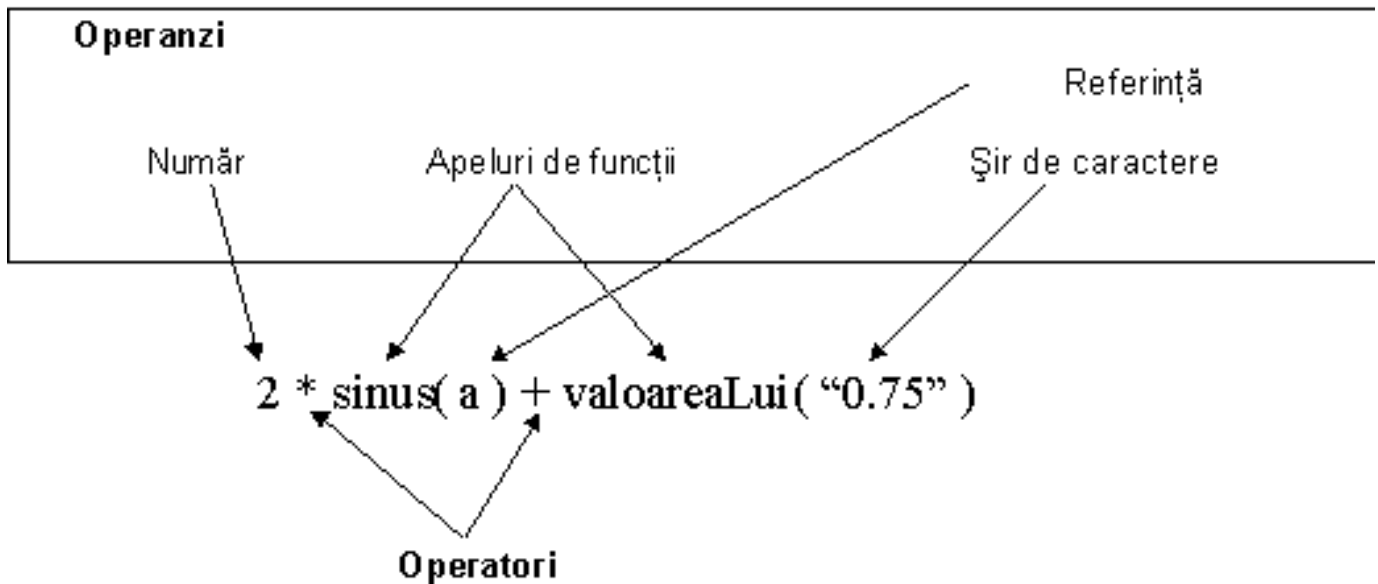


Figura 2.1 Un exemplu de expresie și componentele acesteia 🏠

2.8 Variabile

Uneori, atunci când calculele sunt complexe, avem nevoie să păstrăm rezultate parțiale ale acestor calcule în locații temporare de memorie. Alteori, chiar datele inițiale ale problemei trebuie memorate pentru o folosire ulterioară. Aceste locații de memorie folosite ca depozit de valori le vom numi *variabile*. Variabilele pot juca rol de operanzi în expresii.

Pentru a regăsi valoarea memorată într-o anumită variabilă, este suficient să memorăm poziția locației variabilei în memorie și tipul de dată memorată la această locație, numit și *tipul variabilei*. Cunoașterea tipului variabilei este esențială la memorarea și regăsirea datelor. La locația variabilei găsim întotdeauna o configurație de cifre binare. Interpretarea acestor cifre binare se poate face numai cunoscând convenția de reprezentare care s-a folosit la memorarea acelor cifre binare. Mai mult, tipul de variabilă ne și spune câți octeți de memorie ocupă locația respectivă.

Pentru a putea referi variabilele în interiorul unui program, trebuie să atribuim câte un nume pentru

fiecare dintre acestea și să rezervăm locațiile de memorie destinate lor. Această rezervare a locațiilor se poate face fie la pornirea programului fie pe parcurs.

Putem împărți variabilele în funcție de perioada lor de existență în *variabile statice*, care există pe tot parcursul programului și *variabile locale* care se creează doar în secțiunea de program în care este nevoie de ele pentru a fi distruse imediat ce se părăsește secțiunea respectivă. Variabilele statice păstrează de obicei informații esențiale pentru execuția programului precum numele celui care a pornit programul, data de pornire, ora de pornire sau numărul p . Variabilele locale păstrează valori care au sens doar în contextul unei anumite secțiuni din program. De exemplu, variabilele care sunt utilizate la calculul sinusului dintr-un număr, sunt inutile și trebuie eliberate imediat ce calculul a fost terminat. În continuare, doar valoarea finală a sinusului este importantă.

În plus, există variabile care se creează doar la cererea explicită a programului și nu sunt eliberate decât atunci când programul nu mai are nevoie de ele. Aceste variabile se numesc *variabile dinamice*. De exemplu, se creează o variabilă dinamică atunci când utilizatorul programului introduce un nou nume într-o listă de persoane. Crearea și ștergerea acestui nume nu are legătură cu faza în care se află rularea programului ci are legătură cu dorința celui care utilizează programul de a mai adăuga sau șterge un nume în lista persoanelor cu care lucrează, pentru a le trimite, de exemplu, felicitări de anul nou. 🏠

2.9 Instrucțiuni

Operațiile care trebuie executate de un anumit program sunt exprimate în limbajele de programare cu ajutorul unor instrucțiuni. Spre deosebire de limbajele naturale, există doar câteva tipuri standard de instrucțiuni care pot fi combinate între ele pentru a obține funcționalitatea programelor. Sintaxa de scriere a acestor tipuri de instrucțiuni este foarte rigidă. Motivul acestei rigidități este faptul că un compilator recunoaște un anumit tip de instrucțiune după sintaxa ei și nu după sensul pe care îl are ca în cazul limbajelor naturale.

Tipurile elementare de instrucțiuni nu s-au schimbat de-a lungul timpului pentru că ele sunt conținute în însuși modelul de funcționare al calculatoarelor Von Neumann.

În primul rând, avem *instrucțiuni de atribuire*. Aceste instrucțiuni presupun existența unei locații de memorie căreia dorim să-i atribuim o anumită valoare. Atribuirea poate să pară o operație foarte simplă dar realitatea este cu totul alta. În primul rând, valoarea respectivă poate fi rezultatul evaluării (calculului) unei expresii complicate care trebuie executată înainte de a fi memorată în locația de memorie dorită.

Apoi, însăși poziția locației în memorie ar putea fi rezultatul unui calcul în urma căruia să rezulte o anumită adresă. De exemplu, am putea dori să atribuim o valoare unui element din interiorul unui tablou. Într-o astfel de situație, locația de memorie este calculată prin adunarea unui deplasament la adresa de început a tabloului.

În fine, locația de memorie și valoarea pe care dorim să i-o atribuim ar putea avea tipuri diferite. În acest caz, operația de atribuire presupune și transformarea, dacă este posibilă, a valorii într-o nouă valoare de același tip cu locația de memorie. În plus, această transformare trebuie efectuată în așa fel încât să nu se piardă semnificația valorii originale. De exemplu, dacă valoarea originală era -1 întreg iar locația de memorie este de tip flotant, valoarea obținută după transformare trebuie să fie -1.0.

Transformarea valorilor de la un tip la altul se numește în termeni informatici *conversie*. Conversiile pot apărea și în alte situații decât instrucțiunile de atribuire, de exemplu la transmiterea parametrilor unei funcții, în calculul unei expresii.

Forma generală a unei instrucțiuni de atribuire este:

$$\text{Locație} = \text{Valoare}$$

Locația poate fi o adresă cu tip, un nume de variabilă sau o expresie în urma căreia să rezulte o adresă cu tip. Printr-o adresă cu tip înțelegem o adresă de memorie pentru care a fost specificat tipul valorilor care pot fi memorate în interior. O adresă simplă, fără tip, nu poate intra într-o atribuire pentru că nu putem ști care este convenția de reprezentare care trebuie folosită la memorarea valorii.

Al doilea tip de instrucțiune elementară este *instrucțiunea condițională*. Această instrucțiune permite o primă formă de ramificare a ordinii în care se execută instrucțiunile. Ramificarea depinde de o condiție care poate fi testată pe setul de date cu care lucrează aplicația. În funcție de valoarea de adevăr care rezultă în urma evaluării condiției se poate executa o instrucțiune sau alta. Modelul de exprimare a unei expresii condiționale este următorul:

Dacă condiția este adevărată

execută instrucțiunea 1

altfel

execută instrucțiunea 2.

Cele două ramuri de execuție sunt disjuncte în sensul că dacă este executată instrucțiunea de pe o ramură atunci cu siguranță instrucțiunea de pe cealaltă ramură nu va fi executată. După execuția uneia sau a celeilalte instrucțiuni ramurile se reunifică și execuția își continuă drumul cu instrucțiunea care urmează după instrucțiunea condițională.

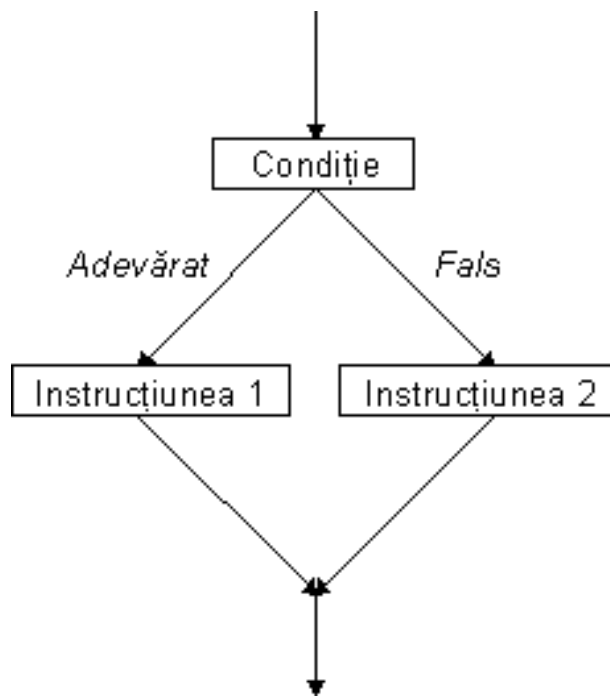


Figura 2.2 Schema de funcționare a unei instrucțiuni condiționale

Condiția care hotărăște care din cele două instrucțiuni va fi executată este de obicei un test de egalitate între două valori sau un test de ordonare în care o valoare este testată dacă este mai mică sau nu decât o altă valoare. Condițiile pot fi compuse prin conectori logici de tipul “și”, “sau” sau “non”, rezultând în final condiții de forma: dacă variabila numită *Temperatură* este mai mică decât 0 și mai mare decât -10, atunci...

Instrucțiunile condiționale au și o variantă în care, în funcție de o valoare întreagă alege o instrucțiune dintr-un set de instrucțiuni și apoi o execută. Această formă este în realitate derivată din instrucțiunea condițională clasică, ea putând fi exprimată prin:

Dacă Variabila este egală cu Valoarea 1 atunci

execută instrucțiunea 1

altfel, dacă Variabila este egală cu Valoarea 2 atunci

execută instrucțiunea 2

altfel, dacă Variabila ...

...

altfel

execută instrucțiunea implicită.

Un al treilea tip de instrucțiuni elementare sunt *instrucțiunile de ciclare* sau *repetitive* sau *buclele*. Acestea specifică faptul că o instrucțiune trebuie executată în mod repetat. Controlul ciclurilor de instrucțiuni se poate face pe diverse criterii. De exemplu se poate executa o instrucțiune de un număr fix de ori sau se poate executa instrucțiunea până când o condiție devine adevărată sau falsă.

Condiția de terminare a buclei poate fi testată la începutul buclei sau la sfârșitul acesteia. Dacă condiția este testată de fiecare dată înainte de execuția instrucțiunii, funcționarea ciclului se poate descrie prin:

Atâta timp cât Condiția este adevărată

execută Instrucțiunea

Acest tip de instrucțiuni de ciclare poartă numele de *cicluri while*, cuvântul *while* însemnând în limba engleză "atâta timp cât". În cazul ciclurilor *while*, există posibilitatea ca instrucțiunea din interiorul ciclului să nu se execute niciodată, dacă condiția este de la început falsă.

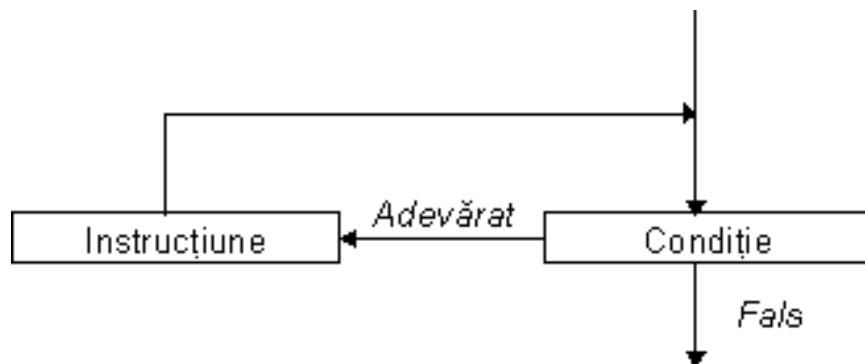


Figura 2.3 Schema de funcționare a unui ciclu while

Dacă condiția este testată de fiecare dată după execuția instrucțiunii, funcționarea ciclului se poate descrie prin:

Execută Instrucțiunea

atâta timp cât Condiția este adevărată.

Acest tip de instrucțiuni de ciclare poartă numele de *cicluri do-while*, cuvântul *do* însemnând în limba engleză "execută". În cazul ciclurilor *do-while* instrucțiunea din interiorul ciclului este garantat că se execută cel puțin o dată, chiar dacă valoarea condiției de terminare a buclei este de la început fals.

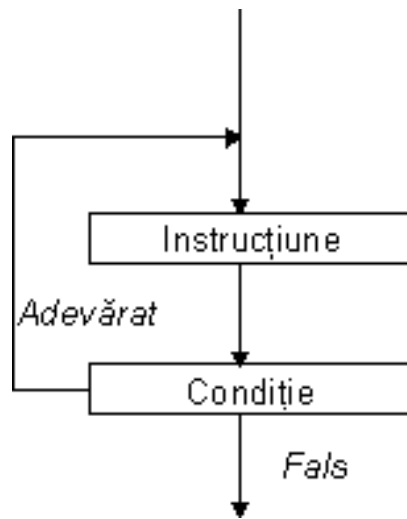


Figura 2.4 Schema de funcționare a unui ciclu do-while

După cum ați observat, probabil, dacă instrucțiunea din interiorul buclei nu afectează în nici un fel valoarea de adevăr a condiției de terminare a buclei, există șansa ca bucla să nu se termine niciodată.

Al patrulea tip de instrucțiuni sunt *apelurile de proceduri*. O *procedură* este o secvență de instrucțiuni de sine stătătoare și parametrizată. Un apel de procedură este o instrucțiune prin care forțăm programul să execute pe loc instrucțiunile unei anumite proceduri. După execuția procedurii, se continuă cu execuția instrucțiunii de după apelul de procedură.

Procedurile sunt utile atunci când, în zone diferite ale programului, dorim să executăm aceeași secvență de instrucțiuni. În aceste situații, putem să scriem secvența de instrucțiuni o singură dată și s-o apelăm apoi oriunde din interiorul programului.

De exemplu, ori de câte ori vom dori să ștergem toate semnele de pe ecranul calculatorului, putem apela la o procedură, pentru a nu fi nevoiți să scriem de fiecare dată secvența de instrucțiuni care duce la ștergerea ecranului. Procedura în sine, o vom scrie o singură dată și îi vom da un nume prin care o vom putea apela ulterior ori de câte ori avem nevoie.

Uneori secvența de instrucțiuni dintr-o procedură depinde de niște *parametri* adică de un set de valori care sunt precizate doar în momentul apelului procedurii. În aceste cazuri, procedura se poate comporta în mod diferit în funcție de valorile de apel ale acestor parametri. În funcție de tipul declarat al unor parametri și de tipul real al valorilor care sunt trimise ca parametri într-un apel de procedură, se poate întâmpla ca înainte de apelul propriu-zis să fie necesară o conversie de tip.

De exemplu, în cazul procedurii care calculează valoarea sinusului unui număr, parametrul de intrare este însuși numărul pentru care trebuie calculat sinusul. Rezultatul acestui calcul va fi diferit, în funcție de valoarea la apel a parametrului. În mod normal, parametrul va fi un număr flotant iar dacă la apel vom transmite ca parametru o valoare întreagă, ea va fi convertită spre o valoare flotantă.

În urma apelului unei proceduri poate rezulta o valoare pe care o vom numi *valoare de retur a procedurii*. De exemplu, în urma apelului procedurii de calcul a sinusului unui număr, rezultă o valoare flotantă care este valoarea calculată a sinusului. Acest tip de proceduri, care întorc valori de retur se mai numesc și *funcții*. Funcțiile definesc întotdeauna un tip al valorii de retur pe care o întorc. În acest fel, apelurile de funcții pot fi implicate și în formarea unor expresii de calcul sub formă de operanzi, expresia cunoscând tipul de valoare care trebuie așteptată ca valoare a apelului unei anumite funcții.

Analogia cu funcțiile pe care le-ați studiat la matematică este evidentă dar nu totală. În cazul funcțiilor scrise într-un limbaj de programare, două apeluri consecutive ale aceleiași funcții cu aceleași valori ca parametri se pot comporta diferit datorită dependenței de condiții exterioare sistemului format din funcție și parametrii de apel. În plus, se poate întâmpla ca modul de comportare al funcției și valoarea de retur să nu fie definite pentru anumite valori de apel ale parametrilor. Deși scrierea acestui tip de funcții este nerecomandabilă, în practică întâlnim astfel de funcții la tot pasul. 🏠

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul III

Reprezentarea informațiilor cu obiecte

[3.1](#) *Obiecte*

[3.2](#) *Încapsularea informațiilor în interiorul obiectelor*

[3.3](#) *Clase de obiecte*

[3.4](#) *Derivarea claselor de obiecte*

[3.5](#) *Interfețe spre obiecte*

3.1 Obiecte

Informațiile pe care le reprezentăm în memoria calculatorului sunt rareori atât de simple precum culorile sau literele. În general, dorim să reprezentăm informații complexe, care să descrie obiectele fizice care ne înconjoară sau noțiunile cu care operăm zilnic, în interiorul cărora culoarea sau o secvență de litere reprezintă doar o mică parte. Aceste obiecte fizice sau noțiuni din lumea reală trebuie reprezentate în memoria calculatorului în așa fel încât informațiile specifice lor să fie păstrate la un loc și să se poată prelucra ca un tot unitar. Să nu uităm însă că, la nivelul cel mai de jos, informația atașată acestor obiecte continuă să fie tratată de către compilator ca un șir de numere naturale, singurele informații reprezentabile direct în memoria calculatoarelor actuale.

Pentru a reprezenta în memoria internă obiecte fizice sau noțiuni, este nevoie să izolăm întregul set de proprietăți specifice acestora și să îl reprezentăm prin numere. Aceste numere vor ocupa în memorie o zonă compactă pe care, printr-un abuz de limbaj, o vom numi, într-o primă aproximare, *obiect*. Va trebui însă să aveți întotdeauna o imagine clară a deosebirii fundamentale dintre un obiect fizic sau o noțiune și reprezentarea acestora în memoria calculatorului.

De exemplu, în memoria calculatorului este foarte simplu să creăm un nou obiect, identic cu altul deja existent, prin simpla duplicare a zonei de memorie folosite de obiectul pe care dorim să-l dedublăm. În realitate însă, este mult mai greu să obținem o copie identică a unui obiect fizic, fie el o simplă foaie de hârtie sau o bancnotă de 10000 de lei.

Să revenim însă la numerele naturale. Din moment ce ele sunt singurele entități reprezentabile în memoria calculatorului, este firesc ca acesta să fie echipat cu un set bogat de operații predefinite de prelucrare a numerelor. Din păcate, atunci când facem corespondența dintre numere și litere de exemplu, nu ne putem aștepta la un set la fel de bogat de operații predefinite care să lucreze cu litere. Dar, ținând cont de faptul că în cele din urmă lucrăm tot cu numere, putem construi propriile noastre operații specifice literelor, combinând în mod corespunzător operațiile numerice predefinite.

De exemplu, pentru a obține dintr-o literă majusculă, să spunem ‘A’, corespondenta ei minusculă ‘a’, este suficient să adunăm un deplasament numeric corespunzător, presupunând că literele mari și cele

mici sunt numerotate în ordine alfabetică și imediat una după cealaltă în convenția de reprezentare folosită. În setul ASCII deplasamentul este 32, reprezentarea lui 'A' fiind 65 iar reprezentarea lui 'a' fiind 97. Acest deplasament se păstrează pentru toate literele din alfabetul englez și român, la cel din urmă existând totuși o excepție în cazul literei 'a'.

Putem să extindem cerințele noastre mai departe, spunând că, atunci când analizăm un obiect fizic sau o noțiune pentru a le reprezenta în calculator, trebuie să analizăm nu numai proprietățile acestora dar și modul în care acestea pot fi utilizate și care sunt operațiile care pot fi executate asupra lor sau cu ajutorul lor. Acest set de operații va trebui ulterior să-l redefinim în contextul mulțimii de numere care formează proprietățile obiectului din memoria calculatorului, și să îl descompunem în operații numerice preexistente. În plus, trebuie să analizăm modul în care reacționează obiectul atunci când este supus efectului unor acțiuni exterioare. Uneori, setul de operații specifice unui obiect împreună cu modul în care acesta reacționează la stimuli exteriori se numește *comportamentul obiectului*.

De exemplu, dacă dorim să construim un obiect care reprezintă o minge de formă sferică în spațiu, este necesar să definim trei numere care să reprezinte coordonatele x , y și z relativ la un sistem de axe dat, precum și o valoare pentru raza sferei. Aceste valori numerice vor face parte din setul de proprietăți ale obiectului minge. Dacă mai târziu vom dori să construim o operație care să reprezinte mutarea în spațiu a obiectului minge, este suficient să ne folosim de operațiile cu numere pentru a modifica valorile coordonatelor x , y și z .

Desigur, obiectul minge este insuficient descris prin aceste coordonate și, pentru a simula în calculator obiectul real este nevoie de multe proprietăți suplimentare precum și de multe operații în plus. Dar, dacă problema pe care o avem de rezolvat nu necesită aceste proprietăți și operații, este preferabil să nu le definim în obiectul folosit pentru reprezentare. Rezultatul direct al acestui mod de abordare este acela că vom putea defini același obiect real în mai multe feluri pentru a-l reprezenta în memoria internă. Modul de definire depinde de problema de rezolvat și de programatorul care a gândit reprezentarea. De altfel, aceste diferențe de percepție ale unui obiect real există și între diverși observatori umani.

Din punctul de vedere al programării, un obiect este o reprezentare în memoria calculatorului a proprietăților și comportamentului unei noțiuni sau ale unui obiect real.

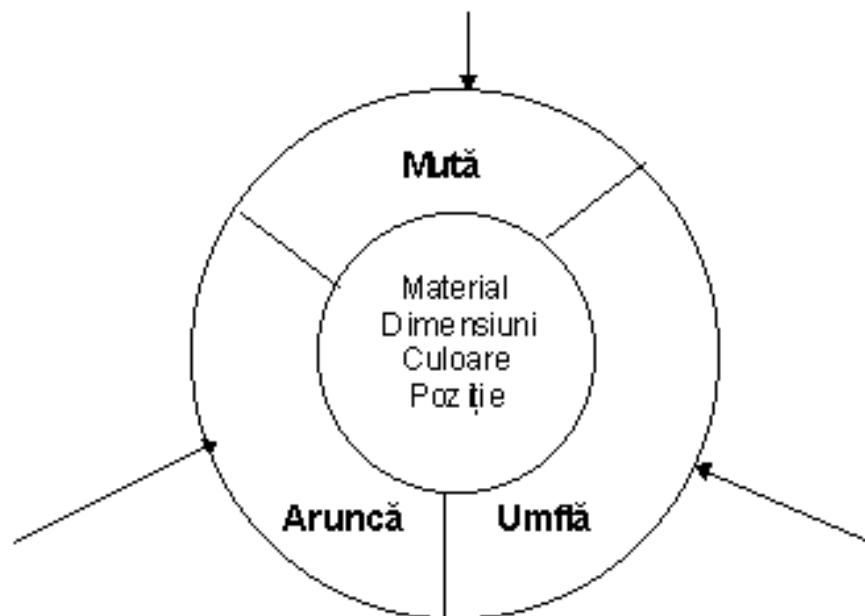


Figura 3.1 Modelul de reprezentare al unui obiect în memorie. Stratul exterior reprezintă doar operațiile care oferă calea de a interacționa cu proprietățile obiectului și nu are corespondent direct în zona de memorie ocupată de obiect. 🏠

3.2 Încapsularea informațiilor în interiorul obiectelor

Există situații în care accesul din exterior la proprietățile unui obiect poate să pună probleme acestuia. Ce s-ar întâmpla de exemplu dacă s-ar putea accesa direct valorile care definesc funcționalitatea corpului uman? Desigur, există cazuri în care acest lucru ar fi îmbucurător. N-ar mai fi nevoie să acționăm indirect asupra concentrațiilor de enzime în corp ci am putea să modificăm aceste valori în mod direct. Dar, în același timp, am putea provoca mari necazuri în cazul în care am modifica aceste valori în afara pragului suportabil de către organism. Din aceste motive, este preferabil să lăsăm modificarea acestor parametri în sarcina exclusivă a unor operații definite de către obiect, operații care vor verifica noile valori înainte de a le schimba în interiorul obiectului. În lipsa acestui filtru, putem să stricăm coerența valorilor memorate în interiorul unui obiect, făcându-l inutilizabil.

Din acest punct de vedere, putem privi obiectul ca pe un set de valori care formează miezul obiectului și un set de operații care îmbracă aceste valori, protejându-le. Vom spune că proprietățile obiectului sunt *încapsulate* în interiorul acestora. Mai mult, obiectul încapsulează și modul de funcționare a operațiilor lui specifice, din exterior neputându-se observa decât modul de apelare a acestor operații și rezultatele apelurilor. Cu alte cuvinte, *procesul de încapsulare* este procesul de ascundere a detaliilor neimportante sau sensibile de construcție a obiectului.

Dar nu numai proprietățile unui obiect trebuie protejate ci și operațiile definite de către acesta. Unele dintre operațiile definite pentru un obiect, cum ar fi de exemplu procesul respirator al omului, sunt periculos de lăsat la dispoziția oricui. Este preferabil să putem controla foarte exact cine ce operații poate apela pentru un anumit obiect. În acest mod, din punctul de vedere al unui observator străin, omul este perceput ca un obiect mult mai simplu decât este în realitate, pentru că acel observator nu poate

vedea decât acele valori și nu poate apela decât acele operații care sunt făcute publice.

Desigur, în practică este nevoie de o oarecare rafinare a gradului de protejare a fiecărei operații sau proprietăți în așa fel încât însuși accesul observatorilor exteriori să poată fi nuanțat în funcție de gradul de similitudine și apropiere al observatorului față de obiectul accesat. Rafinarea trebuie să meargă până la a putea specifica pentru fiecare proprietate și operație în parte care sunt observatorii care au acces și care nu.

Această protejare și încapsulare a proprietăților și operațiilor ce se pot executa cu ajutorul unui obiect are și o altă consecință și anume aceea că utilizatorul obiectului respectiv este independent de detaliile constructive ale obiectului respectiv. Structura internă a obiectului poate fi astfel schimbată și perfecționată în timp fără ca funcționalitatea de bază să fie afectată. 🏠

3.3 Clase de obiecte

În lumea reală se pot identifica ușor familii de obiecte. Este greu să descriem într-un limbaj de programare fiecare minge din lume dar, pentru a putea folosi orice minge din lume, este suficient să descriem o singură dată care sunt proprietățile unei mingi în general, precum și operațiile care pot fi executate cu aceasta. Aceasta nu înseamnă că toate obiectele minge din lume sunt identice. Diferența dintre ele se află reprezentată în primul rând în valorile proprietăților lor care sunt mărimi numerice variabile, adică diferă de la un obiect de același fel la altul. De exemplu, în fiecare obiect minge vom avea un număr natural care reprezintă culoarea mingii. Acest număr poate să difere de la o minge la alta exact așa cum, în realitate, culoarea diferă de la o minge la alta. La fel coordonatele poziției mingii la un moment dat sau raza mingii precum și materialul din care este confecționată au valori care variază de la o minge la alta.

Cu alte cuvinte, fiecare minge din lume are același set de proprietăți, dar valorile acestora pot să difere de la o minge la alta. Modelul de reprezentare în memorie a unui obiect este întotdeauna același, dar valorile memorate în locațiile corespunzătoare proprietăților sunt în general diferite.

În ceea ce privește operațiile, acestea sunt întotdeauna aceleași dar rezultatul aplicării lor poate să difere în funcție de valorile proprietăților obiectului asupra căruia au fost aplicate. De exemplu, atunci când aruncăm o minge spre pământ ea va ricoșa din acesta ridicându-se din nou în aer. Înălțimea la care se va ridica însă, este dependentă de dimensiunile și materialul din care a fost confecționată mingea. Cu alte cuvinte, noua poziție în spațiu se va calcula printr-o operație care va ține cont de valorile memorate în interiorul obiectului. Se poate întâmpla chiar ca operația să hotărască faptul că mingea va străpunge podeaua în loc să fie respinsă de către aceasta.

Să mai observăm că operațiile nu depind numai de proprietățile obiectului ci și de unele valori exterioare acestuia. Atunci când aruncăm o minge spre pământ, înălțimea la care va ricoșa aceasta depinde și de viteza cu care a fost aruncată mingea. Această viteză este un parametru al operației de aruncare. Nu are nici un rost să transmitem ca parametrii ai unei operații valorile proprietăților unui

obiect pentru că acestea sunt întotdeauna disponibile operației. Nici o operație nu se poate aplica asupra unui obiect fără să știm exact care este obiectul respectiv și ce proprietăți are acesta. Este absurd să ne gândim la ce înălțime se va ridica o minge în general, fără să facem presupuneri asupra valorilor proprietăților acesteia. Să mai observăm însă că, dacă toate mingile ar avea aceleași valori pentru proprietățile implicate în operația descrisă mai sus, am putea să calculăm înălțimea de ricoșeu în general, fără să fim dependenți de o anumită minge.

În concluzie, putem spune că obiectele cu care lucrăm fac parte întotdeauna dintr-o familie mai mare de obiecte cu proprietăți și comportament similar. Aceste familii de obiecte le vom numi în continuare *clase de obiecte* sau *concepte* în timp ce obiectele aparținând unei anumite clase le vom numi *instanțe* ale clasei de obiecte respective. Putem vorbi despre clasa de obiecte minge și despre instanțele acesteia, mulțimea tuturor obiectelor minge care există în lume. Fiecare instanță a clasei minge are un loc bine precizat în spațiu și în timp, un material și o culoare. Aceste proprietăți diferă de la o instanță la alta, dar fiecare instanță a aceleiași clase va avea întotdeauna aceleași proprietăți și aceleași operații vor putea fi aplicate asupra ei. În continuare vom numi *variabile* aceste proprietăți ale unei clase de obiecte și vom numi *metode* operațiile definite pentru o anumită clasă de obiecte.

Pentru a clarifica, să mai reluăm încă o dată: O clasă de obiecte este o descriere a proprietăților și operațiilor specifice unui nou tip de obiecte reprezentabile în memorie. O instanță a unei clase de obiecte este un obiect de memorie care respectă descrierea clasei. O variabilă a unei clase de obiecte este o proprietate a clasei respective care poate lua valori diferite în instanțe diferite ale clasei. O metodă a unei clase este descrierea unei operații specifice clasei respective.

Să mai precizăm faptul că, spre deosebire de variabilele unei clase, metodele acesteia sunt memorate o singură dată pentru toate obiectele. Comportarea diferită a acestora este dată de faptul că ele depind de valorile variabilelor.

O categorie aparte a claselor de obiecte este categoria acelor clase care reprezintă concepte care nu se pot instanția în mod direct, adică nu putem construi instanțe ale clasei respective, de obicei pentru că nu avem destule informații pentru a le putea construi. De exemplu, conceptul de om nu se poate instanția în mod direct pentru că nu putem construi un om despre care nu știm exact dacă este bărbat sau femeie. Putem în schimb instanția conceptul de bărbat și conceptul de femeie care sunt niște subconcepte ale conceptului om.

Clasele abstracte, neinstanțiabile, servesc în general pentru definirea unor proprietăți sau operații comune ale mai multor clase și pentru a putea generaliza operațiile referitoare la acestea. Putem, de exemplu să definim în cadrul clasei de obiecte om modul în care acesta se alimentează ca fiind independent de apartenența la conceptul de bărbat sau femeie. Această definiție va fi valabilă la amândouă subconceptele definite mai sus. În schimb, nu putem decât cel mult să precizăm faptul că un om trebuie să aibă un comportament social. Descrierea exactă a acestui comportament trebuie făcută în cadrul conceptului de bărbat și a celui de femeie. Oricum, este interesant faptul că, indiferent care ar fi clasa acestuia, putem să ne bazăm pe faptul că acesta va avea definit un comportament social, specific clasei lui.

Cele două metode despre care am vorbit mai sus, definite la nivelul unui superconcept, sunt profund diferite din punctul de vedere al subconceptelor acestuia. În timp ce metoda de alimentație este definită exact și amândouă subconceptele pot să o folosească fără probleme, metoda de comportament social este doar o metodă abstractă, care trebuie să existe, dar despre care nu se știe exact cum trebuie definită.

Fiecare dintre subconcepte trebuie să-și definească propriul său comportament social pentru a putea deveni instanțibil. Dacă o clasă de obiecte are cel puțin o metodă abstractă, ea devine în întregime o *clasă abstractă* și nu poate fi instanțiată, adică nu putem crea instanțe ale unei clase de obiecte abstracte.

Altfel spus, o *clasă abstractă de obiecte* este o clasă pentru care nu s-au precizat suficient de clar toate metodele astfel încât să poată fi folosită în mod direct. 🏠

3.4 Derivarea claselor de obiecte

O altă proprietate interesantă a claselor de obiecte este aceea de ierarhizare. Practic, ori de câte ori definim o nouă clasă de obiecte care să reprezinte un anumit concept, specificăm clasa de obiecte care reprezintă conceptul original din care provine noul concept împreună cu diferențele pe care le aduce noul concept derivat față de cel original. Această operație de definire a unei noi clase de obiecte pe baza uneia deja existente o vom numi *derivare*. Conceptul mai general se va numi *superconcept* iar conceptul derivat din acesta se va numi *subconcept*. În același mod, clasa originală se va numi *superclasă* a noii clase în timp ce noua clasă de obiecte se va numi *subclasă* a clasei derivate.

Uneori, în loc de derivare se folosește termenul de extindere. Termenul vine de la faptul că o subclasă își extinde superclasa cu noi variabile și metode.

În spiritul acestei ierarhizări, putem presupune că toate clasele de obiecte sunt derivate dintr-o clasă inițială, să-i spunem clasa de obiecte generice, în care putem defini proprietățile și operațiile comune tuturor obiectelor precum ar fi testul de egalitate dintre două instanțe, duplicarea instanțelor sau aflarea clasei de care aparține o anumită instanță.

Ierarhizarea se poate extinde pe mai multe nivele, sub formă arborescentă, în fiecare punct nodal al structurii arborescente rezultate aflându-se clase de obiecte. Desigur, clasele de obiecte de pe orice nivel pot avea instanțe proprii, cu condiția să nu fie clase abstracte, imposibil de instanțiat.

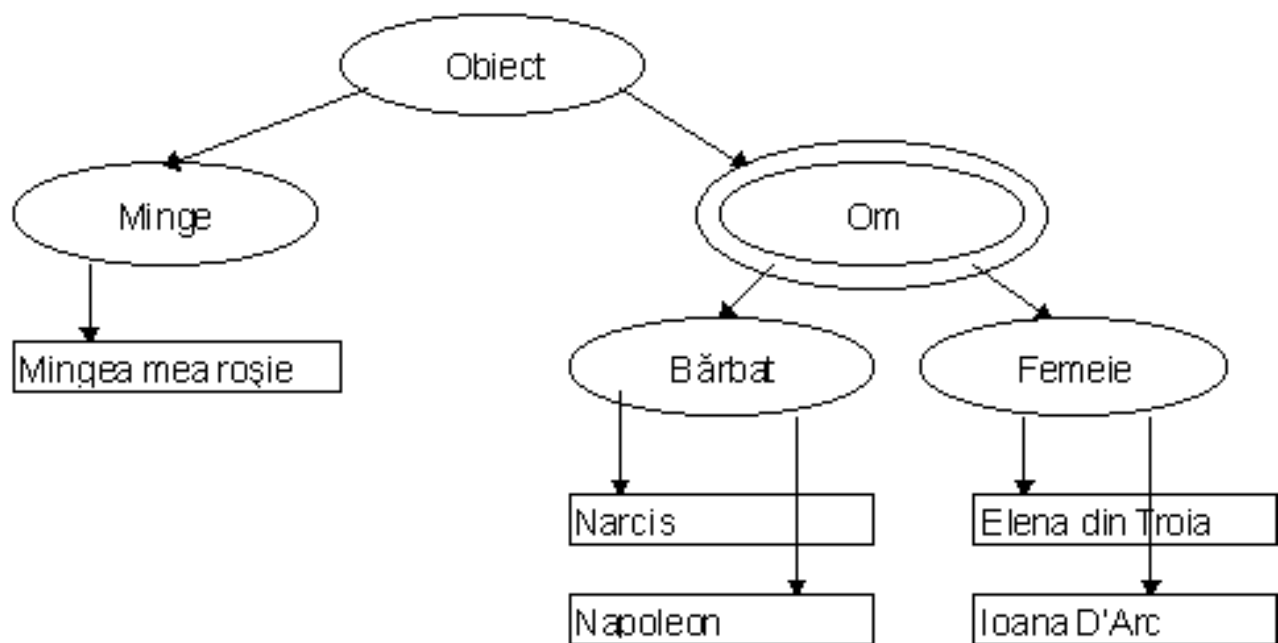


Figura 3.2 O ierarhie de clase de obiecte în care clasele sunt reprezentate în câmpuri eliptice iar instanțele acestora în câmpuri dreptunghiulare. Clasele abstracte de obiecte au elipsa dublată.

Desigur, este foarte dificil să construim o ierarhie de clase de obiecte completă, care să conțină clase de obiecte corespunzătoare fiecărui concept cunoscut. Din fericire, pentru o problemă dată, conceptele implicate în rezolvarea ei sunt relativ puține și pot fi uor izolate, simplificate și definite. Restrângerea la minimum a arborelui de concepte necesar rezolvării unei anumite probleme fără a se afecta generalitatea soluției este un talent pe care fiecare programator trebuie să și-l descopere și să și-l cultive cu atenție. De alegerea acestor concepte depinde eficiența și flexibilitatea aplicației.

O clasă de obiecte derivată dintr-o altă clasă păstrează toate proprietățile și operațiile acesteia din urmă aducând în plus proprietăți și operații noi. De exemplu, dacă la nivelul clasei de obiecte om am definit forma bipedă a acestuia și capacitatea de a vorbi și de a înpelege, toate acestea vor fi moținute și de către clasele derivate din clasa om, și anume clasa bărbatilor și cea a femeilor. Fiecare dintre aceste clase de obiecte derivate își vor defini propriile lor proprietăți și operații pentru a descrie diferența dintre ele și clasa originală.

Unele dintre proprietățile și operațiile definite în superclasă pot fi redefinite în subclasele de obiecte derivate. Vechile proprietăți și operații sunt disponibile în continuare, doar că pentru a le putea accesa va trebui să fie specificată explicit superclasa care deține copia redefinită. Operația de redefinire a unor operații sau variabile din interiorul unei clase în timpul procesului de derivare o vom numi *rescriere*.

Această redefinire ne dă de fapt o mare flexibilitate în construcția ierarhiei unei probleme date pentru că nici o proprietate sau operație definită într-un punct al ierarhiei nu este impusă definitiv pentru conceptele derivate din acest punct direct sau nu.

Revenind pentru un moment la protejarea informațiilor interne ale unui obiect să precizăm faptul că gradul de similitudine de care vorbeam mai sus este mărit în cazul în care vorbim de două clase derivate

una din cealaltă. Cu alte cuvinte, o subclasă a unei clase are acces de obicei la mult mai multe informații memorate în superclasa sa decât o altă clasă de obiecte oarecare. Acest lucru este firesc ținând cont de faptul că, uneori, o subclasă este nevoită să redefinească o parte din funcționalitatea superclasei sale. 🏠

3.5 Interfețe spre obiecte

Un obiect este o entitate complexă pe care o putem privi din diverse puncte de vedere. Omul de exemplu poate fi privit ca un mamifer care naște pui vii sau poate fi privit ca o ființă gânditoare care învăță să programeze calculatoare sau poate fi privit ca un simplu obiect spațio-temporal care are propria lui formă și poziție în funcție de timp.


Această observație ne spune că trebuie să dăm definiții despre ce înseamnă cu adevărat faptul că un obiect poate fi privit ca un mamifer sau ca o ființă gânditoare sau ca un obiect spațio-temporal. Aceste definiții, pe care le vom numi în continuare *interfețe*, sunt aplicabile nu numai clasei de obiecte om dar și la alte clase de obiecte derivate sau nu din acesta, superclase sau nu ale acesteia. Putem să găsim o mulțime de clase de obiecte ale căror instanțe pot fi privite ca obiecte spațio-temporale dar care să nu aibă mare lucru în comun cu omul. Practic, atunci când construim o interfață, definim un set minim de operații care trebuie să aparțină obiectelor care respectă această interfață. Orice clasă de obiecte care declară că respectă această interfață va trebui să definească toate operațiile.

Operațiile însă, sunt definite pe căi specifice fiecărei clase de obiecte în parte. De exemplu, orice obiect spațial trebuie să definească o operație de modificare a poziției în care se află. Dar această operație este diferită la un om, care poate să-și schimbe singur poziția, față de o minge care trebuie ajutată din exterior pentru a putea fi mutată. Totuși, dacă știm cu siguranță că un obiect este o instanță a unui clase de obiecte care respectă interfața spatio-temporală, putem liniști să executăm asupra acestuia o operație de schimbare a poziției, fără să trebuiască să cunoaștem amănunte despre modul în care va fi executată această operație. Tot ceea ce trebuie să știm este faptul că operația este definită pentru obiectul respectiv.

În concluzie, o interfață este un set de operații care trebuiesc definite de o clasă de obiecte pentru a se înscrie într-o anumită categorie. Vom spune despre o clasă care definește toate operațiile unei interfețe că implementează interfața respectivă.

Cu alte cuvinte, putem privi interfețele ca pe niște reguli de comportament impuse claselor de obiecte. În clipa în care o clasă implementează o anumită interfață, obiectele din clasa respectivă pot fi privite în exclusivitate din acest punct de vedere. Interfețele pot fi privite ca niște filtre prin care putem privi un anumit obiect, filtre care nu lasă la vedere decât proprietățile specifice interfeței, chiar dacă obiectul în vizor este mult mai complicat în realitate.

Interfețele crează o altă împărțire a obiectelor cu care lucrăm. În afară de împărțirea normală pe clase, putem să împărțim obiectele și după interfețele pe care le implementează. Și, la fel cu situația în care definim o operație doar pentru obiectele unei anumite clase, putem defini și operații care lucrează doar

cu obiecte care implementează o anumită interfață, indiferent de clasa din care acestea fac parte. 

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul IV

Structura lexicală Java

[4.1](#) *Setul de caractere*

[4.2](#) *Unități lexicale*

[4.2.1](#) *Cuvinte cheie*

[4.2.2](#) *Identificatori*

[4.2.3](#) *Literali*

[4.2.3.1](#) *Literali întregi*

[4.2.3.2](#) *Literali flotanți*

[4.2.3.3](#) *Literali booleeni*

[4.2.3.4](#) *Literali caracter*

[4.2.3.5](#) *Literali ȳir de caractere*

[4.2.4](#) *Separatori*

[4.2.5](#) *Operatori*

[4.3](#) *Comentarii*

4.1 Setul de caractere

Limbajului Java lucrează ȳn mod nativ folosind setul de caractere Unicode. Acesta este un standard internaȳional care ȳnlocuieȳte vechiul set de caractere ASCII. Motivul acestei ȳnlocuiri a fost necesitatea de a reprezenta mai mult de 256 de caractere. Setul de caractere Unicode, fiind reprezentat pe 16 biȳi are posibilităȳi mult mai mari.

Vechiul standard ASCII este ȳnsă un subset al setului Unicode, ceea ce ȳnseamnă că vom regăsi caracterele ASCII cu exact aceleaȳi coduri ca ȳi mai ȳnainte ȳn noul standard.

Java foloseȳte setul Unicode ȳn timpul rulării aplicaȳiilor ca ȳi ȳn timpul compilării acestora. Folosirea Unicode ȳn timpul execuȳiei nu ȳnseamnă nimic altceva decât faptul că o variabilă Java de tip caracter este reprezentată pe 16 biȳi iar un ȳir de caractere va ocupa fizic ȳn memorie de doua ori mai mulȳi octeȳi decât numărul caracterelor care formează ȳirul.

ȳn ceea ce priveȳte folosirea Unicode ȳn timpul compilării, compilatorul Java acceptă la intrare fiȳiere sursă care pot conȳine orice caractere Unicode. Se poate lucra ȳi cu fiȳiere ASCII obiȳnuite ȳn care putem introduce caractere Unicode folosind secvenȳe escape. Fiȳierele sursă sunt fiȳiere care conȳin declaraȳii

și instrucțiuni Java. Aceste fișiere trec prin trei pași distincți la citirea lor de către compilator:

1. Șirul de caractere Unicode sau ASCII, memorat în fișierul sursă, este transformat într-un șir de caractere Unicode. Caracterele Unicode pot fi introduse și ca secvențe escape folosind doar caractere ASCII.
2. Șirul de caractere Unicode este transformat într-un șir de caractere în care sunt evidențiate separat caracterele de intrare față de caracterele de sfârșit de linie.
3. Șirul de caractere de intrare și de sfârșit de linie este transformat într-un șir de cuvinte ale limbajului Java.

În primul pas al citirii fișierului sursă, sunt generate secvențe escape. *Secvențele escape* sunt secvențe de caractere ASCII care încep cu caracterul *backslash* \. Pentru secvențele escape Unicode, al doilea caracter din secvență trebuie să fie **u** sau **U**. Orice alt caracter care urmează după backslash va fi considerat ca fiind caracter nativ Unicode și lăsat nealterat. Dacă al doilea caracter din secvența escape este **u**, următoarele patru caractere ASCII sunt tratate ca și cifre hexazecimale (în baza 16) care formează împreună doi octeți de memorie care reprezintă un caracter Unicode.

Se pot folosi la intrare și fișiere ASCII normale, pentru că ASCII este un subset al Unicode. De exemplu, putem scrie:

```
int f\u0660 = 3;
```

Numele variabilei are două caractere și al doilea caracter este o cifră arabic-indic.

Exemple de secvențe Unicode:

```
\uaa08 \U0045 \uu6abe
```

În al doilea pas al citirii fișierului sursă, sunt recunoscute ca și caractere de sfârșit de linie caracterele ASCII CR și ASCII LF. În același timp, secvența de caractere ASCII CR-ASCII LF este tratată ca un singur sfârșit de linie și nu două. În acest mod, Java suportă în comun standardele de terminare a liniilor folosite de diferite sisteme de operare: MacOS, Unix și DOS.


Este important să separăm caracterele de sfârșit de linie de restul caracterelor de intrare pentru a ști unde se termină comentariile de o singură linie (care încep cu secvența //) precum și pentru a raporta odată cu erorile de compilare și linia din fișierul sursă în care au apărut acestea.

În pasul al treilea al citirii fișierului sursă, sunt izolate elementele de intrare ale limbajului Java, și anume: spații, comentarii și unități lexicele.

Spațiile pot fi caracterele ASCII SP (spațiu), FF (avans de pagină) sau HT (tab orizontal) precum și orice caracter terminator de linie. Comentariile le vom discuta în paragraful 4.3. 🏠

4.2 Unități lexicale

Unitățile lexicale sunt elementele de bază cu care se construiește semantica programelor Java. În ȋrul de cuvinte de intrare, unitățile lexicale sunt separate ȋntre ele prin comentarii și spaȋii. Unitățile lexicale ȋn limbajul Java pot fi:

- Cuvinte cheie
- Identificatori
- Literali
- Separatori
- Operatori 

4.2.1 Cuvinte cheie

Cuvintele cheie sunt secvenȃe de caractere ASCII rezervate de limbaj pentru uzul propriu. Cu ajutorul lor, Java ȋȃi definește unitățile sintactice de bază. Nici un program nu poate să utilizeze aceste secvenȃe altfel decȃt ȋn modul ȋn care sunt definite de limbaj. Singura excepȃie este aceea cȃ nu existȃ nici o restricȃionare a apariȃiei cuvintelor cheie ȋn ȋiruri de caractere sau comentarii.

Cuvintele cheie ale limbajului Java sunt:

abstract	for	public
boolean	future	rest
break	generic	return
byte	goto	short
case	if	static
cast	implements	super
catch	import	switch
char	inner	synchronized
class	instanceof	this
const	interface	throw
continue	long	throws
default	native	transient
do	new	try
double	null	var
else	operator	void
extends	outer	volatile
final	package	while
finally	private	byvalue
float	protected	

Dintre acestea, cele îngro^oate sunt efectiv folosite, iar restul sunt rezervate pentru viitoare extensii ale limbajului. 🏠

4.2.2 Identificatori

Identificatorii Java sunt secvențe nelimitate de litere și cifre Unicode, începând cu o literă. Identificatorii nu au voie să fie identici cu cuvintele rezervate.

Cifrele Unicode sunt definite în următoarele intervale:

Reprezentare Unicode	Caracter ASCII	Explicație
\u0030-\u0039	0-9	cifre ISO-LATIN-1
\u0660-\u0669		cifre Arabic-Indic
\u06f0-\u06f9		cifre Eastern Arabic-Indic
\u0966-\u096f		cifre Devanagari
\u09e6-\u09ef		cifre Bengali
\u0a66-\u0a6f		cifre Gurmukhi
\u0ae6-\u0aef		cifre Gujarati
\u0b66-\u0b6f		cifre Oriya
\u0be7-\u0bef		cifre Tamil
\u0c66-\u0c6f		cifre Telugu
\u0ce6-\u0cef		cifre Kannada
\u0d66-\u0d6f		cifre Malayalam
\u0e50-\u0e59		cifre Thai
\u0ed0-\u0ed9		cifre Lao
\u1040-\u1049		cifre Tibetan

Tabelul 4.1 Cifrele Unicode.

Un caracter Unicode este o literă dacă este în următoarele intervale ȳi nu este cifră:


Reprezentare Unicode	Caracter ASCII	Explicaȳie
\u0024	\$	semnul dolar (din motive istorice)
\u0041-\u005a	A-Z	litere majuscule Latin
\u005f	_	underscore (din motive istorice)
\u0061-\u007a	a-z	litere minuscule Latin
\u00c0-\u00d6		diferite litere Latin cu diacritice
\u00d8-\u00f6		diferite litere Latin cu diacritice
\u00f8-\u00ff		diferite litere Latin cu diacritice
\u0100-\u1fff		alte alfabete ȳi simboluri non-CJK
\u3040-\u318f		Hiragana, Katakana, Bopomofo, ȳi Hangul
\u3300-\u337f		cuvinte pătrărice CJK
\u3400-\u3d2d		simboluri Hangul coreene
\u4e00-\u9fff		Han (Chinez, Japonez, Corean)
\uf900-\ufaff		compatibilitate Han

Tabelul 4.2 Literele Unicode. 

4.2.3 Literali

Un *literal* este modalitatea de bază de exprimare în fiȳierul sursă a valorilor pe care le pot lua tipurile primitive ȳi tipul ȳir de caractere. Cu ajutorul literalilor putem introduce valori constante în variabilele de tip primitiv sau în variabilele de tip ȳir de caractere.

În limbajul Java există următoarele tipuri de literali:

- literali întregi
- literali flotanți
- literali booleeni
- literali caracter
- literali șir de caractere 

4.2.3.1 Literali întregi

Literalii întregi pot fi reprezentați în baza 10, 16 sau 8. Toate caracterele care se folosesc pentru scrierea literalilor întregi fac parte din subsetul ASCII al setului Unicode.

Literalii întregi pot fi întregi normali sau lungi. Literalii lungi se recunosc prin faptul că se termină cu sufixul **L** sau **l**. Un literal întreg este reprezentat pe 32 de biți iar unul lung pe 64 de biți.

Un *literal întreg în baza 10* începe cu o cifră de la 1 la 9 și se continuă cu un șir de cifre de la 0 la 9. Un literal întreg în baza 10 nu poate să înceapă cu cifra 0, pentru că acesta este semnul folosit pentru a semnaliza literalii scriși în baza 8.

Exemple de literali întregi în baza 10:

```
12356L234871234567890L
```

Exemplul al doilea și al patrulea sunt literali întregi lungi.

Pentru a exprima un *literal întreg în baza 16* trebuie să definim cifrele de la 10 la 15. Convenția va fi următoarea:

```
10 - a, A 13 - d, D
11 - b, B 14 - e, E
12 - c, C 15 - f, F
```

În plus, pentru că un literal întreg în baza 16 poate începe cu o literă, vom adăuga prefixul **0x** sau **0X**. Dacă nu am adăuga acest sufix, compilatorul ar considera că este vorba despre un identificator.

Exemple de literali întregi în baza 16:

```
0xa340X1230x2c45L0xde123abccdL
```

Ultimele două exemple sunt literali întregi lungi.

0234500123001234567712345677L

Cea mai mică valoare a unui literal întreg normal este -2147483648 (-2^{31}), respectiv 0x80000000 și 020000000000. Valorile 0xffffffff și 037777777777 reprezintă amândouă valoarea -1.

[illegible]

La fel ca ȳ la literalii ȳntregi normali, depȳirea acestor limite este o eroare de compilare. 

Literalii flotanți reprezintă numere reale. Ei sunt formați dintr-o parte întreagă, o parte fracționară, un exponent și un sufix de tip. Exponentul, dacă există, este introdus de litera **e** sau **E** urmată opțional de un semn al exponentului.

Este obligatoriu să existe măcar o cifră fie în partea întreagă fie în partea zecimală și punctul zecimal sau litera **e** pentru exponent.

Sufixul care indică tipul flotantului poate fi **F** sau **D** în cazul în care avem o valoare flotantă normală și **d** sau **D** dacă avem o valoare flotantă dublă. Dacă nu este specificat nici un sufix, valoarea este implicit dublă.

Valoarea maximă a unui literal flotant normal este 3.40282347e+38f iar valoarea cea mai mică


reprezentabilă este `1.40239846e-45f`, ambele reprezentate pe 32 de biți.

Valoarea maximă reprezentabilă a unui literal flotant dublu este de `1.79769313486231570e+308` iar valoarea cea mai mică reprezentabilă este `4.94065645841246544e-324`, ambele reprezentate pe 64 de biți.


La fel ca și la literalii întregi, este o eroare să avem exprimat în sursă un literal mai mare decât valoarea maximă reprezentabilă sau mai mic decât cea mai mică valoare reprezentabilă.

Exemple de literalii flotați:

```
1.0e45f-3.456f0..01e-3
```

Primele două exemple reprezintă literalii flotați normali, iar celelalte literalii flotați dubli. 

4.2.3.3 Literalii booleeni

Literalii booleeni nu pot fi decât **true** sau **false**, primul reprezentând valoarea booleană de adevăr iar celălalt valoarea booleană de fals. **True** și **false** nu sunt cuvinte rezervate ale limbajului Java, dar nu veți putea folosi aceste cuvinte ca identificatori. 

4.2.3.4 Literalii caracter

Un *literal de tip caracter* este utilizat pentru a exprima caracterele codului Unicode. Reprezentarea se face fie folosind o literă, fie o secvență escape. Secvențele escape ne permit reprezentarea caracterelor care nu au reprezentare grafică și reprezentarea unor caractere speciale precum backslash și însăși caracterul apostrof.

Caracterele care au reprezentare grafică pot fi reprezentate între apostrofe, ca în exemplele:

```
'a' 'a' ', '
```

Pentru restul caracterelor Unicode trebuie să folosim secvențe escape. Dintre acestea, câteva sunt predefinite în Java, și anume:

Secvență escape	Caracterul reprezentat
<code>'\b'</code>	caracterul backspace BS <code>\u0008</code>

<code>'\t'</code>	caracterul tab orizontal HT <code>\u0009</code>
<code>'\n'</code>	caracterul linefeed LF <code>\u000a</code>
<code>'\f'</code>	caracterul formfeed FF <code>\u000c</code>
<code>'\r'</code>	caracterul carriage return CR <code>\u000d</code>
<code>'\"'</code>	caracterul ghilimele <code>\u0022</code>
<code>'\''</code>	caracterul apostrof <code>\u0027</code>
<code>'\\'</code>	caracterul backslash <code>\u005c</code>

Tabelul 4.3 Secvențe escape predefinite în Java.

În formă generală, o secvență escape se scrie sub una din formele:

```
'\o' '\oo' '\too'
```

unde *o* este o cifră octală iar *t* este o cifră octală între 0 și 3.

Nu este corect să folosiți ca valori pentru literale caracter secvența `'\u000d'` (caracterul ASCII CR), sau altele care reprezintă caractere speciale, pentru că acestea fiind secvențe escape Unicode sunt transformate foarte devreme în timpul procesării sursei în caractere CR și sunt interpretate ca terminatori de linie.

Exemple de secvențe escape:

```
'\n' '\u23a' '\34'
```

dacă după caracterul backslash urmează altceva decât: `b`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7` se va semnala o eroare de compilare.

În acest moment secvențele escape Unicode au fost deja înlocuite cu caractere Unicode native. Dacă **u** apare după `\`, se semnalează o eroare de compilare. 🚫

4.2.3.5 Literali ır de caractere

Un *literal ır de caractere* este format din zero sau mai multe caractere între ghilimele. Caracterele care

formează  irul de caractere pot fi caractere grafice sau secven e escape ca cele definite la literalii caracter.

Dac  un literal  ir de caractere con ine  n interior un caracter terminator de linie va fi semnalat  o eroare de compilare. Cu alte cuvinte, nu putem avea  n surs  ceva de forma:

```
"Acesta este
gre it!"
```

chiar dac  aparent exprimarea ar reprezenta un  ir format din caracterele A, c, e, s, t, a, spa iu, e, s, t, e, linie nou , g, r, e,  , i, t, !. Dac  dorim s  introducem astfel de caractere terminatoare de linie  ntr-un  ir va trebui s  folosim secven e escape ca  n:

```
"Acesta este\ngre it"
```

Dac   irul de caractere este prea lung, putem s -l spargem  n buc  i mai mici pe care s  le concaten m cu operatorul +.

Fiecare  ir de caractere este  n fapt o instan  a a clasei de obiecte **String** declarat  standard  n pachetul *java.lang*.

Exemple de  iruri de caractere:

```
" "\ " " ir de caractere" "unu" + "doi"
```

Primul  ir de caractere din exemplu nu con ine nici un caracter  i se nume te * irul vid*. Ultimul exemplu este format din dou   iruri distincte concatenate. 🏠

4.2.4 Separatori

Un *separator* este un caracter care indic  sf r itul unei unit  i lexicale  i  nceputul alteia. Separatorii sunt necesari atunci c nd unit  i lexicale diferite sunt scrise f r  spa ii  ntre ele. Acestea se pot totu i separa dac  unele dintre ele con in caractere separatori.  n Java separatorii sunt urm torii:


```
( ) { } [ ] ; , .
```

Exemple de separare:

```
a[i]sin(56)
```

 n primul exemplu nu avem o singur  unitate lexical  ci patru: a, [, i,]. Separatorii [ i] ne dau aceast 

informație. În al doilea exemplu, unitățile lexicale sunt tot 4 sin, (, 56,).

Atenție, separatorii participă în același timp și la construcția sintaxei limbajului. Ei nu sunt identici cu spațiile deși, ca și acestea, separă unități lexicale diferite. 

4.2.5 Operatori

Operatorii reprezintă simboluri grafice pentru operațiile elementare definite de limbajul Java. Despre operatori vom discuta mai mult atunci când vom prezenta expresiile. Deocamdată, iată lista tuturor operatorilor limbajului Java:

```
= > < ! ~ ? :
== < > = ! = & & | | ++ --
+ - * / & | ^ % < < > > > >
+ = - = * = / = & = | = ^ = % = < < = > > = > > =
```

Să mai precizăm deocamdată că toți operatorii joacă și rol de separatori. Cu alte cuvinte, din secvența de caractere:

```
vasile+gheorghe
```

putem extrage trei unități lexicale, `vasile`, `+` și `gheorghe`. 

4.3 Comentarii

Un *comentariu* este o secvență de caractere existentă în fișierul sursă dar care servește doar pentru explicarea sau documentarea sursei și nu afectează în nici un fel semantica programelor.


În Java există trei feluri de comentarii:

- Comentarii pe mai multe linii, închise între `/*` și `*/`. Toate caracterele dintre cele două secvențe sunt ignorate.
- Comentarii pe mai multe linii care țin de documentație, închise între `/**` și `*/`. Textul dintre cele două secvențe este automat mutat în documentația aplicației de către generatorul automat de documentație.
- Comentarii pe o singură linie care încep cu `//`. Toate caracterele care urmează acestei secvențe până la primul caracter sfârșit de linie sunt ignorate.

În Java, nu putem să scriem comentarii în interiorul altor comentarii. La fel, nu putem introduce comentarii în interiorul literalilor caracter sau șir de caractere. Secvențele `/*` și `*/` pot să apară pe o linie după secvența `//` dar își pierde semnificația. La fel se întâmplă cu secvența `//` în comentarii care încep cu `/`

* sau /**.

Ca urmare, următoarea secvență de caractere formează un singur comentariu:

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

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul V

Componente de bază ale programelor Java

[5.1 Variabile](#)

[5.2 Expresii](#)

[5.3 Instrucțiuni](#)

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul VI

Obiecte Java

[6.1](#) *Declarapia unei noi clase de obiecte*

[Pasul 1](#): *Stabilirea conceptului reprezentat de clasa de obiecte*

[Pasul 2](#): *Stabilirea numelui clasei de obiecte*

[Pasul 3](#): *Stabilirea superclasei*

[Pasul 4](#): *Stabilirea interfefelor pe care le respectă clasa*

[Pasul 5](#): *Stabilirea modifcatorilor clasei*

[Pasul 6](#): *Scrierea corpului declarapiei*

[Stop](#): *Forma generală a unei declarapii de clasă*

[6.2](#) *Variabilele unei clase*

[6.2.1](#) *Modificatori*

[6.2.2](#) *Protecție*

[6.2.3](#) *Accesarea unei variabile*

[6.2.4](#) *Vizibilitate*

[6.2.5](#) *Variabile predefinite: this și super*

[6.3](#) *Metodele unei clase*

[6.3.1](#) *Declararea metodelor*

[6.3.1.1](#) *Numele și parametrii metodelor*

[6.3.1.2](#) *Modificatori de metode*

[6.3.1.2.1](#) *Metode statice*

[6.3.1.2.2](#) *Metode abstracte*

[6.3.1.2.3](#) *Metode finale*

[6.3.1.2.4](#) *Metode native*

[6.3.1.2.5](#) *Metode sincronizate*

[6.3.1.3](#) *Protejarea metodelor*

[6.3.2](#) *Clauze throws*

[6.3.3](#) *Apelul metodelor*

[6.3.4](#) *Valoarea de retur a unei metode*

[6.3.5 Vizibilitate](#)

[6.4 Inițializatori statici](#)

[6.5 Constructori și finalizatori](#)

[6.5.1 constructori](#)

[6.5.2 Finalizatori](#)

[6.5.3 Crearea instanțelor](#)

[6.6 Derivarea claselor](#)

[6.7 Interfețe](#)

În primul rând să observăm că, atunci când scriem programe în Java nu facem altceva decât să definim noi și noi clase de obiecte. Dintre acestea, unele vor reprezenta însăși aplicația noastră în timp ce altele vor fi necesare pentru rezolvarea problemei la care lucrăm. Ulterior, atunci când dorim să lansăm aplicația în execuție nu va trebui decât să instanțiem un obiect reprezentând aplicația în sine și să apelăm o metodă de pornire definită de către aplicație, metodă care de obicei are un nume și un set de parametri bine fixate. Totuși, numele acestei metode depinde de contextul în care este lansată aplicația noastră.

Această abordare a construcției unei aplicații ne spune printre altele că vom putea lansa oricâte instanțe ale aplicației noastre dorim, pentru că fiecare dintre acestea va reprezenta un obiect în memorie având propriile lui valori pentru variabile. Execuția instanțelor diferite ale aplicației va urma desigur căi diferite în funcție de interacțiunea fiecăreia cu un utilizator, eventual același, și în funcție de unii parametri pe care îi putem defini în momentul creării fiecărei instanțe. 🏠

6.1 Declarația unei noi clase de obiecte

Pasul 1: Stabilirea conceptului reprezentat de clasa de obiecte

Să vedem ce trebuie să definim atunci când dorim să creăm o nouă clasă de obiecte. În primul rând trebuie să stabilim care este conceptul care este reprezentat de către noua clasă de obiecte și să definim informațiile memorate în obiect și modul de utilizare a acestuia. Acest pas este cel mai important din tot procesul de definire al unei noi clase de obiecte. Este necesar să încercați să respectați două reguli oarecum antagonice. Una dintre ele spune că nu trebuie să creați mai multe clase de obiecte decât este nevoie, pentru a nu face dificilă înțelegerea modului de lucru al aplicației la care lucrați. Cea de-a doua regulă spune că nu este bine să mixați într-un singur obiect funcționalități care nu au nimic în comun, creând astfel clase care corespund la două concepte diferite.

Medierea celor două reguli nu este întotdeauna foarte ușoară. Oricum, vă va fi mai ușor dacă păstrați în minte faptul că fiecare clasă pe care o definiți trebuie să corespundă unui concept real bine definit, necesar la rezolvarea problemei la care lucrați. Și mai păstrați în minte și faptul că este inutil să lucrați cu concepte foarte generale atunci când aplicația dumneavoastră nu are nevoie decât de o particularizare a acestora. Riscăți să pierdeți controlul dezvoltării acestor clase de obiecte prea generale și să îngreunați dezvoltarea

aplicației. 🏠

Pasul 2: Stabilirea numelui clasei de obiecte

După ce ați stabilit exact ce doriți de la noua clasă de obiecte, sunteți în măsură să găsiți un nume pentru noua clasă, nume care trebuie să urmeze regulile de construcție ale identificatorilor limbajului Java definite în capitolul anterior.

Stabilirea unui nume potrivit pentru o nouă clasă nu este întotdeauna un lucru foarte ușor. Problema este că acest nume nu trebuie să fie exagerat de lung dar trebuie să exprime suficient de bine destinația clasei. Regulile de denumire ale claselor sunt rezultatul experienței fiecăruia sau al unor convenții de numire stabilite anterior. De obicei, numele de clase este bine să înceapă cu o literă majusculă. Dacă numele clasei conține în interior mai multe cuvinte, aceste cuvinte trebuie de asemenea începute cu literă majusculă. Restul caracterelor vor fi litere minuscule.

De exemplu, dacă dorim să definim o clasă de obiecte care implementează conceptul de motor Otto vom folosi un nume ca `MotorOtto` pentru noua clasă ce trebuie creată. La fel, vom defini clasa `MotorDiesel` sau `MotorCuReacție`. Dacă însă avem nevoie să definim o clasă separată pentru un motor Otto cu cilindri în V și carburator, denumirea clasei ca `MotorOttoCuCilindriÎnVșiCarburator` nu este poate cea mai bună soluție. Poate că în acest caz este preferabilă o prescurtare de forma `MotorOttoVC`. Desigur, acestea sunt doar câteva remarci la adresa acestei probleme și este în continuare necesar ca în timp să vă creați propria convenție de denumire a claselor pe care le creați. 🏠

Pasul 3: Stabilirea superclasei

În cazul în care ați definit deja o parte din funcționalitatea de care aveți nevoie într-o altă superclasă, puteți să derivați noua clasă de obiecte din clasa deja existentă. Dacă nu există o astfel de clasă, noua clasă va fi automat derivată din clasa de obiecte predefinită numită `Object`. În Java, clasa `Object` este superclasă direct sau indirect pentru orice altă clasă de obiecte definită de utilizator.

Alegerea superclasei din care derivați noua clasă de obiecte este foarte importantă pentru că vă ajută să refolosiți codul deja existent. Totuși, nu alegeți cu ușurință superclasa unui obiect pentru că astfel puteți încărca obiectele cu o funcționalitate inutilă, existentă în superclasă. Dacă nu există o clasă care să vă ofere doar funcționalitatea de care aveți nevoie, este preferabil să derivați noua clasă direct din clasa `Object` și să apelați indirect funcționalitatea pe care o doriți. 🏠

Pasul 4: Stabilirea interfețelor pe care le respectă clasa

Stabilirea acestor interfețe are dublu scop. În primul rând ele instruiesc compilatorul să verifice dacă noua clasă respectă cu adevărat toate interfețele pe care le-a declarat, cu alte cuvinte definește toate metodele declarate în aceste interfețe. A doua finalitate este aceea de a permite compilatorului să folosească instanțele noii clase oriunde aplicația declară că este nevoie de un obiect care implementează interfețele declarate.

O clasă poate să implementeze mai multe interfețe sau niciuna. 🏠

Pasul 5: Stabilirea modificatorilor clasei

În unele cazuri trebuie să oferim compilatorului informații suplimentare relative la modul în care vom folosi clasa nou creată pentru ca acesta să poată executa verificări suplimentare asupra descrierii clasei. În acest scop, putem defini o clasă ca fiind abstractă, finală sau publică folosindu-ne de o serie de cuvinte rezervate numite modificatori. Modificatorii pentru tipurile de clase de mai sus sunt respectiv: **abstract**, **final** și **public**.

În cazul în care declarăm o clasă de obiecte ca fiind abstractă, compilatorul va interzice instanțierea acestei clase. Dacă o clasă este declarată finală, compilatorul va avea grijă să nu putem deriva noi subclase din această clasă. În cazul în care declarăm în același timp o clasă de obiecte ca fiind abstractă și finală, eroarea va fi semnalată încă din timpul compilării pentru că cei doi modificatori se exclud.

Pentru ca o clasă să poată fi folosită și în exteriorul contextului în care a fost declarată ea trebuie să fie declarată publică. Mai mult despre acest aspect în paragraful referitor la structura programelor. Până atunci, să spunem că orice clasă de obiecte care va fi instanțiată ca o aplicație trebuie declarată publică. 🏠

Pasul 6: Scrierea corpului declarației

În sfârșit, după ce toți ceilalți pași au fost efectuați, putem trece la scrierea corpului declarației de clasă. În principal, aici vom descrie variabilele clasei împreună cu metodele care lucrează cu acestea. Tot aici putem preciza și gradele de protejare pentru fiecare dintre elementele declarației. Uneori numim variabilele și metodele unei clase la un loc ca fiind câmpurile clasei. Subcapitolele următoare vor descrie în amănunt corpul unei declarații. 🏠

Stop: Forma generală a unei declarații de clasă

Sintaxa exactă de declarare a unei clase arată în felul următor:

```
{ abstract | final | public }* class NumeClasă
```

```
[ extends NumeSuperclasă ]
```

```
[ implements NumeInterfață [ , NumeInterfață ]* ]
```

```
{ [ CâmpClasă ]* } 🏠
```

6.2 Variabilele unei clase

În interiorul claselor se pot declara variabile. Aceste variabile sunt specifice clasei respective. Fiecare dintre

ele trebuie să aibă un tip, un nume și poate avea inițializatori. În afară de aceste elemente, pe care le-am prezentat deja în secțiunea în care am prezentat variabilele, variabilele definite în interiorul unei clase pot avea definiți o serie de modificatori care alterează comportarea variabilei în interiorul clasei, și o specificație de protecție care definește cine are dreptul să acceseze variabila respectivă. 🏠

6.2.1 Modificatori

Modificatorii sunt cuvinte rezervate Java care precizează sensul unei declarații. Iată lista acestora:

```
static
final
transient
volatile
```

Dintre acestea, **transient** nu este utilizat în versiunea curentă a limbajului Java. Pe viitor va fi folosit pentru a specifica variabile care nu conțin informații care trebuie să rămână persistente la terminarea programului.

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

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

Modificatorul **static** este folosit pentru a specifica faptul că variabila are o singură valoare comună tuturor instanțelor clasei în care este declarată. Modificarea valorii acestei variabile din interiorul unui obiect face ca modificarea să fie vizibilă din celelalte obiecte. Variabilele statice sunt inițializate la încărcarea codului specific unei clase și există chiar și dacă nu există nici o instanță a clasei respective. Din această cauză, ele pot fi folosite de metodele statice. 🏠

6.2.2 Protecție

În Java există patru grade de protecție pentru o variabilă aparținând unei clase:


- privată
- protejată
- publică
- prietenoasă

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

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

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

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

O variabilă nu poate avea declarate mai multe grade de protecție în același timp. O astfel de declarație este semnalată ca eroare de compilare. 

6.2.3 Accesarea unei variabile

Accesarea unei variabile declarate în interiorul unei clasei se face folosindu-ne de o expresie de forma:


ReferințăInstanță.NumeVariabilă

Referința către o instanță trebuie să fie referință către clasa care conține variabila. Referința poate fi valoarea unei expresii mai complicate, ca de exemplu un element dintr-un tablou de referințe.

În cazul în care avem o variabilă statică, aceasta poate fi accesată și fără să deținem o referință către o instanță a clasei. Sintaxa este, în acest caz:

NumeClasă.NumeVariabilă 

6.2.4 Vizibilitate

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

NumeSuperClasă.NumeVariabilă

sau

super.*NumeVariabilă*

în cazul în care superclasa este imediată.


La fel, o variabilă a unei clase poate fi ascunsă de o declarație de variabilă dintr-un bloc de instrucțiuni. Orice referință la ea va trebui făcută în mod explicit. Expresia de referire este, în acest caz:

this.NumeVariabilă 

6.2.5 Variabile predefinite: **this** și **super**

În interiorul fiecărei metode non-statice dintr-o clasă există predefinite două variabile cu semnificație specială. Cele două variabile sunt de tip referință și au aceeași valoare și anume o referință către obiectul curent. Diferența dintre ele este tipul.


Prima dintre acestea este variabila **this** care are tipul referință către clasa în interiorul căreia apare metoda. A doua este variabila **super** al cărei tip este o referință către superclasa imediată a clasei în care apare metoda. În interiorul obiectelor din clasa **Object** nu se poate folosi referința **super** pentru că nu există nici o superclasă a clasei de obiecte **Object**.

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

6.3 Metodele unei clase

Fiecare clasă își poate defini propriile sale metode pe lângă metodele pe care le moștenește de la superclasa sa. Aceste metode definesc operațiile care pot fi executate cu obiectul respectiv. În cazul în care una dintre metodele moștenite nu are o implementare corespunzătoare în superclasă, clasa își poate redefini metoda după cum dorește.

În plus, o clasă își poate defini metode de construcție a obiectelor și metode de eliberare a acestora. Metodele de construcție sunt apelate ori de câte ori este alocat un nou obiect din clasa respectivă. Putem declara mai multe metode de construcție, ele diferind prin parametrii din care trebuie construit obiectul.

Metodele de eliberare a obiectului sunt cele care eliberează resursele ocupate de obiect în momentul în care acesta este distrus de către mecanismul automat de colectare de gunoarie. Fiecare clasă are o singură metodă de eliberare, numită și finalizator. Apelarea acestei metode se face de către sistem și nu există nici o cale de control al momentului în care se produce acest apel. 

6.3.1 Declaraarea metodelor

Pentru a declara o metodă, este necesar să declarăm numele acesteia, tipul de valoare pe care o întoarce, parametrii metodei precum și un bloc în care să descriem instrucțiunile care trebuiesc executate atunci când metoda este apelată. În plus, orice metodă are un număr de modificatori care descriu proprietățile metodei și

modul de lucru al acesteia.

Declararea precum și implementarea metodelor se face întotdeauna în interiorul declarației de clasă. Nu există nici o cale prin care să putem scrie o parte dintre metodele unei clase într-un fișier separat care să facă referință apoi la declarația clasei.

În formă generală, declarația unei metode arată în felul următor:

[Modificator] TipRezultat Declarație [ClauzeThrows]* CorpulMetodei*

Modificatorii precum și clauzele **throws** pot să lipsească. 

6.3.1.1 Numele și parametrii metodelor

Recunoașterea unei anumite metode se face după numele și tipul parametrilor săi. Pot exista metode cu același nume dar având parametri diferiți. Acest fenomen poartă numele de *supraîncărcarea numelui* unei metode.

Numele metodei este un identificator Java. Avem toată libertatea în a alege numele pe care îl dorim pentru metodele noastre, dar în general este preferabil să alegem nume care sugerează utilizarea metodei.

Numele unei metode începe de obicei cu literă mică. Dacă acesta este format din mai multe cuvinte, litera de început a fiecărui cuvânt va fi majusculă. În acest mod numele unei metode este foarte ușor de citit și de depistat în sursă.

Parametrii metodei sunt în realitate niște variabile care sunt inițializate în momentul apelului cu valori care controlează modul ulterior de execuție. Aceste variabile există pe toată perioada execuției metodei. Se pot scrie metode care să nu aibă nici un parametru.

Fiind o variabilă, fiecare parametru are un tip și un nume. Numele trebuie să fie un identificator Java. Deși avem libertatea să alegem orice nume dorim, din nou este preferabil să alegem nume care să sugereze scopul la care va fi utilizat parametrul respectiv.

Tipul unui parametru este oricare dintre tipurile valide în Java. Acestea poate fi fie un tip primitiv, fie un tip referință către obiect, interfață sau tablou.

În momentul apelului unei metode, compilatorul încearcă să găsească o metodă în interiorul clasei care să aibă același nume cu cel apelat și același număr de parametri ca și apelul. Mai mult, tipurile parametrilor de apel trebuie să corespundă cu tipurile parametrilor declarați ai metodei găsite sau să poată fi convertiți la aceștia.

Dacă o astfel de metodă este găsită în declarația clasei sau în superclasele acesteia, parametrii de apel sunt convertiți către tipurile declarate și se generează apelul către metoda respectivă.


Este o eroare de compilare să declarăm două metode cu același nume, același număr de parametri și același tip pentru parametrii corespunzători. Într-o asemenea situație, compilatorul n-ar mai ști care metodă trebuie apelată la un moment dat.

De asemenea, este o eroare de compilare să existe două metode care se potrivesc la același apel. Acest lucru se întâmplă când nici una dintre metodele existente nu se potrivește exact și când există două metode cu același nume și același număr de parametri și, în plus, parametrii de apel se pot converti către parametrii declarați ai ambelor metode.

Rezolvarea unei astfel de probleme se face prin conversia explicită (cast) de către programator a valorilor de apel spre tipurile exacte ale parametrilor metodei pe care dorim să o apelăm în realitate.


În fine, forma generală de declarație a numelui și parametrilor unei metode este:

NumeMetodă([*TipParametru NumeParametru*]

[,*TipParametru NumeParametru*]*) 

6.3.1.2 Modificatori de metode

Modificatorii sunt cuvinte cheie ale limbajului Java care specifică proprietăți suplimentare pentru o metodă. Iată lista completă a acestora în cazul metodelor:

- **static** - pentru metodele statice
- **abstract** - pentru metodele abstracte
- **final** - pentru metodele finale
- **native** - pentru metodele native
- **synchronized** - pentru metodele sincronizate 

6.3.1.2.1 Metode statice

În mod normal, o metodă a unei clase se poate apela numai printr-o instanță a clasei respective sau printr-o instanță a unei subclase. Acest lucru se datorează faptului că metoda face apel la o serie de variabile ale clasei care sunt memorate în interiorul instanței și care au valori diferite în instanțe diferite. Astfel de metode se numesc *metode ale instanțelor clasei*.

După cum știm deja, există și un alt tip de variabile, și anume variabilele de clasă sau variabilele statice care sunt comune tuturor instanțelor clasei respective și există pe toată perioada de timp în care clasa este încărcată în memorie. Aceste variabile pot fi accesate fără să avem nevoie de o instanță a clasei respective.

În mod similar există și *metode statice*. Aceste metode nu au nevoie de o instanță a clasei sau a unei subclase pentru a putea fi apelate pentru că ele nu au nevoie să folosească variabile care sunt memorate în interiorul instanțelor. În schimb, aceste metode pot să folosească variabilele statice declarate în interiorul clasei.


Orice metodă statică este implicit `final`. 

6.3.1.2.2 Metode abstracte

Metodele abstracte sunt metode care nu au corp de implementare. Ele sunt declarate numai pentru a forța subclasele care vor să aibă instanțe să implementeze metodele respective.


Metodele abstracte trebuie declarate numai în interiorul claselor care au fost declarate abstracte. Altfel compilatorul va semnala o eroare de compilare. Orice subclasă a claselor abstracte care nu este declarată abstractă trebuie să ofere o implementare a acestor metode, altfel va fi generată o eroare de compilare.

Prin acest mecanism ne asigurăm că toate instanțele care pot fi convertite către clasa care conține definiția unei metode abstracte au implementat metoda respectivă dar, în același timp, nu este nevoie să implementăm în nici un fel metoda chiar în clasa care o declară pentru că nu știm pe moment cum va fi implementată.

O metodă statică nu poate fi declarată `abstract` pentru că o metodă statică este implicit `final` și nu poate fi rescrisă. 


6.3.1.2.3 Metode finale

O *metodă finală* este o metodă care nu poate fi rescrisă în subclasele clasei în care a fost declarată. O metodă este rescrisă într-o subclasă dacă aceasta implementează o metodă cu același nume și același număr și tip de parametri ca și metoda din superclasă.

Declararea metodelor finale este utilă în primul rând compilatorului care poate genera metodele respective direct în codul rezultat fiind sigur că metoda nu va avea nici o altă implementare în subclase. 

6.3.1.2.4 Metode native

Metodele native sunt metode care sunt implementate pe o cale specifică unei anumite platforme. De obicei aceste metode sunt implementate în C sau în limbaj de asamblare. Metoda propriu-zisă nu poate avea corp de implementare pentru că implementarea nu este făcută în Java.

În rest, metodele native sunt exact ca orice altă metodă Java. Ele pot fi `modificate`, pot fi `statice` sau nu, pot fi finale sau nu, pot să rescrie o metodă din superclasă și pot fi la rândul lor rescrise în subclase. 

6.3.1.2.5 Metode sincronizate

O *metodă sincronizată* este o metodă care conține cod critic pentru un anumit obiect sau clasă și nu poate fi rulată în paralel cu nici o altă metodă critică sau cu o instrucțiune **synchronized** referitoare la același obiect sau clasă.

Înainte de executarea metodei, obiectul sau clasa respectivă sunt blocate. La terminarea metodei, acestea sunt deblocate.

Dacă metoda este statică atunci este blocată o întreagă clasă, clasa din care face parte metoda. Altfel, este blocată doar instanța în contextul căreia este apelată metoda. 🏠

6.3.1.3 Protejarea metodelor

Accesul la metodele unei clase este protejat în același fel ca și accesul la variabilele clasei. În Java există patru grade de protecție pentru o metodă aparținând unei clase:

- privată
- protejată
- publică
- prietenoasă

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

O metodă declarată protejată este accesibilă în orice clasă din pachetul căreia îi aparține clasa în care este declarată. În același timp, metoda este accesibilă în toate subclasele clasei date, chiar dacă ele aparțin altor pachete. Cuvântul rezervat este **protected**.

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

O metodă care nu are nici o declarație relativă la gradul de protecție este automat o metodă prietenoasă. O metodă prietenoasă este accesibilă în pachetul din care face parte clasa în interiorul căreia a fost declarată la fel ca și o metodă protejată. Dar, spre deosebire de metodele protejate, o metodă prietenoasă nu este accesibilă în subclasele clasei date dacă aceste sunt declarate ca aparținând unui alt pachet. Nu există un cuvânt rezervat pentru specificarea explicită a metodelor prietenoase.

O metodă nu poate avea declarate mai multe grade de protecție în același timp. O astfel de declarație este semnalată ca eroare de compilare. 🏠

6.3.2 Clauze throws

Dacă o metodă poate arunca o excepție, adică să apeleze instrucțiunea **throw**, ea trebuie să declare tipul acestor excepții într-o clauză **throws**. Sintaxa acesteia este:

throws *NumeTip* [, *NumeTip*]*

Numele de tipuri specificate în clauza **throws** trebuie să fie accesibile și să fie asignabile la tipul de clasă

Throwable. Dacă o metodă conține o clauză **throws**, este o eroare de compilare ca metoda să arunce un obiect care nu este asignabil la compilare la tipurile de clase **Error**, **RuntimeException** sau la tipurile de clase specificate în clauza **throws**.

Dacă o metodă nu are o clauză **throws**, este o eroare de compilare ca aceasta să poată arunca o excepție normală din interiorul corpului ei. 🏠

6.3.3 Apelul metodelor

Pentru a apela o metodă a unei clase este necesar să dispunem de o cale de acces la metoda respectivă. În plus, trebuie să dispunem de drepturile necesare apelului metodei.

Sintaxa efectivă de acces este următoarea:

CaleDeAcces.Metodă(Parametri)

În cazul în care metoda este statică, pentru a specifica o cale de acces este suficient să furnizăm numele clasei în care a fost declarată metoda. Accesul la numele clasei se poate obține fie importând clasa sau întreg pachetul din care face parte clasa fie specificând în clar numele clasei și drumul de acces către aceasta.

De exemplu, pentru a accesa metoda **random** definită static în clasa **Math** aparținând pachetului **java.lang** putem scrie:

```
double aleator = Math.random();
```

sau, alternativ:

```
double aleator = java.lang.Math.random();
```

În cazul claselor definite în pachetul **java.lang** nu este necesar nici un import pentru că acestea sunt implicit importate de către compilator.

Cea de-a doua cale de acces este existența unei instanțe a clasei respective. Prin această instanță putem accesa metodele care nu sunt declarate statice, numite uneori și metode ale instanțelor clasei. Aceste metode au nevoie de o instanță a clasei pentru a putea lucra, pentru că folosesc variabile non-stactice ale clasei sau apelează alte metode non-stactice. Metodele primesc acest obiect ca pe un parametru ascuns.

De exemplu, având o instanță a clasei **Object** sau a unei subclase a acesteia, putem obține o reprezentare sub formă de șir de caractere prin:

```
Object obiect = new Object();
String sir = obiect.toString();
```


În cazul în care apelăm o metodă a clasei din care face parte o metoda apelantă putem să renunțăm la calea de acces în cazul metodelor statice, scriind doar numele metodei și parametrii. Pentru metodele specifice instanțelor, putem renunța la calea de acces, dar în acest caz metoda accesează aceeași instanță ca și metoda apelantă. În cazul în care metoda apelantă este statică, specificarea unei instanțe este obligatorie în cazul metodelor de instanță.

Parametrii de apel servesc împreună cu numele la identificarea metodei pe care dorim să o apelăm. Înainte de a fi transmiși, aceștia sunt convertiți către tipurile declarate de parametri ai metodei, după cum este descris mai sus.

Specificarea parametrilor de apel se face separându-i prin virgulă. După ultimul parametru nu se mai pune virgulă. Dacă metoda nu are nici un parametru, parantezele rotunde sunt în continuare necesare. Exemple de apel de metode cu parametri:

```
String numar = String.valueOf( 12 );
// 12 -> String
double valoare = Math.abs( 12.54 );
// valoare absolută
String prima = numar.substring( 0, 1 );
// prima litera ^
```

6.3.4 Valoarea de retur a unei metode

O metodă trebuie să-și declare tipul valorii pe care o întoarce. În cazul în care metoda dorește să specifice explicit că nu întoarce nici o valoare, ea trebuie să declare ca tip de retur tipul **void** ca în exemplul:

```
void a() { ... }
```

În caz general, o metodă întoarce o valoare primitivă sau un tip referință. Putem declara acest tip ca în:

```
Thread cautăFirulCurent() { ... }
long abs( int valoare ) { ... }
```

Pentru a returna o valoare ca rezultat al execuției unei metode, trebuie să folosim instrucțiunea **return**, așa cum s-a arătat în secțiunea dedicată instrucțiunilor. Instrucțiunea **return** trebuie să conțină o expresie a cărei valoare să poată fi convertită la tipul declarat al valorii de retur a metodei.

De exemplu:

```
long abs( int valoare ) {
return Math.abs( valoare );
}
```

Metoda statică **abs** din clasa **Math** care primește un parametru întreg returnează tot un întreg. În exemplul

nostru, instrucțiunea **return** este corectă pentru că există o cale de conversie de la întreg la întreg lung, conversie care este apelată automat de compilator înainte de ieșirea din metodă.

În schimb, în exemplul următor:

```
int abs( long valoare ) {
    return Math.abs( valoare );
}
```

compilatorul va genera o eroare de compilare pentru că metoda statică **abs** din clasa **Math** care primește ca parametru un întreg lung întoarce tot un întreg lung, iar un întreg lung nu poate fi convertit sigur la un întreg normal pentru că există riscul deteriorării valorii, la fel ca la atribuire.

Rezolvarea trebuie să conțină un cast explicit:

```
int abs( long valoare ) {
    return ( int )Math.abs( valoare );
}
```

În cazul în care o metodă este declarată **void**, putem să ne întoarcem din ea folosind instrucțiunea **return** fără nici o expresie. De exemplu:

```
void metoda() {
    ...
    if( ... )
        return;
    ...
}
```

Specificarea unei expresii în acest caz duce la o eroare de compilare. La fel și în cazul în care folosim instrucțiunea **return** fără nici o expresie în interiorul unei metode care nu este declarată **void**. 🏠

6.3.5 Vizibilitate

O metodă este vizibilă dacă este declarată în clasa prin care este apelată sau într-una din superclasele acesteia. De exemplu, dacă avem următoarea declarație:

```
class A {
    ...
    void a() { ... }
}
class B extends A {
    void b() {
        a();
    }
}
```

```

    c();
    ...
}
void c() { .. }
...
}

```

Apelul metodei *a* în interiorul metodei *b* din clasa B este permis pentru că metoda *a* este declarată în interiorul clasei A care este superclasă pentru clasa B. Apelul metodei *c* în aceeași metodă *b* este permis pentru că metoda *c* este declarată în aceeași clasă ca și metoda *a*.

Uneori, o subclasă rescrie o metodă dintr-o superclasă a sa. În acest caz, apelul metodei respective în interiorul subclasei duce automat la apelul metodei din subclasă. Dacă totuși dorim să apelăm metoda a^oa cum a fost ea definită în superclasă, putem prefixa apelul cu numele superclasei. De exemplu:

```

class A {
    ...
    void a() { ... }
}
class B extends A {
    void a() { .. }
    void c() {
        a(); // metoda a din clasa B
        A.a(); // metoda a din clasa A
    }
    ...
}

```

Desigur, pentru a "vedea" o metodă și a o putea apela, este nevoie să avem drepturile necesare. 🏠

6.4 Inițializatori statici

La încărcarea unei clase sunt automat inițializate toate variabilele statice declarate în interiorul clasei. În plus, sunt apelați toți inițializatorii statici ai clasei.

Un *inițializator static* are următoarea sintaxă:

```
static BlocDeInstrucțiuni
```

Blocul de instrucțiuni este executat automat la încărcarea clasei. De exemplu, putem defini un inițializator static în felul următor:

```

class A {
    static double a;
    static int b;
    static {
        a = Math.random();
        // număr dublu între 0.0 și 1.0
        b = ( int )( a * 500 );
        // număr întreg între 0 și 500
    }
    ...
}

```

Declarațiile de variabile statice și inițializatorii statici sunt executate în ordinea în care apar în clasă. De exemplu, dacă avem următoarea declarație de clasă:

```

class A {
    static int i = 11;
    static {
        i += 100;
        i %= 55;
    }
    static int j = i + 1;
}

```

valoarea finală a lui *i* va fi 1 ((11 + 100) % 55) iar valoarea lui *j* va fi 2. 🏠

6.5 Constructori și finalizatori

6.5.1 constructori

La crearea unei noi instanțe a unei clase sistemul alocă automat memoria necesară instanței și o inițializează cu valorile inițiale specificate sau implicite. Dacă dorim să facem inițializări suplimentare în interiorul acestei memorii sau în altă parte putem descrie metode speciale numite *constructori* ai clasei.

Putem avea mai mulți constructori pentru aceeași clasă, aceștia diferind doar prin parametrii pe care îi primesc. Numele tuturor constructorilor este același și este identic cu numele clasei.

Declarația unui constructor este asemănătoare cu declarația unei metode oarecare, cu diferența că nu putem specifica o valoare de retur și nu putem specifica nici un fel de modificatori. Dacă dorim să returnăm dintr-un constructor, trebuie să folosim instrucțiunea **return** fără nici o expresie. Putem însă să specificăm gradul de protecție al unui constructor ca fiind public, privat, protejat sau prietenos.

Constructorii pot avea clauze **throws**.

Dacă o clasă nu are constructori, compilatorul va crea automat un *constructor implicit* care nu ia nici un parametru și care inițializează toate variabilele clasei și apelează constructorul superclasei fără argumente prin **super()**. Dacă superclasa nu are un constructor care ia zero argumente, se va genera o eroare de compilare.

Dacă o clasă are cel puțin un constructor, constructorul implicit nu mai este creat de către compilator.

Când construim corpul unui constructor avem posibilitatea de a apela, pe prima linie a blocului de instrucțiuni care reprezintă corpul constructorului, un *constructor explicit*. Constructorul explicit poate avea două forme:

```
this( [Parametri] );
```

```
super( [Parametri] );
```

Cu această sintaxă apelăm unul dintre constructorii superclasei sau unul dintre ceilalți constructori din aceeași clasă. Aceste linii nu pot apărea decât pe prima poziție în corpul constructorului. Dacă nu apar acolo, compilatorul consideră implicit că prima instrucțiune din corpul constructorului este:

```
super();
```

și în acest caz se va genera o eroare de compilare dacă nu există un constructor în superclasă care să lucreze fără nici un parametru.

După apelul explicit al unui constructor din superclasă cu sintaxa **super**(...) este executată în mod implicit inițializarea tuturor variabilelor de instanță (non-static) care au inițializatori expliți. După apelul unui constructor din aceeași clasă cu sintaxa **this**(...) nu există nici o altă acțiune implicită, deci nu vor fi inițializate nici un fel de variabile. Aceasta datorită faptului că inițializarea s-a produs deja în constructorul apelat.

Iată și un exemplu:

```
class A extends B {
    String valoare;
    A( String val ) {
        // aici există apel implicit
        // al lui super(), adică B()
        valoare = val;
    }
    A( int val ) {
        this( String.valueOf( val ) ); // alt constructor
    }
} ^
_
```

6.5.2 Finalizatori

În Java nu este nevoie să apelăm în mod explicit distrugerea unei instanțe atunci când nu mai este nevoie de ea. Sistemul oferă un mecanism de colectare a gunoaielor care recunoaște situația în care o instanță de obiect sau un tablou nu mai sunt referite de nimeni și le distruge în mod automat.

Acest mecanism de colectare a gunoaielor rulează pe un fir de execuție separat, de prioritate mică. Nu avem nici o posibilitate să aflăm exact care este momentul în care va fi distrusă o instanță. Totuși, putem specifica o funcție care să fie apelată automat în momentul în care colectorul de gunoaie încearcă să distrugă obiectul.

Această funcție are nume, număr de parametri și tip de valoare de retur fixe:

```
void finalize()
```

După apelul *metodei de finalizare* (numită și *finalizator*), instanța nu este încă distrusă până la o nouă verificare din partea colectorului de gunoaie. Această comportare este necesară pentru că instanța poate fi revitalizată prin crearea unei referințe către ea în interiorul finalizatorului.

Totuși, finalizatorul nu este apelat decât o singură dată. Dacă obiectul revitalizat redevine candidat la colectorul de gunoaie, acesta este distrus fără a i se mai apela finalizatorul. Cu alte cuvinte, un obiect nu poate fi revitalizat decât o singură dată.

Dacă în timpul finalizării apare o excepție, ea este ignorată și finalizatorul nu va mai fi apelat din nou. 🏠

6.5.3 Crearea instanțelor

O instanță este creată folosind o expresie de alocare care folosește cuvântul rezervat **new**. Iată care sunt pașii care sunt executați la apelul acestei expresii:

- Se creează o nouă instanță de tipul specificat. Toate variabilele instanței sunt inițializate pe valorile lor implicite.
- Se apelează constructorul corespunzător în funcție de parametrii care sunt transmiși în expresia de alocare. Dacă instanța este creată prin apelul metodei **newInstance**, se apelează constructorul care nu ia nici un argument.
- După creare, expresia de alocare returnează o referință către instanța nou creată.

Exemple de creare:

```
A o1 = new A();
B o2 = new B();
class C extends B {
    String valoare;
    C( String val ) {
```

```
// aici există apel implicit
// al lui super(), adică B()
valoare = val;
}
C( int val ) {
this( String.valueOf( val ) );
}
}
C o3 = new C( "Vasile" );
C o4 = new C( 13 );
```

O altă cale de creare a unui obiect este apelul metodei **newInstance** declarate în clasa **Class**. Iată pașii de creare în acest caz:

- Se creează o nouă instanță de același tip cu tipul clasei pentru care a fost apelată metoda **newInstance**. Toate variabilele instanței sunt inițializate pe valorile lor implicite.
- Este apelat constructorul obiectului care nu ia nici un argument.
- După creare referința către obiectul nou creat este returnată ca

valoare a metodei **newInstance**. Tipul acestei referințe va fi **Object** în timpul compilării și tipul clasei reale în timpul execuției. 🏠

6.6 Derivarea claselor

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

Sintaxa generală este:

```
class SubClasă extends SuperClasă ...
```

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

Clasa derivată moștenește toate variabilele și metodele superclasei sale. Totuși, ea nu poate accesa decât acele variabile și metode care nu sunt declarate private.

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

Exemplu:

```

class A {
    int a = 1;
    void unu() {
        System.out.println( a );
    }
}
class B extends A {
    double a = Math.PI;
    void unu() {
        System.out.println( a );
    }
    void doi() {
        System.out.println( A.a );
    }
    void trei() {
        unu();
        super.unu();
    }
}

```

Dacă apelăm metoda *unu* din clasa A, aceasta va afi^oa la consolă numărul 1. Acest apel se va face cu instrucțiunea:

```
( new A() ).unu();
```

Dacă apelăm metoda *unu* din clasa B, aceasta va afi^oa la consolă numărul π . Apelul îl putem face de exemplu cu instrucțiunea:

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

Observați că în metoda *unu* din clasa B, variabila referită este variabila *a* din clasa B. Variabila *a* din clasa A este ascunsă. Putem însă să o referim prin sintaxa *A.a* ca în metoda *doi* din clasa B.

În interiorul clasei B, apelul metodei *unu* fără nici o altă specificație duce automat la apelul metodei *unu* definite în interiorul clasei B. Metoda *unu* din clasa B rescrie metoda *unu* din clasa A. Vechea metodă este accesibilă pentru a o referi în mod explicit ca în metoda *trei* din clasa B. Apelul acestei metode va afi^oa mai întâi numărul π și apoi numărul 1.

Dacă avem declarată o variabilă de tip referință către o instanță a clasei A, această variabilă poate să conțină în timpul execuției și o referință către o instanță a clasei B. Invers, afirmația nu este valabilă.

În clipa în care apelăm metoda *unu* pentru o variabilă referință către clasa A, sistemul va apela metoda *unu* a clasei A sau B în funcție de adevăratul tip al referinței din timpul execuției. Cu alte cuvinte, următoarea secvență de instrucțiuni:

```
A tablou[] = new A[2];
tablou[0] = new A();
tablou[1] = new B();
for( int i = 0; i < 2; i++ ) {
    tablou[i].unu();
}
```

va afișa două numere diferite, mai întâi 1 și apoi PI. Aceasta din cauză că cel de-al doilea element din tablou este, în timpul execuției, de tip referință la o instanță a clasei B chiar dacă la compilare este de tipul referință la o instanță a clasei A.

Acest mecanism se numește *legare târzie*, și înseamnă că metoda care va fi efectiv apelată este stabilită doar în timpul execuției și nu la compilare.

Dacă nu declarăm nici o superclasă în definiția unei clase, atunci se consideră automat că noua clasă derivă direct din clasa Object, moștenind toate metodele și variabilele acesteia. 🏠

6.7 Interfețe

O *interfață* este în esență o declarație de tip ce constă dintr-un set de metode și constante pentru care nu s-a specificat nici o implementare. Programele Java folosesc interfețele pentru a suplini lipsa moștenirii multiple, adică a claselor de obiecte care derivă din două sau mai multe alte clase.

Sintaxa de declarație a unei interfețe este următoarea:

Modificatori **interface** *NumeInterf* [**extends** [*Interfață*][, *Interfață*]*]

Corp

Modificatorii unei interfețe pot fi doar cuvintele rezervate **public** și **abstract**. O interfață care este publică poate fi accesată și de către alte pachete decât cel care o definește. În plus, fiecare interfață este în mod implicit abstractă. Modificatorul **abstract** este permis dar nu obligatoriu.

Numele interfețelor trebuie să fie identificatori Java. Convențiile de numire a interfețelor le urmează în general pe cele de numire a claselor de obiecte.

Interfețele, la fel ca și clasele de obiecte, pot avea *subinterfețe*. Subinterfețele moștenesc toate constantele și declarațiile de metode ale interfeței din care derivă și pot defini în plus noi elemente. Pentru a defini o subinterfață, folosim o clauză **extends**. Aceste clauze specifică superinterfața unei interfețe. O interfață

poate avea mai multe superinterfețe care se declară separate prin virgulă după cuvântul rezervat **extends**. Circularitatea definiției subinterfețelor nu este permisă.

În cazul interfețelor nu există o rădăcină comună a arborelui de derivare așa cum există pentru arborele de clase, clasa **Object**.

În corpul unei declarații de interfață pot să apară declarații de variabile și declarații de metode. Variabilele sunt implicit statice și finale. Din cauza faptului că variabilele sunt finale, este obligatoriu să fie specificată o valoare inițială pentru aceste variabile. În plus, această valoare inițială trebuie să fie constantă (să nu depindă de alte variabile).

Dacă interfața este declarată publică, toate variabilele din corpul său sunt implicit declarate publice.

În ceea ce privește metodele declarate în interiorul corpului unei interfețe, acestea sunt implicit declarate abstracte. În plus, dacă interfața este declarată publică, metodele din interior sunt implicit declarate publice.

Iată un exemplu de declarații de interfețe:

```
public interface ObiectSpatial {
    final int CUB = 0;
    final int SFERA = 1;
    double greutate();
    double volum();
    double raza();
    int tip();
}
public interface ObiectSpatioTemporal extends ObiectSpatial {
    void centrulDeGreutate( long moment,
        double coordonate[] );
    long momentInitial();
    long momentFinal();
}
```

Cele două interfețe definesc comportamentul unui obiect spațial respectiv al unui obiect spațio-temporal. Un obiect spațial are o greutate, un volum și o rază a sferei minime în care se poate înscrie. În plus, putem defini tipul unui obiect folosindu-ne de o serie de valori constante predefinite precum ar fi SFERA sau CUB.

Un obiect spațio-temporal este un obiect spațial care are în plus o poziție pe axa timpului. Pentru un astfel de obiect, în afară de proprietățile deja descrise pentru obiectele spațiale, trebuie să avem în plus un moment inițial, de apariție, pe axa timpului și un moment final. Obiectul nostru nu există în afara acestui interval de timp. În plus, pentru un astfel de obiect putem afla poziția centrului său de greutate în fiecare moment aflat în intervalul de existență.

Pentru a putea lucra cu obiecte spațiale și spațio-temporale este nevoie să definim diverse clase care să

implementeze aceste interfețe. Acest lucru se face specificând clauza **implements** în declarația de clasă. O clasă poate implementa mai multe interfețe. Dacă o clasă declară că implementează o anumită interfață, ea este obligatoriu să implementeze toate metodele declarate în interfața respectivă.

De exemplu, putem spune că o minge este un obiect spațial de tip sferă. În plus, mingea are o poziție în funcție de timp și un interval de existență. Cu alte cuvinte, mingea este chiar un obiect spațio-temporal. Desigur, în afară de proprietățile spațio-temporale mingea mai are și alte proprietăți precum culoarea, proprietarul sau prețul de cumpărare.

Iată cum ar putea arăta definiția clasei de obiecte de tip minge:

```
import java.awt.Color;
class Minge extends Jucarie implements ObiectSpatioTemporal {
    int culoare = Color.red;
    double pret = 10000.0;
    double raza = 0.25;
    long nastere;
    long moarte;
    // metodele din ObiectSpatial
    double greutate() {
        return raza * 0.5;
    }
    double raza() {
        return raza;
    }
    double volum() {
        return ( 4.0 / 3.0 ) * Math.PI * raza * raza * raza;
    }
    int tip() {
        return SFERA;
    }

    // metodele din interfața ObiectSpatioTemporal
    boolean centrulDeGreutate( long moment,
        double coordonate[] ) {
        if( moment < nastere || moment > moarte ) {
            return false;
        }
        ...
        coordonate[0] = x;
        coordonate[1] = y;
        coordonate[2] = z;
        return true;
    }
    long momentInitial() {
```

```

return nastere;
}
long momentFinal() {
return moarte;
}
int ceCuloare() {
return culoare;
}
double cePret() {
return pret;
}
}

```

Observați că noua clasă `Minge` implementează toate metodele definite în interfața `ObiectSpatioTemporal` și, pentru că aceasta extinde interfața `ObiectSpatial`, și metodele definite în cea din urmă. În plus, clasa își definește propriile metode și variabile.

Să presupunem în continuare că avem și o altă clasă, `Rezervor`, care este tot un obiect spațio-temporal, dar de formă cubică. Declarația acestei clase ar arăta ca:

```

class Rezervor extends Constructii implements ObiectSpatioTemporal {

    ...
}

```

Desigur, toate metodele din interfețele de mai sus trebuie implementate, plus alte metode specifice.

Să mai observăm că cele două obiecte derivă din clase diferite: `Mingea` din `Jucării` iar `Rezervorul` din `Construcții`. Dacă am putea deriva o clasă din două alte clase, am putea deriva `Minge` din `Jucarie` și `ObiectSpatioTemporal` iar `Rezervor` din `Constructie` și `ObiectSpatioTemporal`. Într-o astfel de situație, nu ar mai fi necesar ca `ObiectSpatioTemporal` să fie o interfață, ci ar fi suficient ca acesta să fie o altă clasă.

Din păcate, în Java, o clasă nu poate deriva decât dintr-o singură altă clasă, așa că este obligatoriu în astfel de situații să folosim interfețele. Dacă `ObiectSpatioTemporal` ar fi putut fi o clasă, am fi avut avantajul că puteam implementa acolo metodele cu funcționare identică din cele două clase discutate, acestea fiind automat moștenite fără a mai fi nevoie de definirea lor de două ori în fiecare clasă în parte.

Putem crea în continuare metode care să lucreze cu obiecte spațio-temporale, de exemplu o metodă care să afle distanța unui corp spațio-temporal față de un punct dat la momentul său inițial. O astfel de metodă se poate scrie o singură dată, și poate lucra cu toate clasele care implementează interfața noastră. De exemplu:

```

...
double distanta( double punct[], ObiectSpatioTemporal obiect ) {

```

```

double coordonate[] = new double[3];
obiect.centruDeGreutate( obiect.momentInitial(),
    coordonate );
double x = coordonate[0] - punct[0];
double y = coordonate[1] - punct[1];
double z = coordonate[2] - punct[2];
return Math.sqrt( x * x + y * y + z * z );
}

```

Putem apela metoda atât cu un obiect din clasa *Minge* cât și cu un obiect din clasa *Rezervor*.

Compilatorul nu se va plânge pentru că el știe că ambele clase implementează interfața *ObiectSpatioTemporal*, așa că metodele apelate în interiorul calculului distanței (*momentInitial* și *centruDeGreutate*) sunt cu siguranță implementate în ambele clase. Deci, putem scrie:

```

Minge minge;
Rezervor rezervor;
double punct[] = { 10.0, 45.0, 23.0 };
distanța( punct, minge );
distanța( punct, rezervor );

```

Desigur, în mod normal ar fi trebuit să proiectăm și un constructor sau mai mulți care să inițializeze obiectele noastre cu valori rezonabile. Acești constructori ar fi stat cu siguranță în definiția claselor și nu în definiția interfețelor. Nu avem aici nici o cale de a forța definirea unui anumit constructor cu ajutorul interfeței. 🏠

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul VII

Structura programelor

- [8.1 Pachete de clase](#)
- [8.2 Importul claselor](#)
- [8.3 Fișiere sursă](#)
- [8.4 Compilare și execuție](#)

8.1 Pachete de clase

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

$[NumePachet.]^* NumeComponentăPachet$

Numele de pachete și de componente ale acestora sunt identificatori

Java. De obicei, aceste nume urmează structura de directoare în care sunt memorate clasele compilate. Rădăcina arborelui de directoare în care

sunt memorate clasele este indicată de o variabilă sistem CLASSPATH. În DOS aceasta se setează în felul următor:

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

În Unix se poate seta cu comanda:

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

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

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

```
COM.Microsoft.OLE
COM.Apple.quicktime.v2
```

și aºa mai departe. 

8.2 Importul claselor

Desigur, este nevoie ca o clasă să poată folosi obiecte aparținând unei alte clase. Pentru aceasta, definiția clasei respective trebuie să importe codul binar al celeilalte clase pentru a ºti care sunt variabilele și metodele clasei respective.

Importul se face cu o instrucțiune specială:

```
import numeClasă ;
```


unde numele clasei include și pachetul din care aceasta face parte. De exemplu:

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

Se poate importa și un pachet întreg, adică toate clasele aparținând aceluia pachet, printr-o instrucțiune de forma:

```
import numePachet.*;
```

De exemplu:

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

8.3 Fiºiere sursă

Codul sursă Java trebuie introdus cu un editor într-un fiºier text pe care îl vom numi în continuare *fiºier sursă*. Un fiºier sursă poate să conțină declarația mai multor clase și interfețe, dar doar una dintre acestea poate fi declarată publică. Utilizarea celorlalte clase este limitată la fiºierul respectiv. Mai mult, nu putem avea în același timp o interfață publică și o clasă publică declarate în același fiºier sursă.

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

```
package numePachet;
```

dacă această declarație lipseºte, clasa va fi plasată în pachetul implicit, care nu are nume.

Structura generală a unui fișier sursă este următoarea:

[*DeclarațiePachet*] [*InstrucțiuneImport*] * [*DeclarațieDeTip*] *

unde declarația de tip poate fi o declarație de clasă sau de interfață. 

8.4 Compilare și execuție

Fișierele sursă Java au obligatoriu extensia **.java**. Numele lor este identic cu numele clasei sau interfeței publice declarate în interior. În urma compilării rezultă fișiere cu nume identice cu numele claselor dar cu extensia **.class** indiferent dacă este vorba de o clasă sau o interfață. Fișierul **.class** este generat în directorul local și nu direct la locația pachetului.

Compilarea se face cu o comandă de forma:

```
javac FișierSursă .java
```

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

La compilare, variabila sistem CLASSPATH trebuie să fie deja setată pentru că însuși compilatorul Java actual este scris în Java.


Pentru lansarea în execuție a unei aplicații Java, trebuie să introducem comanda:

```
java NumeClasă
```

unde numele clasei este numele aplicației care conține metoda **main**. Interpretorul va căuta un fișier cu numele **NumeClasă.class** și va încerca să instanțieze clasa respectivă.

Pentru lansarea unui applet veți avea nevoie de un document HTML care conține tagul **APPLET** și ca parametru al acesteia

```
name=NumeClasă.class
```

La lansarea unui applet, clasele care sunt apelate de clasa principală sunt mai întâi căutate pe sistemul pe care rulează navigatorul. Dacă nu sunt acolo, ele vor fi transferate în rețea. Asta înseamnă că transferul de cod este relativ mic, trebuie transferat doar codul specific aplicației. 

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul IX

Fire de execuție și sincronizare

- [9.1 Crearea firelor de execuție](#)
- [9.2 Stările unui fir de execuție](#)
- [9.3 Prioritatea firelor de execuție](#)
- [9.4 Grupuri de fire de execuție](#)
- [9.5 Enumerarea firelor de execuție](#)
- [9.6 Sincronizare](#)
- [9.7 Un exemplu](#)
- [9.8 Un exemplu Runnable](#)

O aplicație Java rulează în interiorul unui proces al sistemului de operare. Acest *proces* constă din segmente de cod și segmente de date mapate într-un spațiu virtual de adresare. Fiecare proces deține un număr de resurse alocate de către sistemul de operare, cum ar fi fișiere deschise, regiuni de memorie alocate dinamic, sau fire de execuție. Toate aceste resurse deținute de către un proces sunt eliberate la terminarea procesului de către sistemul de operare.

Un *fir de execuție* este unitatea de execuție a unui proces. Fiecare fir de execuție are asociate o secvență de instrucțiuni, un set de regiștri CPU și o stivă. Atenție, un proces nu execută nici un fel de instrucțiuni. El este de fapt un spațiu de adresare comun pentru unul sau mai multe fire de execuție. Execuția instrucțiunilor cade în responsabilitatea firelor de execuție. În cele ce urmează vom prescurta uneori denumirea firelor de execuție, numindu-le pur și simplu *fire*.

În cazul aplicațiilor Java interpretate, procesul deține în principal codul interpretorului iar codul binar Java este tratat ca o zonă de date de către interpretor. Dar, chiar și în această situație, o aplicație Java poate avea mai multe fire de execuție, create de către interpretor și care execută, seturi distincte de instrucțiuni binare Java.

Fiecare dintre aceste fire de execuție poate rula în paralel pe un procesor separat dacă mașina pe care rulează aplicația este o mașină cu mai multe procesoare. Pe mașinile monoprocesor, senzația de execuție în paralel a firelor de execuție este creată prin rotirea acestora pe rând la controlul unității centrale, câte o cantitate de timp fiecare. Algoritmul de rotire al firelor de execuție este de tip round-robin.

Mediul de execuție Java execută propriul său control asupra firelor de execuție. Algoritmul pentru planificarea firelor de execuție, prioritățile și stările în care se pot afla acestea sunt specifice aplicațiilor Java și implementate identic pe toate platformele pe care a fost portat mediul de execuție Java. Totuși, acest mediu trebuie să profite de resursele sistemului pe care lucrează. Dacă sistemul gazdă lucrează cu mai multe procesoare, Java va folosi toate aceste procesoare pentru a-și planifica firele de execuție. Dacă sistemul oferă multitasking preemptiv, multitaskingul Java va fi de asemenea preemptiv, etc.

În cazul mașinilor multiprocesor, mediul de execuție Java și sistemul de operare sunt responsabile cu repartizarea firelor de execuție pe un procesor sau altul. Pentru programator, acest mecanism este complet transparent, neexistând nici o diferență între scrierea unei aplicații cu mai multe fire pentru o mașină cu un singur procesor sau cu mai multe. Desigur, există însă diferențe în cazul scrierii aplicațiilor pe mai multe fire de execuție față de acelea cu un singur fir de execuție, diferențe care provin în principal din cauza necesității de sincronizare între firele de execuție aparținând aceluiași proces.

Sincronizarea firelor de execuție înseamnă că acestea se așteaptă unul pe celălalt pentru completarea anumitor operații care nu se pot executa în paralel sau care trebuie executate într-o anumită ordine. Java oferă și în acest caz mecanismele sale proprii de sincronizare, extrem de ușor de utilizat și înglobate în chiar sintaxa de bază a limbajului.

La lansarea în execuție a unei aplicații Java este creat automat și un prim fir de execuție, numit firul principal. Acesta poate ulterior să creeze alte fire de execuție care la rândul lor pot crea alte fire, și așa mai departe. Firele de execuție dintr-o aplicație Java pot fi grupate în grupuri pentru a fi manipulate în comun.

În afară de firele normale de execuție, Java oferă și fire de execuție cu prioritate mică care lucrează în fundalul aplicației atunci când nici un alt fir de execuție nu poate fi rulat. Aceste fire de fundal se numesc *demoni* și execută operații costisitoare în timp și independente de celelalte fire de execuție. De exemplu, în Java colectorul de gunoarie lucrează pe un fir de execuție separat, cu proprietăți de demon. În același fel poate fi gândit un fir de execuție care execută operații de încărcare a unor imagini din rețea.

O aplicație Java se termină atunci când se termină toate firele de execuție din interiorul ei sau când nu mai există decât fire demon. Terminarea firului principal de execuție nu duce la terminarea automată a aplicației. 🏠

9.1 Crearea firelor de execuție

Există două căi de definire de noi fire de execuție: derivarea din clasa **Thread** a noi clase și implementarea într-o clasă a interfeței **Runnable**.

În primul caz, noua clasă moștenește toate metodele și variabilele clasei **Thread** care implementează în mod standard, în Java, funcționalitatea de lucru cu fire de execuție. Singurul lucru pe care trebuie să-l facă noua clasă este să reimplementeze metoda **run** care este apelată automat de către mediul de execuție la lansarea unui nou fir. În plus, noua clasă ar putea avea nevoie să implementeze un constructor care să permită atribuirea unei denumiri firului de execuție.

Dacă firul are un nume, acesta poate fi obținut cu metoda **getName** care returnează un obiect de tip **String**.

Iată un exemplu de definire a unui nou tip de fir de execuție:

```
class FirNou extends Thread {
    public FirNou( String nume ) {
        // apelează constructorul din Thread
        super( nume );
    }
    public void run() {
        while( true ) { // fără sfârșit
            System.out.println( getName() +
                " Tastati ^C" );
        }
    }
}
```

Dacă vom crea un nou obiect de tip **FirNou** și îl lansăm în execuție acesta va afișa la infinit mesajul "Tastați ^C". Întreruperea execuției se poate face într-adevăr prin tastarea caracterului ^C, caz în care întreaga aplicație este terminată. Atâta timp însă cât noul obiect nu va fi întrerupt din exterior, aplicația va continua să se execute pentru că

mai există încă fire de execuție active și indiferent de faptul că firul de execuție principal s-a terminat sau nu.

Iată și un exemplu de aplicație care folosește această clasă:

```
public TestFirNou {
    public static void main( String[] ) {
        new FirNou( "Primul" ).start();
    }
}
```

Metoda **start**, predefinită în obiectul **Thread** lansează execuția propriu-zisă a firului. Desigur există și căi de a opri execuția la nesfârșit a firului creat fie prin apelul metodei **stop**, prezentată mai jos, fie prin rescrierea funcției **run** în așa fel încât execuția sa să se termine după un interval finit de timp.

A doua cale de definiție a unui fir de execuție este implementarea interfeței **Runnable** într-o anumită clasă de obiecte. Această cale este cea care trebuie aleasă atunci când clasa pe care o creăm nu se poate deriva din clasa **Thread** pentru că este important să fie derivată din altă clasă. Desigur, moștenirea multiplă ar rezolva această problemă, dar Java nu are moștenire multiplă.

Această nouă cale se poate folosi în modul următor:

```
class Oclasa {
    ...
}
class FirNou extends Oclasa implements Runnable {
    public void run() {
        for( int i = 0; i < 100; i++ ) {
            System.out.println( "pasul " + i );
        }
    }
    ...
}
public class TestFirNou {
    public static void main( String argumente[] ) {
        new Thread( new FirNou() ).start();
        // Obiectele sunt create și folosite imediat
        // La terminarea instrucțiunii, ele sunt automat
        // eliberate nefiind referite de nimic
    }
}
```

După cum observați, clasa **Thread** are și un constructor care primește ca argument o instanță a unei clase care implementează interfața **Runnable**. În acest caz, la lansarea în execuție a noului fir, cu metoda **start**, se apelează metoda **run** din acest obiect și nu din instanța a clasei **Thread**.

Atunci când dorim să creăm un aplet care să ruleze pe un fir de execuție separat față de pagina de navigator în care rulează pentru a putea executa operații în fereastra apletului și în același timp să putem folosi în continuare navigatorul, suntem obligați să alegem cea de-a doua cale de implementare. Aceasta pentru că apletul nostru trebuie să fie derivat din clasa standard **Applet**. Singura alternativă care ne rămâne este aceea de a implementa în aplet interfața

9.2 Stările unui fir de execuție

Un fir de execuție se poate afla în Java în mai multe stări, în funcție de ce se întâmplă cu el la un moment dat.

Atunci când este creat, dar înainte de apelul metodei `start`, firul se găsește într-o stare pe care o vom numi `Fir Nou Creat`. În această stare, singurele metode care se pot apela pentru firul de execuție sunt metodele `start` și `stop`. Metoda `start` lansează firul în execuție prin apelul metodei `run`. Metoda `stop` omoară firul de execuție încă înainte de a fi lansat. Orice altă metodă apelată în această stare provoacă terminarea firului de execuție prin generarea unei excepții de tip `IllegalThreadStateException`.

Dacă apelăm metoda `start` pentru un `Fir Nou Creat` firul de execuție va trece în starea `Rulează`. În această stare, instrucțiunile din corpul metodei `run` se execută una după alta. Execuția poate fi oprită temporar prin apelul metodei `sleep` care primește ca argument un număr de milisecunde care reprezintă intervalul de timp în care firul trebuie să fie oprit. După trecerea intervalului, firul de execuție va porni din nou.

În timpul în care se scurge intervalul specificat de `sleep`, obiectul nu poate fi repornit prin metode obișnuite. Singura cale de a ieși din această stare este aceea de a apela metoda `interrupt`. Această metodă aruncă o excepție de tip `InterruptedException` care nu este prinsă de `sleep` dar care trebuie prinsă obligatoriu de metoda care a apelat metoda `sleep`. De aceea, modul standard în care se apelează metoda `sleep` este următorul:

```
...
try {
    sleep( 1000 ); // o secundă
} catch( InterruptedException ) {
    ...
}
...
```

Dacă dorim oprirea firului de execuție pe timp nedefinit, putem apela metoda `suspend`. Aceasta trece firul de execuție într-o nouă stare, numită `Nu Rulează`. Această stare este folosită și pentru oprirea temporară cu `sleep`. În cazul apelului `suspend` însă, execuția nu va putea fi reluată decât printr-un apel al metodei `resume`. După acest apel, firul va intra din nou în starea `Rulează`.

Pe timpul în care firul de execuție se găsește în starea `Nu Rulează`, acesta nu este planificat niciodată la controlul unității centrale, aceasta fiind cedată celorlalte fire de execuție din aplicație.

Firul de execuție poate intra în starea `Nu Rulează` și din alte motive. De exemplu se poate întâmpla ca firul să aștepte pentru terminarea unei operații de intrare/ieșire de lungă durată caz în care firul va intra din nou în starea `Rulează` doar după terminarea operației.

O altă cale de a ajunge în starea `Nu Rulează` este aceea de a apela o metodă sau o secvență de instrucțiuni sincronizată după un obiect. În acest caz, dacă obiectul este deja blocat, firul de execuție va fi oprit până în clipa în care obiectul cu pricina apelează metoda `notify` sau `notifyAll`.

În fine, atunci când metoda `run` și-a terminat execuția, obiectul intră în starea `Mort`. Această stare este păstrată până în clipa în care obiectul

este eliminat din memorie de mecanismul de colectare a gunoaielor. O altă posibilitate de a intra în starea **Mort** este aceea de a apela metoda **stop**.

Atunci când se apelează metoda **stop**, aceasta aruncă cu o instrucțiune **throw** o eroare numită **ThreadDeath**. Aceasta poate fi prinsă de către cod pentru a efectua curățenia necesară. Codul necesar este următorul:

```
...
try {
    firDeExecutie.start();
...
} catch( ThreadDeath td ) {
... // curățenie
    throw td; // se aruncă obiectul mai departe
    // pentru a servi la distrugerea
    // firului de execuție
}
```

Desigur, firul de execuție poate fi terminat și pe alte căi, caz în care metoda **stop** nu este apelată și eroarea **ThreadDeath** nu este aruncată. În aceste situații este preferabil să ne folosim de o clauză **finally** ca în:

```
...
try {
    firDeExecutie.start();
...
} finally {
    ..// curățenie
}
```

În fine, dacă nu se mai poate face nimic pentru că firul de execuție nu mai răspunde la comenzi, puteți apela la calea disperată a metodei **destroy**. Din păcate, metoda **destroy** termină firul de execuție fără a proceda la curățirile necesare în memorie.

Atunci când un fir de execuție este oprit cu comanda **stop**, mai este nevoie de un timp până când sistemul efectuează toate operațiile necesare opririi. Din această cauză, este preferabil să așteptăm în mod explicit terminarea firului prin apelul metodei **join**:

```
    firDeExecutie.stop()
    try {
        firDeExecutie.join();
    } catch( InterruptedException e ) {
...
    }
```

Excepția de întrerupere trebuie prinsă obligatoriu. Dacă nu apelăm metoda **join** pentru a aștepta terminarea și metoda **stop** este de exemplu apelată pe ultima linie a funcției **main**, există șansa ca sistemul să creadă că firul auxiliar de execuție este încă în viață și aplicația Java să nu se mai termine rămânând într-o stare de așteptare. O puteți desigur termina tastând ^C. 🚪

9.3 Prioritatea firelor de execuție

Fiecare fir de execuție are o prioritate cuprinsă între valorile `MIN_PRIORITY` și `MAX_PRIORITY`. Aceste două variabile finale sunt declarate în clasa **Thread**. În mod normal însă, un fir de execuție are prioritatea `NORM_PRIORITY`, de asemenea definită în clasa **Thread**.

Mediul de execuție Java planifică firele de execuție la controlul unității centrale în funcție de prioritatea lor. Dacă există mai multe fire cu prioritate maximă, acestea sunt planificate după un algoritm round-robin. Firele de prioritate mai mică intră în calcul doar atunci când toate firele de prioritate mare sunt în starea `Nu Rulează`.

Prioritatea unui fir de execuție se poate interoga cu metoda **getPriority** care întoarce un număr întreg care reprezintă prioritatea curentă a firului de execuție. Pentru a seta prioritatea, se folosește metoda **setPriority** care primește ca parametru un număr întreg care reprezintă prioritatea dorită.

Schimbarea priorității unui fir de execuție este o treabă periculoasă dacă metoda cu prioritate mare nu se termină foarte repede sau dacă nu are opriri dese. În caz contrar, celelalte metode nu vor mai putea primi controlul unității centrale.

Există însă situații în care putem schimba această prioritate fără pericol, de exemplu când avem un fir de execuție care nu face altceva decât să citească caractere de la utilizator și să le memoreze într-o zonă temporară. În acest caz, firul de execuție este în cea mai mare parte a timpului în starea `Nu Rulează` din cauză că așteaptă terminarea unei operații de intrare/ieșire. În clipa în care utilizatorul tastează un caracter, firul va ieși din starea de așteptare și va fi primul planificat la execuție din cauza priorității sale ridicate. În acest fel utilizatorul are senzația că aplicația răspunde foarte repede la comenzile sale.

În alte situații, avem de executat o sarcină cu prioritate mică. În aceste cazuri, putem seta pentru firul de execuție care execută aceste sarcini o prioritate redusă.

Alternativ, putem defini firul respectiv de execuție ca un demon. Dezavantajul în această situație este faptul că aplicația va fi terminată atunci când există doar demoni în lucru și există posibilitatea pierderii de date. Pentru a declara un fir de execuție ca demon, putem apela metoda **setDaemon**. Această metodă primește ca parametru o valoare booleană care dacă este **true** firul este făcut demon și dacă nu este adus înapoi la starea normală. Putem testa faptul că un fir de execuție este demon sau nu cu metoda **isDaemon**. 🏠

9.4 Grupuri de fire de execuție

Uneori avem nevoie să acționăm asupra mai multor fire de execuție deodată, pentru a le suspenda, reporni sau modifica prioritatea în bloc. Din acest motiv, este util să putem grupa firele de execuție pe grupuri. Această funcționalitate este oferită în Java de către o clasă numită **ThreadGroup**.

La pornirea unei aplicații Java, se creează automat un prim grup de fire de execuție, numit grupul principal, *main*. Firul principal de execuție

face parte din acest grup. În continuare, ori de câte ori creăm un nou fir de execuție, acesta va face parte din același grup de fire de execuție ca și firul de execuție din interiorul căruia a fost creat, în afară de cazurile în care în constructorul firului specificăm explicit altceva.

Într-un grup de fire de execuție putem defini nu numai fire dar și alte grupuri de execuție. Se creează astfel o arborescență a cărei rădăcină este grupul principal de fire de execuție.

Pentru a specifica pentru un fir un nou grup de fire de execuție, putem apela constructorii obișnuiți dar introducând un prim parametru suplimentar de tip **ThreadGroup**. De exemplu, putem folosi următorul cod:

```
ThreadGroup tg = new ThreadGroup( "Noul grup" );
Thread t = new Thread( tg, "Firul de execuție" );
```

Acest nou fir de execuție va face parte dintr-un alt grup de fire decât firul principal. Putem afla grupul de fire de execuție din care face parte un anumit fir apelând metoda **getThreadGroup**, ca în secvența:

```
Thread t = new Thread( "Firul de Execuție" );
ThreadGroup tg = t.getThreadGroup();
```

Operațiile definite pentru un grup de fire de execuție sunt clasificabile în operații care acționează la nivelul grupului, cum ar fi aflarea numelui, setarea unei priorități maxime, etc., și operații care acționează asupra fiecărui fir de execuție din grup, cum ar fi **stop**, **suspend** sau **resume**. Unele dintre aceste operații necesită aprobarea controlorilor de securitate acest lucru făcându-se printr-o metodă numită **checkAccess**. De exemplu, nu puteți seta prioritatea unui fir de execuție decât dacă aveți drepturile de acces necesare. 🏠

9.5 Enumerarea firelor de execuție

Pentru a enumera firele de execuție active la un moment dat, putem folosi metoda **enumerate** definită în clasa **Thread** precum și în clasa **ThreadGroup**. Această metodă primește ca parametru o referință către un tablou de referințe la obiecte de tip **Thread** pe care îl umple cu referințe către fiecare fir activ în grupul specificat.

Pentru a afla câte fire active sunt în grupul respectiv la un moment dat, putem apela metoda **activeCount** din clasa **ThreadGroup**. De exemplu:

```
public listeazaFire {
    ThreadGroup grup = Thread.currentThread().getThreadGroup();
    int numarFire = grup.activeCount();
    Thread fire[] = new Thread[numarFire];
    grup.enumerate( fire );
    for( int i = 0; i < numar; i++ ) {
        System.out.println( fire[i].toString() );
    }
}
```

Metoda **enumerate** întoarce numărul de fire memorate în tablou, care este identic cu numărul de fire active. 🏠

9.6 Sincronizare

În unele situații se poate întâmpla ca mai multe fire de execuție să vrea să acceseze aceeași variabilă. În astfel de situații, se pot produce încurcături dacă în timpul unuia dintre accese un alt fir de execuție modifică valoarea variabilei.

Limbajul Java oferă în mod nativ suport pentru protejarea acestor variabile. Suportul este construit de fapt cu granulație mai mare decât o singură variabilă, protecția făcându-se la nivelul obiectelor. Putem defini metode, în cadrul claselor, care sunt sincronizate.

Pe o instanță de clasă, la un moment dat, poate lucra o singură metodă sincronizată. Dacă un alt fir de execuție încearcă să apeleze aceeași metodă pe aceeași instanță sau o altă metodă a clasei de asemenea declarată sincronizată, acest al doilea apel va trebui să aștepte înainte de execuție eliberarea instanței de către cealaltă metodă.

În afară de sincronizarea metodelor, se pot sincroniza și doar blocuri de instrucțiuni. Aceste sincronizări se fac tot în legătură cu o anumită instanță a unei clase. Aceste blocuri de instrucțiuni sincronizate se pot executa doar când instanța este liberă. Se poate întâmpla ca cele două tipuri de sincronizări să se amestece, în sensul că obiectul poate fi blocat de un bloc de instrucțiuni și toate metodele sincronizate să aștepte, sau invers.

Declararea unui bloc de instrucțiuni sincronizate se face prin:

```
synchronize ( Instanță ) {  
  
    Instrucțiuni  
  
}
```

iar declararea unei metode sincronizate se face prin folosirea modificatorului **synchronize** la implementarea metodei. 🏠

9.7 Un exemplu

Exemplul următor implementează soluția următoare probleme: Într-o țară foarte îndepărtată trăiau trei înțelepți filozofi. Acești trei înțelepți își pierdeau o mare parte din energie certându-se între ei pentru a afla care este cel mai înțelept. Pentru a tranșa problema o dată pentru totdeauna, cei trei înțelepți au pornit la drum către un al patrulea înțelept pe care cu toții îl recunoșteau că ar fi mai bun decât ei.

Când au ajuns la acesta, cei trei i-au cerut să le spună care dintre ei este cel mai înțelept. Acesta, a scos cinci pălării, trei negre și două albe, și li le-a arătat explicându-le că îi va lega la ochi și le va pune în cap câte o pălărie, cele două rămase ascunzându-le. După aceea, le va dezlega ochii, și fiecare dintre ei va vedea culoarea pălăriei celorlalți dar nu și-o va putea vedea pe a sa. Cel care își va da primul seama ce culoare are propria pălărie, acela va fi cel mai înțelept.

După explicație, înțeleptul i-a legat la ochi, le-a pus la fiecare câte o pălărie neagră și le-a ascuns pe celelalte două. Problema este aceea de a descoperi care a fost raționamentul celui care a ghicit primul că pălăria lui este neagră.

Programul următor rezolvă problema dată în felul următor: Fiecare înțelept privește pălăriile celorlalți doi. Dacă ambele sunt albe, problema este rezolvată, a lui nu poate fi decât neagră. Dacă vede o pălărie albă și una neagră, atunci el va trebui să aștepte puțin să vadă ce spune cel cu pălăria neagră. Dacă acesta nu găsește soluția, înseamnă că el nu vede două pălării albe, altfel ar fi găsit imediat răspunsul. După un scurt timp de așteptare, înțeleptul poate să fie sigur că pălăria lui este neagră.

În fine, dacă ambele pălării pe care le vede sunt negre, va trebui să aștepte un timp ceva mai lung pentru a vedea dacă unul dintre concurenții săi nu ghicește pălăria. Dacă după scurgerea timpului nici unul nu spune nimic, înseamnă că

nici unul nu vede o pălărie albă și una neagră. Înseamnă că propria pălărie este neagră.

Desigur, raționamentul pleacă de la ideea că ne putem baza pe faptul că toți înțelepții gândesc și pot rezolva probleme ușoare. Cel care câștigă a gândit doar un pic mai repede. Putem simula viteza de gândire cu un interval aleator de așteptare până la luarea deciziilor. În realitate, intervalul nu este aleator ci dictat de viteza de gândire a fiecărui înțelept.

Cei trei înțelepți sunt implementați identic sub formă de fire de execuție. Nu câștigă la fiecare rulare același din cauza caracterului aleator al implementării. Înțeleptul cel mare este firul de execuție principal care controlează activitatea celorlalte fire și le servește cu date, culoarea pălăriilor, doar în măsura în care aceste date trebuie să fie accesibile. Adică nu se poate cere propria culoare de pălărie.

Culoarea inițială a pălăriilor se poate rescrie din linia de comandă.

```
import java.awt.Color;
// clasa Filozof implementează comportamentul
// unui concurent
class Filozof extends Thread {
// părerea concurentului despre culoarea
// pălăriei sale. Null dacă încă nu și-a
// format o părere.
Color parere = null;
Filozof( String nume ) {
super( nume );
}
public void run() {
// concurentii firului curent
Filozof concurenti[] = new Filozof[2];
// temporar
Thread fire[] = new Thread[10];
int numarFire = enumerate( fire );
for( int i = 0, j = 0; i < numarFire && j < 2; i++ ) {
if( fire[i] instanceof Filozof &&
fire[i] != this ) {
concurenti[j++] = (Filozof)fire[i];
}
}
while( true ) {
Color primaCuloare = Concurrency.culoare( this,
concurenti[0] );
Color adouaCuloare =
Concurrency.culoare( this, concurenti[1] );
if( primaCuloare == Color.white &&
adouaCuloare == Color.white ) {
synchronized( this ) {
parere = Color.black;
}
} else if( primaCuloare == Color.white ){
try{
sleep( 500 );
```

```

    } catch( InterruptedException e ){
    };
    if( Conkurs.culoare( this, concurenti[1]) != concurenti[1].aGhicit() ) {
    synchronized( this ) {
    parere = Color.black;
    };
    }
    } else if( adouaCuloare == Color.white ) {
    try{
        sleep( (int)( Math.random()*500));
    } catch( InterruptedException e ) {
    };
    if( Conkurs.culoare(this, concurenti[0] ) != concurenti[0].aGhicit() ) {
    synchronized( this ) {
    parere = Color.black;
    };
    }
    } else {
    try {
    sleep( (int)( Math.random()*500)+500 );
    } catch( InterruptedException e ) {
    };
    if( Conkurs.culoare(this, concurenti[0]) != concurenti[0].aGhicit() &&
        Konkurs.culoare( this,
        concurenti[1] ) !=
        concurenti[1].aGhicit() ) {
    synchronized( this ) {
    parere = Color.black;
    };
    }
    }
    }
    }

    public synchronized Color aGhicit() {
    return parere;
    }
    }

    public class Konkurs {
    private static Color palarii[] = {
    Color.black, Color.black, Color.black
    };
    private static Filozof filozofi[] = new Filozof[3];
    public static void main( String args[] ) {
    for( int i = 0; i < args.length && i < 3; i++ ) {
    if( args[i].equalsIgnoreCase( "alb" ) ) {
    palarii[i] = Color.white;
    } else if(args[i].equalsIgnoreCase("negru")) {
    palarii[i] = Color.black;
    }
    }
    }
    }

```

```

for( int i = 0; i < 3; i++ ) {
    filozofi[i] = new Filozof( "Filozoful " +
        ( i + 1 ) );
}
for( int i = 0; i < 3; i++ ) {
    filozofi[i].start();
}
System.out.println( "Concurenti:" );
for( int i = 0; i < 3; i++ ) {
    System.out.println( "\t" +
        filozofi[i].getName() + " "
        + ( ( palarii[i] == Color.white ) ?
            "alb":"negru" ) );
}
gata:
while( true ) {
    for( int i = 0; i < 3; i++ ) {
        if( filozofi[i].aGhicit()==palarii[i] ) {
            System.out.println(
                filozofi[i].getName() +
                " a ghicit." );
            break gata;
        }
    }
}
for( int i = 0; i < 3; i++ ) {
    filozofi[i].stop();
    try {
        filozofi[i].join();
    } catch( InterruptedException e ) {};
}
public static Color culoare( Filozof filozof,
    Filozof concurent ) {
    if( filozof != concurent ) {
        for( int i = 0; i < 3; i++ ) {
            if( filozofi[i] == concurent ) {
                return palarii[i];
            }
        }
    }
    return null;
}

```

9.8 Un exemplu Runnable

Exemplul următor implementează problema celor 8 dame, și anume: găsește toate posibilitățile de a așeza pe o tablă de 8x8 regine în așa fel încât acestea să nu se bată între ele. Reginele se bat pe linie, coloană sau în diagonală.

Solupia de față extinde problema la o tablă de $N \times N$ căsuțe și la N regine. Parametrul N este citit din tagul HTML asociat apletului.

```
import java.awt.*;
import java.applet.Applet;
public
class QueensRunner extends Applet implements Runnable {
    int n;
    int regine[];
    int linie;
    Image queenImage;
    Thread myThread;
    public void start() {
if( myThread == null ) {
myThread = new Thread( this, "Queens" );
myThread.start();
}
}
    public void stop() {
myThread.stop();
myThread = null;
}
    public void run() {
while( myThread != null ) {
    nextSolution();
    repaint();
    try {
myThread.sleep( 1000 );
    } catch ( InterruptedException e ){
    }
    }
}
    boolean isGood() {
for( int i = 0; i < linie; i++ ) {
if( regine[linie] == regine[i] ||
Math.abs( regine[i] -
regine[linie] ) == Math.abs( i - linie ) ) {
return false;
}
}
return true;
}

    void nextSolution() {
while( true ) {
if( linie < 0 ) {
linie = 0;
}
}
```

```

regine[linie]++;
if( regine[linie] > n ) {
regine[linie] = 0;
linie--;
} else {
if( isGood() ) {
linie++;
if( linie >= n ) {
break;
}
}
}
}
}
}
}
}

```

```

public void init() {
String param = getParameter( "Dimension" );
if( param == null ) {
n = 4;
} else {
try {
n = Integer.parseInt( param );
} catch( NumberFormatException e ) {
n = 4;
}
if( n < 4 ) {
n = 4;
}
}
regine = new int[n + 1];
for( int i = 0; i < n; i++ ) {
regine[i] = 0;
}
linie = 0;
queenImage = getImage(getCodeBase(), "queen.gif" );
}
public void paint( Graphics g ) {
Dimension d = size();
g.setColor( Color.red );
int xoff = d.width / n;
int yoff = d.height / n;
for( int i = 1; i < n; i++ ) {
g.drawLine( xoff * i, 0, xoff * i, d.height );
g.drawLine( 0, yoff * i, d.width, yoff * i );
}
for( int i = 0; i < n; i++ ) {
for( int j = 0; j < n; j++ ) {
if( regine[i] - 1 == j ) {
g.drawImage(queenImage,

```

```
i*xoff + 1, j*yoff + 1, this);  
    }  
    }  
    }  
    }  
    public String getAppletInfo() {  
        return "Queens by E. Rotariu";  
    }  
}
```

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Bibliografie

1. Dumitru Rădoiu,
HTML - Publicații Web, *Computer Press Agora SRL*
2. James Gostling, Henry McGilton,
The Java Language Environment, A white paper, *Sun Microsystems, Inc.*
3. David Flanagan,
Java in a Nutshell: A Desktop Quick Reference for Java Programmers,
O'Reilly & Associates, Inc.
4. Ed Anuff,
The Java Sourcebook, *Wiley Computer Publishing*
5. Mike Daconta,
Java for C/C++ Programmers, *Wiley Computer Publishing*
6. Arthur van Hoff, Sami Shaio, Orca Starbuck,
Hooked on Java, *Addison-Wesley Publishing Company*
7. Laura Lemay, Charles L. Perkins,
Teach your self Java in 21 days, *Sams.net Publishing*
8. *** - **The Java Language Specification**, *Sun Microsystems, Inc.*

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul V

5.1 Variabile

[5.1.1](#) *Declarații de variabile*

[5.1.1.1](#) *Tipul unei variabile*

[5.1.1.2](#) *Numele variabilelor*

[5.1.1.3](#) *Inițializarea variabilelor*

[5.1.2](#) *Tipuri primitive*

[5.1.2.1](#) *Tipul boolean*

[5.1.2.2](#) *Tipul caracter*

[5.1.2.3](#) *Tipuri întregi*

[5.1.2.3.1](#) *Tipul octet*

[5.1.2.3.2](#) *Tipul întreg scurt*

[5.1.2.3.3](#) *Tipul întreg*

[5.1.2.3.4](#) *Tipul întreg lung*

[5.1.2.4](#) *Tipuri flotante*

[5.1.2.4.1](#) *Tipul flotant*

[5.1.2.4.2](#) *Tipul flotant dublu*

[5.1.2.4.3](#) *Reali speciali definiți de IEEE*

[5.1.3](#) *Tipuri referință*

[5.1.3.1](#) *Tipul referință către o clasă*

[5.1.3.2](#) *Tipul referință către o interfață*

[5.1.3.3](#) *Tipul referință către un tablou*

[5.1.4](#) *Clasa de memorare*

[5.1.4.1](#) *Variabile locale*

[5.1.4.2](#) *Variabile statice*

[5.1.4.3](#) *Variabile dinamice*

[5.1.5](#) *Tablouri de variabile*

[5.1.5.1](#) *Declarația variabilelor de tip tablou*

[5.1.5.2](#) *Inițializarea tablourilor.*

[5.1.5.3](#) *Lungimea tablourilor*

[5.1.5.4](#) *Referirea elementelor din tablou*

[5.1.5.5](#) *Alocarea și eliberarea tablourilor*

[5.1.6](#) *Conversii*

[5.1.6.1](#) *Conversii de extindere a valorii*

[5.1.6.2](#) *Conversii de trunchiere a valorii*

[5.1.6.3](#) *Conversii pe tipuri referință*

[5.1.6.4](#) *Conversii la operația de atribuire*

[5.1.6.5](#) *Conversii explicite*


[5.1.6.6](#) *Conversii de promovare aritmetică*

5.1.1 Declarații de variabile

O *variabilă* în limbajul Java este o locație de memorie care poate păstra o valoare de un anumit tip. În ciuda denumirii, există variabile care își pot modifica valoarea și variabile care nu și-o pot modifica, numite în Java *variabile finale*. Orice variabilă trebuie să fie declarată pentru a putea fi folosită. Această declarație trebuie să conțină un tip de valori care pot fi memorate în locația rezervată variabilei și un nume pentru variabila declarată. În funcție de locul în sursa programului în care a fost declarată

variabila, aceasta primește o clasă de memorare locală sau statică. Această clasă de memorare definește intervalul de existență al variabilei în timpul execuției.

În forma cea mai simplă, declarația unei variabile arată în felul următor:

```
Tip NumeVariabilă [, NumeVariabilă]*; 
```

5.1.1.1 Tipul unei variabile

Tipul unei variabile poate fi fie unul dintre tipurile primitive definite de limbajul Java fie o referință. Creatorii limbajului Java au avut grijă să

definească foarte exact care sunt caracteristicile fiecărui tip primitiv în parte și care este setul de valori care se poate memora în variabilele care au tipuri primitive. În plus, a fost exact definită și modalitatea de reprezentare a acestor tipuri primitive în memorie. În acest fel, variabilele Java devin independente de

platforma hardware și software pe care lucrează.

În același spirit, Java definește o valoare implicită pentru fiecare tip de dată, în cazul în care aceasta nu a primit nici o valoare de la utilizator. În acest fel, știm întotdeauna care este valoarea cu care o variabilă intră în calcul. Este o practică bună însă aceea ca programele să nu depindă niciodată de aceste inițializări implicite. 🏠

5.1.1.2 Numele variabilelor

Numele variabilei poate fi orice identificator Java. Convenția nescrisă de formare a numelor variabilelor este aceea că orice variabilă care nu este finală are un nume care începe cu literă minusculă în timp ce

variabilele finale au nume care conțin numai majuscule. Dacă numele unei variabile care nu este finală conține mai multe cuvinte, cuvintele începând cu cel de-al doilea se scriu cu litere minuscule dar cu prima literă majusculă. Exemple de nume de variabile care nu sunt finale ar putea fi:

```
culoarea numaruluiDePași următorulElement
```

Variabilele finale ar putea avea nume precum:

```
PORTOCALIUVERDEALBASTRUDESCHIS 🏠
```

5.1.1.3 Inițializarea variabilelor

Limbajul Java permite inițializarea valorilor variabilelor chiar în momentul declarării acestora. Sintaxa este următoarea:

```
Tip NumeVariabilă = ValoareInițială;
```

Desigur, valoarea inițială trebuie să fie de același tip cu tipul variabilei sau să poată fi convertită într-o valoare de acest tip.

Deși limbajul Java ne asigură că toate variabilele au o valoare inițială bine precizată, este preferabil să executăm această inițializare în mod explicit pentru fiecare declarație. În acest fel mărim claritatea propriului cod.

Regula ar fi deci următoarea: nici o declarație fără inițializare. 🏠

5.1.2 Tipuri primitive

5.1.2.1 Tipul boolean

Tipul boolean este folosit pentru memorarea unei valori de adevăr. Pentru acest scop, sunt suficiente doar două valori: adevărat și fals. În Java aceste două valori le vom nota prin literalii **true** și respectiv **false**.

Aceste valori pot fi reprezentate în memorie folosindu-ne de o singură cifră binară, adică pe un bit.

Valorile booleene sunt foarte importante în limbajul Java pentru că ele sunt valorile care se folosesc în condițiile care controlează instrucțiunile repetitive sau cele condiționale. Pentru a exprima o condiție este suficient să scriem o expresie al cărui rezultat este o valoare booleană, adevărat sau fals.

Valorile de tip boolean nu se pot transforma în valori de alt tip în mod nativ. La fel, nu există transformare nativă dinspre celelalte valori înspre tipul boolean. Cu alte cuvinte, având o variabilă de tip boolean nu putem memora în interiorul acesteia o valoare întreagă pentru că limbajul Java nu face pentru noi nici un fel de presupunere legată de ce înseamnă o anumită valoare întreagă din punctul de vedere al valorii de adevăr. La fel, dacă avem o variabilă întreagă, nu îi putem atribui o valoare de tip boolean.

Orice variabilă booleană nou creată primește automat valoarea implicită **false**. Putem modifica această comportare specificând în mod explicit o valoare inițială **true** după modelul pe care îl vom descrie mai târziu.

Pentru a declara o variabilă de tip boolean, în Java vom folosi cuvântul rezervat **boolean** ca în exemplele de mai jos:

```
boolean terminat;  
boolean areDreptate;
```

Rândurile de mai sus reprezintă declarația a două variabile de tip boolean numite `terminat` respectiv `areDreptate`. Cele două variabile au, după declarație, valoarea **false**. Adică nu e terminat dar nici n-are dreptate. 🏠

5.1.2.2 Tipul caracter

Orice limbaj de programare ne oferă într-un fel sau altul posibilitatea de a lucra cu caractere grafice care să reprezinte litere, cifre, semne de punctuație, etc. În cazul limbajului Java acest lucru se poate face folosind tipul primitiv numit tip caracter.

O variabilă de tip caracter poate avea ca valoare coduri Unicode reprezentate pe 16 biți, adică doi octeți. Codurile reprezentabile astfel sunt foarte multe, putând acoperi caracterele de bază din toate limbile scrise existente.

În Java putem combina mai multe caractere pentru a forma cuvinte sau șiruri de caractere mai lungi.

Totuși, trebuie să precizăm că aceste ıruri de caractere nu trebuiesc confundate cu tablourile de caractere pentru că ele conțin în plus informații legate de lungimea ırului.

Codul nu este altceva decât o corespondență între numere și caractere fapt care permite conversii între variabile întregi și caractere în ambele sensuri. O parte din aceste transformări pot să altereze valoarea originală din cauza dimensiunilor diferite ale zonelor în care sunt memorate cele două tipuri de valori. Convertirea caracterelor în numere și invers poate să fie utilă la prelucrarea în bloc a caracterelor, cum ar fi trecerea tuturor literelor minuscule în majuscule și invers.

Atunci când declarăm un caracter fără să specificăm o valoare inițială, el va primi automat ca valoare implicită caracterul null al codului Unicode, \u0000?.

Pentru a declara o variabilă de tip caracter folosim cuvântul rezervat **char** ca în exemplele următoare:

```
char primaLiteră;
char prima, ultima;
```

În cele două linii de cod am declarat trei variabile de tip caracter care au fost automat inițializate cu caracterul null. În continuare, vom folosi interschimbabil denumirea de tip caracter cu denumirea de tip char, care are avantajul că este mai aproape de declarațiile Java. 🏠

5.1.2.3 Tipuri întregi

5.1.2.3.1 Tipul octet

Între tipurile întregi, acest tip ocupă un singur octet de memorie, adică opt cifre binare. Într-o variabilă de tip octet sunt reprezentate întotdeauna valori cu semn, ca de altfel în toate variabilele de tip întreg definite în limbajul Java. Această convenție simplifică schema de tipuri primitive care, în cazul altor limbaje include separat tipuri întregi cu semn și fără.

Fiind vorba de numere cu semn, este nevoie de o convenție de reprezentare a semnului. Convenția folosită de Java este reprezentarea în complement față de doi. Această reprezentare este de altfel folosită de majoritatea limbajelor actuale și permite memorarea, pe 8 biți a 256 de numere începând de la -128 până la 127. Dacă aveți nevoie de numere mai mari în valoare absolută, apălați la alte tipuri întregi.

Valoarea implicită pentru o variabilă neinițializată de tip octet este valoarea 0 reprezentată pe un octet.

Iată și câteva exemple de declarații care folosesc cuvântul Java rezervat **byte**:

```
byte octet;
byte eleviPeClasa;
```

În continuare vom folosi interschimbabil denumirea de tip octet cu cea de tip byte. 🏠

5.1.2.3.2 Tipul întreg scurt

Tipul întreg scurt este similar cu tipul octet dar valorile sunt reprezentate pe doi octeți, adică 16 biți. La fel ca ȳi la tipul octet, valorile

sunt ȳntotdeauna cu semn ȳi se folosește reprezentarea ȳn complement fașă de doi. Valorile de ȳntregi scurȳi reprezentabile sunt de la -32768 la 32767 iar valoarea implicită este 0 reprezentat pe doi octeți.

Pentru declararea variabilelor de tip ȳntreg scurt ȳn Java se folosește cuvântul rezervat **short**, ca ȳn exemplele următoare:

```
short i, j;
short valoareNuPreaMare;
```

În continuare vom folosi interschimbabil denumirea de tip ȳntreg scurt ȳi cea de tip short. 🏠

5.1.2.3.3 Tipul ȳntreg

Singura diferenșă dintre tipul ȳntreg ȳi tipurile precedente este faptul că valorile sunt reprezentate pe patru octeți adică 32 biți. Valorile reprezentabile sunt de la -2147483648 la 2147483647 valoarea implicită fiind 0. Cuvântul rezervat este **int** ca ȳn:

```
int salariu;
```

În continuare vom folosi interschimbabil denumirea de tip ȳntreg ȳi cea de tip int. 🏠


5.1.2.3.4 Tipul ȳntreg lung

ȳn fine, pentru cei care vor șă reprezinte numerele ȳntregi cu semn pe 8 octeți, 64 de biți, există tipul ȳntreg lung. Valorile reprezentabile sunt de la -9223372036854775808 la 9223372036854775807 iar valoarea implicită este 0L.

Pentru cei care nu au calculatoare care lucrează pe 64 de biți este bine de precizat faptul că folosirea acestui tip duce la operaȳii lente pentru că nu există operaȳii native ale procesorului care șă lucreze cu numere așă de mari.

Declarașia se face cu cuvântul rezervat **long**. În continuare vom folosi interschimbabil denumirea de tip ȳntreg lung cu cea de tip long. 🏠

5.1.2.4 Tipuri flotante

Acest tip este folosit pentru reprezentarea numerelor reale sub formă de exponent și cifre semnificative. Reprezentarea se face pe patru octeți, 32 biți, așa cum specifică standardul IEEE 754. 

5.1.2.4.1 Tipul flotant

Valorile finite reprezentabile într-o variabilă de tip flotant sunt de forma:

$$sm2e$$

unde s este semnul $+1$ sau -1 , m este partea care specifică cifrele reprezentative ale numărului, numită și mantisă, un întreg pozitiv mai mic decât 2^{24} iar e este un exponent întreg între -149 și 104 .

Valoarea implicită pentru variabilele flotante este $0.0f$. Pentru declararea unui număr flotant, Java definește cuvântul rezervat **float**. Declarațiile se fac ca în exemplele următoare:

```
float procent;
float noi, ei;
```

În continuare vom folosi interschimbabil denumirea de tip flotant și cea de tip float. 

5.1.2.4.2 Tipul flotant dublu

Dacă valorile reprezentabile în variabile flotante nu sunt destul de precise sau destul de mari, puteți folosi tipul flotant dublu care folosește opt octeți pentru reprezentare, urmând același standard IEEE 754

Valorile finite reprezentabile cu flotați dubli sunt de forma:

$$sm2e$$

unde s este semnul $+1$ sau -1 , m este mantisa, un întreg pozitiv mai mic decât 2^{53} iar e este un exponent întreg între -1045 și 1000 . Valoarea implicită în acest caz este $0.0d$.

Pentru a declara flotați dubli, Java definește cuvântul rezervat **double** ca în:

```
double distanțaPânăLaLună;
```

În continuare vom folosi interschimbabil denumirea de tip flotant dublu și cea de tip double.

În afară de valorile definite până acum, standardul IEEE definește câteva valori speciale reprezentabile

pe un flotant sau un flotant dublu. 🏠

5.1.2.4.3 Reali speciali definiți de IEEE

Prima dintre acestea este NaN (Not a Number), valoare care se obține atunci când efectuăm o operație a cărei rezultat nu este definit, de exemplu $0.0 / 0.0$.

În plus, standardul IEEE definește două valori pe care le putem folosi pe post de infinit pozitiv și negativ. Și aceste valori pot rezulta în urma unor calcule.

Aceste valori sunt definite sub formă de constante și în ierarhia standard Java, mai precis în clasa `java.lang.Float` și respectiv în `java.lang.Double`. Numele constantelor este `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN`.

În plus, pentru tipurile întregi și întregi lungi și pentru tipurile flotante există definite clase în ierarhia standard Java care se numesc respectiv `java.lang.Integer`, `java.lang.Long`, `java.lang.Float` și `java.lang.Double`. În fiecare dintre aceste clase numerice sunt definite două constante care reprezintă valorile minime și maxime care se pot reprezenta în tipurile respective. Aceste două constante se numesc în mod uniform `MIN_VALUE` și `MAX_VALUE`. 🏠

5.1.3 Tipuri referință

Tipurile referință sunt folosite pentru a referi un obiect din interiorul unui alt obiect. În acest mod putem înlănțui informațiile aflate în memorie.

Tipurile referință au, la fel ca și toate celelalte tipuri o valoare implicită care este atribuită automat oricărei variabile de tip referință care nu a fost inițializată. Această valoare implicită este definită de către limbajul Java prin cuvântul rezervat **`null`**.

Puteți înțelege semnificația referinței nule ca o referință care nu trimite nicăieri, a cărei destinație nu a fost încă fixată.

Simpla declarație a unei referințe nu duce automat la rezervarea spațiului de memorie pentru obiectul referit. Singura rezervare care se face este aceea a spațiului necesar memorării referinței în sine. Rezervarea obiectului trebuie făcută explicit în program printr-o expresie de alocare care folosește cuvântul rezervat **`new`**.

O variabilă de tip referință nu trebuie să trimită pe tot timpul existenței sale către același obiect în memorie. Cu alte cuvinte, variabila își poate schimba locația referită în timpul execuției. 🏠

5.1.3.1 Tipul referință către o clasă

Tipul *referință către o clasă* este un tip referință care trimite către o instanță a unei clasei de obiecte. Clasa instanței referite poate fi oricare clasă validă definită de limbaj sau de utilizator.

Clasa de obiecte care pot fi referite de o anumită variabilă de tip referință la clasă trebuie declarată explicit. De exemplu, pentru a declara o referință către o instanță a clasei `Minge`, trebuie să folosim următoarea sintaxă:

```
Minge mingeaMea;
```

Din acest moment, variabila referință de clasă numită *mingeaMea* va putea păstra doar referințe către obiecte de tip `Minge` sau către obiecte aparținând unor clase derivate din clasa `Minge`. De exemplu, dacă avem o altă clasă, derivată din `Minge`, numită `MingeDeBaschet`, putem memora în referința *mingeaMea* și o trimitere către o instanță a clasei `MingeDeBaschet`.

În mod general însă, nu se pot păstra în variabila *mingeaMea* referințe către alte clase de obiecte. Dacă se încercă acest lucru, eroarea va fi semnalată chiar în momentul compilării, atunci când sursa programului este examinată pentru a fi transformată în instrucțiuni ale mașinii virtuale Java.

Să mai observăm că o referință către clasa de obiecte **Object**, rădăcina ierarhiei de clase Java, poate păstra și o referință către un tablou. Mai multe lămuriri asupra acestei afirmații mai târziu. 🏠

5.1.3.2 Tipul referință către o interfață

Tipul *referință către o interfață* permite păstrarea unor referințe către obiecte care respectă o anumită interfață. Clasa obiectelor referite poate fi oricare, atâta timp cât clasa respectivă implementează interfața cerută.

Declarația se face cu următoarea sintaxă:

```
ObiectSpazioTemporal mingeaLuiVasile;
```

În care tipul este chiar numele interfeței cerute. Dacă clasa de obiecte `Minge` declară că implementează această interfață, atunci variabila referință *mingeaLuiVasile* poate lua ca valoare referința către o instanță a clasei `Minge` sau a clasei `MingeDeBaschet`.

Prin intermediul unei variabile referință către o interfață nu se poate apela decât la funcționalitatea cerută în interfața respectivă, chiar dacă obiectele reale oferă și alte facilități, ele aparținând unor clase mai bogate în metode. 🏠

5.1.3.3 Tipul referință către un tablou

Tipul *referință către un tablou* este un tip referință care poate păstra o trimitere către locația din memorie a unui tablou de elemente. Prin intermediul acestei referințe putem accesa elementele tabloului furnizând indexul elementului dorit.

Tablourile de elemente nu există în general ci ele sunt tablouri formate din elemente de un tip bine precizat. Din această cauză, atunci când declarăm o referință către un tablou, trebuie să precizăm ȳi de ce tip sunt elementele din tabloul respectiv.

La declaraȳia referinȳei către tablou nu trebuie să precizăm ȳi numărul de elemente din tablou.

Iată cum se declară o referință către un tablou de ȳntregi lungi:

```
long numere[ ] ;
```

Numele variabilei este *numere*. Un alt exemplu de declaraȳie de referință către un tablou:

```
Minge echipament[ ] ;
```

Declaraȳia de mai sus construiește o referință către un tablou care păstrează elemente de tip referință către o instanȳă a clasei *Minge*. Numele variabilei referință este *echipament*. Parantezele drepte de după numele variabilei specifică faptul că este vorba despre un tablou.

Mai multe despre tablouri ȳntr-un paragraf următor. 🏠

5.1.4 Clasa de memorare

Fiecare variabilă trebuie să aibă o anumită clasă de memorare. Această clasă ne permite să aflăm care este intervalul de existenȳă ȳi vizibilitatea unei variabile ȳn contextul execuȳiei unui program.

Este important să ȳnȳelegem exact această noȳiune pentru că altfel vom ȳncerca să referim variabile ȳnainte ca acestea să fi fost create sau după ce au fost distruse sau să referim variabile care nu sunt vizibile din zona de program ȳn care le apelăm. Soluȳia simplă de existenȳă a tuturor variabilelor pe tot timpul execuȳiei este desigur afară din discuȳie atât din punct de vedere al eficienȳei cât ȳi a eleganȳei ȳi stabilităȳii codului. 🏠

5.1.4.1 Variabile locale

Aceste variabile nu au importanȳă prea mare ȳn contextul ȳntregii aplicaȳii, ele servind la rezolvarea unor probleme locale. Variabilele locale sunt declarate, rezervate ȳn memorie ȳi utilizate doar ȳn interiorul unor blocuri de instrucȳiuni, fiind distruse automat la ieșirea din aceste blocuri. Aceste variabile sunt vizibile doar ȳn interiorul blocului ȳn care au fost create ȳi ȳn subblocurile acestuia. 🏠

5.1.4.2 Variabile statice

Variabilele statice sunt în general legate de funcționalitatea anumitor clase de obiecte ale căror instanțe folosesc în comun aceste variabile. Variabilele statice sunt create atunci când codul specific clasei în care au fost declarate este încărcat în memorie și nu sunt distruse decât atunci când acest cod este eliminat din memorie.

Valorile memorate în variabile statice au importanță mult mai mare în aplicație decât cele locale, ele păstrând informații care nu trebuie să se piardă la dispariția unei instanțe a clasei. De exemplu, variabila în care este memorat numărul de picioare al obiectelor din clasa `Om` nu trebuie să fie distrusă la dispariția unei instanțe din această clasă. Aceasta din cauză că și celelalte instanțe ale clasei folosesc aceeași valoare. Și chiar dacă la un moment dat nu mai există nici o instanță a acestei clase, numărul de picioare ale unui `Om` trebuie să fie accesibil în continuare pentru interogare de către celelalte clase.

Variabilele statice nu se pot declara decât ca variabile ale unor clase și conțin în declarație cuvântul rezervat **static**. Din cauza faptului că ele aparțin clasei și nu unei anumite instanțe a clasei, variabilele statice se mai numesc uneori și *variabile de clasă*. 🏠

5.1.4.3 Variabile dinamice

Un alt tip de variabile sunt variabilele a căror perioadă de existență este stabilită de către programator. Aceste variabile pot fi alocate la cerere, dinamic, în orice moment al execuției programului. Ele vor fi distruse doar atunci când nu mai sunt referite de nicăieri.

La alocarea unei variabile dinamice, este obligatoriu să păstrăm o referință către ea într-o variabilă de tip referință. Altfel, nu vom putea accesa în viitor variabila dinamică. În momentul în care nici o referință nu mai trimite către variabila dinamică, de exemplu pentru că referința a fost o variabilă locală și blocul în care a fost declarată și-a terminat execuția, variabila dinamică este distrusă automat de către sistem printr-un mecanism numit *colector de gunoarie*.

Colectorul de gunoarie poate porni din inițiativa sistemului sau din inițiativa programatorului la momente bine precizate ale execuției.

Pentru a rezerva spațiu pentru o variabilă dinamică este nevoie să apelăm la o *expresie de alocare* care folosește cuvântul rezervat **new**. Această expresie alocă spațiul necesar pentru un anumit tip de valoare. De exemplu, pentru a rezerva spațiul necesar unui obiect de tip `Minge`, putem apela la sintaxa:

```
Minge mingeaMea = new Minge();
```

iar pentru a rezerva spațiul necesar unui tablou de referințe către obiecte de tip `Minge` putem folosi declarația:

```
Minge echipament[] = new Minge[5];
```

Am alocat astfel spațiu pentru un tablou care conține 5 referințe către obiecte de tip `Minge`. Pentru alocarea tablourilor conținând tipuri primitive se folosește aceeași sintaxă. De exemplu, următoarea linie de program alocă spațiul necesar unui tablou cu 10 întregi, creând în același timp și o variabilă referință spre acest tablou, numită *numere*:

```
int numere[] = new int[10];
```

5.1.5 Tablouri de variabile

Tablourile servesc, după cum spuneam, la memorarea secvențelor de elemente de același tip. Tablourile unidimensionale au semnificația vectorilor de elemente. Se poate întâmpla să lucrăm și cu tablouri de referințe către tablouri, în acest caz modelul fiind acela al unei matrici bidimensionale. În fine, putem extinde definiția și pentru mai mult de două dimensiuni.

5.1.5.1 Declarația variabilelor de tip tablou

Pentru a declara variabile de tip tablou, trebuie să specificăm tipul elementelor care vor umple tabloul și un nume pentru variabila referință care va păstra trimiterea către zona de memorie în care sunt memorate elementele tabloului.

Deși putem declara variabile referință către tablou și separat, de obicei declarația este făcută în același timp cu alocarea spațiului ca în exemplele din paragraful anterior.

Sintaxa Java permite plasarea parantezelor drepte care specifică tipul tablou înainte sau după numele variabilei. Astfel, următoarele două declarații sunt echivalente:

```
int[] numere;
int numere[];
```

Dacă doriți să folosiți tablouri cu două dimensiuni ca matricile, puteți să declarați un tablou de referințe către tablouri cu una dintre următoarele trei sintaxe echivalente:

```
float[][] matrice;
float[] matrice[];
float matrice[][];
```

De precizat că și în cazul dimensiunilor multiple, declarațiile de mai sus nu fac nimic altceva decât să rezerve loc pentru o referință și să precizeze numărul de dimensiuni. Alocarea spațiului pentru elementele tabloului trebuie făcută explicit.

Despre rezervarea spațiului pentru tablourile cu o singură dimensiune am vorbit deja. Pentru tablourile cu mai multe dimensiuni, rezervarea spațiului se poate face cu următoarea sintaxă:

```
byte [][]octeti = new byte[23][5];
```

În expresia de alocare sunt specificate în clar numărul elementelor pentru fiecare dimensiune a tabloului.



5.1.5.2 Inițializarea tablourilor.

Limbajul Java permite și o sintaxă pentru inițializarea elementelor unui tablou. Într-un astfel de caz este rezervat automat și spațiul de memorie necesar memorării valorilor inițiale. Sintaxa folosită în astfel de cazuri este următoarea:

```
char []caractere = { a, b, c, d };
```

Acest prim exemplu alocă spațiu pentru patru elemente de tip caracter și inițializează aceste elemente cu valorile dintre acolade. După aceea, creează variabila de tip referință numită *caractere* și o inițializează cu referința la zona de memorie care păstrează cele patru valori.

Inițializarea funcționează și la tablouri cu mai multe dimensiuni ca în exemplele următoare:

```
int [][]numere = {
    { 1, 3, 4, 5 },
    { 2, 4, 5 },
    { 1, 2, 3, 4, 5 }
};
double [][][]reali = {
    { { 0.0, -1.0 }, { 4.5 } },
    { { 2.5, 3.0 } }
};
```

După cum observați numărul inițializatorilor nu trebuie să fie același pentru fiecare element.



5.1.5.3 Lungimea tablourilor

Tablourile Java sunt alocate dinamic, ceea ce înseamnă că ele își pot schimba dimensiunile pe parcursul execuției. Pentru a afla numărul de elemente dintr-un tablou, putem apela la următoarea sintaxă:

```
float []tablou = new float[25];
int dimensiune = tablou.length;
// dimensiune primește valoarea 25
```

sau

```
float [][]multiTablou = new float[3][4];
int dimensiune1 = multiTablou[2].length;
// dimensiune1 primește valoarea 4
int dimensiune2 = multiTablou.length;
// dimensiune2 primește valoarea 3 ^
```

5.1.5.4 Referirea elementelor din tablou

Elementele unui tablou se pot referi prin numele referinței tabloului și indexul elementului pe care dorim să-l referim. În Java, primul element din tablou este elementul cu numărul 0, al doilea este elementul numărul 1 și așa mai departe.

Sintaxa de referire folosește parantezele pătrate [și]. Între ele trebuie specificat indexul elementului pe care dorim să-l referim. Indexul nu trebuie să fie constant, el putând fi o expresie de complexitate oarecare.

Iată câteva exemple:

```
int []tablou = new int[10];
tablou[3] = 1;
// al patrulea element primește valoarea 1
float [][]reali = new float[3][4];
reali[2][3] = 1.0f;
// al patrulea element din al treilea tablou
// primește valoarea 1
```

În cazul tablourilor cu mai multe dimensiuni, avem în realitate tablouri de referințe la tablouri. Asta înseamnă că dacă considerăm următoarea declarație:

```
char [][]caractere = new char [5][];
```

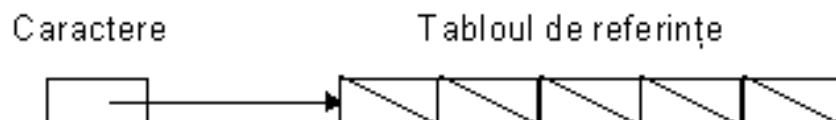


Figura 5.1 Elementele tabloului sunt de tip referință, inițializate implicit la valoarea null.

Variabila referință numită *caractere* conține deocamdată un tablou de 5 referințe la tablouri de caractere. Cele cinci referințe sunt inițializate cu **null**. Putem inițializa aceste tablouri prin atribuiri de

expresii de alocare:

```
caractere[0] = new char [3];
caractere[4] = new char [5];
```

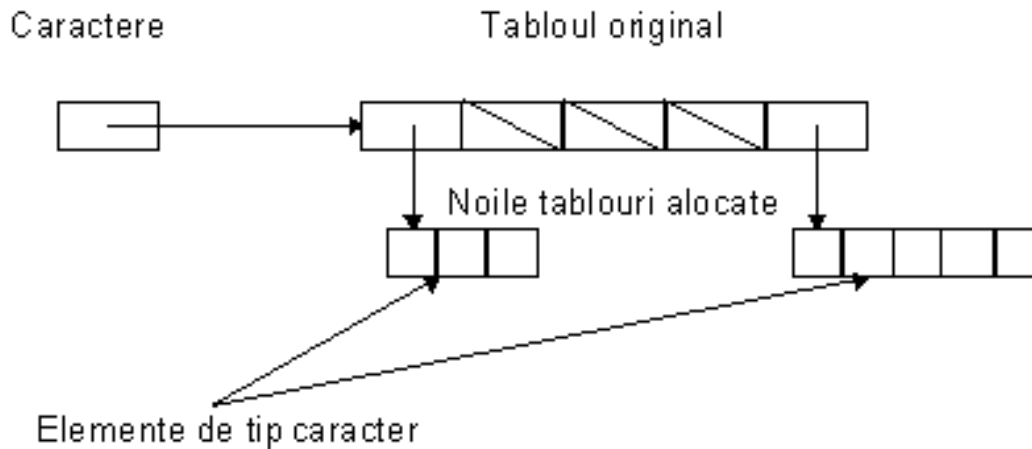


Figura 5.2 Noile tablouri sunt referite din interiorul tabloului original. Elementele noilor tablouri sunt caractere.

La fel, putem scrie:

```
char []tablouDeCaractere = caractere[0];
```

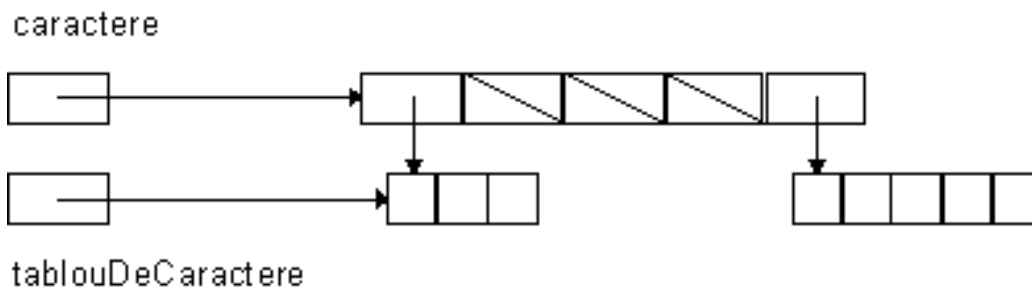


Figura 5.3 Variabilele de tip referință *caractere[0]* și *tablouDeCaractere* trimit spre același tablou rezervat în memorie.

Variabila *tablouDeCaractere* trimite către același tablou de caractere ca și cel referit de primul element al tabloului referit de variabila *caractere*.

Să mai precizăm că referirea unui element de tablou printr-un index mai mare sau egal cu lungimea tabloului duce la oprirea execuției programului cu un mesaj de eroare de execuție corespunzător. 🚫

5.1.5.5 Alocarea și eliberarea tablourilor

Despre alocarea tablourilor am spus deja destul de multe. În cazul în care nu avem inițializatori, variabilele sunt inițializate cu valorile implicite definite de limbaj pentru tipul corespunzător. Aceasta înseamnă că, pentru tablourile cu mai multe dimensiuni, referințele sunt inițializate cu **null**.

Pentru eliberarea memoriei ocupate de un tablou, este suficient să tăiem toate referințele către tablou. Sistemul va sesiza automat că tabloul nu mai este referit și mecanismul colector de gunoaie va elibera zona. Pentru a tăia o referință către un tablou dăm o altă valoare variabilei care referă tabloul. Valoarea poate fi **null** sau o referință către un alt tablou.

De exemplu:

```
float []reali = new float[10];
?
reali = null; // eliberarea tabloului
sau
reali = new float[15]; // eliberarea în alt fel
sau
{
float []reali = new float[10];
?
} // eliberare automată, variabila reali a fost
// distrusă la ieșirea din blocul în care a
// fost declarată, iar tabloul de 10 floatanți
// nu mai este referit ^
```

5.1.6 Conversii

Operațiile definite în limbajul Java au un tip bine precizat de argumente. Din păcate, există situații în care nu putem transmite la apelul acestora exact tipul pe care compilatorul Java îl așteaptă. În asemenea situații, compilatorul are două alternative: fie respinge orice operație cu argumente greșite, fie încearcă să convertească argumentele către tipurile necesare. Desigur, în cazul în care conversia nu este posibilă, singura alternativă rămâne prima.

În multe situații însă, conversia este posibilă. Să luăm de exemplu tipurile întregi. Putem să convertim întotdeauna un întreg scurt la un întreg. Valoarea rezultată va fi exact aceeași. Conversia inversă însă, poate pune probleme dacă valoarea memorată în întreg depășește capacitatea de memorare a unui întreg scurt.

În afară de conversiile implicite, pe care compilatorul le hotărăște de unul singur, există și conversii explicite, pe care programatorul le poate forța la nevoie. Aceste conversii efectuează de obicei operații în care există pericolul să se piardă o parte din informații. Compilatorul nu poate hotărî de unul singur în aceste situații.

Conversiile implicite pot fi un pericol pentru stabilitatea aplicației dacă pot să ducă la pierderi de informații fără avertizarea programatorului. Aceste erori sunt de obicei extrem de greu de depistat.

În fiecare limbaj care lucrează cu tipuri fixe pentru datele sale există conversii imposibile, conversii periculoase și conversii sigure. Conversiile imposibile sunt conversiile pe care limbajul nu le permite pentru că nu ți se cum să le execute sau pentru că operația este prea periculoasă. De exemplu, Java refuză să convertească un tip primitiv către un tip referință. Deși s-ar putea imagina o astfel de conversie bazată pe faptul că o adresă este în cele din urmă un număr natural, acest tip de conversii

sunt extrem de periculoase, chiar și atunci când programatorul cere explicit această conversie. 🏠

5.1.6.1 Conversii de extindere a valorii

În aceste conversii valoarea se reprezintă într-o zonă mai mare fără să se piardă nici un fel de informații. Iată conversiile de extindere pe tipuri primitive:

- **byte** la **short**, **int**, **long**, **float** sau **double**
- **short** la **int**, **long**, **float** sau **double**
- **char** la **int**, **long**, **float** sau **double**
- **int** la **long**, **float** sau **double**
- **long** la **float** sau **double**
- **float** la **double**

Să mai precizăm totuși că, într-o parte din aceste cazuri, putem pierde din precizie. Această situație apare de exemplu la conversia unui **long** într-un **float**, caz în care se pierde o parte din cifrele semnificative păstrându-se însă ordinul de mărime. De altfel această observație este evidentă dacă ținem cont de faptul că un **long** este reprezentat pe 64 de biți în timp ce un **float** este reprezentat doar pe 32 de biți.

Precizia se pierde chiar și în cazul conversiei **long** la **double** sau **int** la **float** pentru că, deși dimensiunea zonei alocată pentru cele două tipuri este aceeași, numerele flotante au nevoie de o parte din această zonă pentru a reprezenta exponentul.

În aceste situații, se va produce o rotunjire a numerelor reprezentate. 🏠

5.1.6.2 Conversii de trunchiere a valorii

Convențiile de trunchiere a valorii pot produce pierderi de informație pentru că ele convertesc tipuri mai bogate în informații către tipuri mai sărace. Conversiile de trunchiere pe tipurile elementare sunt următoarele:

- **byte** la **char**

- **short** la **byte** sau **char**
- **char** la **byte** sau **short**
- **int** la **byte**, **short** sau **char**
- **long** la **byte**, **short** **char**, sau **int**
- **float** la **byte**, **short**, **char**, **int** sau **long**
- **double** la **byte**, **short**, **char**, **int**, **long** sau **float**.

În cazul conversiilor de trunchiere la numerele cu semn, este posibil să se schimbe semnul pentru că, în timpul conversiei, se îndepărtează pur și simplu octeții care nu mai încap și poate rămâne primul bit diferit de vechiul prim bit. Copierea se face începând cu octeții mai puțin semnificativi iar trunchierea se face la octeții cei mai semnificativi.

Prin *octeții cei mai semnificativi* ne referim la octeții în care sunt reprezentate cifrele cele mai semnificative. *Cifrele cele mai semnificative* sunt cifrele care dau ordinul de mărime al numărului. De exemplu, la numărul 123456, cifrele cele mai semnificative sunt primele, adică: 1, 2, etc. La același număr, *cifrele cele mai puțin semnificative* sunt ultimele, adică: 6, 5, etc. 🏠

5.1.6.3 Conversii pe tipuri referință

Conversiile tipurilor referință nu pun probleme pentru modul în care trebuie executată operația din cauză că, referința fiind o adresă, în timpul conversiei nu trebuie afectată în nici un fel această adresă. În schimb, se pun probleme legate de corectitudinea logică a conversiei. De exemplu, dacă avem o referință la un obiect care nu este tablou, este absurd să încercăm să convertim această referință la o referință de tablou.

Limbajul Java definește extrem de strict conversiile posibile în cazul tipurilor referință pentru a salva programatorul de eventualele necazuri care pot apare în timpul execuției. Iată conversiile posibile:

- O referință către un obiect aparținând unei clase *C* poate fi convertit la o referință către un obiect aparținând clasei *S* doar în cazul în care *C* este chiar *S* sau *C* este derivată direct sau indirect din *S*.
- O referință către un obiect aparținând unei clase *C* poate fi convertit către o referință de interfață *I* numai dacă clasa *C* implementează interfața *I*.
- O referință către un tablou poate fi convertită la o referință către o clasă numai dacă clasa respectivă este clasa **Object**.
- O referință către un tablou de elemente ale cărui elemente sunt de tipul *T1* poate fi convertită la o referință către un tablou de elemente de tip *T2* numai dacă *T1* și *T2* reprezintă același tip primitiv sau *T2* este un tip referință și *T1* poate fi convertit către *T2*. 🏠

5.1.6.4 Conversii la operația de atribuire

Conversiile pe care limbajul Java le execută implicit la atribuire sunt foarte puține. Mai exact, sunt

executate doar acele conversii care nu necesită validare în timpul execuției și care nu pot pierde informații în cazul tipurilor primitive.

În cazul valorilor aparținând tipurilor primitive, următorul tabel arată conversiile posibile. Pe coloane avem tipul de valoare care se atribuie iar pe linii avem tipurile de variabile la care se atribuie:

	boolean	char	byte	short	int	long	float	double
boolean	Da	Nu	Nu	Nu	Nu	Nu	Nu	Nu
char	Nu	Da	Da	Da	Nu	Nu	Nu	Nu
byte	Nu	Da	Da	Nu	Nu	Nu	Nu	Nu
short	Nu	Da	Da	Da	Nu	Nu	Nu	Nu
int	Nu	Da	Da	Da	Da	Nu	Nu	Nu
long	Nu	Da	Da	Da	Da	Da	Nu	Nu
float	Nu	Da	Da	Da	Da	Da	Da	Nu
double	Nu	Da	Da	Da	Da	Da	Da	Da

Tabloul 5.1 Conversiile posibile într-o operație de atribuire cu tipuri primitive. Coloanele reprezintă tipurile care se atribuie iar liniile reprezintă tipul de variabilă către care se face atribuirea.

După cum observați, tipul boolean nu poate fi atribuit la o variabilă de alt tip.

Valorile de tip primitiv nu pot fi atribuite variabilelor de tip referință. La fel, valorile de tip referință nu pot fi memorate în variabile de tip primitiv. În ceea ce privește tipurile referință între ele, următorul tabel definește situațiile în care conversiile sunt posibile la atribuirea unei valori de tipul *T* la o variabilă de tipul *S*:

	T este o clasă care nu este finală	T este o clasă care este finală	T este o interfață	T = B[] este un tablou cu elemente de tipul B
S este o clasă care nu este finală	T trebuie să fie subclasă a lui S	T trebuie să fie o subclasă a lui S	eroare la compilare	S trebuie să fie Object

S este o clasă care este finală	T trebuie să fie aceeași clasă ca S	T trebuie să fie aceeași clasă ca S	eroare la compilare	eroare la compilare
S este o interfață	T trebuie să implementeze interfața S	T trebuie să implementeze interfața S	T trebuie să fie o subinterfață a lui S	eroare la compilare
S = A[] este un tablou cu elemente de tipul A	eroare la compilare	eroare la compilare	eroare la compilare	A sau B sunt același tip primitiv sau A este un tip referință și B poate fi atribuit lui A

Tabloul 5.2 Conversiile posibile la atribuirea unei valori de tipul T la o variabilă de tipul S. 🏠

5.1.6.5 Conversii explicite

Conversiile de tip cast, sau *casturile*, sunt apelate de către programator în mod explicit. Sintaxa pentru construcția unui cast este scrierea tipului către care dorim să convertim în paranteze în fața valorii pe care dorim să o convertim. Forma generală este:

(*Tip*) Valoare

Conversiile posibile în acest caz sunt mai multe decât conversiile implicite la atribuire pentru că în acest caz programatorul este prevenit de eventuale pierderi de date el trebuind să apeleze conversia explicit.

Dar, continuă să existe conversii care nu se pot apela nici măcar în mod explicit, după cum am explicat înainte.

În cazul conversiilor de tip cast, orice valoare numerică poate fi convertită la orice valoare numerică.

În continuare, valorile de tip boolean nu pot fi convertite la nici un alt tip.

Nu există conversii între valorile de tip referință și valorile de tip primitiv.

În cazul conversiilor dintr-un tip referință într-altul putem separa două cazuri. Dacă compilatorul poate decide în timpul compilării dacă conversia este corectă sau nu, o va decide. În cazul în care compilatorul nu poate decide pe loc, se va efectua o verificare a conversiei în timpul execuției. Dacă conversia se dovedește greșită, va apărea o eroare de execuție și programul va fi întrerupt.

Iată un exemplu de situație în care compilatorul nu poate decide dacă conversia este posibilă sau nu:

```
Minge mingeaMea;
?
MingeDeBaschet mingeaMeaDeBaschet;
// MingeDeBaschet este o clasă
// derivată din clasa Minge
mingeaMeaDeBaschet=(MingeDeBaschet)mingeaMea;
```

În acest caz, compilatorul nu poate fi sigur dacă referința memorată în variabila *mingeaMea* este de tip *MingeDeBaschet* sau nu pentru că variabilei de tip *Minge* i se pot atribui și referințe către instanțe de tip *Minge* în general, care nu respectă întru totul definiția clasei *MingeDeBaschet* sau chiar referințe către alte tipuri de minge derivate din clasa *Minge*, de exemplu *MingeDePolo* care implementează proprietăți și operații diferite față de clasa *MingeDeBaschet*.

Iată și un exemplu de conversie care poate fi decisă în timpul compilării:

```
Minge mingeaMea;
MingeDeBaschet mingeaMeaDeBaschet;
?
mingeaMea = ( Minge ) mingeaMeaDeBaschet;
```

În următorul exemplu însă, se poate decide în timpul compilării imposibilitatea conversiei:

```
MingeDeBaschet mingeaMeaDeBaschet;
MingeDePolo mingeaMeaDePolo;
?
mingeaMeaDePolo = ( MingeDePolo ) mingeaMeaDeBaschet;
```

În fine, tabelul următor arată conversiile de tip cast a căror corectitudine poate fi stabilită în timpul compilării. Conversia încearcă să transforme printr-un cast o referință de tip *T* într-o referință de tip *S*.

	T este o clasă care nu este finală	T este o clasă care este finală	T este o interfață	T = B[] este un tablou cu elemente de tipul B
S este o clasă care nu este finală	T trebuie să fie subclasă a lui S	T trebuie să fie o subclasă a lui S	Totdeauna corectă la compilare	S trebuie să fie Object

S este o clasă care este finală	S trebuie să fie subclasă a lui T	T trebuie să fie aceeași clasă ca și S	S trebuie să implementeze interfața T	eroare la compilare
S este o interfață	Totdeauna corectă la compilare	T trebuie să implementeze interfața S	Totdeauna corectă la compilare	eroare la compilare
S = A[] este un tablou cu elemente de tipul A	T trebuie să fie Object	eroare la compilare	eroare la compilare	A sau B sunt același tip primitiv sau A este un tip referință și B poate fi convertit cu un cast la A

Tabloul 5.3 Cazurile posibile la convertirea unei referințe de tip T într-o referință de tip S. 🏠

5.1.6.6 Conversii de promovare aritmetică

Promovarea aritmetică se aplică în cazul unor formule în care operanzii pe care se aplică un operator sunt de tipuri diferite. În aceste cazuri,

compilatorul încearcă să promoveze unul sau chiar amândoi operanzii la același tip pentru a putea fi executată operația.

Există două tipuri de promovare, promovare aritmetică unară și binară.

În cazul *promovării aritmetice unare*, există un singur operand care în cazul că este **byte** sau **short** este transformat la **int** altfel rămâne nemodificat.

La *promovarea aritmetică binară* se aplică următorul algoritm:

1. Dacă un operand este **double**, celălalt este convertit la **double**.
2. Altfel, dacă un operand este de tip **float**, celălalt operand este convertit la **float**.
3. Altfel, dacă un operand este de tip **long**, celălalt este convertit la **long**.
4. Altfel, amândoi operanzii sunt convertiți la **int**.

De exemplu, în următoarea operație amândoi operanzii vor fi convertiți la **float** prin promovare aritmetică binară:

```
float f;
double i = f + 3;
```

După efectuarea operației, valoarea obținută va fi convertită implicit la **double**.

În următorul exemplu, se produce o promovare unară la int de la short.

```
short s, r;
?
int min = ( r < -s ) ? r : s;
```

În expresia condițională, operandul `-s` se traduce de fapt prin aplicarea operatorului unar `-` la variabila `s` care este de tip **short**. În acest caz, se va produce automat promovarea aritmetică unară de la **short** la **int**, apoi se va continua evaluarea expresiei. 🏠

[\[capitolul V\]](#)

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul V

5.2 Expresii

[5.2.1](#) *Valoarea și tipul unei expresii*

[5.2.2](#) *Ordinea de evaluare*

[5.2.3](#) *Operatori*

[5.2.3.1](#) *Operatori unari*

[5.2.3.1.1](#) *Operatorii de preincrementare și postincrementare*

[5.2.3.1.2](#) *Operatorul + unar*

[5.2.3.1.3](#) *Operatorul - unar*

[5.2.3.1.4](#) *Operatorul de complementare*

[5.2.3.1.5](#) *Operatorul de negare logică*

[5.2.3.1.6](#) *Casturi*

[5.2.3.2](#) *Operatori binari*

[5.2.3.2.1](#) *Operatori multiplicativi: *, /, %*

[5.2.3.2.2](#) *Operatori aditivi: +, -, concatenare pentru șiruri de caractere*

[5.2.3.2.3](#) *Operatori de deplasare: >>, <<, >>>*

[5.2.3.2.4](#) *Operatori relaționali: <, >, <=, >=, instanceof*

[5.2.3.2.5](#) *Operatori de egalitate: ==, !=*

[5.2.3.2.6](#) *Operatori la nivel de bit: &, |, ^*

[5.2.3.2.7](#) *Operatori logici: &&, ||*

[5.2.3.3](#) *Operatorul condițional ?:*

[5.2.3.4](#) *Operații întregi*

[5.2.3.5](#) *Operații flotante*

[5.2.3.6](#) *Apeluri de metode*

5.2.1 Valoarea și tipul unei expresii

Fiecare expresie a limbajului Java are un rezultat și un tip. Rezultatul poate fi:

- o valoare
- o variabilă

- nimic

În cazul în care valoarea unei expresii are ca rezultat o *variabilă*, expresia poate apare în stânga unei operații de atribuire. De exemplu expresia:

```
tablou[i]
```

este o expresie care are ca rezultat o variabilă și anume locația elementului cu indexul i din tabloul de elemente numit *tablou*.

O expresie nu produce nimic doar în cazul în care este un apel de metodă și acest apel nu produce nici un rezultat (este declarat de tip **void**).

Fiecare expresie are în timpul compilării un tip cunoscut. Acest tip poate fi o valoare primitivă sau o referință. În cazul expresiilor care au tip referință, valoarea expresiei poate să fie și referința neinițializată, **null**. 🏠

5.2.2 Ordinea de evaluare

În Java operanzii sunt evaluați întotdeauna de la stânga spre dreapta. Acest lucru nu trebuie să ne îndemne să scriem cod care să depindă de ordinea de evaluare pentru că, în unele situații, codul rezultat este greu de citit. Regula este introdusă doar pentru a asigura generarea uniformă a codului binar, independent de compilatorul folosit.

Evaluarea de la stânga la dreapta implică următoarele aspecte:

- în cazul unui operator binar, operandul din stânga este întotdeauna complet evaluat atunci când se trece la evaluarea operandului din dreapta. De exemplu, în expresia:

```
( i++ ) + i
```

dacă i avea valoarea inițială 2, toată expresia va avea valoarea 5 pentru că valoarea celui de-al doilea i este luată după ce s-a executat incrementarea $i++$.

- în cazul unei referințe de tablou, expresia care numește tabloul este complet evaluată înainte de a se trece la evaluarea expresiei care dă indexul. De exemplu, în expresia:

```
( a = b )[i]
```

- indexul va fi aplicat după ce s-a executat atribuirea valorii referință b la variabila referință de tablou a . Cu alte cuvinte, rezultatul va fi al i -lea element din tabloul b .
- în cazul apelului unei metode, expresia care numește obiectul este complet evaluată atunci când se

trece la evaluarea expresiilor care servesc drept argumente.

- în cazul unui apel de metodă, dacă există mai mult decât un parametru, la evaluarea parametrului numărul i , toți parametrii de la 1 la $i-1$ sunt deja evaluați complet.
- în cazul unei expresii de alocare, dacă avem mai multe dimensiuni exprimate în paranteze drepte, dimensiunile sunt de asemenea evaluate de la stânga la dreapta. De exemplu, în expresia:

```
new int[i++][i]
```

- tabloul rezultat nu va fi o matrice pătratică ci, dacă i avea valoarea 3, de dimensiune 3 x 4. 🏠

5.2.3 Operatori

5.2.3.1 Operatori unari

Operatorii unari se aplică întotdeauna unui singur operand. Acești operatori sunt, în general exprimați înaintea operatorului asupra căruia se aplică. Există însă și două excepții, operatorii de incrementare și decrementare care pot apare și înainte și după operator, cu semnificații diferite.

Operatorii unari sunt următorii:

- ++ preincrement
- -- predecrement
- ++ postincrement
- -- postdecrement
- + unar
- - unar
- ~ complementare
- ! negație logică
- cast 🏠

5.2.3.1.1 Operatorii de preincrementare și postincrementare

Operatorii ++ preincrement și postincrement au același rezultat final, și anume incrementează variabila asupra căreia acționează cu 1. Operandul asupra căruia sunt aplicați trebuie să fie o variabilă de tip aritmetic. Nu are sens să apelăm un operand de incrementare asupra unei valori (de exemplu valoarea unei expresii sau un literal), pentru că aceasta nu are o locație de memorie fixă în care să memorăm valoarea după incrementare.

În cazul operatorului prefix, valoarea rezultat a acestei expresii este valoarea variabilei după incrementare în timp ce, la operatorul postfix, valoarea rezultat a expresiei este valoarea de dinainte de incrementare. De exemplu, după execuția următoarei secvențe de instrucțiuni:

```
int i = 5;
int j = i++;
```

valoarea lui j este 5, în timp ce, după execuția următoarei secvențe de instrucțiuni:

```
int i = 5;
int j = ++i;
```

valoarea lui j va fi 6. În ambele cazuri, valoarea finală a lui i va fi 6.

În cazul operatorilor $--$ predecrement și $--$ postdecrement, sunt valabile aceleași considerații ca mai sus, cu diferența că valoarea variabilei asupra căreia se aplică operandul va fi decrementată cu 1. De exemplu, următoarele instrucțiuni:

```
int i = 5;
int j = i--;
```


fac ca j să aibă valoarea finală 5 iar următoarele instrucțiuni:

```
int i = 5;
int j = --i;
```


fac ca j să aibă valoarea finală 4. În ambele cazuri, valoarea finală a lui i este 4.

Operatorii de incrementare și de decrementare se pot aplica și pe variabile de tip flotant. În asemenea cazuri, se convertește valoarea la tipul variabilei incrementate sau decrementate, după care valoarea rezultată este adunată respectiv scăzută din vechea valoare a variabilei. De exemplu, următoarele instrucțiuni:

```
double f = 5.6;
double g = ++f;
```

au ca rezultat final valoarea 6.6 pentru f și pentru g . 

5.2.3.1.2 Operatorul + unar

Operatorul $+$ unar se aplică asupra oricărei valori primitive aritmetice. Valoarea rămâne neschimbată. 

5.2.3.1.3 Operatorul - unar

Operatorul $-$ unar se aplică asupra oricărei valori primitive aritmetice. Rezultatul aplicării acestui operand este negarea aritmetică a valorii. În cazul valorilor întregi, acest lucru este echivalent cu


scăderea din 0 a valorii originale. De exemplu, instrucțiunile:

```
int i = 5;
int j = -i;
```


îi dau lui j valoarea -5 . În cazul valorilor speciale definite de standardul IEEE pentru reprezentarea numerelor flotante, se aplică următoarele reguli:

- Dacă operandul este NaN rezultatul negării aritmetice este tot NaN pentru că NaN nu are semn.
- Dacă operandul este unul dintre infinii, rezultatul este infinitul opus ca semn.
- Dacă operandul este zero de un anumit semn, rezultatul este zero de semn diferit. [^](#)

5.2.3.1.4 Operatorul de complementare

Operatorul de complementare \sim se aplică asupra valorilor primitive de tip întreg. Rezultatul aplicării operandului este complementarea bit cu bit a valorii originale. De exemplu, dacă operandul era de tip byte având valoarea, în binar, 00110001, rezultatul va fi 11001110. În realitate, înainte de complementare se face \circ i extinderea valorii la un întreg, deci rezultatul va fi de fapt: 11111111 11111111 11001110. 

5.2.3.1.5 Operatorul de negare logică


Operatorul de negare logică $!$ se aplică în exclusivitate valorilor de tip boolean. În cazul în care valoarea inițială a operandului este **true** rezultatul va fi **false** \circ i invers. 

5.2.3.1.6 Casturi

Casturile sunt expresii de conversie dintr-un tip într-altul, a \circ a cum deja am arătat la paragraful destinat conversiilor. Rezultatul unui cast este valoarea operandului convertită la noul tip de valoare exprimat de cast. De exemplu, la instrucțiunile:


```
double f = 5.6;
int i = ( int )f;
double g = -5.6;
int j = ( int )g;
```

valoarea variabilei f este convertită la o valoare întreagă, anume 5, \circ i noua valoare este atribuită variabilei i . La fel, j primește valoarea -5.

Să mai precizăm că nu toate casturile sunt valide în Java. De exemplu, nu putem converti o valoare întreagă într-o valoare de tip referință. 

5.2.3.2 Operatori binari

Operatorii binari au întotdeauna doi operanzi. Operatorii binari sunt următorii:


- Operatori multiplicativi: `*`, `/`, `%`
- Operatori aditivi: `+`, `-`, `+` (concatenare) pentru şiruri de caractere
- Operatori de şifare: `>>`, `<<`, `>>>`
- Operatori relaţionali: `<`, `>`, `<=`, `>=`, `instanceof`
- Operatori de egalitate: `==`, `!=`
- Operatori la nivel de bit: `&`, `|`, `^`
- Operatori logici: `&&`, `||` 

5.2.3.2.1 Operatori multiplicativi: `*`, `/`, `%`

Operatorii multiplicativi reprezintă respectiv operaţiunile de înmulţire (`*`), împărţire (`/`) şi restul împărţirii (`%`). Prioritatea acestor operaţii este mai mare relativ la operaţiunile aditive, deci aceşti operatori se vor executa mai întâi. Exemple:

```
10 * 5 == 50
10.3 * 5.0 == 51.5
10 / 2.5 == 4.0 // împărţire reală
3 / 2 == 1 // împărţire întreagă
7 % 2 == 1 // restul împărţirii întregi
123.5 % 4 == 3.5 // 4 * 30 + 3.5
123.5 % 4.5 == 2.0 // 4.5 * 27 + 2.0
```

După cum observaţi, operanzii sunt convertiţi mai întâi la tipul cel mai puternic, prin promovare aritmetică, şi apoi se execută operaţia. Rezultatul este de acelaşi tip cu tipul cel mai puternic.

În cazul operatorului pentru restul împărţirii, dacă lucrăm cu numere flotante, rezultatul se calculează în felul următor: se calculează de câte ori este cuprins cel de-al doilea operand în primul (un număr întreg de ori) după care rezultatul este diferenţa care mai rămâne, întotdeauna mai mică strict decât al doilea operand. 

5.2.3.2.2 Operatori aditivi: `+`, `-`, concatenare pentru şiruri de caractere

Operatorii aditivi reprezintă operaţiunile de adunare (`+`), scădere (`-`) şi concatenare (`+`) de şiruri. Observaţiunile despre conversia tipurilor făcute la operatorii multiplicativi rămân valabile. Exemple:

```
2 + 3 == 5
2.34 + 3 == 5.34
34.5 - 23.1 == 11.4
```

```
"Acesta este" + " un sir" == "Acesta este un sir"
"Sirul: " + 1 == "Sirul: 1"
"Sirul: " + 3.4444 == "Sirul: 3.4444"
"Sirul: " + null == "Sirul: null"
"Sirul: " + true == "Sirul: true"
Object obiect = new Object();
"Sirul: " + obiect == "java.lang.Object@1393800"
```

La concatenarea ȳirurilor de caractere, lungimea ȳirului rezultat este suma lungimii ȳirurilor care intră ȳn operaȳie. Caracterele din ȳirul rezultat sunt caracterele din primul ȳir, urmate de cele dintr-al doilea ȳir ȳn ordine.

Dacă cel de-al doilea operand nu este de tip **String** ci este de tip referinȳă, se va apela metoda sa **toString**, ȳi apoi se va folosi ȳn operaȳie rezultatul. Metoda **toString** este definită ȳn clasa **Object** ȳi este moȳtenită de toate celelalte clase.

Dacă cel de-al doilea operand este un tip primitiv, acesta este convertit la un ȳir rezonabil de caractere care să reprezinte valoarea operandului. 🏠

5.2.3.2.3 Operatori de ȳiftare: >>, <<, >>>

Operatorii de ȳiftare se pot aplica doar pe valori primitive ȳntregi. Ei reprezintă respectiv operaȳiile de ȳiftare cu semn stȳnga (<<) ȳi dreapta (>>) ȳi operaȳia de ȳiftare fȳră semn spre dreapta (>>>).

ȳiftările cu semn lucrează la nivel de cifre binare. Cifrele binare din locaȳia de memorie implicată sunt mutate cu mai multe poziȳii spre stȳnga sau spre dreapta. Poziȳia binară care reprezintă semnul rȳmâne neschimbată. Numărul de poziȳii cu care se efectuează mutarea este dat de al doilea operand. Locaȳia de memorie ȳn care se execută operaȳia este locaȳia ȳn care este memorat primul operand.

ȳiftarea cu semn la stȳnga reprezintă o operaȳie identică cu ȳnmulȳirea cu 2 de n ori, unde n este al doilea operand. ȳiftarea cu semn la dreapta reprezintă ȳmpȳrȳirea ȳntreagă. ȳn acest caz, semnul este copiat ȳn mod repetat ȳn locurile rȳmase goale. Iată cȳteva exemple:

```
255 << 3 == 2040
// 00000000 11111111 -> 00000111 11111000
255 >> 5 == 7
// 00000000 11111111 -> 00000000 00000111
```

ȳiftarea fȳră semn la dreapta, mută cifrele binare din operand completȳnd spaȳiul rȳmas cu zerouri:

```
0xffffffff >>> -1 == 0x00000001
0xffffffff >>> -2 == 0x00000003
```

```
0xffffffff >>> -3 == 0x00000007
0xffffffff >>> 3 == 0x1ffffffff
0xffffffff >>> 5 == 0x07ffffff ^
```

5.2.3.2.4 Operatori relaționali: <, >, <=, >=, instanceof

Operatorii relaționali întorc valori booleene de adevărat sau fals. Ei reprezintă testele de mai mic (<), mai mare (>), mai mic sau egal (<=), mai mare sau egal (>=) și testul care ne spune dacă un anumit obiect este sau nu instanță a unei anumite clase (**instanceof**). Iată câteva exemple:

```
1 < 345 == true
1 <= 0 == false
Object o = new Object();
String s = new String();
o instanceof Object == true
s instanceof String == true
o instanceof String == false
s instanceof Object == true
```

Să mai observăm că **String** este derivat din **Object**. 🏠


5.2.3.2.5 Operatori de egalitate: ==, !=

Acești operatori testează egalitatea sau inegalitatea dintre două valori. Ei reprezintă testul de egalitate (==) și de inegalitate (!=). Rezultatul aplicării acestor operatori este o valoare booleană.

Exemple:

```
( 1 == 1.0 ) == true
( 2 != 2 ) == false
Object o = new Object();
String s1 = "vasile";
String s2 = s1;
String s4 = "e";
String s3 = "vasil" + s4;
( o == s1 ) == false
( s1 == s2 ) == true // același obiect referit
( s3 == s1 ) == false // același șir de caractere
// dar obiecte diferite
```

Să observăm că egalitatea a două obiecte de tip **String** reprezintă egalitatea a două referințe de obiecte și nu egalitatea conținutului șirului de caractere. Două referințe sunt egale dacă referă exact același

obiect, nu dacă obiectele pe care le referă sunt egale între ele. Egalitatea conținutului a două șiruri de caractere se testează folosind metoda **equals**, definită în clasa **String**. 

5.2.3.2.6 Operatori la nivel de bit: &, |, ^

Operatorii la nivel de bit reprezintă operațiile logice obișnuite, dacă considerăm că 1 ar reprezenta adevărul și 0 falsul. Operatorii la nivel de bit, consideră cei doi operanzi ca pe două șiruri de cifre binare și fac operațiile pentru fiecare dintre perechile de cifre binare corespunzătoare în parte. Rezultatul este un nou șir de cifre binare. De exemplu, operația de *și* (&) logic are următorul tabel de adevăr:

1	&	1	==	1
1	&	0	==	0
0	&	1	==	0
0	&	0	==	0

Dacă apelăm operatorul & pe numerele reprezentate binar:

```
00101111
01110110
```

rezultatul este:

```
00100110
```

Primul număr reprezintă cifra 47, al doilea 118 iar rezultatul 38, deci:

```
47 & 118 == 38
```

În mod asemănător, tabela de adevăr pentru operația logică *sau* (|) este:

1		1	==	1
1		0	==	1
0		1	==	1
0		0	==	0

iar tabela de adevăr pentru operația logică de *sau exclusiv* (^) este:

1	^	1	==	0
1	^	0	==	1
0	^	1	==	1
0	^	0	==	0

Iată ȳi alte exemple:

```
1245 ^ 2345 == 3572
128 & 255 == 128
127 & 6 == 6
128 | 255 == 255
127 | 6 == 127
32 ^ 64 == 96 ^
```

5.2.3.2.7 Operatori logici: &&, ||

Operatorii logici se pot aplica doar asupra unor operanzi de tip boolean. Rezultatul aplicării lor este tot boolean ȳi reprezintă operaȳia logică de ȳi (&&) sau operaȳia logică de sau (| |) ȳntre cele două valori booleene. Iată toate posibilităȳile de combinare:

```
true && true == true
true && false == false
false && true == false
false && false == false
true || true == true
true || false == true
false || true == true
false || false == false
```

ȳn cazul operatorului && este evaluat mai ȳntâi operandul din stânga. Dacă acesta este fals, operandul din dreapta nu mai este evaluat, pentru că oricum rezultatul este fals. Acest lucru ne permite să testăm condiȳiile absolut necesare pentru corectitudinea unor operaȳii ȳi să nu executăm operaȳia decât dacă aceste condiȳii sunt ȳndeplinite.


De exemplu, dacă avem o referinȳă ȳi dorim să citim o valoare de variabilă din obiectul referit, trebuie să ne asigurăm că referinȳa este diferită de **null**. ȳn acest caz, putem scrie:

```
String s = "sir de caractere";
if( s != null && s.length < 5 ) ?
```

ȳn cazul ȳn care *s* este **null**, a doua operaȳie nu are sens ȳi nici nu va fi executată.

ȳn mod similar, la operatorul | |, se evaluează mai ȳntâi primul operand. Dacă acesta este adevărat, nu se mai evaluează ȳi cel de-al doilea operand pentru că rezultatul este oricum adevărat. Faptul se poate folosi ȳn mod similar ca mai sus:

```
if( s == null || s.length == 0 ) ?
```

În cazul în care s este **null**, nu se merge mai departe cu evaluarea. 

5.2.3.3 Operatorul condițional ?:

Este singurul operator definit de limbajul Java care acceptă trei operanzi. Operatorul primește o expresie condițională booleană pe care o evaluează și alte două expresii care vor fi rezultatul aplicării operandului. Care dintre cele două expresii este rezultatul adevărat depinde de valoarea rezultat a expresiei booleene. Forma generală a operatorului este:

ExpresieCondițională ? Expresie1 : Expresie2

Dacă valoarea expresiei condiționale este **true**, valoarea operației este valoarea expresiei 1. Altfel, valoarea operației este valoarea expresiei 2.

Cele două expresii trebuie să fie amândouă aritmetice sau amândouă booleene sau amândouă de tip referință.


Iată și un exemplu:

```
int i = 5;
int j = 4;
double f = ( i < j ) ? 100.5 : 100.4;
```

Parantezele nu sunt obligatorii.

După execuția instrucțiunilor de mai sus, valoarea lui f este 100.4. Iar după:

```
int a[] = { 1, 2, 3, 4, 5 };
int b[] = { 10, 20, 30 };
int k = ( ( a.length < b.length ) ? a : b )[0];
```

valoarea lui k va deveni 10. 

5.2.3.4 Operații întregi

Dacă amândoi operanzii unei operator sunt întregi atunci întreaga operație este întreagă.

Dacă unul dintre operanzi este întreg lung, operația se va face cu precizia de 64 de biți. Operanzii sunt eventual convertiți. Rezultatul este un întreg lung sau o valoare booleană dacă operatorul este un operator condițional.

Dacă nici unul dintre operanzi nu e întreg lung, operația se face întotdeauna pe 32 de biți, chiar dacă cei doi operanzi sunt întregi scurți sau octeți. Rezultatul este întreg sau boolean.

Dacă valoarea obținută este mai mare sau mai mică decât se poate reprezenta pe tipul rezultat, nu este semnalată nici o eroare de execuție, dar rezultatul este trunchiat. 🏠

5.2.3.5 Operatii flotante

Dacă un operand al unei operații este flotant, atunci întreaga operație este flotantă. Dacă unul dintre operanzi este flotant dublu, operația este pe flotanți dubli. Rezultatul este un flotant dublu sau boolean.

În caz de operații eronate nu se generează erori. În loc de aceasta se obține rezultatul NaN. Dacă într-o operație participă un NaN rezultatul este de obicei NaN.

În cazul testelor de egalitate, expresia

$$\text{NaN} == \text{NaN}$$

are întotdeauna rezultatul fals pentru că un NaN nu este egal cu nimic. La fel, expresia:

$$\text{NaN} != \text{NaN}$$

este întotdeauna adevărată.

În plus, întotdeauna expresia:

$$-0.0 == +0.0$$

este adevărată, unde +0.0 este zeroul pozitiv iar -0.0 este zeroul negativ. Cele două valori sunt definite în standardul IEEE 754.

În alte operații însă, zero pozitiv diferă de zero negativ. De exemplu 1.0 / 0.0 este infinit pozitiv iar 1.0 / -0.0 este infinit negativ. 🏠

5.2.3.6 Apeluri de metode

La apelul unei metode, valorile întregi nu sunt automat extinse la întreg sau la întreg lung ca la operații. Asta înseamnă că, dacă un operand este transmis ca întreg scurt de exemplu, el rămâne întreg scurt în interiorul metodei apelate.

[\[capitolul V\]](#)

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)

Capitolul V

5.3 Instrucțiuni

[5.3.1](#) *Blocuri de instrucțiuni*

[5.3.1.1](#) *Declarații de variabile locale*

[5.3.2](#) *Tipuri de instrucțiuni*

[5.3.2.1](#) *Instrucțiuni de atribuire*

[5.3.2.1.1](#) *Atribuire cu operație*

[5.3.2.1.2](#) *Atribuiri implicite*

[5.3.2.2](#) *Instrucțiuni etichetate*

[5.3.2.3](#) *Instrucțiuni condiționale*

[5.3.2.3.1](#) *Instrucțiunea if*

[5.3.2.3.2](#) *Instrucțiunea switch*

[5.3.2.4](#) *Instrucțiuni de ciclare*

[5.3.2.4.1](#) *Instrucțiunea while*

[5.3.2.4.2](#) *Instrucțiunea do*

[5.3.2.4.3](#) *Instrucțiunea for*

[5.3.2.5](#) *Instrucțiuni de salt*

[5.3.2.5.1](#) *Instrucțiunea break*

[5.3.2.5.2](#) *Instrucțiunea continue*

[5.3.2.5.3](#) *Instrucțiunea return*

[5.3.2.5.4](#) *Instrucțiunea throw*

[5.3.2.6](#) *Instrucțiuni de protecție*

[5.3.2.6.1](#) *Instrucțiunea try*

[5.3.2.6.2](#) *Instrucțiunea synchronized*

5.3.2.7 Instrucțiunea *vidă*

5.3.1 Blocuri de instrucțiuni

Un bloc de instrucțiuni este o secvență, eventual vidă, de instrucțiuni și declarații de variabile locale. Aceste instrucțiuni se execută în ordinea în care apar în interiorul blocului. Sintactic, blocurile de instrucțiuni sunt delimitate în sursă de caracterele { și }.

În limbajul Java, regula generală este aceea că oriunde putem pune o instrucțiune putem pune și un bloc de instrucțiuni, cu câteva excepții pe

care le vom sesiza la momentul potrivit, specificând în acest fel că instrucțiunile din interiorul blocului trebuiesc privite în mod unitar și tratate ca o singură instrucțiune. 🏠

5.3.1.1 Declarații de variabile locale

O declarație de variabilă locală introduce o nouă variabilă care poate fi folosită doar în interiorul blocului în care a fost definită. Declarația trebuie să conțină un nume și un tip. În plus, într-o declarație putem specifica o valoare inițială în cazul în care valoarea implicită a tipului variabilei, definită standard de limbajul Java, nu ne satisface.

Numele variabilei este un identificador Java. Acest nume trebuie să fie diferit de numele celorlalte variabile locale definite în blocul respectiv și de eventualii parametri ai metodei în interiorul căreia este declarat blocul.

De exemplu, este o eroare de compilare să declarăm cea de-a doua variabilă x în blocul:

```
{
int x = 3;
...
{
int x = 5;
...
}
}
```

Compilatorul va semnală faptul că deja există o variabilă cu acest nume în interiorul metodei. Eroarea de compilare apare indiferent dacă cele două variabile sunt de același tip sau nu.

Nu același lucru se întâmplă însă dacă variabilele sunt declarate în două blocuri de instrucțiuni complet disjuncte, care nu se includ unul pe celălalt. De exemplu, declarațiile următoare sunt perfect valide:

```

{
{
int x = 3;
...
}
...
{
int x = 5;
...
}
}

```

Practic, fiecare bloc de instrucțiuni definește un domeniu de existență a variabilelor locale declarate în interior. Dacă un bloc are subblocuri declarate în interiorul lui, variabilele din aceste subblocuri trebuie să fie distincte ca nume față de variabilele din superbloc. Aceste domenii se pot reprezenta grafic ca în figura următoare:

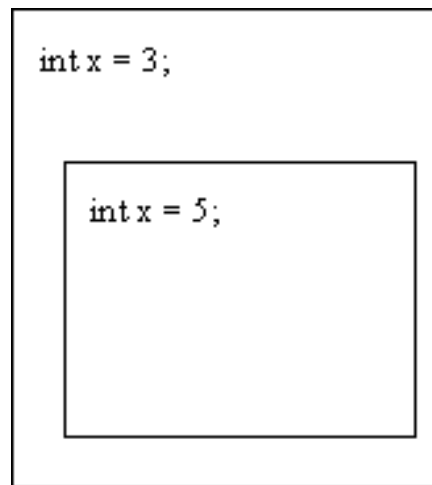


Figura 5.4 Reprezentarea grafică a domeniilor de existență a variabilelor incluse unul într-altul.

Figura reprezintă situația din primul exemplu. Observați că domeniul blocului interior este complet inclus în domeniul blocului exterior. Din această cauză, x apare ca fiind definită de două ori. În cel de-al doilea exemplu, reprezentarea grafică este următoarea:

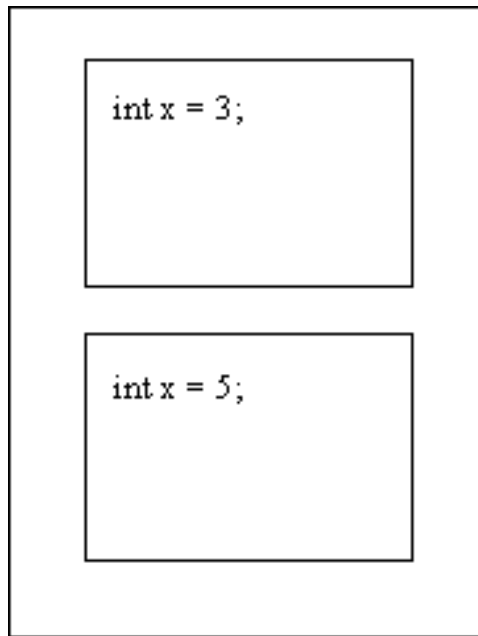


Figura 5.5 Reprezentarea grafică a domeniilor de existență a variabilelor disjuncte.

În acest caz, domeniile în care sunt definite cele două variabile sunt complet disjuncte și compilatorul nu va avea nici o dificultate în a identifica la fiecare referire a unei variabile cu numele x despre care variabilă este vorba. Să mai observăm că, în cel de-al doilea exemplu, referirea variabilelor x în blocul mare, în afara celor două subblocuri este o eroare de compilare.

În cazul în care blocul mare este blocul de implementare al unei metode și metoda respectivă are parametri, acești parametri sunt luați în considerare la fel ca niște declarații de variabile care apar chiar la începutul blocului. Acest lucru face ca numele parametrilor să nu poată fi folosit în nici o declarație de variabilă locală din blocul de implementare sau subblocuri ale acestuia.

În realitate, o variabilă locală poate fi referită în interiorul unui bloc abia după declarația ei. Cu alte cuvinte, domeniul de existență al unei variabile locale începe din punctul de declarație și continuă până la terminarea blocului.

Astfel, următoarea secvență de instrucțiuni:

```
{
x = 4;
int x = 3;
}
```

va determina o eroare de compilare, cu specificația că variabila x nu a fost încă definită în momentul primei atribuirii.

Acest mod de lucru cu variabilele face corectă următoarea secvență de instrucțiuni:


```

{
{
int x = 4;
}
int x = 3;
}

```

pentru că, în momentul declarației din interiorul subblocului, variabila x din exterior nu este încă definită. La ieșirea din subbloc, prima declarație își termină domeniul, așa că se poate defini fără probleme o nouă variabilă cu același nume. 🏠

5.3.2 Tipuri de instrucțiuni

5.3.2.1 Instrucțiuni de atribuire

O atribuire este o setare de valoare într-o locație de memorie. Forma generală a unei astfel de instrucțiuni este:

Locație = valoare ;

Specificarea locației se poate face în mai multe feluri. Cel mai simplu este să specificăm un nume de variabilă locală. Alte alternative sunt acelea de a specifica un element dintr-un tablou de elemente sau o variabilă care nu este declarată finală din interiorul unei clase sau numele unui parametru al unei metode.

Valoarea care trebuie atribuită poate fi un literal sau rezultatul evaluării unei expresii.

Instrucțiunea de atribuire are ca rezultat chiar valoarea atribuită. Din această cauză, la partea de valoare a unei operații de atribuire putem avea chiar o altă operație de atribuire ca în exemplul următor:

```

int x = 5;
int y = 6;
x = ( y = y / 2 );

```

Valoarea finală a lui x va fi identică cu valoarea lui y și va fi egală cu 3. Parantezele de mai sus sunt puse doar pentru claritatea codului, pentru a înțelege exact care este ordinea în care se execută atribuiri. În realitate, ele pot să lipsească pentru că, în mod implicit, Java va executa întâi atribuirea din dreapta. Deci, ultima linie se poate rescrie ca:

```

x = y = y / 2;

```

De altfel, gruparea inversă nici nu are sens:

```
( x = y ) = y / 2;
```

Această instrucțiune va genera o eroare de compilare pentru că, după executarea atribuirii din paranteze, rezultatul este valoarea 6 iar unei valori nu i se poate atribui o altă valoare.

În momentul atribuirii, dacă valoarea din partea dreaptă a operației nu este de același tip cu locația din partea stângă, compilatorul va încerca conversia valorii la tipul locației în modul pe care l-am discutat în paragraful referitor la conversii. Această conversie trebuie să fie posibilă, altfel compilatorul va semnală o eroare.

Pentru anumite conversii, eroarea s-ar putea să nu poată fi depistată decât în timpul execuției. În aceste cazuri, compilatorul nu va semnală eroare dar va fi semnalată o eroare în timpul execuției și rularea programului va fi abandonată. 🚫

5.3.2.1.1 Atribuire cu operație

Se întâmplă deseori ca valoarea care se atribuie unei locații să depindă de vechea valoare care era memorată în locația respectivă. De exemplu, în instrucțiunea:

```
int x = 3;
x = x * 4;
```

vechea valoare a lui x este înmulțită cu 4 și rezultatul înmulțirii este memorat înapoi în locația destinată lui x . Într-o astfel de instrucțiune, calculul adresei locației lui x este efectuat de două ori, o dată pentru a lua vechea valoare și încă o dată pentru a memora noua valoare. În realitate, acest calcul nu ar trebui executat de două ori pentru că locația variabilei x nu se schimbă în timpul instrucțiunii.

Pentru a ajuta compilatorul la generarea unui cod eficient, care să calculeze locația lui x o singură dată, în limbajul Java au fost introduse instrucțiuni mixte de calcul combinat cu atribuire. În cazul nostru, noua formă de scriere, mai eficientă, este:

```
int x = 3;
x *= 4;
```

Eficiența acestei exprimări este cu atât mai mare cu cât calculul locației este mai complicat. De exemplu, în secvența următoare:

```
int x = 3, y = 5;
double valori[] = new double[10];
valori[( x + y ) / 2] += 3.5;
```

calculul locației de unde se va lua o valoare la care se va aduna 3.5 și unde se va memora înapoi


rezultatul acestei operații este efectuat o singură dată. În acest exemplu, calculul locației presupune execuția expresiei:

$$(x + y) / 2$$


și indexarea tabloului numit *valori* cu valoarea rezultată. Valoarea din partea dreaptă poate fi o expresie arbitrar de complicată, ca în:

```
valori[x] += 7.0 * ( y * 5 );
```

Iată toți operatorii care pot fi mixați cu o atribuire:

`*`, `/`, `%`, `+`, `-`, `<=`, `>=`, `>>=`, `&=`, `|=`, `^=` 

5.3.2.1.2 Atribuiri implicite

Să mai observăm că, în cazul folosirii operatorilor de incrementare și decrementare se face și o atribuire implicită pentru că pe lângă valoarea rezultată în urma operației, se modifică și valoarea memorată la locația pe care se aplică operatorul. La fel ca și la operațiile mixte de atribuire, calculul locației asupra căreia se aplică operatorul se efectuează o singură dată. 

5.3.2.2 Instrucțiuni etichetate

Unele dintre instrucțiunile din interiorul unui bloc de instrucțiuni trebuie referite din altă parte a programului pentru a se putea direcționa execuția spre aceste instrucțiuni. Pentru referirea acestor instrucțiuni, este necesar ca ele să aibă o *etichetă* asociată. O etichetă este deci un nume dat unei instrucțiuni din program prin care instrucțiunea respectivă poate fi referită din alte părți ale programului.

Eticheta cea mai simplă este un simplu identificator urmat de caracterul `:` și de instrucțiunea pe care dorim să o etichetăm:

Etichetă: Instrucțiune


ca în exemplul următor:

```
int a = 5;
Eticheta: a = 3;
```


Pentru instrucțiunea **switch**, descrisă mai jos, sunt definite două moduri diferite de etichetare. Acestea au forma:

case Valoare: Instrucțiune

default: Instrucțiune

Exemple pentru folosirea acestor forme sunt date la definirea instrucțiunii. 

5.3.2.3 Instrucțiuni condiționale

Instrucțiunile condiționale sunt instrucțiuni care selectează pentru execuție o instrucțiune dintr-un set de instrucțiuni în funcție de o anumită condiție. 

5.3.2.3.1 Instrucțiunea if

Instrucțiunea **if** primește o expresie a cărei valoare este obligatoriu de tipul boolean. Evaluarea acestei expresii poate duce la doar două valori: adevărat sau fals. În funcție de valoarea rezultată din această evaluare, se execută unul din două seturi de instrucțiuni distincte, specificate de instrucțiunea **if**.

Sintaxa acestei instrucțiuni este următoarea:

if(*Expresie*) *Instrucțiune1* [**else** *Instrucțiune2*]

După evaluarea expresiei booleene, dacă valoarea rezultată este **true**, se execută instrucțiunea 1.

Restul instrucțiunii este opțional, cu alte cuvinte partea care începe cu cuvântul rezervat **else** poate să lipsească. În cazul în care această parte nu lipsește, dacă rezultatul evaluării expresiei este **false** se va executa instrucțiunea 2.

Indiferent de instrucțiunea care a fost executată în interiorul unui **if**, după terminarea acesteia execuția continuă cu instrucțiunea de după instrucțiunea **if**, în afară de cazul în care instrucțiunile executate conțin în interior o instrucțiune de salt.

Să mai observăm că este posibil ca într-o instrucțiune **if** să nu se execute nici o instrucțiune în afară de evaluarea expresiei în cazul în care expresia este falsă iar partea **else** din instrucțiunea **if** lipsește. Expresia booleană va fi întotdeauna evaluată.

Iată un exemplu de instrucțiune **if** în care partea **else** lipsește:

```
int x = 3;
if( x == 3 )
    x *= 7;
```

Ți iată un exemplu în care sunt prezente ambele părți:

```
int x = 5;
if( x % 2 == 0 )
x = 100;
else
x = 1000;
```

Instrucțiunea 1 nu poate lipsi niciodată. Dacă totuși pe ramura de adevăr a instrucțiunii condiționale nu dorim să executăm nimic, putem folosi o instrucțiune vidă, după cum este arătat puțin mai departe.


În cazul în care dorim să executăm mai multe instrucțiuni pe una dintre ramurile instrucțiunii **if**, putem să înlocuim instrucțiunea 1 sau 2 sau pe amândouă cu blocuri de instrucțiuni, ca în exemplul următor:

```
int x = 5;
if( x == 0 ) {
x = 3;
y = x * 5;
} else {
x = 5;
y = x * 7;
}
```

Expresia booleană poate să fie și o expresie compusă de forma:

```
int x = 3;
int y = 5;
if( y != 0 && x / y == 2 )
x = 4;
```

În acest caz, așa cum deja s-a specificat la descrierea operatorului **&&**, evaluarea expresiei nu este terminată în toate situațiile. În exemplul nostru, dacă y este egal cu 0, evaluarea expresiei este oprită și rezultatul este fals. Această comportare este corectă pentru că, dacă un termen al unei operații logice de conjuncție este fals, atunci întreaga conjuncție este falsă.

În plus, acest mod de execuție ne permite să evităm unele operații cu rezultat incert, în cazul nostru o împărțire prin 0. Pentru mai multe informații relative la operatorii **&&** și **||** citiți secțiunea destinată operatorilor. 

5.3.2.3.2 Instrucțiunea switch

Instrucțiunea **switch** ne permite saltul la o anumită instrucțiune etichetată în funcție de valoarea unei

expresii. Putem să specificăm câte o etichetă pentru fiecare valoare particulară a expresiei pe care dorim să o diferențiem. În plus, putem specifica o etichetă la care să se facă saltul implicit, dacă expresia nu ia nici una dintre valorile particulare specificate.

Sintaxa instrucțiunii este:

```
switch( Expresie ) {

    [case ValoareParticulară: Instrucțiuni]*

    [default: InstrucțiuniImplicite;]

}
```

Execuția unei instrucțiuni **switch** începe întotdeauna prin evaluarea expresiei dintre parantezele rotunde. Această expresie trebuie să aibă tipul caracter, octet, întreg scurt sau întreg. După evaluarea expresiei se trece la compararea valorii rezultate cu valorile particulare specificate în etichetele **case** din interiorul blocului de instrucțiuni. Dacă una dintre valorile particulare este egală cu valoarea expresiei, se execută instrucțiunile începând de la eticheta **case** corespunzătoare acelei valori în jos, până la capătul blocului. Dacă nici una dintre valorile particulare specificate nu este egală cu valoarea expresiei, se execută instrucțiunile care încep cu eticheta **default**, dacă aceasta există.

Iată un exemplu de instrucțiune **switch**:

```
int x = 4;
...
int y = 0;
switch( x + 1 ) {
case 3:
x += 2;
y++;
case 5:
x = 11;
y++;
default:
x = 4;
y += 3;
}
```

Dacă valoarea lui x în timpul evaluării expresiei este 2 atunci expresia va avea valoarea 3 și instrucțiunile vor fi executate una după alta începând cu cea etichetată cu **case** 3. În ordine, x va deveni 4, y va deveni 1, x va deveni 11, y va deveni 2, x va deveni 4 și y va deveni 5.

Dacă valoarea lui x în timpul evaluării expresiei este 4 atunci expresia va avea valoarea 5 și instrucțiunile vor fi executate pornind de la cea etichetată cu **case** 5. În ordine, x va deveni 11, y va deveni 1, x va deveni 4 și y va deveni 4.

În fine, dacă valoarea lui x în timpul evaluării expresiei este diferită de 2 și 4, se vor executa instrucțiunile începând cu cea etichetată cu **default**. În ordine, x va deveni 4 și y va deveni 3.

Eticheta **default** poate lipsi, caz în care, dacă nici una dintre valorile particulare nu este egală cu valoarea expresiei, nu se va executa nici o instrucțiune din bloc.


În cele mai multe cazuri, această comportare a instrucțiunii **switch** nu ne convine, din cauza faptului că instrucțiunile de după cea etichetată cu **case** 5 se vor executa și dacă valoarea este 3. La fel, instrucțiunile de după cea etichetată cu **default** se execută întotdeauna. Pentru a schimba această comportare trebuie să folosim una dintre instrucțiunile de salt care să oprească execuția instrucțiunilor din bloc înainte de întâlnirea unei noi instrucțiuni etichetate.

Putem de exemplu folosi instrucțiunea de salt **break** care va opri execuția instrucțiunilor din blocul **switch**. De exemplu:

```
char c = '\t';
...
String mesaj = "nimic";
switch( c ) {
case '\t':
    mesaj = "tab";
    break;
case '\n':
    mesaj = "linie noua";
    break;
case '\r':
    mesaj = "retur";
default:
    mesaj = mesaj + " de";
    mesaj = mesaj + " car";
}
```

În acest caz, dacă c este egal cu caracterul tab, la terminarea instrucțiunii **switch**, *mesaj* va avea valoarea "tab". În cazul în care c are valoarea CR, *mesaj* va deveni mai întâi "retur" apoi "retur de" și apoi "retur de car". Lipsa lui **break** după instrucțiunea

```
mesaj = "retur";
```

face ca în continuare să fie executate și instrucțiunile de după cea etichetată cu **default**. 

5.3.2.4 Instrucțiuni de ciclare

Instrucțiunile de ciclare (sau ciclurile, sau buclele) sunt necesare atunci când dorim să executăm de mai multe ori aceeași instrucțiune sau același bloc de instrucțiuni. Necesitatea acestui lucru este evidentă dacă ne gândim că programele trebuie să poată reprezenta acțiuni de forma: execută 10 întoarceri, execută 7 genoflexiuni, execută flotări până ai obosit, etc.

Desigur, sintaxa instrucțiunilor care se execută trebuie să fie aceeași, pentru că ele vor fi în realitate scrise în Java o singură dată. Totuși, instrucțiunile nu sunt neapărat aceleași. De exemplu, dacă executăm în mod repetat instrucțiunea:

```
int tablou[] = new int[10];
int i = 0;
tablou[i++] = 0;
```

în realitate se va memora valoarea 0 în locații diferite pentru că variabila care participă la calculul locației își modifică la fiecare iterație valoarea. La primul pas, se va face 0 primul element din tablou și în același timp *i* va primi valoarea 1. La al doilea pas, se va face 0 al doilea element din tablou și *i* va primi valoarea 2, și așa mai departe.

Lucrurile par și mai clare dacă ne gândim că instrucțiunea executată în mod repetat poate fi o instrucțiune **if**. În acest caz, în funcție de expresia condițională din **if** se poate executa o ramură sau alta a instrucțiunii.

De exemplu instrucțiunea din buclă poate fi:

```
int i = 0;
if( i++ % 2 == 0 )
...
else
...

```

În acest caz, *i* este când par când impar și se execută alternativ cele două ramuri din **if**. Desigur, comportarea descrisă este valabilă dacă valoarea lui *i* nu este modificată în interiorul uneia dintre ramuri.



5.3.2.4.1 Instrucțiunea while

Această instrucțiune de buclare se folosește atunci când vrem să executăm o instrucțiune atâta timp cât o anumită expresie condițională rămâne adevărată. Expresia condițională se evaluează și testează înainte

de execuția instrucțiunii, astfel că, dacă expresia era de la început falsă, instrucțiunea nu se mai execută niciodată.

Sintaxa acestei instrucțiuni este:

```
while( Test ) Corp
```

Test este o expresie booleană iar *Corp* este o instrucțiune normală, eventual vidă. Dacă avem nevoie să repetăm mai multe instrucțiuni, putem înlocui corpul buclei cu un bloc de instrucțiuni.

Iată ȳi un exemplu:

```
int i = 0;
int tablou[] = new int[20];
while( i < 10 )
    tablou[i++] = 1;
```


Buclea **while** de mai sus se execută de 10 ori primele 10 elemente din tablou fiind inițializate cu 1. În trecut fie spus, celelalte rămân la valoarea 0 care este valoarea implicită pentru întregi. După cei 10 pași iterativi, *i* devine 10 și testul devine fals ($10 < 10$).

În exemplul următor, corpul nu se execută nici măcar o dată:

```
int i = 3;
while( i < 3 )
    i++;
```

Putem să creăm un *ciclu infinit* (care nu se termină niciodată) prin:

```
while( true )
    ;
```

Înteruperea execuției unui ciclu infinit se poate face introducând în corpul ciclului o instrucțiune de salt. 

5.3.2.4.2 Instrucțiunea do

Buclele **do** se folosesc atunci când testul de terminare a buclei trebuie făcut după execuția corpului buclei. Sintaxa de descriere a instrucțiunii **do** este:

```
do Corp while( Test );
```

Test este o expresie booleană iar *Corp* este o instrucțiune sau un bloc de instrucțiuni. Execuția acestei instrucțiuni înseamnă execuția corpului în mod repetat atâta timp cât expresia *Test* are valoarea adevărat. Testul se evaluează după execuția corpului, deci corpul se execută cel puțin o dată.

De exemplu, în instrucțiunea:

```
int i = 3;
do
i++;
while( false );
```

valoarea finală a lui *i* este 4, pentru că instrucțiunea *i++* care formează corpul buclei se execută o dată chiar dacă testul este întotdeauna fals.

În instrucțiunea:

```
int i = 1;
do {
tablou[i] = 0;
i += 2;
} while( i < 5 );
```

sunt setate pe 0 elementele 1 și 3 din tablou. După a doua iterație, *i* devine 5 și testul evaluează cauzând terminarea iterației. 🚧

5.3.2.4.3 Instrucțiunea for

Instrucțiunea **for** se folosește atunci când putem identifica foarte clar o parte de inițializare a buclei, testul de terminare a buclei, o parte de reluare a buclei și un corp pentru buclă. În acest caz, putem folosi sintaxa:

for(*Inițializare Test ; Reluare*) *Corp*

Corp și *Inițializare* sunt instrucțiuni normale. *Test* este o expresie booleană iar *Reluare* este o instrucțiune căreia îi lipsește caracterul **;** final.

Execuția unei bucle for începe cu execuția instrucțiunii de inițializare. Această instrucțiune stabilește de obicei niște valori pentru variabilele care controlează bucla. Putem chiar declara aici noi variabile. Aceste variabile există doar în interiorul corpului buclei și în instrucțiunile de test și reluare ale buclei.

În partea de inițializare nu putem scrie decât o singură instrucțiune fie ea declarație sau instrucțiune normală. În acest caz instrucțiunea nu se poate înlocui cu un bloc de instrucțiuni. Putem însă să

declaram două variabile cu o sintaxă de forma:

```
int i = 0, j = 1;
```

După execuția părții de inițializare se pornește bucla propriu-zisă. Aceasta constă din trei instrucțiuni diferite executate în mod repetat. Cele trei instrucțiuni sunt testul, corpul buclei și instrucțiunea de reluare. Testul trebuie să fie o expresie booleană. Dacă aceasta este evaluată la valoarea adevărat, bucla continuă cu execuția corpului, a instrucțiunii de reluare și din nou a testului. În clipa în care testul are valoarea fals, bucla este oprită fără să mai fie executat corpul sau reluarea.

Iată un exemplu:

```
int x = 0;
for( int i = 3; i < 30; i += 10 )
    x += i;
```

În această buclă se execută mai întâi crearea variabilei i și inițializarea acesteia cu 3. După aceea se testează variabila i dacă are o valoare mai mică decât 30. Testul are rezultat adevărat (i este $0 < 30$) și se trece la execuția corpului unde x primește valoarea 3 ($0 + 3$). În continuare se execută partea de reluare în care i este crescut cu 10, devenind 13. Se termină astfel primul pas al buclei și aceasta este reluată începând cu testul care este în continuare adevărat ($13 < 30$). Se execută corpul, x devenind 16 ($3 + 13$), și reluarea, unde x devine 23 ($13 + 10$). Se reia bucla de la test care este în continuare adevărat ($23 < 30$). Se execută corpul unde x devine 39 ($16 + 23$) și reluarea unde i devine 33 ($23 + 10$). Se reia testul care în acest caz devine fals ($33 < 30$) și se părăsește bucla, continuându-se execuția cu prima instrucțiune de după buclă.

După ieșirea din buclă, variabila i nu mai există, deci nu se mai poate folosi și nu putem vorbi despre valoarea cu care iese din buclă, iar variabila x rămâne cu valoarea 39.

Pentru a putea declara variabila i în instrucțiunea de inițializare a buclei `for` este necesar ca în blocurile superioare instrucțiunii `for` să nu existe o altă variabilă i , să nu existe un parametru numit i și nici o etichetă cu acest nume.

Dacă dorim un corp care să conțină mai multe instrucțiuni, putem folosi un bloc. Nu putem face același lucru în partea de inițializare sau în partea de reluare.

Oricare dintre părțile buclei **for** în afară de inițializare poate să lipsească. Dacă aceste părți lipsesc, se consideră că ele sunt reprezentate de instrucțiunea vidă. Dacă nu dorim inițializare trebuie totuși să specificăm implicit instrucțiunea vidă. Putem de exemplu să scriem o buclă infinită prin:

```
int x;
for( ; ; )
```

```
x = 0;
```

Putem specifica funcționarea instrucțiunii **for** folosindu-ne de o instrucțiune **while** în felul următor:

Inițializare

```
while( Test ) {
```

Corp

```
    Reluare ;
```

```
}
```

În fine, schema următoare reprezintă funcționarea unei bucle **for**:

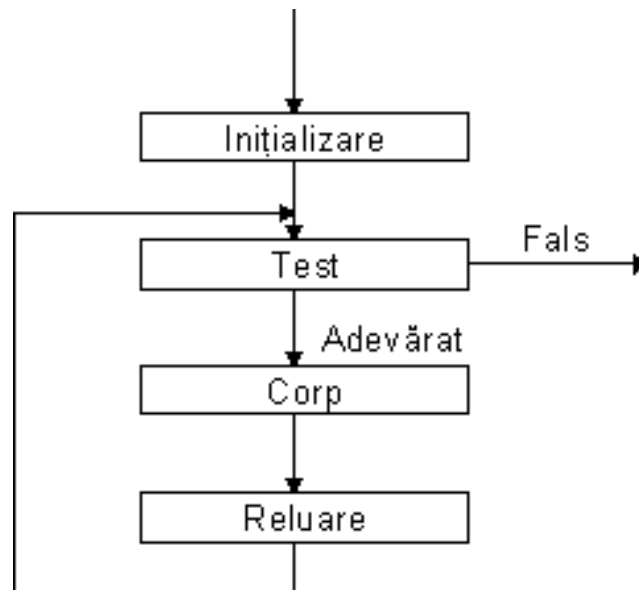


Figura 5.6 Schema de funcționare a buclei for. 🏠

5.3.2.5 Instrucțiuni de salt

Instrucțiunile de salt provoacă întreruperea forțată a unei bucle, a unui bloc de instrucțiuni sau ieșirea dintr-o metodă. Instrucțiunile de salt sunt destul de periculoase pentru că perturbă curgerea uniformă a programului ceea ce poate duce la o citire și înțelegere eronată a codului rezultat. 🏠

5.3.2.5.1 Instrucțiunea **break**

Instrucțiunea **break** produce întreruperea unei bucle sau a unui bloc **switch**. Controlul este dat instrucțiunii care urmează imediat după bucla întreruptă sau după blocul instrucțiunii **switch**.

De exemplu în secvența:

```
int i = 1, j = 3;
int tablou[] = new int[10];
while( i < 10 ) {
    tablou[i++] = 0;
    if( i == j )
        break;
}
i += 2;
```

sunt setate pe 0 elementele 1 și 2 ale tabloului. După a doua iterație i devine 3 și testul $i == j$ are valoarea adevărat, producându-se execuția instrucțiunii **break**.

Instrucțiunea **break** cauzează ieșirea forțată din buclă, chiar dacă testul buclei $i < 10$ este în continuare valid. La terminarea tuturor instrucțiunilor de mai sus, i are valoarea 5.

În exemplul următor:

```
int i, j = 3;
int tablou[] = new int[10];
do {
    tablou[i] = 0;
    if( i++ >= j )
        break;
} while( i < 10 );
```

sunt setate pe 0 elementele 0, 1, 2 și 3 din tablou. Dacă vă întrebați la ce mai folosește testul buclei, atunci să spunem că este o măsură suplimentară de siguranță. Dacă cumva j intră cu o valoare greșită, testul ne asigură în continuare că nu vom încerca să setăm un element care nu există al tabloului. De fapt, pericolul există încă, dacă valoarea inițială a lui i este greșită, deci testul ar trebui mutat la început și bucla transformată într-un **while**, ca cel de mai sus.

În cazul buclelor **for**, să mai precizăm faptul că la o ieșire forțată nu se mai execută instrucțiunea de reluare.

Instrucțiunea **break** poate avea ca argument opțional o etichetă, ca în:

```
break Identificator;
```


În acest caz, identificatorul trebuie să fie eticheta unei instrucțiuni care să includă instrucțiunea **break**.

Prin faptul că instrucțiunea include instrucțiunea **break**, înțelegem că instrucțiunea **break** apare în corpul instrucțiunii care o include sau în corpul unei instrucțiuni care se găsește în interiorul corpului instrucțiunii care include instrucțiunea **break**. Controlul revine instrucțiunii de după instrucțiunea etichetată cu identificatorul specificat.

De exemplu, în instrucțiunile:

```
int i, j;
asta: while( i < 3 ) {
do {
i = j + 1;
if( i == 3 )
break asta;
} while( j++ < 3 );
}
j = 10;
```

instrucțiunea **break** termină bucla **do** și bucla **while** controlul fiind dat instrucțiunii de după **while**, și anume atribuirea:

```
j = 10; 
```

5.3.2.5.2 Instrucțiunea continue

Instrucțiunea **continue** permite reluarea unei bucle fără a mai termina execuția corpului acesteia. Reluarea se face ca și cum corpul buclei tocmai a fost terminat de executat. Bucla nu este părăsită.

De exemplu, în instrucțiunea:

```
int i;
while( i < 10 ) {
i++;
continue;
i++;
}
```

corpul buclei se execută de 10 ori, pentru că a doua incrementare a lui *i* nu se execută niciodată. Execuția începe cu testul și apoi urmează cu prima incrementare. După aceasta se execută instrucțiunea **continue** care duce la reluarea buclei, pornind de la test și urmând cu incrementarea și din nou instrucțiunea **continue**.

În exemplul următor:

```

int i;
do {
    i++;
    continue;
    i++;
} while( i < 10 );

```

corpul se execută de 10 ori la fel ca ȳi mai sus. Instrucțiunea **continue** duce la evitarea celei de-a doua incrementări, dar nu ȳi la evitarea testului de sfârșit de buclă.

În sfârșit, în exemplul următor:

```

for( int i = 0; i < 10; i++ ) {
    continue;
    i++;
}

```

corpul se execută tot de 10 ori, ceea ce înseamnă că reluarea buclei duce la execuția instrucțiunii de reluare a buclei **for** ȳi apoi a testului. Doar ceea ce este în interiorul corpului este evitat.

Instrucțiunea **continue** poate avea, la fel ca ȳi instrucțiunea **break**, un identificator opțional care specifică eticheta buclei care trebuie continuată. Dacă există mai multe bucle imbricate una în cealaltă buclele interioare celei referite de etichetă sunt abandonate.

De exemplu, în secvența:

```

asta: for( int i, j = 1; i < 10; i++ ) {
    while( j < 5 ) {
        j++;
        if( j % 2 == 0 )
            continue asta;
    }
}

```

instrucțiunea **continue** provoacă abandonarea buclei **while** ȳi reluarea buclei **for** cu partea de reluare ȳi apoi testul. 🚧

5.3.2.5.3 Instrucțiunea return

Instrucțiunea **return** provoacă părăsirea corpului unei metode. În cazul în care **return** este urmată de o expresie, valoarea expresiei este folosită ca valoare de retur a metodei. Această valoare poate fi eventual convertită către tipul de valoare de retur declarat al metodei, dacă acest lucru este posibil. Dacă

nu este posibil, va fi semnalată o eroare de compilare.

Este o eroare de compilare specificarea unei valori de retur într-o instrucțiune **return** din interiorul unei metode care este declarată **void**, cu alte cuvinte care nu întoarce nici o valoare.

Instrucțiunea **return** fără valoare de retur poate fi folosită și pentru a părăsi execuția unui inițializator static.

Exemple de instrucțiuni **return** veți găsi în secțiunea care tratează metodele unei clase de obiecte. 🏠

5.3.2.5.4 Instrucțiunea throw

Instrucțiunea **throw** este folosită pentru a semnaliza o excepție de execuție. Această instrucțiune trebuie să aibă un argument și acesta trebuie să fie un tip obiect, de obicei dintr-o subclasă a clasei de obiecte **Exception**.

La execuția instrucțiunii **throw**, fluxul normal de execuție este părăsit și se termină toate instrucțiunile în curs până la prima instrucțiune **try** care specifică într-o clauză **catch** un argument formal de același tip cu obiectul aruncat sau o superclasă a acestuia. 🏠

5.3.2.6 Instrucțiuni de protecție

Aceste instrucțiuni sunt necesare pentru tratarea erorilor și a excepțiilor precum și pentru sincronizarea unor secvențe de cod care nu pot rula în paralel. 🏠

5.3.2.6.1 Instrucțiunea try

Instrucțiunea **try** inițiază un context de tratare a excepțiilor. În orice punct ulterior inițializării acestui context și înainte de terminarea acestuia, o excepție semnalată prin execuția unei instrucțiuni **throw** va returna controlul la nivelul instrucțiunii **try**, abandonându-se în totalitate restul instrucțiunilor din corpul acestuia.

La semnalarea unei excepții aceasta poate fi prinsă de o clauză **catch** și, în funcție de tipul obiectului aruncat, se pot executa unele instrucțiuni care repun programul într-o stare stabilă. De obicei, o excepție este generată atunci când s-a produs o eroare majoră și continuarea instrucțiunilor din contextul curent nu mai are sens.

În finalul instrucțiunii, se poate specifica și un bloc de instrucțiuni care se execută imediat după blocul **try** și blocurile **catch** indiferent cum s-a terminat execuția acestora. Pentru specificarea acestor instrucțiuni, trebuie folosită o clauză **finally**.


Iată sintaxa unei instrucțiuni **try**:

try *Bloc1* [**catch**(*Argument*) *Bloc2*]*[**finally** *Bloc3*]

Dacă, undeva în interiorul blocului 1 sau în metodele apelate din interiorul acestuia, pe oricâte nivele, este apelată o instrucțiune **throw**, execuția blocului și a metodelor în curs este abandonată și se revine în instrucțiunea **try**. În continuare, obiectul aruncat de **throw** este comparat cu argumentele specificate în clauzele **catch**. Dacă unul dintre aceste argumente este instanță a aceleiași clase sau a unei superclase a clasei obiectului aruncat, se execută blocul de instrucțiuni corespunzător clauzei **catch** respective. Dacă nici una dintre clauzele **catch** nu se potrivește, obiectul este aruncat mai departe. Dacă excepția nu este nicăieri prinsă în program, acesta se termină cu o eroare de execuție.

Indiferent dacă a apărut o excepție sau nu, indiferent dacă s-a executat blocul unei clauze **catch** sau nu, în finalul execuției instrucțiunii **try** se execută blocul specificat în clauza **finally**, dacă aceasta există.

Clauza **finally** se execută chiar și dacă în interiorul blocului 1 s-a executat o instrucțiune **throw** care a aruncat un obiect care nu poate fi prins de clauzele **catch** ale acestei instrucțiuni **try**. În astfel de situații, execuția instrucțiunii **throw** se oprește temporar, se execută blocul **finally** și apoi se aruncă mai departe excepția.

Exemple de utilizare a instrucțiunii **try** găsiți în paragraful 9.2 

5.3.2.6.2 Instrucțiunea **synchronized**

Instrucțiunea **synchronized** introduce o secvență de instrucțiuni critică. *O secvență critică de instrucțiuni* trebuie executată în așa fel încât nici o altă parte a programului să nu poată afecta obiectul cu care lucrează secvența dată. Secvențele critice apar de obicei atunci când mai multe părți ale programului încearcă să acceseze în același timp aceleași resurse.


Gândiți-vă, de exemplu, ce s-ar întâmpla dacă mai multe părți ale programului ar încerca să incrementeze în același timp valoarea unei variabile. Una dintre ele ar citi vechea valoare a variabilei, să spunem 5, ar incrementa-o la 6 și, când să o scrie înapoi, să presupunem că ar fi întreruptă de o altă parte a programului care ar incrementa-o la 6. La revenirea în prima parte, aceasta ar termina prima incrementare prin scrierea valorii 6 înapoi în variabilă. Valoarea finală a variabilei ar fi 6 în loc să fie 7 așa cum ne-am așteptat dacă cele două incrementări s-ar face pe rând.

Spunem că cele două regiuni în care se face incrementarea aceleiași variabile sunt regiuni critice. Înainte ca una dintre ele să se execute, ar trebui să ne asigurăm că cealaltă regiune critică nu rulează deja. Cea mai simplă cale de a face acest lucru este să punem o condiție de blocare chiar pe variabila incrementată. Ori de câte ori o regiune critică va încerca să lucreze, va verifica dacă variabila noastră este liberă sau nu.

Instrucțiunea **synchronized** îi blochează obiectul pe care îl primește ca parametru și apoi execută secvența critică. La sfârșitul acesteia obiectul este deblocat înapoi. Dacă instrucțiunea nu poate bloca imediat obiectul pentru că acesta este blocat de o altă instrucțiune, așteaptă până când obiectul este deblocat.

Mai mult despre această instrucțiune precum și exemple de utilizare veți găsi în partea a treia, capitolul 9. Până atunci, iată sintaxa generală a acestei instrucțiuni:

synchronized (*Expresie*) *Instrucțiune*

Expresia trebuie să aibă ca valoare o referință către un obiect sau un tablou care va servi drept dispozitiv de blocare. Instrucțiunea poate fi o instrucțiune simplă sau un bloc de instrucțiuni. 

5.3.2.7 Instrucțiunea vidă


Instrucțiunea vidă este o instrucțiune care nu execută nimic. Ea este folosită uneori, atunci când este obligatoriu să avem o instrucțiune, dar nu dorim să executăm nimic în acea instrucțiune. De exemplu, în cazul unei instrucțiuni **if**, este obligatoriu să avem o instrucțiune pe ramura de adevăr. Dacă însă nu dorim să executăm nimic acolo, putem folosi un bloc vid sau o instrucțiune vidă.

Sintaxa pentru o instrucțiune vidă este următoarea:

;

Iată și un exemplu:

```
int x = 3;
if( x == 5 ) ; else x = 5;
```

Caracterul **;** care apare după condiția din **if** reprezintă o instrucțiune vidă care specifică faptul că, în cazul în care *x* are valoarea 5 nu trebuie să se execute nimic. 

[\[capitolul V\]](#)

[\[cuprins\]](#)

[\(C\) IntegraSoft 1996-1998](#)