

1.1 CALCULATOARE ȘI PROGRAMAREA LOR

1.1.1 Structura unui calculator

Un *calculator* este un dispozitiv capabil să execute calcule și să ia decizii logice cu o viteză de execuție incomparabil mai mare decât viteza cu care le-am face noi, oamenii. Sute de milioane de adunări pe secundă este o performanță pe care nici un om nu o poate atinge, dar pe care calculatoarele o pot realiza și chiar depăși.

Calcululele realizate de calculator se efectuează asupra *datelor*, adică asupra unor informații codificate într-o formă convenabilă calculului. Aceste informații pot fi numere, caractere simple, șiruri de caractere sau orice alte informații reprezentând mărimi cuantificabile. Forma convenabilă a datelor din calculator este forma binară: combinații liniare de 0 și 1 de diverse lungimi.

Operațiile din care este alcătuit calculul și ordinea efectuării lor sunt specificate în *programul* calculatorului. Orice calculator funcționează pe baza unui program, adică a unei secvențe de *instrucțiuni-mașină*, fiecare instrucțiune indicând o operație elementară, specifică acelui calculator și care urmează a fi efectuată asupra unor date memorate de calculator.

Programele sunt concepute și scrise de *programatori*, dar pentru a fi executate, ele trebuie mai întâi memorate în calculator împreună cu datele de elaborat într-un același format binar. Execuția instrucțiunilor din program este asigurată de *unitatea de prelucrare centrală* (CPU), iar înmagazinarea datelor de prelucrat sau a rezultatelor operațiilor se face în *memoria* calculatorului.

CPU îndeplinește următoarele funcțiuni:

- extrage una după alta instrucțiunile din programul de executat,
- interpretează și execută instrucțiuni, efectuând operații aritmetice, logice și de alte tipuri după ce mai înainte și-a procurat datele necesare din memorie,
- comandă celelalte componente,
- supraveghează modul în care acestea își îndeplinesc sarcinile.

Orice calculator are un *set de instrucțiuni-mașină*, limitat ca număr de instrucțiuni, și specific tipului de CPU utilizat. De multe ori se utilizează termenul de *procesor* sau *microprocesor* pentru a identifica CPU.

Memoria este locul în care programatorul poate stoca date și programe. Există două tipuri de memorie:

- *memoria primară* alcătuită din chip-uri de *memorie RAM* (*random access memory*) din care se poate citi și în care se poate scrie, și o *memorie ROM* (*read only memory*) din care se poate doar citi. Memoria RAM este o mulțime ordonată de celule identice identificate în mod univoc printr-un număr întreg, nenegativ numit *adresa celulei*. Într-o celulă se pot memora date dintr-un domeniu finit de valori, determinat de capacitatea de memorare limitată a celulei. Memoria RAM este *volatilă* în sensul că oprit calculatorul programul, conținutul ei se pierde. Memoria ROM servește la găzduirea unor programe și date de configurare a calculatorului necesare la pornirea acestuia;
- *memoria secundară* formată din spațiile de memorare oferite de dispozitive interne sau externe calculatorului cum ar fi: discuri magnetice (hard disk sau floppy disk), discuri optice, flash memory etc. Memoria secundară este mult mai lentă decât memoria primară, dar are o capacitate de memorare mult mai mare decât a acesteia. În plus, ea este de regulă memorie persistentă: la oprirea calculatorului datele memorate în memoria secundară nu se pierd și se regăsesc intacte la repornirea calculatorului. Organizarea datelor în această memorie este bazată pe noțiunea de *file system*, în esență o structură arborescentă de containere numite *directoare* (directories) care găzduiesc *fișiere* (files) și alte directoare. Un fișier este o secvență de celule la care accesul în *citire* sau în *scriere* se poate face prin adresa celulei (random access file) sau strict secvențial (sequential file).

Un calculator este conectat la *dispozitive periferice* sau *dispozitive de intrare/ieșire* (dispozitive I/O) necesare pentru a putea interacționa cu utilizatorul când este vorba de obținerea de date necesare calculului (intrare de date) sau de prezentarea datelor rezultate din calcule (ieșire de date). Cele mai importante dispozitive periferice sunt: *tastatura* (keyboard) și *mouse-ul* pentru intrări și *ecranul* (display) și *imprimanta* (printer) pentru ieșiri. Mai există și alte dispozitive I/O precum: modem-ul cu care calculatorul poate comunica cu alte calculatoare folosind linii telefonice, scanner-ul pentru a putea captura imagini etc. Discurile magnetice și, în general, dispozitivele ce oferă memorie

externă sunt și ele dispozitive de intrare/ieșire deoarece permit citirea și scrierea datelor pe care le memorează în mod persistent. În toate cazurile de operații de intrare/ieșire calculatorul folosește niște componente care se interpun între CPU și memorie, pe de o parte, și dispozitive, pe de altă parte, și care sunt specializate în a lucra cu respectivele dispozitive. De exemplu:

- pentru a citi de pe sau scrie pe discuri, calculatorul folosește un *controller* legat la una sau mai multe unități de discuri magnetice sau optice pe care le comandă și controlează în cursul operațiilor I/O;
- pentru a interacționa cu mouse-ul, tastatura, imprimanta sau modem-ul, calculatorul folosește niște *porți* de acces combinate cu un sistem de prelucrări bazate pe întreruperi care se activează automat oricând are un eveniment de comunicație între calculator și dispozitive;
- pentru a afișa pe ecran mesaje, desene, ferestre sau imagini, calculatorul folosește o *memorie video* aflată pe *placa video* (video card) specializată în gestionarea imaginilor pe ecran. Memoria video este "scrisă" celulă cu celulă pe baza comenzilor primite de la CPU și apoi informația scrisă este vizualizată pe ecran sub formă de puncte luminoase de diverse culori și intensități (pixels);
- pentru a comunica cu alte calculatoare în rețea, calculatorul folosește un *adaptor de rețea* aflat pe *placa de rețea* și specializat în captarea semnalelor care parcurg rețeaua, reținând doar cele care interesează calculatorul în care este instalată. Aceste semnale, transformate și buffer-izate (memorate temporar într-o memorie locală – buffer – pentru a le transmite toate dintr-odată când memoria locală s-a umplut), sunt trimise apoi către CPU și memorie. În mod analog, adaptorul de rețea injectează în rețea secvențe de date destinate altor calculatoare cu care calculatorul este legat.

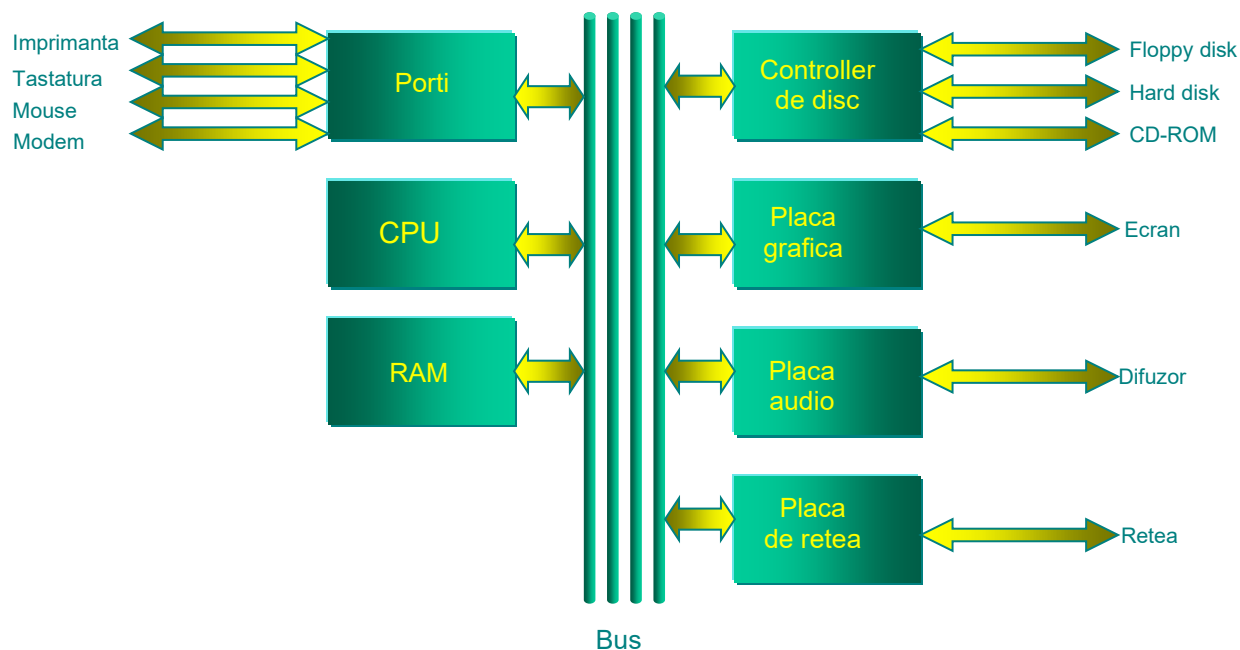


Figura Errore. Nel documento non esiste testo dello stile specificato.-1. Schema structurii unui calculator personal

După cum se vede din Figura **Errore. Nel documento non esiste testo dello stile specificato.-1** în structura calculatorului intră multe componente care trebuie să comunice extrem de frecvent, transmițându-și una alteia numeroase date. Pentru a facilita aceste interacțiuni, calculatorul are o componentă specializată în comunicații interne de înaltă viteză: *bus-ul*. Toate componentele sunt legate la acest bus și au deci posibilitatea să schimbe în orice moment date între ele.

Răspândirea enormă a calculatoarelor personale a condus la standardizarea atât a structurii lor interne articulată în jurul bus-ului, cât și a modului în care diferitele componente comunică cu bus-ul. Standardizarea bus-ului calculatoarelor personale a fost un pas important în evoluția lor, deoarece a permis legarea la bus a unor plăci aparținând unor furnizori diferiți și care trebuie doar

să respecte standardul de comunicații impus de bus. Și din acest motiv astăzi calculatoarele personale au devenit bunuri de larg consum, precum televizoarele sau automobilele.

Componentele care alcătuiesc un calculator (unitatea centrală, memoria, adaptorii, controllerii, tastatura, ecranul, discurile etc.) sunt numite în mod generic *hardware* pentru a se deosebi de componentele “invizibile” ale calculatorului care sunt programele și datele lor, numite generic *software*.

Prin definiție, un *program* este o secvență finită de elemente ce aparțin alfabetului limbajului în care este scris programul. Alfabetul unui limbaj este o mulțime de elemente atomice, adică elemente ce nu pot fi descompuse. De exemplu, cuvintele cheie, semnele de punctuație, operatori și alte caractere speciale formează alfabetul unui limbaj de programare.

1.1.2 Sistemul de operare

A devenit o regulă ca actualele calculatoare să fie dotate încă de la livrare cu un *sistem de operare* și care de multe ori poate fi ales de cumpărător. Sistemul de operare este un software specializat în administrarea resurselor hardware și software utilizate de programe: CPU, memorie, dispozitive periferice, fișiere, rutine de întreruperi, protocoale de comunicare etc.

Sistemele de operare actuale permit *multiprogramarea*, adică oferă posibilitatea executării în paralel a mai multor programe (*execuție concurrentă*) oferind fiecărui program, atunci când îl lansează în execuție, posibilitatea de a utiliza CPU. În vederea asigurării multiprogramării, fiecare execuție de program este organizată ca un *proces* (numit și *task*) izolat de celelalte procese prin încapsularea stării programului și a datelor necesare execuției sale într-un unic spațiu de memorie, exclusiv dedicat procesului respectiv. Multiprogramarea are la bază o operație de comutare a CPU de la un proces la altul, operație realizată de către sistemul de operare într-un mod transparent pentru utilizator. Sistemul de operare alocă CPU unui singur proces, procesul curent, oferindu-i astfel posibilitatea de a-și executa instrucțiunile programului, dar o face doar pentru un timp limitat. La sfârșitul perioadei de timp acordate sau în cazul apariției unor evenimente speciale, sistemul de operare comută execuția de la un proces la altul: procesul curent este întrerupt, sistemul de operare îi ia unitatea centrală și o acordă unuia din procesele în așteptare de CPU care devine proces curent. Uneori se întâmplă ca însuși procesul curent să decidă să se întrerupă.

O alternativă mai recentă de execuție concurrentă este *multithreading*. În acest caz concurența se manifestă în interiorul spațiului de adresare a unui proces. Programatorul delimitează în programul care stă la baza procesului un “segment de cod” pe care îl organizează într-un *thread*, adică o unitate de program pregătită pentru a fi lansată în execuție în paralel cu alte unități asemănătoare. Specificația de thread stă la baza creării “instanțelor” de thread (sau *firelor de execuție*) concurente. Oricâte ori în cursul execuției programului, prin intermediul unor operații speciale, se crează o instanță de thread, execuția acesteia intră în competiție cu execuții în curs ale altor instanțe de thread. Competiția se referă tot la obținerea CPU, dar comutarea CPU de la un fir de execuție la altul se desfășoară în interiorul aceluiași proces. Aceasta înseamnă că diversele fire de execuție pot accesa pentru a citi sau modifica aceleași celule de memorie, utilizându-le în mod partajat (shared). În acest fel, firele de execuție pot comunica cu ușurință unul cu altul, ca și cum ar utiliza căsuțe poștale comune.

Din cele prezentate mai sus rezultă clar că sistemul de operare realizează o “simulare” de concurență între procese sau fire de execuție, simulare transparentă utilizatorului. Aceștia programele îi apar ca executându-se în paralel, deși calculatorul cu o unică CPU nu execută decât un singur program la un moment dat. Când calculatorul are mai multe procesoare, caz din ce în ce mai frecvent, lucrurile nu se schimbă prea mult deoarece dacă numărul proceselor sau threadurilor în curs de execuție depășește numărul procesoarelor, soluția este tot cea oferită de multiprogramare, adică partajarea între procese și thread-uri a procesoarelor disponibile.

O altă “image fabricată” de sistemul de operare poate fi oferită unor utilizatori care folosesc mai multe terminale (console formate din tastatură, ecran și un minim de software de comunicație cu calculatorul central) pentru a accesa un același calculator. Și în acest caz sistemul de operare va organiza execuții ale unor mici porțiuni din programul fiecărui utilizator, trecând rapid de la un utilizator la altul. Trecerile sunt așa de rapide că în cursul unei secunde calculatorul va putea servi de mai multe ori același utilizator. Un asemenea serviciu furnizat de sistemul de operare se numește *timesharing*. Prin timesharing fiecare utilizator percepe interacțiunea cu calculatorul ca și cum l-ar avea complet dedicat.

Din exemplele de mai sus putem spune că prin serviciile sale un sistem de operare ascunde particularitățile hardware-ului pe care este instalat, oferind utilizatorilor o imagine diferită, de fapt imaginea unui sistem de calcul abstract, având avantajul ca acesta este mult mai ușor de utilizat și administrat decât calculatorul real. De multe ori se utilizează termenul de *platformă* (hardware+software) pentru a desemna un sistem de calcul alcătuit dintr-un calculator real și sistemul său de operare.

1.1.3 Software de bază și software de aplicații

Sistemul de operare este un software creat pentru a oferi servicii cu caracter general, folosite de marea majoritate a utilizatorilor de calculatoare. Alte exemple de astfel de software sunt:

- compilatoarele limbajelor de programare,
- mașinile virtuale, cum ar fi Mașina Virtuală Java,
- browser-ele pentru navigare în internet,
- software-ul de comunicații în rețea sau
- software-ul de acces la o bază de date.

Acest tip de software se numește *software de bază* spre a-l deosebi de software-ul creat ad hoc pentru a răspunde unor cerințe specifice ale unui număr redus de utilizatori, numit *software de aplicații*.

Un software de aplicații este un program scris de un dezvoltator sau grup de dezvoltatori pentru a rezolva cu mijloace informatice o problemă concretă cum ar fi gestiunea comenzilor într-o întreprindere sau rezervarea locurilor pentru zborurile organizate de o companie de aviație. Un asemenea program se numește *aplicație*.

Deși diferența între software de bază și cel de aplicații nu este foarte netă, ea ne este necesară pentru plasa granițele acestui curs strict în domeniul software-ului de aplicații. Cursul prezintă cum se scriu aplicațiile în Java pentru un calculator dotat cu un software de baza consistent, alcătuit pe lângă sistemul de operare, din mașina virtuală, compilatoare, librării de servicii, editoare, depanatoare, elemente absolut necesare programării și pe care le vom introduce treptat în secțiunile următoare.

1.1.2 Limbaje de programare

Instrucțiunile-mașină folosite de calculator au două dezavantaje suficiente pentru a descuraja un programator să le folosească în scrierea aplicației sale: 1) sunt elementar de simple, deci pentru a descrie calcule complexe programatorul va trebui să scrie foarte multe instrucțiuni cu un nivel de detaliu exagerat de mare, și 2) sunt reprezentate în calculator ca niște numere binare (șiruri de cifre 0 și 1), deci într-o formă puțin familiară unui programator. Acest limbaj este inacceptabil pentru scrierea de aplicații complexe și de aceea, în afara unei scurte perioade inițiale în istoria calculatoarelor moderne, a fost abandonat în favoarea unor limbaje mai apropiate de limbajul natural și cel logico-matematic. De altfel, această istorie care se întinde pe perioada ultimilor 60 de ani poate fi văzută ca o perfecționare continuă a limbajului cu care programatorii și-au putut scrie programele, apropiindu-l de modul natural în care oamenii comunică și îndepărtându-l tot mai mult de modul în care calculatorul îl pretinde.

Un prim pas a fost făcut dând nume simbolice mnemonice comenzilor din setul de instrucțiuni-mașină și nume simbolice celulelor din memorie care sunt folosite în program. Un limbaj de acest fel se cheamă *limbaj de asamblare* și este specific calculatorului pentru care a fost creat. El necesită un traducător care să mapeze mnemonicele în coduri binare de comenzi și adresele simbolice în adrese de memorie adevărate asamblând totul în celule care memorează configurații de 0 și 1. Traducătorul este un software de bază numit *asamblor*.

Utilizarea altor limbaje mai apropiate de limbajul logico-matematic decât de limbajul instrucțiunilor-mașină pentru descrierea calculelor a fost pasul următor și el a fost posibil datorită utilizării unor programe de traducere automată a noilor limbaje în limbaje-mașină. Asemenea programe se cheamă *compilatoare* și sunt specifice atât limbajului de programare, cât și platformei utilizate pentru executarea programului. Pentru a putea executa un program, programatorul va trebui mai întâi să își compileze programul și abia după aceea să își poată pune problema executării lui. Așa cum se constată în Figura 1-1, programul scris în limbaj de programare este supus unui proces de compilare care produce un program echivalent în limbaj de asamblare sau chiar în limbaj mașină.

Limbajele recente de programare sunt independente de structura calculatoarelor și permit utilizarea de operații și date complexe, adecvate domeniului în care se plasează problema de programat. Aceasta conduce la o economie importantă de cod scris și la o legibilitate sporită a programului. Asemenea limbaje se numesc *limbaje de programare de nivel înalt* (high-level languages) pentru a le deosebi de *limbajele de programare de nivel coborât* (low-level languages), cele în care transpar particularitățile calculatoarelor, fiind mai apropiate de limbajul-mașină. De exemplu, s-a spus despre limbajul C că este un limbaj de nivel coborât datorită posibilităților sale de a lucra cu adrese de memorie, precum și a multor operații din limbaj care se inspirau din aritmetica calculatoarelor.

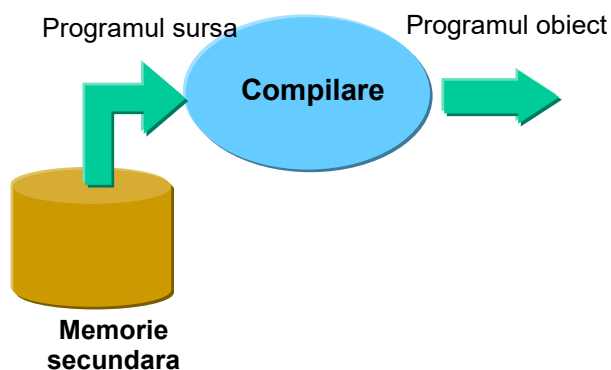


Figura 1-1. Compilarea programelor scrise în limbaj de programare

Principalele caracteristici ale limbajelor de programare de nivel înalt sunt:

- permit efectuarea de operații complexe, orientate spre problemă,

- admit folosirea de structuri de date,
- folosesc reguli de definire de noi operații și structuri de date specificate ad hoc de utilizator.

Față de limbajele de nivel coborât, limbajele de programare de nivel înalt prezintă următoarele avantaje:

- naturalețe, pentru că sunt apropiate de limbajele naturale și/sau limbajul matematic,
- ușurință de înțelegere și utilizare,
- portabilitate, dând posibilitatea ca aceleași programe să fie executate fără modificări sau cu modificări minime pe calculatoare de tipuri diferite,
- eficiență în scriere datorată facilităților de definire de tipuri și structuri de date noi, de compunere de operații în noi operații oricât de complexe, etc.

1.1.3 Sintaxa și semantica limbajelor de programare de nivel înalt

Limbajele de programare de nivel înalt sunt încă departe de limbajul natural și se supun unor reguli sau convenții mult mai rigide decât cele care constrâng limbajul natural. Să vedem care sunt grupurile de reguli care reglementează utilizarea unui limbaj de programare.

Prin *sintaxa* unui limbaj de programare înțelegem totalitatea regulilor care descriu modul de alcătuire corectă a programelor care în respectivul limbaj devin "propoziții". Regulile sintactice se împart în două tipuri de sintaxă prezente în definițiile limbajelor de programare: *sintaxa concretă* și *sintaxa abstractă*. Sintaxa concretă definește scrierea corectă a elementelor componente ale limbajului. Sintaxa abstractă selectează din componentele limbajului doar cele absolut necesare eliminând redundanțele în scrierea programelor și reținând doar componentele sintactice care au o semantică asociată, cele pe care se fundamentează semantica limbajului.

Sintaxa se poate prezenta formalizat prin diagrame de sintaxă sau prin specificații într-un metalimbaj specializat pentru specificarea sintaxei limbajelor precum *BNF* (Backus-Naur Form).

Mulțimea componentelor sintactice utilizate în definiția sintactică a unui limbaj împreună cu regulile lor de definiție în funcție de alte componente formează *domeniul sintactic* al limbajului.

Așa cum sintaxa furnizează construcțiile corecte ale limbajului, semantica furnizează înțelesul acestor construcții. Semantica poate fi *statică* și *dinamică*. Semantica statică definește modul în care o "instanță" a unei construcții intră în relație cu alte "instanțe" pentru a avea un înțeles. În acest scop, noua construcție trebuie să îndeplinească anumite reguli, numite și condiții semantice, care verifică dacă construcția este bine-formată. Regulile se bazează pe o mulțime de concepte care sunt bine cunoscute și înțelese de toți utilizatorii limbajului. Mulțimea conceptelor semantice împreună cu regulile definite asupra acestora formează *domeniul semantic* al limbajului.

Semantica dinamică realizează maparea celor două domenii, sintactic și semantic, ale unui limbaj, prin maparea componentelor sintactice și relațiilor dintre ele din sintaxa abstractă în mulțimea conceptelor și a relațiilor dintre ele din semantica statică.

În Figura 1-2 se prezintă definiții sintactice și semantice ale instrucțiunii condiționale din Java. În figură, sintaxa concretă este formulată mai întâi în metalimbajul BNF. Simbolul ::= în BNF se citește "este formată prin definiție din". Între < și > se include un nume de variabilă, unde prin variabila sintactică sau *neterminal* se înțelege un tip de construcție sintactică (un "pattern" sintactic) care se poate regăsi în definițiile altor variabile. De obicei unei variabile i se asociază atribute semantice. Restul simbolurilor (**if else** ()) sunt *terminale* și apar ca atare în programe. Terminalele nu au o structură internă și nu au asociate atribute semantice.

Simbolurile ::= < > nu aparțin limbajului de programare, ci metalimbajului BNF. De aceea se numesc metasimboluri. În BNF există și metasimbolul | care separă alternative de construcție ale aceleași variabile. De exemplu, definiția:

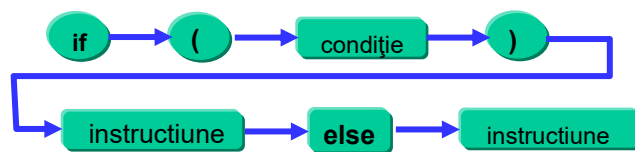
```
<instrucțiune condițională> ::=      if (<expresie>)
                                     <instrucțiune>
                                     | if (<expresie>)
                                     <instrucțiune>
                                     else
                                     <instrucțiune>
```

indică două structuri posibile pentru instrucțiunea condițională: prima fără alternativa **else**, a doua cu alternativa **else**.

- [] pentru structuri opționale
- { } pentru structuri repetitive etc.

```
<instrucțiune condițională> ::=
    if (<expresie>)
    <instrucțiune>
    [ else
    <instrucțiune> ]
```

```
<instrucțiune condițională> ::= if (<expresie>)  
                                <instrucțiune>  
                                else  
                                <instrucțiune>
```



```
<instrucțiune condițională> ::=      E:<expresie>
                                     I1:<instrucțiune>
                                     I2:<instrucțiune>
```

```
type(val(E)) = Boolean
```

```

graph TD
    Entry(( )) --> E{E}
    E -- true --> I1[I1]
    E -- false --> I2[I2]
    I1 --> Merge(( ))
    I2 --> Merge
    Merge --> Exit(( ))
  
```

În restul acestui curs vom folosi BNF pentru a descrie sintaxa construcțiilor din limbajul Java, dar fără ajutorul metasimbolurilor `< si >`, preferând scrierea cursivă a variabilelor. În acest caz nu mai pot apare spații între cuvintele distincte ce formează numele componenteii sintactice. Pentru a le marca, vom utiliza litere mari pentru fiecare cuvând inclus în nume. În schimb, vom menține scrierea “bold” a cuvintelor cheie și altor terminale. Cu aceste convenții, vom scrie ultima regulă sintactică:

```
InstrucțiuneCondițională ::= if (expresie)  
                                instrucțiune  
                                [else  
                                instrucțiune ]
```

Așa cum se vede din Figura 1-2, sintaxa concretă poate fi specificată și prin intermediul grafurilor de sintaxă.

În sintaxa abstractă nu mai apar elemente inutile precum terminalele, rămân doar componentele esențiale ale construcției <instrucțiune condițională>. În definiție componentele au fost etichetate pentru a putea “agăța” semantica.

Semantica statică indică faptul că această construcție este corectă doar dacă rezultatul evaluării expresiei este o valoare logică (true sau false).

În fine, semantica dinamică descrie înțelesul instrucțiunii condiționale care - în funcție de valoarea lui E – este format din înțelesul lui I1 sau al lui I2.

1.2 CARACTERISTICILE LIMBAJULUI JAVA

1.2.1 Scurt istoric

Java s-a născut dintr-un proiect inițiat de către Sun Microsystems în 1991 și care se ocupă de utilizarea microprocesoarelor în dispozitive electronice inteligente, de larg consum. Proiectul s-a încheiat cu proiectarea unui limbaj derivat din limbajele C și C++ și care a fost numit Java. Însă piața dispozitivelor inteligente nu a explodat în primii ani '90 cum se credea în Sun. În schimb exact în acei ani a explodat internet-ul cu rețeaua WWW (World Wide Web). Sun a constatat că limbajul proiectat era exact ceea ce trebuia pentru a crea pagini Web cu conținut dinamic. Proiectul a fost relansat cu noul obiectiv și în mai 1995 Sun a anunțat formal lansarea pe piață a noului limbaj. Spre deosebire de alte limbaje, Java a fost proiectat din rațiuni comerciale pentru a răspunde unor cereri precise ale noii piețe în expansiune care este WWW. De aceea a avut imediat un succes extraordinar și o evoluție rapidă și în același timp surprinzătoare pentru masa utilizatorilor.

1.2.2 Caracteristicile limbajului Java

Limbajul Java prezintă următoarele caracteristici:

- este un limbaj de programare orientat spre obiecte,
- este un limbaj relativ mic conceput în jurul unor concepte de programare riguros respectate,
- programele Java sunt robuste, cu posibilități reduse de a comite erori,
- programele Java sunt sigure, protejate de tentativele de a provoca daune accidentale sau intenționate,
- este un limbaj portabil, adică independent de platformă,
- programele Java pot fi distribuite în rețea și deci adaptate folosirii lor în internet și în intranet,
- programele Java sunt interpretate. Este utilizată o Mașină Virtuală Java (JVM) care interpretează un cod compilat al programelor,
- este un limbaj concurrent pentru că include facilități ce permit crearea și gestiunea de fire de execuție concurente.

1.2.3 Orientarea spre obiecte în programele Java

Java este un limbaj de programare orientat spre obiecte cu următoarele caracteristici:

- un program Java constă din module de cod Java numite clase,
- în timpul execuției sale, un program este un ansamblu de obiecte care comunică între ele,
- un obiect în Java este format din date și metode (obiect = date+metode),
- obiectele sunt grupate în clase,
- se pot construi ierarhii de clase bazate pe moștenire,
- limbajul conține biblioteci de clase standard, specializate pe diferite tematici. În Java un programator poate scrie toate clasele necesare, așa cum poate folosi clase existente într-o extrem de bogată colecție de biblioteci organizate în pachete. În acest sens putem afirma că:
- Java încurajează reutilizarea software-ului.

Cel care învață Java va trebui să învețe desigur sintaxa și semantica limbajului, dar va trebui să cunoască și care sunt bibliotecile de clase, la ce servesc și cum se utilizează.

Bibliotecile sunt reprezentate de pachete Java, adică de colecții de clase de obiecte înrudite între ele și încapsulate într-o unică structură. Pachetele sunt specializate pe anumite problematici. De exemplu, pachetul *java.io* este specializat în intrări/iesiri, *java.net* în comunicații în rețea, *java.lang* în elementele de bază ale limbajului etc. Unele din biblioteci aparțin specificației Sun, deci sunt standard, și trebuie să existe în toate implementările Java. Altele pot fi cumpărate de la furnizori independenți și utilizate în programe.

Avantajul utilizării bibliotecilor constă în:

1. creșterea productivității în activitatea de scriere a programelor pentru că utilizarea unor clase deja existente implică economie de efort de programare,

- creșterea calității programului final pentru ca implementările claselor din biblioteci sunt foarte probabil mult mai corecte, fiind de obicei îndelung verificate, decât implementările făcute ad hoc pentru un proiect.

Modul de utilizare a claselor din bibliotecile standard este dat de API¹ Java, adică de *interfețele pentru programarea aplicațiilor*. Aceste interfețe reprezintă modul în care programatorul este invitat să “vadă” și să folosească JVM. Ele conțin toate serviciile pe care JVM le oferă programării. Aceste servicii se modifică întrucâtva de la o versiune la alta a limbajului. Servicii noi se pot adăuga, iar unele servicii devin “deprecated” (Sun intenționează să renunțe în viitor la ele sau, pur și simplu, intenționează să descurajeze folosirea lor).

1.2.4 Arhitectura sistemului Java

Când vorbim de un “sistem Java” trebuie să avem în vedere următoarele elemente componente ale sistemului Java:

- limbajul Java, adică regulile de scriere și interpretare a programelor Java,
- bibliotecile Java, care extind facilitățile nucleului Java cu funcții specializate în diverse tematici,
- API Java, care specifică serviciile accesibile programatorului de către mediul de execuție,
- mediul de execuție Java, reprezentând procesul creat de interpretorul Java ce simulează JVM.

Software-ul instalat cu pachetul J2SDK 1.4.x oferit de Sun Microsystems are arhitectura prezentată în Figura 1-1. În figură se observă stratificarea sistemului în două mari componente:

1. JRE, adică mediul de execuție Java (Java Runtime Environment) și
2. Colecția de instrumente pe care kit-ul J2SDK 1.4.x le pune la dispoziția dezvoltatorilor. În acest kit intră: compilatorul (javac), depanatorul (jdb), vizualizatorul de applet-uri (appletviewer), generatorul de documente (javadoc), generatorul de fișiere .jar de clase Java zippate (jar), convertitoare (HtmlConverter, idlj) și alte programe utile dezvoltării de programe Java.

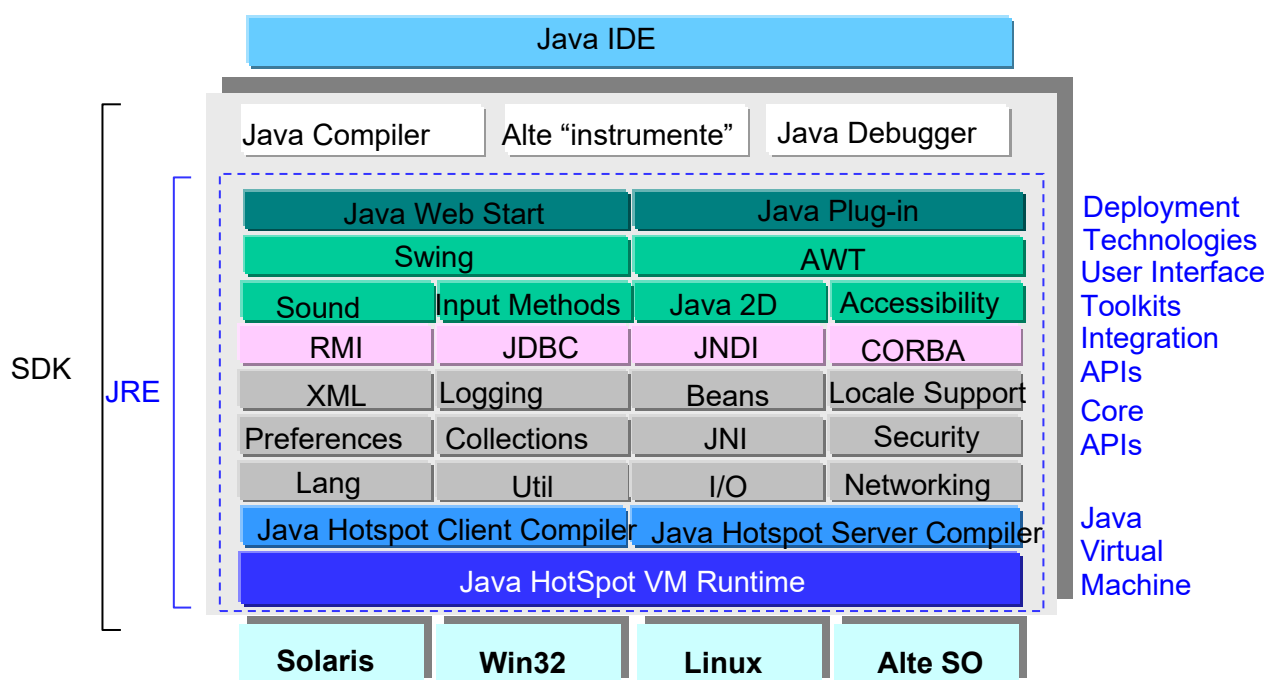


Figura 1-1. Arhitectura platformei Java 2 Standard Edition

Macro-componenta JRE este și ea stratificată pe considerente legate de utilizarea serviciilor oferite. Straturile, de la cel de bază spre cel mai complex sunt:

1. JVM, motorul execuțiilor programelor Java, în două versiuni, Java HotSpot Client VM (JVM predefinită rulează în medii client) și Java HotSpot Server VM (proiectată pentru a avea maximă eficiență în medii server). Oferă serviciile de bază accesibile componentelor furnizoare de API;

¹ API = Application Programming Interface

2. CoreAPIs include serviciile ce reprezintă facilitățile de bază, fundamentale utilizate în scrierea programelor Java;
3. Integration APIs include serviciile oferite celor care vor sa creeze aplicatii cu date și/sau procese distribuite;
4. User Interface Toolkits conțin serviciile care permit construirea de interfețe grafice, multimedia, cu dispozitive de intrare/ieșire neconvenționale etc.;
5. Deployment Technologies conțin servicii necesare execuției programelor Java în medii parțial dotate cu resursele necesare. În aceste situații este necesară o actualizare (plug-in) sau o încărcare automată (web downloading) a resurselor necesare execuției.

1.2.5 Mediul de execuție Java

Un program în Java este alcătuit dintr-o listă de definiții de clase și interfețe. În clase se definesc datele și comportamentul obiectelor care vor participa la prelucrarea de date dorită, iar în interfețe se definesc serviciile comune pe care unele clase decid să le implementeze.

Un astfel de program nu poate fi executat direct de calculator pentru că instrucțiunile lui nu sunt instrucțiuni mașină, deci trebuie mai întâi compilat. Compilarea programelor Java, spre deosebire de cazul programelor în alte limbaje precum C, C++ sau Fortran, nu produce nici ea programe direct executabile. Rezultatul compilării Java sunt programe într-un limbaj care este într-adevăr un limbaj-mașină, dar pe care nici un hardware nu va fi în stare să-l execute deoarece aparține unui calculator abstract numit *Mașina Virtuală Java* (JVM).

Soluția mașinii virtuale a fost adoptată de Sun pentru a face din Java un limbaj portabil: programele Java vor putea să fie rulate pe orice calculator care emulează JVM, iar rezultatul rulării lor pe un calculator va fi identic cu rezultatul rulării pe orice alt calculator ("write once, run anywhere"). Portabilitatea este esențială pentru Java care este limbajul "internet" prin excelență. Fără portabilitate nu s-ar fi putut ca un program Java scris și compilat într-un nod al rețelei Internet să poată fi descărcat și executat în orice alt nod (așa cum își propuseseră autorii limbajului).

Limbajul-mașină al JVM se numește *bytecode* pentru ca programele scrise în acest limbaj au instrucțiunile reprezentate pe un singur byte². Un asemenea program poate fi transferat în rețea ca un flux de bytes, exact ca orice flux de date numerice sau caractere.

La sosirea lui la destinație, indiferent care este noua platformă, programul ar putea fi executat dacă această platformă are instalată o JVM care să execute programul. Dar JVM nu este unica condiție, pentru că limbajul Java se dorește nu numai portabil, dar și sigur, iar cu securitatea datelor în internet nu se poate glumi! Ajuns la destinație, bytecode-ul programului, pentru a fi executat trebuie mai înainte verificat. Verificarea se face analizând în primul rând proveniența programului, care trebuie să vină dintr-o sursă cunoscută ca sigură. În al doilea rând se verifică faptul că operațiile pe care programul sosit are de gând să le execute pe calculatorul de destinație nu sunt periculoase pentru acesta, pentru că nu orice operație este permisă unui program nou sosit!

Dacă primele elemente ale "sistemului" Java au fost explicate mai sus, să vedem ce este mediul de execuție Java.

Figura 1-2 prezintă din alt punct de vedere pașii în dezvoltarea și execuția unui program Java.

² Byte este o unitate de măsură a capacității de memorie și reprezintă un grup de 8 biți, ceea ce este echivalent cu 256 combinații distincte de 0 și 1.

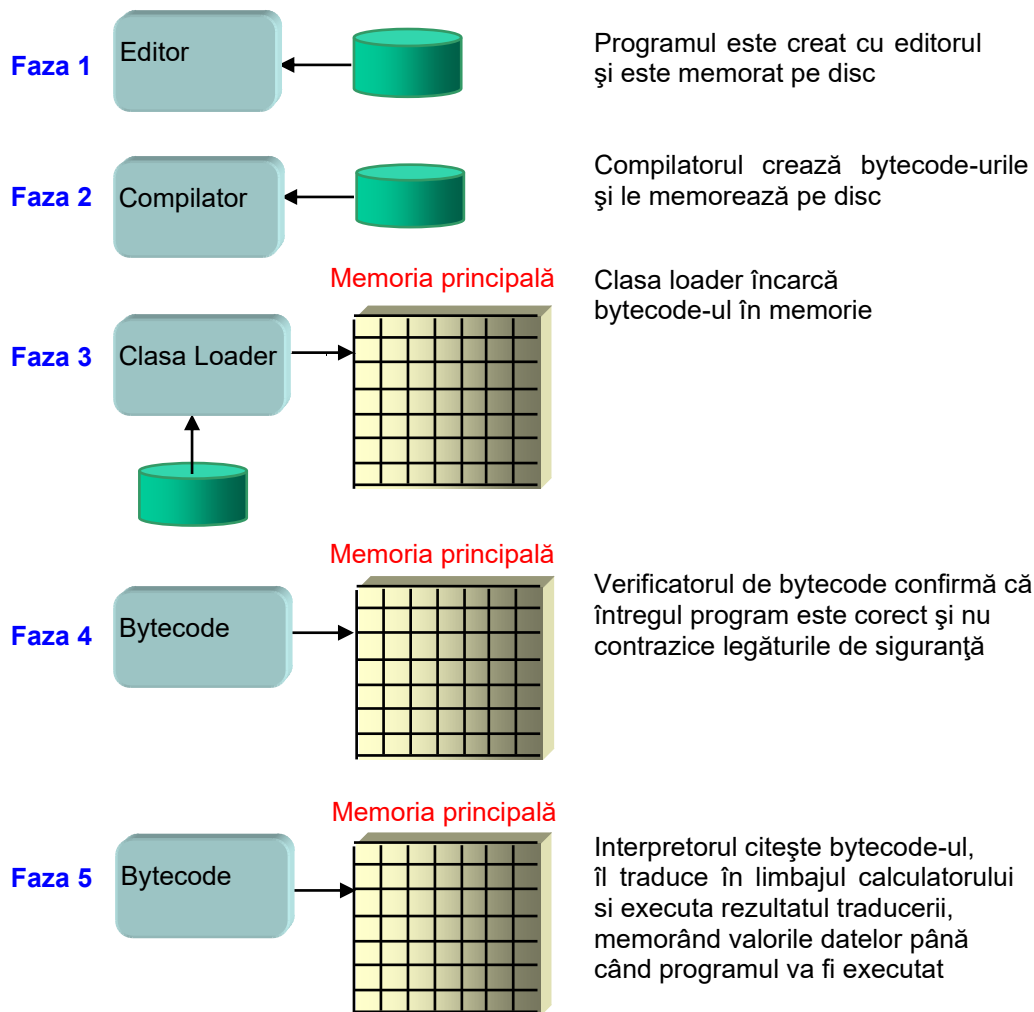


Figura 1-2. Pașii în execuția unui program Java

Acești pași sunt: editarea, compilarea, încărcarea, verificarea și execuția.

1. Editarea este realizată de un *editor*, un program care permite scrierea programului sursă precum *notepad* și *wordpad* în sisteme Windows sau *vi* și *emacs* în sisteme Unix. Se pot utiliza editoarele unor medii Java de programare integrate (IDE) precum JBuilder a lui Borland, Visual Café a lui Symantec sau Visual J++ a lui Microsoft.
2. Compilarea este realizată de *compilatorul Java* care traduce programul sursă editat în primul pas în bytecodes, limbajul interpretorului Java.
3. Încărcarea aduce în memorie bytecodes și este realizată de *încărcătorul de clase* (class loader). De unde își ia încărcătorul clasele? Le poate lua de pe discul calculatorului local sau le poate descărca din rețea.
4. Bytecodes sunt acum în memorie, dar nu pot fi executați până nu sunt verificați de *verificatorul de bytecodes*. Verificarea este necesară atunci când clasele sunt descărcate din alt calculator. Ele nu trebuie să încalce constrângerile de securitate din Java. Aceste constrângeri sunt necesare pentru a nu permite codului care provine din internet să nu creeze daune în sistemul de fișiere și în sistemul de calcul local, în general.
5. Execuția programului este datorată unui program în execuție în calculatorul local și care este *interpretorul de bytecodes*. Interpretorul este un motor care execută din program instrucțiune după instrucțiune creând și menținând mediul de execuție Java care se manifestă în acest fel ca o mașină de calcul, virtuală însă.

1.2.6 Tipuri de programe Java

Dacă ideea execuției în calculatorul local de programe descărcate din rețea a stat la baza conceperii limbajului Java, evoluția lui ulterioară a fost puternic influențată de succesul pe care l-a raportat în masa programatorilor utilizarea lui ca limbaj de programare tradițional, nelegat neapărat de WWW. De aici a rezultat și o diversificare a programelor Java în mai multe categorii.

În Java, programele sunt în principal de trei tipuri:

- aplicații,
- applet,
- servlet.

Cele trei tipuri de programe, deși utilizează același limbaj Java, sunt folosite în scopuri diferite unul de altul și sunt structurate diferit.

Aplicație Java

O *aplicație Java* este un program tradițional așa cum este un procesor de text (word processor) sau o foaie electronică (spreadsheet) și care, în mod normal, se execută în calculatorul local care îl găzduiește. Are, de regulă, o aplicație cu interfață grafică alcătuită din ferestre (frame-uri) ce conțin la rândul lor alte componente grafice. Figura 1-3 prezintă dezvoltarea și rularea unei aplicații Java. Din figură se poate vedea că programul sursă ale aplicației este alcătuit din fișiere cu extensia *.java*, dar care se pot scrie cu un editor de texte obișnuit. Aceste fișiere se pot scrie și folosind un mediu de programare, al cărui editor ne ajută să scriem programe corecte.

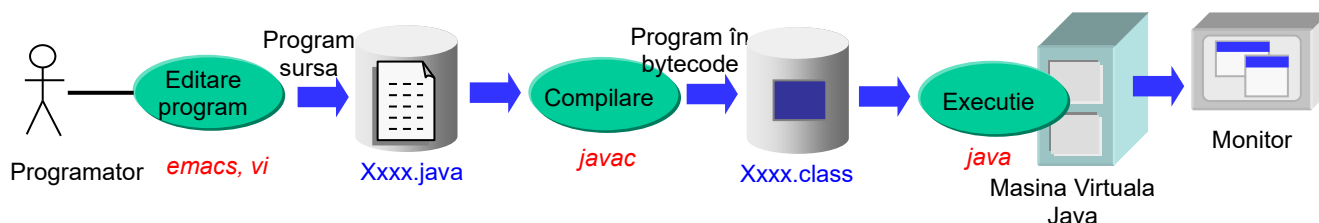


Figura 1-3. Dezvoltarea și execuția unei aplicații Java

Odată scrise, fișierele se compilează cu compilatorul Java (în J2SDK compilatorul se numește *javac*). Programul compilat este o secvență de bytecode care se memorează în una sau mai multe fișiere cu extensia *.class*. Fișierele *.class* sunt portabile pe toate platformele. Pentru a fi executat programul, instrucțiunile din fluxul de bytecode trebuie să fie interpretate de o JVM. În J2SDK, JVM este lansată în execuție cu comanda *java* având ca parametru numele fișierului ce conține programul principal. Interpretarea bytecode-ului de către JVM va vizualiza pe monitor o fereastră principală de la care se poate porni pentru realizarea prelucrării dorite.

Applet Java

Un applet Java este un mic program care este de obicei memorat într-un computer aflat la distanță (remote) față de calculatoarele utilizatorilor și la care utilizatorii se conectează cu ajutorul unei aplicații locale numită browser WWW. Applet-ul este încărcat din calculatorul remote în browser, executat de browser și eliminat când execuția s-a terminat. Pentru a executa din nou applet-ul, utilizatorul trebuie să navigheze cu browser-ul în locația WWW corespunzătoare și să reîncareze programul în browser.

Figura 1-4 prezintă dezvoltarea și execuția unui applet. Programul Java care este applet-ul se editează mai întâi într-un fișier cu extensia *.java*. Fișierul este apoi compilat și memorat cu același nume dar extensia *.class* într-un director al calculatorului remote, chemat *Server Web*, care va fi accesat de browser-ele aflate în calculatoarele utilizatorilor, chemate *Client Web*. Când? Atunci când browser-ul descarcă din Server-ul Web un fișier cu extensia *.html* sau *.htm* numit pagină HTML. Acest fișier specifică browser-ului într-un limbaj numit HTML ce anume trebuie să vizualizeze pe ecranul Client-ului Web: texte, imagini, componente grafice și ... applet-uri. Pentru a vizualiza un applet pagina HTML conține o referință la fișierul *.class* al applet-ului. La întâlnirea referinței, browser-ul lansează încărcătorul de clase Java (Java class loader) care descarcă în calculatorul local applet-ul și îl interpretează folosind o mașină Java integrată în browser. Cum applet-ul este o componentă grafică Java, execuția este echivalentă cu vizualizarea lui pe ecran.

Din cele de mai sus, pentru a executa un applet, trebuie să fie dezvoltată și o pagină HTML care să îl refere. O asemenea pagină se poate dezvolta cu un editor obișnuit sau cu un program specializat cum este FrontPage.

Applet-urile pot să fie executate și din linia de comandă utilizând comanda `appletviewer` furnizată de kit-ul de dezvoltare având ca parametru fișierul HTML al paginei ce punctează către applet.

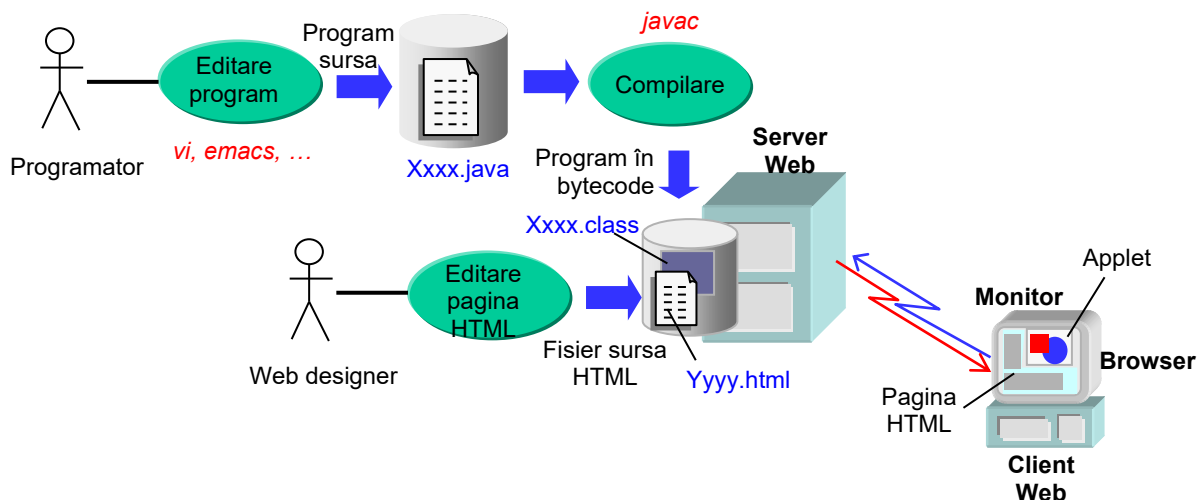


Figura 1-4. Dezvoltarea și execuția unui applet Java

Servlet

Un servlet este un mic program Java, asemănător ca idee unui applet, care însă în loc să se execute în calculatorul-client Web se execută într-un calculator care rulează un server Web. Servlet-urile sunt folosite pentru crearea dinamică de pagini Web. Când un browser cere în loc de o pagină HTML vizualizarea unui servlet, server-ul Web lansează în execuție servlet-ul. Acesta va genera o pagină HTML și o va livra browser-ului care o vizualizează. Spre deosebire de applet care este o componentă grafică Java, servlet-ul nu are nimic de a face cu componentele grafice deoarece este complet transparent client-ului.

2. FUNDAMENTE ALE PROGRAMĂRII

2.1 ELEMENTE DE BAZĂ ALE LIMBAJULUI JAVA

2.1.1 Compilarea și execuția unui program Java

Considerăm că avem instalat kit-ul de dezvoltare J2SDK. Ne propunem să dezvoltăm și să executăm un program Java extrem de simplu care să scrie la ieșire, adică pe ecranul monitorului, în fereastra din care lansăm programul, următoarea propoziție banală:

Acesta este un program simplu!

Pașii procedurii de dezvoltare și execuție pe care îi vom executa împreună, sunt:

1. Scrieți textul următorului program în fereastra unui word-processor pe care îl aveți la îndemână. Se poate utiliza de exemplu NotePad sau WordPad.

```
// Primul program in Java
public class PrimulProgram {
    // Metoda main descrie executia unei aplicatii Java
    public static void main(String[] args){
        System.out.println("Acesta este un program simplu!");
    }
}
```

Programul 1. PrimulProgram

2. Salvați programul în formatul text într-un director numit `CursJava` pe disc, scriind între ghilimele numele și extensia programului astfel: `"PrimulProgram.java"`
3. Deschideți o fereastră **MS-DOS Prompt** urmând traseul: Start>Programs>MS-DOS Prompt
4. Schimbați directorul curent în directorul `CursJava`
5. Executați de la prompter comanda:
`javac PrimulProgram.java`
6. Executați de la prompter comanda:
`java PrimulProgram.`

Pe ecranul consolei MS-DOS Prompt apare ca rezultat al execuției, textul:

Acesta este un program simplu!

Explicarea programului PrimulProgram

- Primul rând al programului, deoarece începe cu `//`, este un comentariu. Un comentariu servește de obicei pentru lămurirea cititorului programului asupra rostului prezenței în program a unor elemente ce se doresc evidențiate. Comentariile nu sunt compilate, deci conținutul lor nu se repercutează în nici un fel asupra codului generat.
- Restul programului este declarația unei clase numită `PrimulProgram`. Clasa este declarată cu `public`, ceea ce o face vizibilă tuturor altor clase din program. Dar în acest program nu există alte clase! Această declarație este totuși necesară deoarece numele fișierului sursă `PrimulProgram.java` are chiar numele clasei.

Regulă: un fișier de program sursă Java trebuie să aibă numele unei clase din fișier. Dintre toate clasele declarate în fișierul respectiv, acea clasă, și numai ea, trebuie să fie declarată `public`.

- După numele clasei se deschide o paranteză `{` care se închide cu `}` la sfârșitul programului. Între cele două paranteze se află corpul clasei `PrimulProgram`.

Regulă: O declarație de clasă este formată dintr-un titlu și un corp cuprins între `{ ... }`. Titlul caracterizează clasa, declarând în primul rând numele și apoi anumite atribute, cum este `public`. Corpul declară și descrie membrii clasei.

- În corpul clasei, primul rând este tot un comentariu care descrie metoda ce îi urmează.
- Restul rândurilor din corpul clasei reprezintă o declarație de metodă. Metoda se numește `main` și este declarată cu atributele `public` și `static`. O *metodă* în Java reprezintă un calcul care primește un nume astfel încât, utilizând în program numele metodei, putem să declanșăm calculul. În cazul acestui program, calculul lui `main` este format din două operații. Metoda nu

produce nici un rezultat, ceea ce este simbolizat de cuvântul `void` care precede numele metodei. În fine, metoda are un parametru numit `args`, iar `String[]` indică tipul parametrului.

Regulă: Metodele în Java se specifică - ca și clasele - printr-un titlu și un corp cuprins între `{ ... }`. Titlul caracterizează metoda, declarând în primul rând numele metodei, anumite atribute, cum sunt `public` și `static`, tipul rezultatului produs de calculul metodei și o listă de parametri cuprinsă între `(...)`. Corpul metodei specifică calculul care este reprezentat de numele metodei.

- Primul și al doilea rând din corpul metodei sunt *instrucțiuni*.

Regulă: Instrucțiunile în Java se termină cu caracterul `;`. Acest caracter face parte din sintaxa instrucțiunii.

- Primul rând din corpul metodei este:

```
System.out.println("Acesta este un program simplu!");
```

`println` este numele unei metode din Core API Java care tipărește un text. Unde? Această informație este dată de variabila `System.out`, care reprezintă un obiect Java ce identifică de obicei ecranul calculatorului. Prin urmare, acest rând din program specifică "chemarea" metodei `println` pentru a tipări un text pe ecranul calculatorului. Ce va tipări? Acest lucru este indicat între parantezele `(...)` și anume textul `Acesta este un program simplu!`. Faptul că acest text este un șir de caractere unitar și nu face parte din limbaj este indicat de închiderea lor între `"... "`.

2.1.2 Structura unei clase Java

Să considerăm un alt program simplu care tipărește pe 5 rânduri același text precedat de numărul de ordine al rândului. În textul programului liniile sunt numerotate pentru a le putea referi mai ușor conținutul în explicarea programului. Desigur, la editarea programului în vederea execuției, se vor omite însă aceste numere.

```
1.      //      Programul BunaZiuaTutoror
2.      /*
3.          Programul scrie de 5 ori
4.          "Buna ziua tutoror!"
5.      */
6.      public class BunaZiuaTutoror {
7.          String mesaj = "Buna ziua, tutoror!";
8.          public void afiseazaMesaj() {
9.              int j = 5;
10.             for (int i = 1; i <= j; i++)
11.                 System.out.println(i + "." + mesaj);
12.         }
13.         public static void main (String[] args) {
14.             BunaZiuaTutoror obj = new BunaZiuaTutoror();
15.             obj.afiseazaMesaj();
16.         }
17.     }
```

Programul 2. BunaZiuaTutoror

Explicarea programului BunaZiuaTutoror:

- Liniile 1-4 sunt comentarii. Prima linie este un tip de comentariu introdus cu `//` pe care l-am văzut și în programul precedent. Un asemenea comentariu poate fi introdus în orice punct al liniei, nu numai la începutul ei, și se va întinde însă până la sfârșitul liniei respective. Liniile 2-4 prezintă un al tip de comentariu care se poate întinde pe mai multe linii. Comentariul se introduce cu perechea de caractere `/*` și se întinde până la întâlnirea perechii `*/` care îl va închide. Și acest comentariu poate fi inițiat în orice punct al unei linii, doar că, având marcaj de sfârșit de comentariu, se poate termina chiar în linia respectivă, oriunde, lăsând spațiu eventual pentru continuarea codului programului pe aceeași linie. De exemplu, linia:

```
public class /* urmeaza numele clasei */ BunaZiuaTutoror {
```

este corectă, deși nu este recomandat să se abuzeze de asemenea întreruperi ale codului.

- Linia 6 reprezintă titlul declarației clasei `BunaZiuaTuturor`, iar liniile 7-16 reprezintă corpul clasei, după cum se vede inclus între `{...}`.
- Linia 7 introduce o noutate: o *variabilă* numită `mesaj` care poate avea ca valori un `String`, adică un șir de caractere. Variabila este inițializată cu `"Buna ziua, tuturor!"` care, așa cum apare, delimitat de `" "`, în Java reprezintă șirul de caractere: `Buna ziua, tuturor!`.
- Liniile 8-12 introduc metoda `afiseazaMesaj` în definiția clasei `BunaZiuaTuturor`. Metoda are atributul `public` și nu întoarce un rezultat ceea ce este indicat de `void`. Metoda nu are parametri.

Regulă: Chiar și definiția unui metode fără parametri trebuie să păstreze parantezele `()`.

- Corpul metodei `afiseazaMesaj` se întinde pe liniile 9-11. Prima linie reprezintă declarația unei variabile `j`. Variabila are valori întregi, fapt indicat de cuvântul `int` ce precede `j`. Variabila `j` se deosebește de variabila `mesaj` nu numai ca tip de valori, dar și ca rol în cadrul programului: `j` este locală metodei, deci cunoscută doar în metodă, pe când `mesaj` este globală clasei, adică este cunoscută în întreaga clasă, inclusiv în metodele acesteia. Variabila `j` este inițializată cu valoarea constantei 5.
- Linia 10 introduce o instrucțiune `for`. Instrucțiunea cere calculatorului să execute în mod repetat instrucțiunile incluse în corpul său care este format dintr-o singură instrucțiune, chemarea de metodă din linia 11. Repetarea este ghidată de variabila `i` așa cum indica titlul instrucțiunii `for`:

```
for (int i = 1; i <= j; i++)
```

Variabila `i` este cunoscută doar în interiorul instrucțiunii `for`. Ea este declarată de tip `int` și inițializată cu 1. Apoi, după fiecare repetare a execuției corpului lui `for`, valoarea lui `i` este incrementată până când `i` depășește valoarea lui `j`. În acest moment repetarea execuției lui `for` se oprește.

- Linia 11 reprezintă chemarea metodei `println` de vizualizare a unui șir de caractere compus din valoarea lui `i`, un caracter `" "` și conținutul variabilei `mesaj`.
- Linia 13 introduce titlul metodei `main` deja întâlnită în programul precedent. Este doar o întâmplare sau este obligatorie această metodă? `main` este o metodă obligatorie în orice aplicație (nu applet sau servlet, ci aplicație!) Java.

Regulă: În programul oricărei aplicații trebuie ca măcar să existe o clasă care conține metoda `main`. Mai mult, clasa care dă numele aplicației *trebuie* să conțină metoda `main`. Această metodă descrie calculul aplicației. Execuția aplicației începe cu chemarea metodei `main` și se încheie, dacă nu apar evenimente neprevăzute, când `main` își încheie calculul.

- Se poate observa în linia 13 că titlul metodei `main` este exact același cu titlul aceleiași metode din clasa `PrimulProgram`.

Regulă: De la aplicație la aplicație titlul metodei `main` rămâne neschimbat:

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

- Linia 14 este o declarație a unei variabile `obj` de tip `BunaZiuaTuturor`. Această variabilă reprezintă un obiect al clasei `BunaZiuaTuturor` pe care tocmai o definim. Obiectul este creat cu expresia:

```
new BunaZiuaTuturor()
```

și inițializează variabila `obj`. Obiectul creat conține o variabilă `mesaj` inițializată cu `Buna ziua, tuturor!` și are o metodă `afiseazaMesaj`.

- Linia 15 utilizează obiectul pentru a chema metoda `afiseazaMesaj`. Execuția metodei produce afișarea liniilor:

```
1.Buna ziua, tuturor!
2.Buna ziua, tuturor!
3.Buna ziua, tuturor!
4.Buna ziua, tuturor!
5.Buna ziua, tuturor!
```

care reprezintă și rezultatul programului, deoarece nemaifiind instrucțiuni de executat în `main`, procesorul oprește execuția programului.

2.2 VARIABLE

2.2.1 Conceptul de variabilă

O *variabilă* este o componentă a programului care identifică un spațiu în memoria calculatorului alocat la *runtime*¹. În spațiul respectiv programul poate memora date necesare calculului.

În textul programului, variabilele apar de obicei ca niște identificatori. Prin *identificator* într-un limbaj de programare se înțelege un nume simbolic dat unei componente a programului: variabilă, clasă, metodă, etc. Acest nume este, de regulă, un șir de litere și cifre în care primul caracter trebuie să fie o literă. De exemplu: `sold`, `anul2`, `x123`, `alfa3bis` sunt identificatori. În Java, la litere și cifre se adaugă și caracterele `$` și `_`.

Anumiți identificatori, numiți *cuvinte-cheie* (key words), au semnificații precise în cadrul programului și sunt rezervați doar scopului de a reprezenta aceste semnificații. Ei nu pot fi utilizați pentru a desemna variabile, funcții, clase sau alte componente introduse de programator, sunt "cuvinte rezervate".

Să reținem că limbajul Java este "case sensitive", adică compilatorul diferențiază literele mari de literele mici. Deci `sold`, `Sold`, și `SOLD` sunt identificatori diferiți.

În Java oricâte ori se folosește o variabilă într-un calcul, JVM "citește" din spațiul alocat variabilei valoarea memorată. Citirea unei variabile nu modifică valoarea memorată. Unei variabile i se poate modifica valoarea prin *atribuirea* unei noi valori. În acest caz, în spațiul respectiv se memorează noua valoare care șterge valoarea precedentă.

De exemplu, următoarele instrucțiuni de atribuire în Java:

```
sold = 2500000;  
cf = "ABCDE45I880";  
numar1=14;  
numar2=-8;  
suma=25;  
suma = numar1 + numar2;
```

modifică valoarea variabilelor `sold`, `cf`, `numar1`, `numar2` și `suma`.

În Figura 2-1 se prezintă schematic execuția instrucțiunii de atribuire a rezultatului adunării:

```
suma = numar1 + numar2;
```

Valorile extrase din variabilele `numar1` și `numar2` sunt însumate și rezultatul este apoi memorat în variabila `suma`. La sfârșitul execuției instrucțiunii, valorile variabilelor `numar1` și `numar2` au rămas neschimbate, în timp ce valoarea variabilei `suma` s-a modificat cu rezultatul adunării.

Legând noțiunea de variabilă de existența unei locații în memorie, apare o dublă ipostază a variabilei: cea de păstrătoare de date și cea de dată ea însăși, deoarece identificarea locației variabilei printre celelalte locații din memorie reprezintă ea însăși o informație și, implicit, o dată utilizabilă în program.

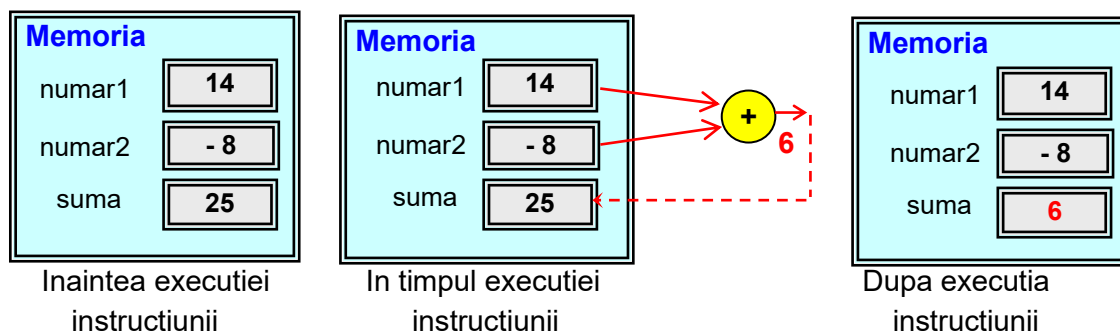


Figura 2-1. Execuția unei instrucțiuni de atribuire

¹ "La runtime" înseamnă la momentul execuției programului.

De aceea, unei variabile îi corespund în realitate două valori (Figura 2-2):

- valoarea stânga, VS (în engleză: left-hand-value sau L-value),
- valoarea dreapta, VD (în engleză: right-hand-value sau R-value).

Valoarea dreapta este ceea ce de obicei se înțelege prin *valoarea variabilei*: valoarea memorată în spațiul alocat variabilei. Această valoare participă la calculele în care este implicată variabila.

Valoarea stânga este în esență adresa din memorie unde este alocat spațiul dedicat variabilei. Această valoare poate fi accesibilă programatorului pentru a o utiliza în calcule, cum se întâmplă în C sau C++, sau inaccesibilă, ca în Java sau Fortran. În Pascal, C și C++, VS a unei variabile se poate "extrage" ca o simplă valoare și se poate apoi lucra cu ea, folosind-o ca VD a unei alte variabile de tip "pointer". Indiferent de posibilitatea dată programatorului de a extrage VS a variabilei și de a o utiliza în program, toate limbajele de programare utilizează VS pentru a defini operații asupra variabilei precum operația de atribuire sau operația de indexare.

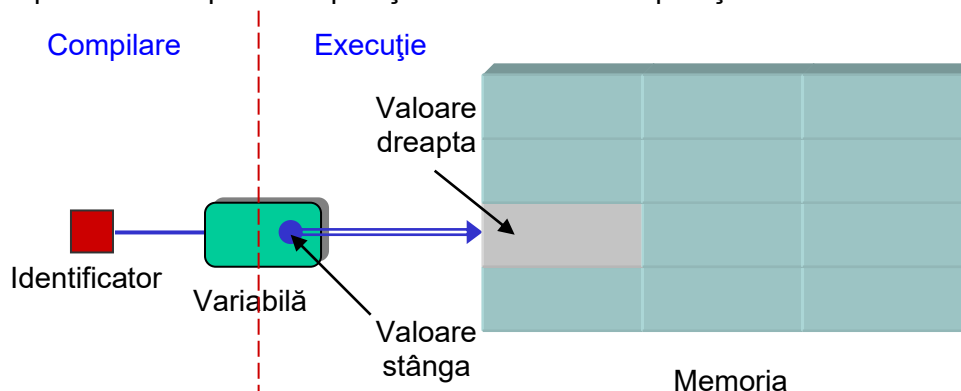


Figura 2-2. Valoarea stânga și valoarea dreapta ale unei variabile

De exemplu, execuția instrucțiunii de atribuire:

$a = b + c;$

utilizează VD ale variabilelor *b* și *c* pentru a face adunarea, dar pentru a salva rezultatul utilizează VS a lui *a*. Semantica instrucțiunii ar putea fi enunțată așa:

Adună valoarea dreapta a lui *c* la valoarea dreapta a lui *b* și rezultatul memorează-l în spațiul indicat de valoarea stânga a lui *a*.

Relația dintre identificatorul variabilei și variabilă se realizează la momentul compilării în urma întâlnirii de către compilator în textul programului a unei declarații care introduce identificatorul variabilei și caracterizează variabila.

Regulă. În limbajele de programare, *legarea*² la o variabilă a identificatorului, precum și a proprietăților ei se face prin *declarații*.

Variabilei *i* se asociază valoarea stânga la execuție.

2.2.2 Declarația de variabilă

În afara celor două valori, VS și VD, și a identificatorului, o variabilă este caracterizată și prin *atribute*. Acestea reprezintă proprietăți pe care variabila le menține pe tot parcursul existenței sale. Declarațiile de variabile sunt instrucțiuni care au ca rol:

- alocarea de spațiu pentru variabile,

² Se numește *momentul legării* (binding time) momentul în care are loc asocierea unei proprietăți la o componentă a programului. Dacă asocierea poate fi făcută la compilare, se spune că avem o *legare statică* (early binding time). Dacă momentul în care se poate face asocierea este la execuție, adică valoarea de atribut ce se asociază variabilei rezultă din calcul, avem o *legare dinamică* (late binding time). Momentul legării are o importanță deosebită pentru alegerea soluției de traducere a limbajului de programare: compilare sau interpretare. Dacă definiția limbajului prevede legarea statică a mării majorități a atributelor, atunci soluția cea mai indicată este compilarea, dar dacă atribute importante cum ar fi tipul variabilei sunt legate la execuție, soluția nu poate fi decât interpretativă, cea în care construcțiile limbajului se execută pe măsură ce sunt compilate.

- introducerea în program de identificatori cu care se accesează aceste variabile,
- asocierea unor atribute variabilelor,
- inițializarea variabilelor.

La execuție, variabila există doar din punctul declarației încolo. În declarație se poate preciza și valoarea cu care variabila este inițializată.

Exemple de declarații:

```
int imprumut;
String mesaj, nume = "Ion";
boolean gata = true;
```

În lipsa inițializării, variabila rămâne de regulă fără o valoare controlată de program până la prima atribuire de valoare. Există, așa cum vom vedea în capitoul următor, cazuri în care compilatorul Java inițializează automat variabila cu o valoare predefinită, chiar dacă programatorul nu a specificat o valoare de inițializare.

Regulă. În Java, o variabilă are o unică declarație. Această declarație trebuie să apară înaintea primei utilizări a variabilei.

Domeniul de valabilitate a unui identificator și domeniul de viață a variabilei

Domeniul de valabilitate al identicatorului unei variabile (în engleza, scope) este reprezentat de acea porțiune din program în care identicatorul poate fi folosit pentru a accesa variabila. Limbajele de programare au reguli precise care guvernează stabilirea domeniului de valabilitate a tuturor identicatorilor. În general vorbind, o variabilă poate fi locală sau globală. Dar această clasificare este întotdeauna relativă: locală sau globală unei metode, locală sau globală unui fișier sursă etc.

Pentru a utiliza corect aceste caracterizări este important să se cunoască că în limbajele de programare există o structurare a programului în unități de program. Aceste unități pot fi blocuri de câteva instrucțiuni, metode, clase, module sau pachete de program, cum ar putea fi chiar întregul program. După cum se vede, este inevitabil ca unitățile de program să fie incluse unele în altele precum cutiile chinezești: un bloc este în interiorul unei metode, o metodă este în interiorul unei clase, iar o clasă este inclusă într-un pachet etc.

Ce au comun toate unitățile de program? Faptul că fiecare din ele reprezintă un *spațiu de nume* (namespace), adică un container în care se află declarate componente ale programului: variabile, metode, clase, fiecare cu identicatorul său folosit ca nume. Numele declarate într-un spațiu de nume trebuie să fie unice în acel spațiu.

În majoritatea limbajelor de programare regula de legare a unui identicator la variabilă depinde de unitatea de program în care apare declarația (*"regula celei mai apropiate unități înconjurătoare"*) ceea ce permite o legare statică (facută de compilator) a identicatorilor la variabile:

Regula stipulează că variabila la care se referă la un moment dat un identicator utilizat într-o unitate de program este cea alocată în cadrul celei mai apropiate unități de program (clasă, metodă, bloc etc) care înconjoară unitatea curentă, inclusiv unitatea însăși, și care conține o declarație a acestuia.

Legătura identicator-variabilă poate rămâne aceeași pe tot parcursul execuției unității de program, așa cum poate să fie pierdută la un moment dat, dacă o altă declarație a aceluiași identicator intervine în interiorul unității de program considerate.

În următoarea declarație de clasă același identicator `x` apare într-o declarație de variabilă globală metodelor în clasa `Clasa` și într-o declarație de variabilă locală a unei metode a clasei:

```

public class Clasa {
    int x=0;           // introduce o variabilă globală, în clasă
    void metodaA() {
        int x=1, y;    // introduce variabile locale a metodei
        y = x;         // OK, y=1 și nu y=0
    }
    void metodaB() {
        int z=1;       // variabila locală
        x = y + z;     //eroare! y este în afara mediului de valabilitate
    }
}

```

Declarația de variabilă locală `int x=1` din `metodaA` dezleagă identificatorul `x` de variabila globală, lăsând-o fără un nume de acces. Astfel, în interiorul metodei `metodaA` variabila globală devine inaccesibilă, iar identificatorul este legat o noua variabilă, locală metodei respective. În `metodaB` însă legătura lui `x` este cea inițială, deoarece declarația variabilei globale nu este mascată de o altă declarație a aceluiași identificator.

Rezultă de aici două consecințe importante:

1. identificatorul nu este adevăratul nume al variabilei. "Numele" variabilei este de obicei ascuns observatorului, fiind conferit de compilator și utilizat apoi de el în codul generat. La execuție numelui `i` se asociază valoarea stângă, neschimbată apoi pe toată durata existenței variabilei;
2. o variabilă continuă să existe și atunci când se execută un cod în care ea este inaccesibilă. De aici necesitatea unui concept care să identifice ciclul de viață al variabilei la execuție. Se numește *domeniul de existență* a variabilei și reprezintă acea porțiune din program în care variabila este prezentă în memorie la execuție. Durata de viață nu coincide întotdeauna cu domeniul de valabilitate a identificatorului variabilei. De altfel cele două caracteristici ale unei variabile se manifestă la momente de timp diferite: domeniul de valabilitate este un concept static, el se manifestă la compilare, iar domeniul de existență a variabilei este un concept dinamic, pentru ca se manifestă la execuție.

În Java, din punct de vedere al modului de stabilire a duratei de viață, variabilele pot fi:

- *variabile ale clasei*, create la momentul încărcării clasei de către încărcătorul de clase în mediul de execuție și menținute cât timp clasa este menținută în mediul de execuție,
- *variabile ale instanței*, create la momentul creării unui obiect și menținute cât timp obiectul există în mediul de execuție sau
- *variabile locale*, create la intrarea execuției într-o metodă sau într-un bloc de instrucțiuni prin executarea declarațiilor variabilelor și menținute până la ieșirea execuției din metodă sau blocul respectiv.

Asupra acestor probleme vom reveni în capitolul 4.

Tipul variabilei

O variabilă nu poate memora orice fel de valori. Cea mai importantă proprietate a unei variabile și în același timp obligatorie în orice declarație de variabilă este *tipul* variabilei. El reprezintă *domeniul de definiție*, adică mulțimea valorilor permise pentru variabila respectivă. Pe lângă domeniul de definiție, tipul determină și operațiile care pot fi efectuate asupra variabilei, uneori furnizând chiar indicații privind implementarea valorilor variabilei.

Vizibilitatea variabilei

Vizibilitatea variabilei este dată de zonele din program din care se poate accesa o variabilă folosind numele ei. Ar rezulta din cele de mai sus că această vizibilitate este dată de domeniul de valabilitate a identificatorului său și este mărginit la unitatea de program în care a fost declarată variabila.

Totuși în Java vizibilitatea variabilei, chiar dacă este un concept înrudit cu domeniul de valabilitate a numelui ei, nu se poate confunda cu acesta.

Din punct de vedere al vizibilității în Java o variabilă poate fi *publică*, *locală pachetului*, *protejată* sau *privată*. Aceste caracterizări pot restrânge sau extinde regulile conferite de domeniul de valabilitate și au semnificații specifice limbajului asupra cărora vom reveni.

Zona de memorare a variabilei

Prin declarație se poate impune și tipul de memorie utilizat pentru a găzdui variabila: memorie statică sau memorie dinamică. Diferențele dintre cele două metode se repercutează, așa cum vom vedea în continuare, asupra duratei de viață a variabilei.

Sintaxa declarației de variabilă

În timp ce numele și tipul variabilei trebuie să existe întotdeauna în declarație, valoarea atributelor de vizibilitate și zona de memorare a variabilei sunt indicate prin prezența unor *modificatori* sau, pur și simplu, prin absența lor. Modificatorii sunt cuvinte-cheie precum: `private`, `public`, `static` etc. care preced tipul în declarație.

Sintaxa declarației de variabilă în Java este următoarea:

```
DeclaratieVariabila ::= { Modificator } Tip Identificator;  
| {Modificator} Tip Identificator [=Expresie] {,Identificator [=Expresie]};
```

În continuare se prezintă câteva exemple de declarații, care asociază anumite atribute variabilelor:

- tipul de date:

```
int a;           // variabilă întreagă, dinamica, cunoscuta în pachet  
Persoana p;     // variabilă de tip Persoana, dinamica etc.
```
- vizibilitatea:

```
private String nume;           // variabilă String privată si dinamica  
public boolean esteVizibil;    // variabilă boolean publică si dinamica
```
- zona de memorie unde este plasată variabila:

```
static int contor; // variabilă întreagă statică, cunoscuta în pachet
```

După cum se vede atributele se pot combina. De exemplu, variabilele, după unitatea de program în care au fost declarate și după tipul de memorie în care trebuie alocate pot fi:

- variabile ale unei instanțe, pe scurt variabile-instanță,
- variabile ale unei clase, pe scurt variabile-clasă,
- variabile locale dinamice,
- variabile locale statice.

Vom explica aceasta clasificare în cele ce urmează.

2.3 MEMORAREA DATELOR DE CĂTRE JVM

Am văzut că Mașina Virtuală Java (JVM) nu este un calculator fizic, ci o mașină simulată prin software. În acest scop, în kit-ul J2SDK există programul `java.exe` care crează mediul pentru execuția unui program Java pe care i-l furnizăm ca argument. Execuția programului se face prin analizarea, instrucțiune cu instrucțiune, a codului său de către un interpretor de bytecode și prin execuția fiecărei instrucțiuni analizate folosind resursele JVM.

Pentru un observator extern funcționarea unui calculator real care execută JVM și care la rândul său execută bytecode poate fi asimilată cu funcționarea unui calculator orientat Java care execută instrucțiuni ale limbajului Java și care are o structură adecvată acestui scop. O asemenea imagine vrea să evoce termenul de mașină virtuală.

Ca orice calculator, JVM are un procesor, o memorie și dispozitive I/O.

Procesorul execută instrucțiuni Java pe care le găsește în memorie, folosind date din memorie și modificând date din memorie. El utilizează regiștrii săi care formează memoria o memorie de lucru, extrem de rapidă, dar nu sunt accesibili în Java. Procesorul accesează memoria internă folosind nume de variabile din program.

Dispozitivele I/O ale JVM sunt capabile să primească și să transmită fluxuri de date de la sau către alte JVM sau sisteme în stare să comunice cu ele.

O resursă importantă a JVM este memoria care este structurată în mai multe componente, fiecare cu rolul ei în arhitectura mașinii.

Așa cum rezultă din Figura 2-3, componentele de memorie utilizate de JVM sunt:

- *memoria dinamică a variabilelor locale ale metodelor*;
- *memoria dinamică a obiectelor și array-urilor din program*. Fiecare obiect sau array creat cu `new` va fi adăugat în memoria heap;

- *memoria statică* conține clasele obiectelor din program, variabilele statice declarate în metode, precum și toate constantele din program;
- *memoria programului* este o memorie read-only din care JVM citește programul.

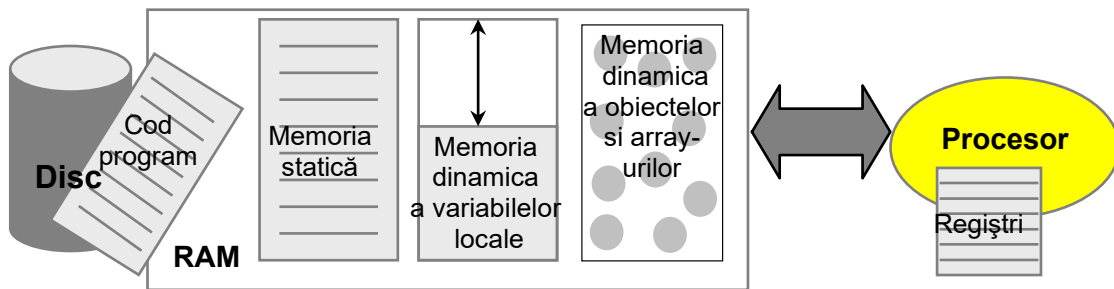


Figura 2-3. Structura memoriei Mașinii Virtuale Java

Fiecare componentă a memoriei mașinii virtuale are propriul său mod de funcționare. Memoria dinamică a variabilelor locale funcționează ca o listă de tip LIFO, *memorie stack*, care impune duratelor de viață a variabilelor locale anumite relații reciproce (duratele de viață a două variabile nu se pot partial suprapune, ci ori sunt disjuncte, ori una este inclusă în cealaltă). În schimb memoria dinamică a obiectelor și array-urilor, numită și *memorie heap*, este dotată cu un mecanism de fragmentare în care nu este important unde se memorează sau de unde sunt eliminate aceste obiecte și array-uri. Cele două memorii dinamice sunt necesare memorării variabilelor locale metodelor (memoria stack) și instantelor de clase și array (memoria heap). Memoria JVM este implementată în memoria RAM și partial pe discul calculatorului real. Programatorul nu trebuie să cunoască cum este implementată, dar ar putea să cunoască structurarea conceptuală a memoriei pentru a ști la ce tip de memorare și acces trebuie să se aștepte.

2.3 TIPURI DE DATE ÎN JAVA

Tipurile de date în Java se împart în două categorii:

- Tipuri *primitive* (built-in), preexistente în limbaj. Ele sunt tipurile întregi, reale, caracter și logic;
- Tipuri *construite*, ad hoc. Aceste tipuri reprezintă entități complexe ce se pot construi folosind mecanisme oferite de limbajul Java și care pot fi clase, array-uri sau interfețe. Aceste tipuri pot fi definite de programator sau definiția lor poate fi extrasă dintr-o bibliotecă existentă de tipuri Java. În ultimul caz avem de-a face cu o reutilizare de tipuri.

După categoria de tipuri cu care se declară, variabilele Java pot fi:

1. variabile de tip primitiv,
2. variabile de tip referință (reference) la un tip T construit existent. Sunt variabile care conțin valoarea stânga a unei variabile de tip T. Un caz particular de valoare de tip reference este `null`. Ea indică, pentru orice tip reference, inexistența unei variabile referite.

2.3.1 Tipuri de date primitive

Nume tip	Descriere	Reprezentare în calculator
<i>Tipuri întregi</i>		
byte	Întreg de un byte	8-biti în complementul lui 2 (1 byte)
short	Întreg scurt	16-biti în complementul lui 2 (2 bytes)
int	Întreg	32-biti în complementul lui 2 (4 bytes)
long	Întreg lung	64-biti în complementul lui 2 (8 bytes)
<i>Tipuri reale</i>		
float	Real cu precizie simplă	32-biti conform standardului IEEE 754 (4 bytes)
double	Real cu precizie dublă	64-biti conform standardului IEEE 754 (8 bytes)
<i>Alte tipuri</i>		
char	Un caracter	16-biti conform standardului Unicode (2 bytes)
boolean	Valoare logică (true sau false)	1 bit/1 byte

Tabelul 1. Tipurile primitive în Java

În Tabelul 1 se prezintă tipurile primitive de date utilizate în Java. Prima coloană prezintă numele tipului, așa cum trebuie inclus în declarațiile de tip. A doua coloană este descrierea semnificației tipului în raport cu mulțimi de valori cunoscute din matematică și logică. A treia coloană prezintă lungimea zonei de memorie alocată pentru găzduirea unei valori a tipului și regulile cu care valoarea respectivă este codificată în șiruri de 0 și 1.

Se știe că în ciuda faptului că numele tipurilor primitive ne amintesc de mulțimi cunoscute din matematică cum ar fi mulțimea numerelor întregi sau mulțimea numerelor reale, nu toate numerele întregi (care sunt o infinitate) se pot reprezenta în calculator și cu atât mai puțin se pot reprezenta toate numerele reale.

Nume tip	Lung. zona	Domeniu de valori	Precizie
boolean	1 bit	true sau false	exactă
byte	1 byte	-128 ÷ 127	exactă
short	2 bytes	-32.768 ÷ 32.767	exactă
char	2 bytes	'\u0000' ÷ '\uFFFF'	exactă
int	4 bytes	-2.147.483.648 ÷ 2.147.438.647	exactă
long	8 bytes	-9.223.372.036.854.775.808L ÷ 9.223.372.036.854.775.807L	exactă
float	4 bytes	±3,40282347E+38F	cca. 10 ⁻⁸
double	8 B	± 1,79769313486231570E+308	cca. 10 ⁻¹⁸

Tabelul 2. Domeniile de valori și precizia reprezentării tipurilor primitive

Regulile de codificare a numerelor întregi și reale în calculator impun niște limite precise domeniului de valori reprezentate în calculator, cât și preciziei de reprezentare a valorilor în cazul numerelor reale. Tabelul 2 prezintă care sunt domeniile de valori acoperite de tipurile din Java și precizia asigurată de Java în cadrul acestor domenii.

Pentru a lucra cu valori în afara acestor domenii și cu precizii sporite, Java oferă alte posibilități decât tipurile primitive. Este vorba de două clase standard specializate în reprezentarea unor asemenea valori: `BigDecimal` și `BigInteger`

Conversii de tipuri primitive

1. *Conversii implicite.* Java realizează, dacă contextul utilizării valorii o cere, conversii implicite a valorilor de tip primitiv. Dacă din context rezultă că pentru a se putea continua calculul o valoare de un anumit tip (a unei variabile sau a unei expresii) trebuie să fie convertită într-o valoare echivalentă de alt tip și dacă această conversie este prevăzută de limbaj ca implicită, atunci compilatorul Java introduce operații care convertesc automat valoarea. Conversiile implicite în Java sunt reprezentate prin sensul săgeților în Figura 2-1.

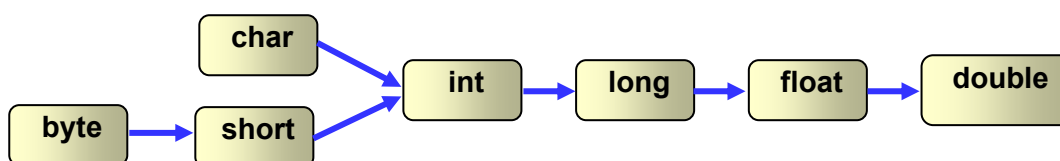


Figura 2-1. Conversii implicite în Java

Conversiile automate se includ de compilator în următoarele cazuri:

1. la atribuirea unei valori de un tip unei variabile de un alt tip,
2. la apelul de metodă, la efectuarea transmiterii parametrilor, dacă tipul argumentelor este diferit de tipul parametrilor formali, și
3. în calculul de expresii aritmetice, când o operație aritmetică (+, -, *, / etc.) trebuie efectuată între două valori de tipuri diferite.

Ideea de bază a acestor conversii este că în cursul lor trebuie să nu se piardă informații, pentru că fiind automate, compilatorul nu își poate lua răspunderea realizării unei operații în care pot fi modificate în mod necontrolat niște informații. De aceea Java crează o ierarhie de tipuri de date primitive (toate în afară de tipul `boolean` care nu este convertibil), dominată de tipul `double`, în care tipurile “mai slabe” sunt cele care au domeniul valorilor inclus în domeniul altor tipuri “mai tari”. De exemplu, toate valorile de tip `short` se regăsesc în valori de tip `int`, iar toate valorile `int` se regăsesc în domeniul valorilor `long`.

2. *Conversii explicite.* *Casting* este o cerere explicită de realizare a unei conversii de tipuri de date. Ea se realizează precedând valoarea de convertit cu un “cast” format din numele tipului în care se dorește să se facă conversia, cuprins între paranteze:

```
short x = (short)75;  
int pixels = (int)(width/scale);
```

În asemenea cazuri compilatorul va efectua conversia cerută chiar dacă ea produce și o pierdere de informație (ca în cazul conversiei de la `float` la `int`, sau de la `double` la `float`). În cazul unei conversii imposibile (de la `boolean` la `int`, de exemplu), compilatorul semnalează eroare de compilare.

2.3.2 Tipuri de date construite

Tipurile de date construite în Java folosesc trei mecanisme principale capabile să adauge la tipurile existente noi tipuri de date. Cele trei mecanisme sunt:

1. definiția de clasă,
2. definiția de array,
3. definiția de interfață.

Odată definit un tip construit, Java dă posibilitatea declarării de variabile cu acest tip.

Definiția de clasă

Definiția de clasă introduce un tip de date având același nume cu numele clasei. Cu o clasă putem să ne creem câte instanțe ne sunt necesare în program. Aceste instanțe sunt efectiv valorile noului tip definit.

O definiție de clasă conține descrierea componentelor interne ale valorilor noului tip: pe de o parte, structuri de date și, pe de alta parte, metode definite peste aceste date.

În Java o clasă se definește printr-o declarație de clasă:

```
DeclaratieClasa ::= class Identificator {  
    ListaDeclaratiiVariabile  
    ListaDeclaratiiMetode  
}
```

Identificator introduce numele noului tip de date. *ListaDeclaratiiVariabile* introduce variabile ale instanței și variabile ale clasei. *ListaDeclaratiiMetode* introduce metode ale instanței și metode ale clasei.

Să presupunem că avem un tip de date construit *T*. El a fost introdus în program pentru ca obiecte ale programului stii și vor să îl folosească. Există în esență două abordări pe care un client al tipului¹ le poate avea când lucrează cu valorile de tip *T* în funcție de modul în care a fost definit *T*:

1. clientul trebuie să cunoască structura internă a valorilor de tip *T*. În acest caz, va accesa direct componentele valorii pentru a le citi sau a le modifica. Noua valoare se comportă deci ca o structură de date complexă formată din alte date mai simple, agregate sub un unic nume. Este echivalentă cu o instanță a unui tip *struct* în C sau a unui *record* în Pascal;
2. clientul nu trebuie să cunoască structura internă a valorilor de tip *T*. În acest caz, va utiliza exclusiv operațiile definite de clasă pentru a citi și modifica aceste valori.

Dintre cele două abordări, a doua este cea urmată de orientarea spre obiecte, deși într-un program putem întâlni ambele abordări și, încă mai frecvent, clase în a căror definiție se regăsesc ambele abordări: acces direct la unele date împreună cu metode de acces la altele.

Regulă. Deși sunt cazuri în care prima abordare este cea indicată, în acest curs vom folosi cu precădere al doilea mod de a construi noi tipuri de date deoarece reflectă mai bine modul de gândire orientat pe obiecte.

Pentru a înțelege mai bine această recomandare și a sesiza care este diferența între cele două scenarii, să pornim de la originile abordării pe obiecte, adică de la tipurile abstracte de date.

Un *tip abstract de date* (ADT-Abstract Data Type) formalizează tipurile de date folosite în diversele limbaje de programare, dându-le un model matematic. Informal, un ADT reprezintă o mulțime de valori și operațiile asociate care sunt specificate în mod independent de orice implementare posibilă.

Definiție. Un tip de date abstract este o algebră $\langle V, F \rangle$ unde $V = \{V_1, \dots, V_m\}$ este o mulțime finită de mulțimi numite *domenii de definiție*, iar F este o mulțime de funcții numite *operații* $f_i : V_{i_1} \times \dots \times V_{i_n} \rightarrow V_{i_k}$, $V_{i_j} \in V$ pentru $j=k$ și $j=1, \dots, n$ împreună cu o mulțime finită de constrângeri asupra operațiilor din F . Un ADT "exportă" celui care îl utilizează doar numele tipului și setul de operații, denumit *interfața* ADT-ului.

Principiul fundamental promovat de ADT-uri este *încapsularea datelor*, principiu uneori referit și ca *ascunderea informațiilor* (information hiding). El stipulează evitarea prezentării detaliilor de implementare a tipului de date celor care sunt interesați în utilizarea tipului, adică clienților săi. Acestora le este suficientă interfața operațiilor pentru a putea lucra cu instanțele de ADT.

Exemplul clasic prezentat în [Liskov 75] și în alte materiale de referință este cel al stivei (listă LIFO) de întregi. Definiția tipului *Stack* conform modelului ADT este următoarea:

¹ Înțelegem prin client al tipului o metodă care are nevoie în algoritmul său de tipul respectiv.

```

ADT Stack is:
  Domains
    V = { Z  $\cup$  {error}, Stack  $\cup$  {error}}
  Interface
    create:       $\rightarrow$  Stack
    push : Stack  $\times$  Z  $\rightarrow$  Stack
    pop  : Stack  $\rightarrow$  Stack  $\cup$  {error}
    top  : Stack  $\rightarrow$  Z  $\cup$  {error}
  Predicates
    C1    top (push (s,i)) = i
    C2    pop (push (s,i)) = s
    C3    top (create) = error
    C4    pop (create) = error

```

■

Traducerea acestui ADT printr-o definiție de clasă în Java, ne conduce la clasa definită în Programul 1.

```

class Stack {
  private int[] list;
  private int v;
  private final int SIZE=100;
  public Stack(){
    list = new int[SIZE];
    v = -1;
  }
  public void push(int i){
    if (v<SIZE)
      list[++v] = i;
  }
  public void pop() throws EmptyStackException{
    if (v>=0)
      --v;
    else
      throw new EmptyStackException();
  }
  public int top() throws EmptyStackException{
    if (v>=0)
      return list[v];
    else
      throw new EmptyStackException();
  }
}

```

Programul 1. Clasa Stack

Explicarea programului Stack

- Stiva este implementată static folosind un array de lungime 100 în care poziția vârfului stivei este indicată de valoarea unei variabile întregi. Definiția de clasă în Java utilizează modificatorii de acces `private` și `public` pentru ascunderea datelor componente noului tip și, respectiv, exportarea operațiilor. Datele interne sunt: `list` și `v`. Ele privesc implementarea tipului și de aceea sunt declarate `private`, și prin aceasta ascunse în interiorul valorilor tipului `Stack`. În schimb, metodele `push`, `pop` și `top` sunt declarate `public` și astfel sunt exportate clienților. Exportarea privește exclusiv numele operației și nu algoritmul ei, care este strâns legat de componentele clasei și de aceea este și el ascuns clientului.
- Operației `create` din definiția ADT a stivei îi corespunde în definiția clasei constructorul `Stack`. Constructorul este și el declarat `public`, făcând astfel parte din interfața exportată clientului. În acest fel, utilizând chemarea constructorului `Stack`, clientul poate crea noi instanțe ale tipului.

Din definiția de clasă `Stack` comparată cu ADT-ul `Stack` se poate deduce ușor că:

Regulă. O clasă de obiecte reprezintă o implementare a unui ADT.

Definiția de array

Un array este o agregare liniară a unui număr fix de elemente de același tip. Numărul de elemente se numește lungimea array-ului. Elementele sunt numărate începând de la 0. Numărul de ordine al elementelor se numește *index*. Java tratează un array ca o unică valoare în care programatorul poate naviga utilizând indecșii.

Definiția unui array introduce un tip ale cărui instanțe sunt array-uri alcătuite din elemente omogene care au tipul indicat de definiție.

Exemple:

```
int list[100];
Persona p[];
```

Definiția de interfață

O definiție de interfață în Java este constituită dintr-o listă de declarații de operații fără a propune și implementările lor. Pentru ADT-ul Stack interfața Java este următoarea:

```
interface IStack {
    public void push(int i);
    public void pop();
    public int top();
}
```

și în afara lipsei lui `create`, este identică cu interfața specificată în ADT.

Regulă. O definiție de interfață în Java descrie un ADT.

Java consideră definiția de interfață ca definiție a unui tip de date. Se pot declara variabile având ca tip o interfață, dar nu se pot crea obiecte dintr-o interfață. Atunci este normal să ne întrebăm: ce valori va memora o variabilă-interfață? Răspunsul este interesant și cu consecințe extrem de importante pentru programatori:

Regulă. O variabilă interfață în Java memorează obiecte ale oricărei clase ce implementează ADT-ul interfeței.

Deci putem scrie în Java următoarea declarație de variabilă:

```
IStack stack = new Stack();
```

unde `Stack` este clasa definită în Programul 1. Mai mult, orice altă instanță de clasă care implementează ADT-ul `Stack` poate fi memorată în `stack`.

Instanțele tipurilor construite

Cele trei categorii de tipuri construite în Java au instanțe memorate în memoria dinamică a JVM, numită și memorie heap. Aici fiecare instanță de obiect sau array ocupă o zonă de bytes în care își memorează informațiile proprii. Aceste zone sunt alocate de mediul de execuție al JVM, în limita spațiului disponibil și sunt complet gestionate de mediu, în mod transparent pentru utilizator.

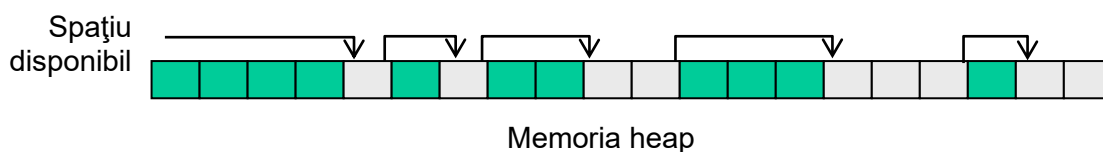


Figura 2-2. Zone alocate și disponibile în memoria heap

Situația la un moment dat în memoria heap poate fi reprezentată ca în Figura 2-2.

Gestiunea instanțelor privește în principal ciclul de viață a acestora. Începutul ciclului este legat de crearea instanței. Ea este înscrisă în cod și decisă la execuția programului (de regulă, prin execuția operatorului `new`). La un moment dat, instanța poate însă să își încheie rostul ei în program, devenind inutilă acestuia și, în plus, ocupând inutil zona alocată de memorie. În majoritatea altor limbaje (C, C++, Pascal) programatorul are posibilitatea de a cere în asemenea situații distrugerea acestor instanțe din memorie. Dar nu în Java! În Java eliminarea instanțelor inutile este decisă de mediul de execuție. JVM decide pe considerente proprii când să elibereze spațiul ocupat de o instanță care a devenit inutilă! Pentru aceasta JVM are un instrument eficient numit *garbage collector*. Acesta este un proces aparținând mediului de execuție Java și care de obicei este suspendat în așteptarea momentului potrivit pentru a interveni în execuție. Când

intervine, el detectează toate instanțele inutile și le elimină, măbind astfel spațiul disponibil în memoria heap. Decizia de a chema garbage collector-ul aparține JVM și este luată de aceasta la apariția unor evenimente dintre care cel mai important este, desigur, ocuparea excesivă a memoriei, peste limitele fixate la configurarea calculatorului.

Variabile referință

Instanțele din memoria heap sunt adevăratele protagoniste ale programelor scrise în Java. Programele le crează și apoi le utilizează pentru a compune încet-încet prelucrarea globală, cea dorită de programator. Utilizarea înseamnă acces în citire sau scriere la informațiile memorate de instanțe, precum și pentru apelarea metodelor instanței. Pentru a accesa aceste instanțe, trebuie să existe o posibilitate de a le localiza în memorie. Ele nu pot fi identificate printr-un identificator în textul programului pentru că pot fi foarte multe în memorie în cursul execuției, deci ar fi greu de gestionat dându-le fiecăreia un nume. Pe de altă parte, ciclul lor de viață nu este determinat de structura statică a programelor care asigură o “primerire” a memoriei oricâte ori se iese dintr-o unitate de program, ci de necesitățile programului la momentul execuției care sunt greu de prevăzut.

Toate acestea au condus la o soluție originală de a lucra cu instanțe de clase și de array: *accesul indirect al instanței*, care se bazează pe *variabile-referință*. O variabilă referință este o variabilă care are ca valori referințe (adrese) ale unor instanțe din program.

Regulă. În Java instanțele de obiecte și array nu au un nume la nivelul programului: accesul la ele se realizează exclusiv prin valoarea lor stânga memorată în variabile-referință.

Tipul variabilei referință este dat de tipul instanțelor a căror valoare stânga se va memora în variabila.

Să exemplificăm variabilele referințe cu ajutorul clasei `Stack` prezentată în Programul 1.

Pentru a lucra cu un obiect `Stack` vom utiliza clasa `TestStack` din Programul 2.

```
1.    public class TestStack {
2.        public static void main(String [] args) throws Exception {
3.            Stack st = new Stack();
4.            st.push(5);
5.            st.push(-12);
6.            System.out.println( "Varful stivei este: " + st.top());
7.            st.pop();
8.            System.out.println( "Varful stivei este: " + st.top());
9.        }
10.    }
```

Programul 2. TestStack

Explicarea programului TestStack

- Metoda `main` în linia 3 declară o variabilă referință `st` de tip `Stack` pe care o inițializează cu un obiect nou creat, de asemenea de tip `Stack`.
- Variabila `st` este locală metodei `main`, deci o vom găsi în memoria stivă ce găzduiește variabilele declarate în metodele claselor. Ea memorează unica referință la obiectul `Stack` creat. Dacă ștergem această referință memorând altceva în `st`, am pierdut orice posibilitate de a mai accesa obiectul creat. Acesta va fi eliminat de garbage collector atunci când va fi nevoie.
- Obiectul, conform definiției clasei `Stack` conține două variabile instanță: `list` și `v`. La momentul creării obiectului, constructorul clasei inițializează ambele variabile. `list` este inițializată cu o referință la un array de 100 de întregi tocmai atunci creat, în timp ce `v` este inițializată cu `-1`, valoare care indică că stiva este goală.

- Accesul la obiectul `Stack` se poate face doar prin variabila ce îi conține referința, așa cum se întâmplă cu cele două chemări de metode `push` aplicate variabilei `st` din liniile 4 și 5 și care depun 5 și -12 în stivă.

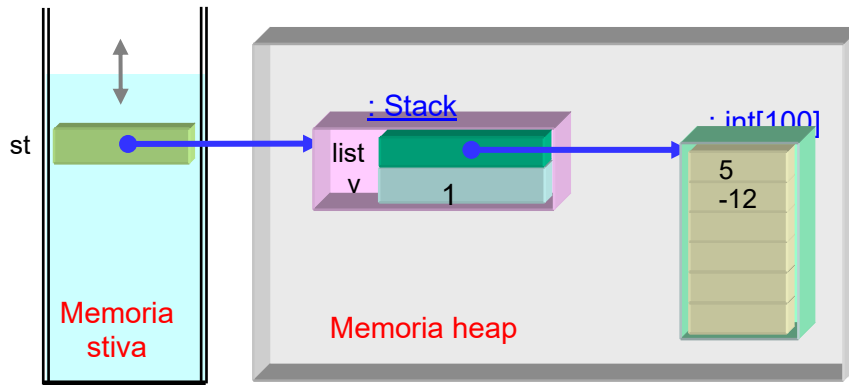


Figura 2-3. Utilizarea variabilelor referință

- Apelările metodei folosesc un algoritm specificat în implementarea metodei `push` pe care o găsim în clasa `Stack`. Algoritmul utilizează variabila referință `list` pentru a depune în array-ul creat, la poziția indicată de `v`, valoarea transmisă prin parametrul `i`; ea este mai întâi 5 și apoi -12. De fiecare dată `v` este incrementat, așa încât ajunge la valoarea 1. Situația din memorie în acest moment este prezentată în Figura 2-3.
- Linia 6 vizualizează pe ecran valoarea din vârful stivei. După cum rezultă din execuția metodei `top` aplicată lui `st`, în acest moment în vârful stivei este -12. Pentru a ajunge la aceasta ultimă valoare memorată în stivă, metoda `top` utilizează variabila referință `list` care punctează la un array. Accesul la array se face indexat. În program se folosește expresia `list[v]`. Aceasta, în mod indirect, plecând de la variabila referință `list` care conține valoarea stânga a array-ului, o incrementează cu 1 ajungând astfel la al doilea element al array-ului în care găsește -12.
- Linia 7 prezintă chemarea metodei `pop` aplicată aceleiași variabile `st`, deci instanței `Stack`. Algoritmul lui `pop` din clasa `Stack` decrementează valoarea lui `v` care astfel devine 0. Se observă că valoarea -12 nu se șterge, dar nu se va mai putea ajunge la ea decât prin eventuale `push`-uri succesive (nu este cazul în acest program), când va fi substituită cu o altă valoare.
- Linia 8 afișează pe ecran valoarea din vârful stivei. După cum rezultă din execuția metodei `top` aplicată lui `st` în acest moment în vârful stivei este 5 (`lista[0]=5`).

2.3.3 Clasa String

În această secțiune vom prezenta pe scurt cazul lui `String`, un tip Java construit pe baza unei definiții de clasă. După numere, el reprezintă tipul de date cel mai important, fiind frecvent utilizat în programe, aceluiași subiect îi vom dedica întregul capitol 5.

`String` face parte din tipurile standard Java, iar clasa tipului aparține pachetului `java.lang`. Apartenența la acest pachet face ca tipul `String` să fie întotdeauna disponibil pentru a fi utilizat de programator, fără a fi necesar importul lui explicit în program.

Regulă. Toate clasele din pachetul `java.lang` sunt automat importate în programele Java. Ele reprezintă conceptele de bază (core concepts) ale limbajului, cele care sunt cel mai frecvent utilizate.

Clasa `String` reprezintă șirurile de caractere în Java. Orice text, pentru a fi prelucrat, trebuie încapsulat într-o instanță a clasei `String`.

Șirurile de caractere pot apărea în programe ca și constante, închise între ghilimele care nu fac parte din șir:

```
"Buna ziua!"
"Astazi magazinul este inchis!"
"m=Suma(xi)/n, i=1..n, este expresia mediei aritmetice"
```

sau memorate în variabile de tip `String`:

```
String salut = "Buna ziua!";  
String mesaj;
```

Într-o variabilă de tip `String` șirurile se pot memora prin atribuiri de șiruri constante:

```
mesaj = "Astazi magazinul este inchis!";
```

sau de șiruri concatenate:

```
mesaj = salut + "Astazi magazinul este inchis!";
```

Numărul de caractere dintr-un șir constituie *lungimea șirului* și este obținută prin apelarea metodei `length` pentru instanța respectivă. De exemplu:

```
salut.length()
```

restituie valoarea 10 care este lungimea șirului "Buna ziua!".

Regulă. În Java șirurile de caractere sunt obiecte, instanțe ale clasei `String`.

Cu orice șir, constantă sau variabilă, se poate lucra chemând metode ale clasei

`String`.

Există și șiruri de lungime 0. Un asemenea șir este constantă și reprezintă "șirul vid": ""

Conversii între șiruri și numere

Sunt numeroase situațiile în care un șir de caractere trebuie convertit într-un alt șir sau într-un număr, dacă știm că șirul este un șir de cifre. Sunt alte situații, de asemenea frecvente, în care un număr ar trebui convertit într-un șir, de exemplu pentru a-l putea afișa.

Să presupunem că trebuie să convertim șirul "Buna ziua!" memorat în variabila `salut` într-un același text, scris însă cu litere mari, adică "BUNA ZIUA!". Se poate utiliza una din metodele clasei `String`, metoda `toUpperCase` astfel:

```
String salutLitereMari = salut.toUpperCase();
```

Invers, putem transcrie același text folosind doar literele mici:

```
String salutLitereMici = salut.toLowerCase();
```

În aceste cazuri, de fiecare dată se crează o nouă instanță `String` în memoria dinamică, instanța inițială rămânând neschimbată.

Regulă. În Java, șirurile de caractere `String` sunt obiecte imutabile (după crearea lor nu mai pot fi modificate).

Dintr-un șir putem să extragem subșiruri folosind metoda `substring` a clasei `String`.

```
String primul = salut.substring(0, 4); // extrage primul cuvânt "Buna"  
String alDoilea = salut.substring(5, 9); // extrage al doilea cuvânt "Ziua"  
String alDoilea = salut.substring(5); // extrage șirul "Ziua!"
```

Când dorim să convertim un număr într-un șir de caractere avem la dispoziție mai multe posibilități. Cea mai simplă este următoarea:

```
int an = 1982;  
String anulNasterii = "" + an;
```

În această soluție se "adună" șirul vid cu un număr întreg. În asemenea situații, Java transformă automat numărul într-un șir de caractere ("1982"), iar adunarea se face între șiruri. În Java operatorul `+` între șiruri de caractere simbolizează operația de "concatenare" între cele două șiruri. Concatenând un șir cu șirul vid se obține exact același șir.

O altă posibilitate constă în utilizarea unei clase Java `Integer` care reprezintă obiectele care conțin un număr întreg. Clasa are diferite metode utile printre care metoda `toString` de conversie a unui număr întreg într-un șir de caractere².

```
String anulNasterii = Integer.toString(an);
```

Dacă numărul de convertit este real, aceste posibilități de conversie rămân aceleași. Ceea ce se schimbă este clasa folosită pentru conversie.

Să vedem mai întâi, printr-un exemplu, cum funcționează "concatenarea" unui șir de caractere cu un număr real.

```
double procentDobanda = 0.0657;  
double imprumut = 2450;  
System.out.println("Dobânda de platit: Euro " + imprumut*procentDobanda);
```

Acest cod afișează textul:

² După cum vom vedea în capitolele următoare toate clasele Java au o asemenea metodă.

Dobânda de platit: Euro 160.475

Acum se poate înțelege de ce putem amesteca în parametrul metodei `println` atât șiruri cât și numere: deoarece rezultatul trebuie să fie un șir, compilatorul va include cod de conversie automată a numerelor la șiruri.

Pentru conversie s-ar putea utiliza și clasa `Double` cu metoda `toString`:

```
String dobanda = Double.toString(imprumut*procentDobanda);
```

Cu un efect similar: valoarea `160.475` este memorată în `dobanda`. Dat fiindcă este vorba de euro, ultima zecimală este inutilă, deci am dori să nu fie generată la conversie (sau să se facă o eventuală rotunjire). Asemenea exigențe nu pot fi realizate de conversiile automate și trebuie intervenit în alt mod, așa cum se va explica într-unul din capitolele următoare.

Conversia de la un șir de caractere la un număr, în schimb, nu mai este automată, deci va trebui să utilizăm metode speciale de conversie.

Dacă se dorește convertirea unui șir de cifre într-un număr întreg se poate utiliza aceeași clasă `Integer` însă cu metoda `parseInt`:

```
String strAn = "1982";  
int iAn = Integer.parseInt(strAn);
```

Dacă se dorește convertirea unui șir într-un număr real se poate utiliza aceeași clasă `Double` însă cu metoda `parseDouble`:

```
String strProcentDobanda = "0.0655";  
double iProcentDobanda = Double.parseDouble(strProcentDobanda);
```

2.4 CONSTANTE

Într-un limbaj de programare prin *constantă* se înțelege un element textual din program care reprezintă o valoare fixă ce nu se poate modifica în cursul execuției programului. Valoarea constantei rezultă direct din modul în care este scrisa sau dintr-o declarație precedentă în cazul constantelor simbolice. De exemplu, în instrucțiunea:

```
int an = 2004;
```

2004 este o constantă. Valoarea ei se poate imediat deduce ca fiind numărul întreg 2004. Fiecare tip de date are propria sintaxă de scriere a constantelor.

În funcție de tipul de date de care aparțin, constantele pot fi:

- constante numerice întregi:

Exemple: `16547` `-654` `987634872L` `0777` `0x5A67`

- constante numerice reale:

Exemple: `2.25` `-4556.43` `3.1415927F` `12.5e6` `19E-65`

- constante booleene: `true` `false`

- constante caractere:

Exemple: `'a'` `'#'` `'5'` `'\377'` `'xA'` `'\u4323'`

Notă. Caracterul special `'\'` (backslash) este folosit în următoarele secvențe "escape" pentru reprezentarea constantelor caracter necesare formătărilor informațiilor afișate la consolă:

`'\n'` Newline. Poziționează cursorul la începutul liniei următoare.

`'\t'` Tab orizontal. Așează cursorul la următorul tab.

`'\r'` Carriage return. Așează cursorul la începutul liniei curente, trecând în modul suprascriere.

`'\f'` Form feed. Avansează cu o pagină.

`'\\'` Backslash. Este folosit pentru a afișa caracterul `'\'`.

`'\''` Apostrof. Este folosit pentru a afișa caracterul apostrof.

`'\"'` Ghilimele. Este folosit pentru a afișa caracterul ghilimele.

- constante șir de caractere. Sintaxa lor este: un șir de caractere cuprins între ghilimele "...". Aceste constante sunt unicul caz de constante-obiect în Java: valoarea unei asemenea constante nu este o valoare a unui tip primitiv ca în cazurile de mai sus, ci o valoare reference care "punctează" la un obiect al clasei `String`. Obiectul conține șirul de caractere conținut între ghilimele.

Exemplu: `"acesta este un șir."`

Constantele simbolice sunt declarate în Java ca niște variabile, doar că sunt precedate de modificatorul *final*. Exemplu:

```
final double PI = 3.141592653589793238;
```

Uneori constantele sunt declarate ca variabile clasă ca `MAX_VAL` de mai sus:

```
static final int MAX_VAL = 14567;
```

3. COMPONENTE DE BAZĂ ALE PROGRAMELOR

3.1 EXPRESII

O *expresie* este descrierea unui calcul a cărui evaluare produce o valoare unică, de un tip bine stabilit. Componentele unei expresii sunt: purtătorii de valori, numiți *operanzi*, și simbolurile folosite pentru operații, numite *operatori*.

Operanzii unei expresii pot fi variabile, constante, apeluri de metode sau alte expresii. Această definiție recursivă indică faptul că putem construi expresii oricât de lungi plecând de la alte expresii.

În funcție de tipul rezultatului expresiei și a operatorilor folosiți pentru a-l obține, expresiile sunt:

- expresii cu rezultat o valoare primitivă și care, la rândul lor, sunt:
 - expresii numerice,
 - expresii booleene,
 - expresii logice;
- expresii cu rezultat un obiect.

3.1.1 Operatori în Java

În Tabelul 1 se prezintă operatorii din Java, împreună cu semnificația lor, clasificați pe categorii.

Operator	Semnificație	Operator	Semnificație	Operator	Semnificație
Operatori de comparație		Operatori logici		Operatori pe șiruri	
<	mai mic	&&	și (evaluare scurtă)	+	concatenare
<=	mai mic sau egal		sau (evaluare scurtă)	Operatori de deplasare	
>	mai mare	&	și (evaluare completă)	<<	depl. la stânga
>=	mai mare sau egal		sau (evaluare completă)	>>	depl. la dreapta cu extensie semn
==	egal	^	sau exclusiv (evaluare completă)	>>>	depl. la dreapta cu extensie 0
!=	diferit	!	not		
		~	not pe biti		
Operatori aritmetici		Operatori multifuncționali			
*	înmulțire	++	incrementează cu 1	/=	împarte cu valoarea specificată
/	împărțire	--	decrementează cu 1	%=	modulo valoarea specificată
%	modulo	+=	incrementează cu valoarea specificată	&=	“și” pe biți cu valoarea specificată
+	adunare	-=	decrementează cu valoarea specificată	=	“sau” pe biți cu valoarea specificată
-	scădere	*=	multiplică cu valoarea specificată	^=	“xor” pe biți cu valoarea specificată
		<<=	deplasează stânga cu valoarea specificată	>>>=	deplasează dreapta fără semn cu valoarea specificată
		>>=	deplasează dreapta cu semn cu valoarea specificată	(tip)	conversie explicită a operandului la tipul specificat
instanceof determină dacă un obiect aparține unei clase					
new crează un nou obiect					
?: decizie					
, secvență					

Tabelul 1. Operatori în Java

Clasificarea din tabel se bazează pe semantica operațiilor indicate de acești operatori.

În afara acestei clasificări, operatorii se mai pot clasifica în:

- Operatori unari, cu un singur operand: `++ -- + - ~ ! new (tip)`
- Operatori binari, cu doi operanzi: `* / % + - << >> >>> <> <= >= > == !=`
`& ^ | && || = += -= *= . . . ,`
- Operatori ternari, cu trei operanzi: `?:`

3.1.2 Expresii numerice

Într-o operație numerică, în funcție de tipul operanzilor se determină și tipul rezultatului. Aceasta înseamnă că folosim același simbol pentru a indica o operație care se poate realiza între operanzi de tipuri diferite. Vom spune în acest caz că operatorii sunt supraîncărcați ("overloaded"), deci semantica lor este multiplă, în funcție de contextul utilizării.

Regulile de evaluare a tipului rezultatului unei operații numerice în Java sunt simple:

dacă operanzii sunt de același tip

se efectuează operația conformă tipului operanzilor, iar rezultatul va fi de același tip

altfel

se convertește valoarea operandului de tip "mai slab" aducând-o la tipul celuilalt operand
 se efectuează operația conformă tipului "mai tare", iar rezultatul va fi de același tip.

■

Exemplu

Să considerăm cazul unui program care calculează valoarea totală în euro și fracțiuni de euro a monedelor dintr-un portmoneu. Se știe că monedele în valută europeană sunt de 1, 2, 5, 10, 20 și 50 de cenți și de 1 și 2 euro.

Programul care rezolvă această problemă îl vom numi Monede și va fi alcătuit, ca toate programele de până acum, dintr-o singură clasă având o singură metodă, `main`.

Explicarea programului Monede

Programul prezentat în Programul 1:

1. memorează în variabile situația monedelor din portmoneu,
2. calculează numărul de cenți aflați în portmoneu,
3. extrage numărul întreg de euro, împărțind numărul de euro la valoarea unui euro,
4. calculează restul împărțirii la valoarea unui euro care reprezintă cenții și, în fine,
5. scrie acest rezultat pe un rând.

Programul a declarat ca niște constante `final int` valoarea în cenți a diverselor monede.

Se poate observa utilizarea unei convenții de scriere a identificatorilor care ar trebui întotdeauna respectată când scriem programe în Java.

Regulă:

- Numele de clase (dar se aplică și la interfețe) încep cu literă mare. Dacă semnificația clasei nu se poate reda într-un singur cuvânt, se pot reuni toate cuvintele într-un singur șir în care fiecare cuvânt începe cu literă mare. De exemplu: `PrimulProgram` sau `BunaZiuaTuturor`;
- Numele de variabile și metode se scriu cu litera mică;
- Numele de constante simbolice se scriu complet cu litere mari.

În calculul variabilei `total` din programul `Monede` se observă existența unei expresii în care apare operatorul `=` și, în mod repetat, operatorul `+`. Este clar că se dorește să se facă întâi suma și apoi atribuirea, dar cine stabilește această ordine?

Regulă. În limbajele de programare, în cazul expresiilor se aplică reguli precise privind ordinea de evaluare a operațiilor. Această ordine este dată de *precedența* și *asociativitatea* operatorilor aflați în alcătuirea expresiei.

În calculul totalului de mai sus ordinea dorită este obținută din faptul că în Java `=` are o precedență mai mică decât `+`. Deci operația indicată de `=` se va declanșa după cele indicate de `+`.

Regulă. În expresii cu operatori cu precedențe diferite, ordinea evaluării lor este dată de ordinea descrescătoare a precedenței.

```

public class Monede {
    public static void main (String[] args) {
        final int VAL_C01 = 1;
        final int VAL_C02 = 2;
        final int VAL_C05 = 5;
        final int VAL_C10 = 10;
        final int VAL_C20 = 20;
        final int VAL_C50 = 50;
        final int VAL_E01 = 100;
        final int VAL_E02 = 200;
        // Se memoreaza în 8 variabile numarul de monede din portmoneu
        int c01 = 3;
        int c02 = 6;
        int c05 = 5;
        int c10 = 7;
        int c20 = 2;
        int c50 = 4;
        int e01 = 3;
        int e02 = 1;
        // Se calculeaza totalul
        int total = c01*VAL_C01 + c02*VAL_C02 + c05*VAL_C05 + c10*VAL_C10
            + c20*VAL_C20 + c50*VAL_C50 + e01*VAL_E01 + e02*VAL_E02;
        // Se calculeaza numarul de euro si centii
        int euro = total / VAL_E01;
        int centi = total % VAL_E01;
        // Se tipareste rezultatul
        System.out.print("Valoarea din portmoneu este de ");
        System.out.print(euro + " euro si ");
        System.out.println(centi + " centi.");
    }
}

```

Programul 1. Calculul sumei monedelor

Tip operator	Operatori	Asociativitate
operatori postfixați	[] . (parametri) expr++ expr--	dreapta
operatori unari	++expr --expr +expr -expr ~ !	dreapta
creare sau cast	new (tip)expr	dreapta
multiplicativi	* / %	stânga
aditivi	+ -	stânga
deplasare	<< >> >>>	stânga
relaționali	< > <= >= instanceof	stânga
egalitate	== !=	stânga
AND pe biți	&	stânga
XOR pe biți	^	stânga
OR pe biți		stânga
AND logic	&&	stânga
OR logic		stânga
condițional	?:	stânga
atribuiri	= += -= *= /= %= &= ^= = <<= >>= >>>=	dreapta
secvența	,	stânga

Tabelul 2. Precedența și asociativitatea operatorilor

În Tabelul 2 se prezintă în ordinea descrescătoare a precedenței operatorii din Java. Se observă că + are o precedență superioară la =, deci va fi executat înaintea acestuia din urmă. Dar în ce ordine se execută suma variabilelor din partea dreaptă a lui =? Deși în acest caz ordinea lor nu are importanță pentru rezultatul final (adunarea o putem într-o primă aproximație considera

comutativă și când este efectuată de calculator!), în alte cazuri este important de știut în ce ordine sunt efectuate operațiile într-o secvență de operații cu aceeași precedență.

Regulă. În expresii cu operatori cu aceeași precedență ordinea operațiilor este de la stânga la dreapta sau de la dreapta la stânga în funcție de asociativitatea operatorilor (asociativitate dreapta, respectiv asociativitate stânga).

În Tabelul 2 se prezintă în ultima coloană asociativitatea fiecărui grup de operatori. Deci în suma din programul precedent, prima operație evaluată va fi cea mai din stânga, adică: `c01*VAL_C01 + c02*VAL_C02` după care se va trece spre dreapta la următorul `+` și se evaluează suma dintre rezultatul sumei precedente și `c05*VAL_C05` s.a.m.d.

Ca și în expresiile matematice, parantezele (...) schimbă ordinea prestabilită a operațiilor.

Regulă. Într-o expresie cu paranteze, operațiile se execută începând de la parantezele interioare spre cele exterioare.

3.1.3 Expresii booleene și logice

Operatorii relaționali verifică dacă două valori numerice sunt în anumite relații: mai mic, mai mare, egal etc. Rezultatul unei asemenea verificări poate fi “adevarat” sau “fals”, ceea ce în Java este reprezentat de tipul `boolean`. De aceea expresiile care utilizează operatorii relaționali se numesc booleene. De exemplu:

```
suma <= sold
temperaturaMasurata > temperaturaReferinta + MARGINE_ADMISIBILA
```

sunt două expresii booleene care compară două expresii întregi. Ele descriu două situații diferite:

- suma pe care dorim să o extragem din cont, dar care nu trebuie să fie mai mare decât soldul;
- temperatura măsurată care a depășit marginea superioară admisibilă față de o temperatură de referință.

Deseori o simplă comparație nu este suficientă pentru a descrie o situație. Trebuie să facem mai multe comparații care ar putea: sau să fie toate îndeplinite sau să fie îndeplinită oricare din ele. Pentru a descrie asemenea situații ne folosim de expresii logice.

O expresie logică poate fi o simplă expresie booleană, așa cum poate fi alcătuită din mai multe expresii booleene legate cu operatori logici. Ca în exemplele următoare:

```
CodClient == IdCont && suma <= sold
temperaturaMasurata > temperaturaReferinta + MARGINE_ADMISIBILA
|| temperaturaMasurata < temperaturaReferinta - MARGINE_ADMISIBILA
```

Față de situațiile precedente lucrurile s-au complicat:

- codul inserat de client trebuie să fie egal cu identificatorul contului, iar suma cerută depășește soldul;
- temperatura măsurată este în afara intervalului admisibil în jurul valorii de referință a temperaturii.

Operatorii `&&` și `||` sunt operatori logici. Ei leagă două expresii booleene și reprezintă operațiile **and** logic și, respectiv, **or** logic. Compunerea expresiilor logice poate continua incluzând și operatorul **not** logic (!). În aceste cazuri sunt importante regulile care dau precedența operatorilor.

Să considerăm expresia logică:

```
temperatura > base + 15 && presiunea < 760 || !alarmOn && semnalizator
```

Expresia conține variabile și constante numerice, precum și două variabile booleene: `alarmOn` și `semnalizator`. Operatorii sunt aritmetici, relaționali și logici.

Figura 3-1 prezintă ordinea operațiilor în evaluarea expresiei. `t1`, `t2`, . . . , `t7` sunt variabilele temporare ce găzduiesc valorile rezultatelor subexpresiilor evaluate.

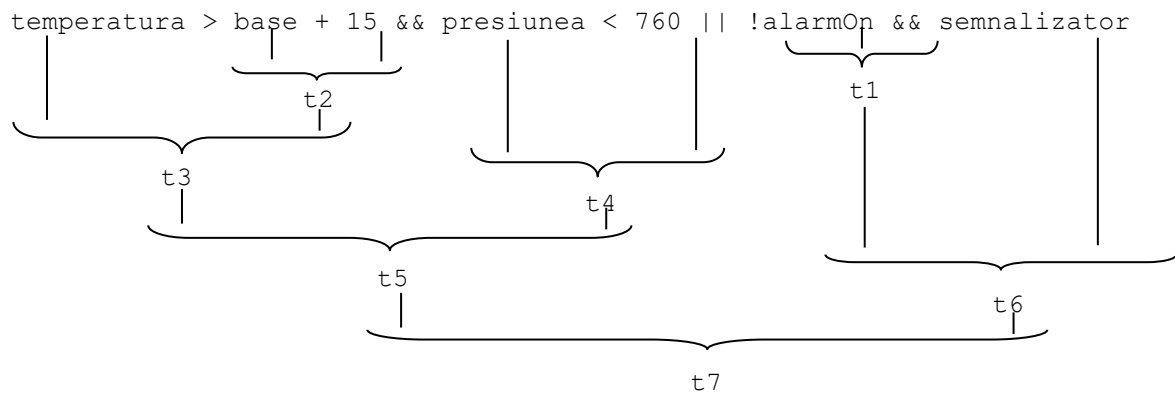


Figura 3-1. Ordinea de evaluare a operațiilor într-o expresie logică

3.1.4 Exemple de expresii în Java

Pentru a da o imagine globală asupra expresiilor, în Tabelul 3 se prezintă diverse categorii de expresii prin exemple împreună cu semnificația lor.

Expresie	Semnificație
<i>Expresii numerice</i>	
$(x+7) * y / 2$	$\frac{(x+7) \times y}{2}$
$2 * a * 3.14$	$2 \times a \times 3,14$
$(x - 'a') + 'A'$	O litera mică este transformată într-o literă mare
<i>Expresii booleene</i>	
$x > 3$	x este strict mai mare decât 3
$x == z$	x este egal cu z
$x != y$	x este diferit de y
b	b este adevărat
<i>Expresii logice</i>	
$x > 3 \ \&\& \ x == z$	$x > 3$ și $x = z$
$a + d < c \ \ b$	$a + d < c$ sau b
<i>Apeluri de metode</i>	
<code>getValue() * 77.4</code>	valoarea obținută prin executarea metodei <code>getValue()</code> este înmulțită cu 77.4
<code>amiculMeu.getPrenume()</code>	apelează metoda instanță <code>getPrenume()</code> a variabilei instanță <code>amiculMeu</code>
<i>Crearea obiectelor și memorarea referințelor lor în variabile instanță</i>	
<code>Mamifer m = new Mamifer()</code>	se crează un obiect al clasei Mamifer și referința la el este memorată în variabila m de același tip: Mamifer
<i>Expresii cu operatori aritmetici și atribuiri</i>	
<code>x = y = z = 82;</code>	$z=82; y=z; x=y$
<code>s = "Buna";</code>	se crează un obiect de tip String cu valoarea "Buna" și referința la el este memorată în variabila de același tip s
<code>x = 123.5 / 8.75f * y;</code>	noua valoare a lui x va fi valoarea expresiei: $123,5 / 8,75 \times \text{valoarea lui } y$
<i>Expresii cu operatorul de concatenare a șirurilor</i>	
<code>s=s+" ziua!"</code>	dacă valoarea lui s era o referință la obiectul ce conține șirul "Buna", noua valoare a lui s va fi o referință la un nou obiect ce conține șirul "Buna ziua!"

Tabelul 3. Exemple de expresii în Java

3.2 INSTRUCȚIUNI ÎN JAVA

Instrucțiunile sunt comenzi ale limbajului de programare ce modifică starea programului. O clasificare a comenzilor constă în instrucțiuni elementare și instrucțiuni de control.

Instrucțiunile elementare sunt instrucțiuni ce realizează o singură modificare a stării programului. Această modificare se obține de obicei prin evaluarea unei expresii și realizarea unei atribuiri, prin declararea unei variabile și inițializarea ei cu o valoare, prin apelarea unei metode, prin crearea unui obiect și memorarea referinței la el într-o variabilă, prin întreruperea execuției unei metode, etc.

Instrucțiunile elementare se clasifică în următoarele categorii: expresii, declarații, întreruperea execuției unei metode sau a unei instrucțiuni de control.

3.2.1 Instrucțiuni-expresie

Instrucțiunile-expresie provin din expresii formate cu ajutorul operatorilor sau apelurilor de metode. Exemple:

- expresii de atribuire: `oValoare = 8933.234;` //instrucțiune-expresie: atribuire
- utilizarea lui ++ sau --: `oValoare++;` //instrucțiune-expresie: incrementarea valorii
//variabilei oValoare
- apeluri de metode: `System.out.println(oValoare);` //instrucțiune-expresie: apelul
//unei metodei
- expresii de creare a obiectelor: `new Integer(4);` //crearea unui obiect ce conține
//valoarea 4

3.2.2 Instrucțiune de declarare

Orice declarare a unei variabile este o instrucțiune care introduce un nou identificador, împreună cu celelalte atribute ale variabile: tipul de date, valoarea inițială, etc. De exemplu,

```
double oValoare = 8933.234;
```

este o instrucțiune ce declară variabila `oValoare` de tip `double` și o inițializează cu valoarea `8933.234`.

Instrucțiunile de control alterează fluxul de control al unui program și conțin în componența lor alte instrucțiuni (elementare sau de control) impunând ordinea de executare a acestora. Ordinea rezultă de obicei dintr-un test asupra stării execuției, mai precis asupra valorilor unor variabile numite *variabile de test* din program. Așadar, ordinea rezultă dinamic în funcție de valorile variabilelor de test.

Instrucțiunile de control sunt de trei tipuri și anume: structura secvență, structura de ramificare a controlului și structura repetitivă.

3.2.3 Instrucțiunea secvență

Instrucțiunea secvență este încorporată în Java. Dacă nu se specifică altfel, compilatorul execută instrucțiunile în ordinea în care sunt scrise. Mai mult, instrucțiunile pot fi grupate. Un *bloc de instrucțiuni* este un grup format din zero sau mai multe instrucțiuni cuprinse între `{ }`. Instrucțiunile unui bloc vor fi executate secvențial, pe rând, în ordinea în care apar în bloc. De exemplu, dacă vrem să declarăm și să inițializăm două multe variabile statice, putem folosi un bloc static:

```
static{  
    int a=2;  
    double b=1.21;  
}
```

3.2.4 Instrucțiunile de ramificare a controlului

Instrucțiunile de decizie sunt instrucțiunea condițională **if** cu una sau doua alternative și instrucțiunea de selecție **switch**.

Sintaxa instrucțiunii **if**

Instrucțiune Condițională ::= **if** (*expresie*)

```

        instrucțiune1
    [ else
        instrucțiune2 ]

```

cu condiția ca *expresie* să fie o expresie logică.

Semantica acestei instrucțiuni se poate enunța astfel:

Dacă expresia are valoarea **true** atunci se va executa instrucțiunea *instrucțiune1*, altfel se va executa *instrucțiune2*, dacă if conține alternativa else. În caz contrar, compilatorul va trece la următoarea instrucțiune de după if.

De exemplu, în clasa *Arhitect* putem avea o metodă care să verifice dacă un arhitect este sau nu specialist în proiectarea "Zgârâie norilor". Pentru aceasta, ar trebui să avem variabila instanță booleană *specialist* pe care o vom folosi în metoda *esteSpecialist()* care implementează predicatul logic "x este specialist". În acest caz, codul clasei *Arhitect* se modifică astfel:

```

class Arhitect{
    private String nume;
    private boolean specialist=true;
    . . .
    boolean esteSpecialist() {
        if(specialist) return true;
        else return false;
    }
}

```

Sintaxa instrucțiunii **switch**

```

InstrucțiuneDeSelectie ::= switch (E)
{
    case v1: ListaInstrucțiuni1
    case v2: ListaInstrucțiuni2
    ...
    case vn: ListaInstrucțiunin
    [default: ListaInstrucțiuniDef]
}

```

Instrucțiunea switch execută o instrucțiune sau un grup de instrucțiuni în funcție de valoarea expresiei specificate în switch. Valoarea expresiei trebuie să fie de tip int, long sau char.

Mai întâi, se evaluează expresia *E*. Apoi, se compară succesiv valoarea expresiei cu constantele *v1*, *v2*, ..., *vn*. Dacă *vk* este prima constantă care este egală cu valoarea expresiei, atunci se execută *ListaInstrucțiunik*. Dacă această listă de instrucțiuni nu conține instrucțiunea **break** vor fi executate toate instrucțiunile *ListaInstrucțiunik+1* până la prima instrucțiune **break** întâlnită sau până la sfârșitul instrucțiunii switch.

De exemplu, executarea următoarei instrucțiuni switch

```

switch(4+3) {
    case 3: System.out.print("3");break;
    case 7: System.out.print("7");
    case 9: System.out.print("9");
    case 10: System.out.print("10");
}

```

determină afișarea șirului "7910".

În cazul în care valoarea expresiei nu este egală cu nici una din constantele *v1*, ..., *vk* și instrucțiunea switch conține alternativa default, atunci se va executa *ListaInstrucțiuniDef*.

3.2.5 Instrucțiunile repetitive

Instrucțiunile repetitive sunt de trei tipuri: instrucțiunea **while**, instrucțiunea **do-while** și instrucțiunea **for**.

Sintaxa instrucțiunii **while**

```

InstrucțiuneWhile ::= while (E) Instrucțiune

```

cu condiția ca E să fie o expresie logică.

Această instrucțiune determină executarea lui *Instrucțiune* cât timp E este adevărată. Dacă de la început valoarea lui E este false, atunci instrucțiune nu se execută și fluxul controlului trece la următoarea instrucțiune de după while.

De exemplu, următoarea secvență de program va afișa toate numerele naturale pare mai mici decât 10.

```
int i=0;
while(i<10)
{
    System.out.print(i+" ");
    i+=2;
}
```

Sintaxa instrucțiunii **do-while**

InstrucțiuneDoWhile ::= **do**

Instrucțiune

while (E);

Această instrucțiune determină executarea în mod repetat a lui *instrucțiune* cât timp expresia E este adevărată. Spre deosebire de instrucțiunea while, *instrucțiune* va fi executată cel puțin o dată.

De exemplu, aceleași cifre pare din exemplul anterior se pot obține folosind instrucțiunea do-while:

```
int i=0;
do
{
    System.out.print(i+" ");
    i+=2;
}while(i<10);
```

Sintaxa instrucțiunii **for**

InstrucțiuneFor ::= **for** (*ExpresieInitiala*; *ExpresieLogica*; *ExpresieIncrement*) *Instrucțiune*

Această instrucțiune permite executarea lui *Instrucțiune* în mod repetat, atâta timp cât *ExpresieLogică* este adevărată. Această expresie conține una sau mai multe variabile numite variabile de contorizare a numărului de execuții. Aceste variabile sunt inițializate (și eventual declarate) în *ExpresieInitială* și reinițializate în *ExpresieIncrement*.

Mai întâi, se evaluează *ExpresieInițială*, după care se evaluează *ExpresieLogică*. Dacă valoarea acesteia este true atunci se execută Instrucțiune. După aceea, se evaluează *ExpresieIncrement* ce determină modificarea valorii variabilelor din *ExpresieInitială*, după care se reia evaluarea lui *ExpresieLogica*, samd. Instrucțiunea for se termină atunci după ce a fost evaluată *ExpresieLogică* se obține valoarea false.

Exercițiul 2. Să scriem un program care calculează și afișează suma numerelor întregi de la 1 la 10.

Soluție. Avem două modalități:

1. varianta 1: scriem întregul cod în metoda main,
2. varianta 2: programul utilizează variabila instanță `suma: int` și metoda instanță `aduna(int n)` care adună numărul `n` la suma.

În metoda main, creăm un obiect al clasei curente (`Demo`), referința la el o memorăm într-o variabilă locală `demo` de același tip și adunăm la `suma` fiecare *număr* de la 1 la 10, apelând metoda `aduna(numar)`.

În continuare, prezentăm codul celor două variante, care au același rezultat: `suma = 55`.

```
//Varianta 1
public class Demo{
public static void main(String[] args){
    int suma = 0; //atribuire
    for(int contor=1;contor<=10;contor++){
        //instrucțiune de control
        suma += contor; //atribuire
    }
    //apel de metoda
    System.out.println("suma= " + suma);
}
}
```

```
//Varianta 2
public class Demo{
    int suma;
    void aduna(int c) {suma+=c;}
    public static void main(String[] args){
        Demo demo=new Demo();//initializare
        //instrucțiune de control
        for(int contor=1;contor<=10; contor++){
            //apel de metoda instanta
            demo.aduna(contor);
        }
        //folosirea variabilei instanta suma
        System.out.println("suma= " +
            demo.suma);
    }
}
```

Execuția unei instrucțiuni repetitive se poate întrerupe dacă folosim instrucțiunea elementară **break**. De aceea, această instrucțiune trebuie să fie folosită numai într-o instrucțiune condițională.

În aceeași condiție, putem folosi instrucțiunea **continue** pentru a întrerupe executarea iterației curente a unei instrucțiuni de control. De exemplu, variabilele *i* și *j* din următoarea secvență de program

```
1. for (int i=0, j=0 ; i<4 && j<3 ; ++i, j++){
2. do{
3. if (i+j<2) break;
4. else {i++; continue;}
5. }while (i<2);
6.
7. }//for
```

vor avea pe linia 6 următoarele valori: (0,0), (2,1), (4,2).

Exercițiul 3. Să se compileze și să se execute Programul 1 care calculează suma a două numere întregi preluate din linia de comandă.

Observație. Programul se va executa de la tastatură cu comanda:

PROMPT>java Suma 12 34

```
public class Suma {

    public static void main(String args[]) {
        int numar1;    // variabila pentru primul numar
        int numar2;    // variabila pentru al doilea numar
        int suma;      // variabila pentru suma

        //converteste primul numar de la tipul String la tipul int
        numar1 = Integer.parseInt(args[0]);

        //converteste al doilea numar de la tipul String la tipul int
        numar2 = Integer.parseInt(args[1]);

        //calculeaza si memoreaza suma a doua numere in variabila suma
        suma = numar1 + numar2;

        //vizualizeaza rezultatul
        System.out.println("Suma este " + suma);

    } //sfarsitul metodei main

} //sfarsitul clasei Suma
```

Programul 1. Suma: Utilizarea argumentelor din linia de comandă

Probleme propuse

1. Să se implementeze un program care realizează operații de schimb al monedelor. Programul are două intrări: suma primită de la client și rata de schimb. Să se calculeze diferența și numărul minim de restituit ca rest: 1, 2, 5, 10, 20 cenți și 1, 2 euro.
2. Să se scrie un program care preia din linia de comandă două numere întregi și apoi afișează suma, diferența, produsul, media aritmetică, maximum și minimum dintre ele.
3. Să se scrie un program care traduce numerele de la 1 la 12 în numele lunilor corespunzătoare: Ianuarie, Februarie, ..., Decembrie.

3.3 METODE STATICE ALE CLASEI JAVA.LANG.MATH

Pachetul standard Java `java.lang` conține o clasă `Math` care oferă metode ale clasei care realizează calcule matematice bine cunoscute. În Tabelul 1 se prezintă câteva din aceste metode.

Metoda	Valoarea returnată	Metoda	Valoarea returnată
<code>abs(x)</code>	valoarea absolută	<code>min(x, y)</code>	minimumul dintre x și y
<code>exp(x)</code>	e^x	<code>pow(x, y)</code>	ridicarea la putere, x^y
<code>log(x)</code>	$\ln x$	<code>sqrt(x)</code>	$\sqrt{x}$
<code>ceil(x)</code>	cel mai mic întreg convertit la <code>double</code> $\geq x$	<code>cos(x)</code>	cosinusul unghiului (în radiani) transmis ca parametru
<code>floor(x)</code>	cel mai mare întreg convertit la <code>double</code> $\leq x$	<code>sin(x)</code>	sinusul unghiului (în radiani) transmis ca parametru
<code>round(x)</code>	valoarea rotunjită a lui x	<code>tan(x)</code>	tangentă unghiului (în radiani) transmis ca parametru
<code>max(x, y)</code>	maximum	<code>random()</code>	număr real (de tip <code>double</code>) aleator între 0.0 și 1.0, exclusiv

Tabelul 1. Metode statice ale clasei `Math`

Exemple de apeluri de metode din clasa `Math`:

```
Math.abs(a-b)
Math.pow(1+rata, ani)
```

Exemple.

1. Program care calculează și afișează rădăcinile unei ecuații de gradul al doilea și ai cărei coeficienți sunt preluați din linia de comandă.

```
public class Ecuatie{
    public static void main (String args[]) {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        double c = Double.parseDouble(args[2]);

        if ((Math.pow(b,2) - 4*a*c)<0) System.out.println("Solutii complexe");
        else if (b*b == 4*a*c)
            System.out.println("Solutii reale si egale");
        else
            System.out.println("Solutii reale");
    } //main
} //clasa
```

2. Program care calculează valoarea dobânzii acordate pe 10 ani, la suma de 100000000 lei. Știm că rata pe an este de 5% din suma inițială din cont.

```

public class Dobanda {
    public static void main (String args[]) {
        long sold=1000000000, total=0;
        double rata=0.05;

        System.out.println("An\t\tTotal ");
        for (int an=1; an<=10; an++){
            total=(long) (an*rata*sold);
            System.out.println(an+ "\t\t"+ total);
        }
    }
}

```

3. Următorul program afișează caracterele alfabetului, împreună cu codurile Unicode asociate.

```

public class ListaDeCaractere{
    public static void main(String[] a){
        for(char c=97; c<=122;c++) System.out.print(c+"-"+(int)c+" ");
        System.out.println();
    }
}

```

3.2 METODE

Metodă = Algoritm care se execută în mediul unui obiect sau al unei clase de obiecte.

Sintaxa declarației de metodă este următoarea:

```
DeclarațieMetodă ::= ( public | protected | private | λ ) (static | λ )
                                     Tip Identificator (ListaParametri) Bloc
NumeMetodă ::= Identificator
ListaParametri ::= λ | Parametru | {[, Parametru ]}+
Parametru ::= Tip Identificator
Bloc ::= { ListaInstrucțiuni }
```

De exemplu, declarația metodei care caută o persoană după nume este:

```
public int cautaPersoana(String nume){. . .}
```

O metodă declarată cu modificatorul `static` este o metodă a clasei în vreme ce metodele declarate fără `static` sunt metode instanță adică aparțin fiecărui obiect din clasa în care este definită metoda.

Execuția algoritmului descris în metodă se realizează la apelul metodei dintr-o altă metodă. O metodă poate să fie apelată în două moduri:

- cu o variabilă referință care conține un obiect al clasei.

Exemplu:

```
Persoana persoana = new Persoana("Popescu", "Ion");
. . .
int v=persoana.getVarsta(); //variabila.numeMetoda(...)
```

- cu numele unei clase (metodă "statică").

Exemplu: Instrucțiunea

```
Math.pow(x, y);
```

calculează și returnează valoarea obținută în urma operației x^y

Metodele care aparțin unui obiect sau unei clase se pot apela între ele:

- dintr-o metodă ce aparține unui obiect se pot apela atât celelalte metode ale obiectului cât și metodele clasei din care face parte metoda respectivă;
- dintr-o metodă a unei clase se pot apela alte metode ale clasei.

Exemplu. Programul 7 calculează și afișează primele 100 de numere întregi, strict pozitive, împreună cu pătratele lor. Pentru aceasta la vizualizarea din metoda `main` se apelează metoda statică și privată `patrat()` care returnează pătratul numărului specificat ca parametru.

```
public class NumerePatrate {
    public static void main (String args[]) {
        for(int i = 1; i<=100; i++)
            System.out.println("patratul lui " + i + " = " + patrat(i));
    }
    private static int patrat(int x){
        return x * x;
    }
}
```

Programul 1. NumerePatrate

Instrucțiuni de întrerupere a execuției unei metode

Ca și alte limbaje de programare, Java conține instrucțiuni care întrerup execuția unei metode: `return`, `throw` și `try-catch`. Sintaxa acestora este prezentată în Figura 3-1:

Denumire instrucțiune

Sintaxa

Înșirea dintr-o metodă cu un rezultat

```
return Expresie;
```

Înșirea dintr-o metodă fără rezultat (void)

```
return Expresie;
```

Lansarea unei excepție

Instrucțiunea **try-catch**

```
throw Expresie;  
try{  
    ListaInstructiuni  
}catch(TipExceptie Identificator)  
{  
    ListaInstructiuni  
}catch(TipExceptie Identificator)  
{  
    ListaInstructiuni  
}  
...
```

Instrucțiunea **try-catch-finally**

```
try{  
    ListaInstructiuni  
}catch(TipExceptie Identificator)  
{  
    ListaInstructiuni  
}catch(TipExceptie Identificator)  
{  
    ListaInstructiuni  
}  
...  
finally{ ListaInstructiuni }
```

Figura 3-1. Instructiuni de întrerupere a executiei unei metode

În metode putem folosi în instrucțiunea `return` expresii booleene pentru a returna o valoare logică. De exemplu, metoda `esteSpecialist()` poate fi optimizată astfel:

//Varianta 1

```
class Arhitect {  
    boolean specializat=true;  
    . . .  
    boolean esteSpecialist() {  
        return specializat==true;  
    }  
}
```

//Varianta 2

```
class Arhitect {  
    boolean specializat=true;  
    . . .  
    boolean esteSpecialist() {  
        return specializat;  
    }  
}
```

Exercițiul 4 (Magazin). Să se gestioneze orarul unui magazin de cărți și jocuri video. Se vor executa următorii pași:

1. Se crează clasa `Magazin` cu variabilele `oraDeschidere` și `oraInchidere`, de tip `int`.
2. Se adaugă 4 metode de acces pentru variabilele `oraDeschidere` și `oraInchidere`: `getDeschidere`, `setDeschidere`, `getInchidere` și `setInchidere`.
3. Se adaugă o metodă `esteDeschis` care întoarce o valoare boolean-ă ce corespunde faptului că magazinul este deschis sau închis. Utilizați metoda `getOrarInt` de tip `Data` furnizată pe disc și prezentată pe pagina următoare.
4. Se adaugă metoda `getMesaj` care întoarce un șir și indică dacă magazinul este deschis sau închis la momentul apelării metodei.
5. Se modifică `vizualizeazăOrar` pentru a raporta orarul magazinului specificat de variabilele instanță `oraDeschidere` și `oraInchidere`.
6. Se adaugă funcționalitate clasei `TestMagazin` pentru a vizualiza "Deschis!" sau "Inchis!".
7. Compilați `Magazin.java`, `Data.java` și `TestMagazin.java`.
8. Executați `TestMagazin`.

Executarea programului va genera următoarea ieșire:

Deschis!

Orarul magazinului:

Toate zilele lucrătoare: 9:00 – 21:00.

Proprietar: Popescu Ion

3.2.1 Metode recursive

În general, un lucru sau un fenomen se definește în mod recursiv dacă în definiția sa există o referire la el însuși.

Recursivitatea este folosită cu multă eficiență în matematică. Exemple de funcții matematice recursive sunt:

1. Definitia functiei factorial fact: $N \rightarrow N$

$$\text{fact}(n) = \begin{cases} 1 & \text{daca } n = 0 \\ n * \text{fact}(n-1) & \text{daca } n > 0 \end{cases}$$

Se observă că în definiția funcției fact există o parte care nu se definește prin ea însăși și anume $\text{fact}(n)=1$ dacă $n=0$.

2. Definitia functiei Fibonacci fib: $N \rightarrow N$

$$\text{fib}(n) = \begin{cases} 1 & \text{daca } n = 0 \text{ sau } n = 1 \\ \text{fib}(n-2) + \text{fib}(n-1) & \text{daca } n > 1 \end{cases}$$

3. Definitia functiei Ackermann

ac: $N \times N \rightarrow N$

$$\text{ac}(m, n) = \begin{cases} n + 1 & \text{daca } m = 0, \\ \text{ac}(m-1, 1) & \text{daca } n = 0, \\ \text{ac}(m-1, \text{ac}(m, n-1)) & \text{daca } m, n > 0 \end{cases}$$

O metodă recursivă M este o metodă care se apelează pe ea însăși în corpul său.

Sunt două cazuri:

- apelul este direct, adica metoda M are un apel la ea însăși în corpul sau;
- apelul este indirect: o metodă M_1 apelează o metodă M_2 , care apelează direct M_3 , samd până cand o metodă M_k din secvență apelează M_1 .

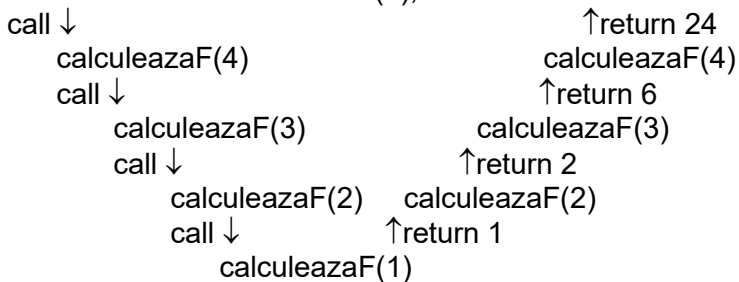
Apelul recursiv al unei metode trebuie sa fie conditionat de o decizie care sa impiedice apelul in cascada (la infinit); aceasta ar duce la o eroare de program – depasirea stivei.

```
void p( ) { //metoda recursiva
    p(); //apel infinit
}
//apelul trebuie conditionat in una din variantele:
if(cond) p();
while(cond) p();
do p() while(cond);
```

De exemplu, implementarea metodei recursive fact in Java este:

```
int calculeazaF(int n) {
    if (n <= 0)
        return 1;
    else return n * calculeazaF(n-1);
}
```

De exemplu, daca apeland $\text{calculeazaF}(4)$, rezultatul este un apel la $\text{calculeazaF}(3)$, care apelează $\text{calculeazaF}(2)$, care apelează $\text{calculeazaF}(1)$. In acest punct, se aplica regula de baza si returneaza 2 lui $\text{calculeazaF}(2)$, samd.



Varianta recursiva a calcularii termenilor sirului lui Fibonacci este următoarea:

```
int calculeazaFib1(int n){
    if (n==0 || n==1) return 1;
    else return calculeazaFib1(n-2)+calculeazaFib1(n-1);
}
```

Notă. Este esențial unei recursivități să facă numai apeluri ce implică argumente cu valori mai mici, deci care se pot calcula. De asemenea, trebuie să ajungem la cazul de bază, adică recursia trebuie să se termine.

Până acum am vorbit despre recursie înainte. Putem avea și recursie înapoi, în care pornim cu baza în care parametrul are valori mari și pasul de recursie rezolvă valorile mai mici ale parametrului în termeni cu valori mai mari.

Avantajele programelor recursive:

- sunt mai succinte și mai ușor de înțeles;
- unele probleme se rezolvă mai ușor prin recursivitate, decât prin programe iterative (ce nu contin recursivitate)

Dezavantaj: - funcțiile recursive sunt consumatoare de timp și spațiu. De aceea, uneori se preferă varianta nerecursivă, iterativă a implementării funcțiilor matematice recursive. De exemplu, varianta iterativă a calculării termenilor sirului lui Fibonacci este:

```
int calculeazaFib2(int n)
{
    if (n==0 || n==1) return 1;
    else {
        int x=1, y=1, z=0;
        for(int i=2; i<=n; i++){z=x+y; x=y; y=z;}
        return z;
    }
}
```

Probleme propuse

1. Să se scrie o metodă recursivă și una nerecursivă pentru calculul valorii polinoamelor Hermite $H(x)$ definite astfel:

$$H_0(x)=1; \quad H_1(x)=2x; \quad H_n(x)=2nH_{n-1}(x)-2(n-1)H_{n-2}(x), \text{ pentru } n \geq 2$$

2. Să se scrie un program ce calculează recursiv suma a n numere preluate din linia de comandă.
3. Se preiau din linia de comandă numerele n și k (numere naturale $n > k$). Calculați recursiv C_n^k utilizând formula de recurență: $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$.
4. Să se scrie un program care calculează și afișează c.m.m.d.c dintre 2 numere întregi preluate din linia de comandă, folosind următoarea definiție recursivă:

$$\text{cmmdc}(u,v) = \begin{cases} \text{cmmdc}(v, u \bmod v), & \text{dacă } v \neq 0 \\ u & \text{dacă } v=0 \end{cases}$$

5. Să se scrie un program recursiv care transformă un număr întreg din baza 10 într-o altă bază dată (2,8).
6. Problema turnurilor din Hanoi. Se consideră trei tije verticale A,B,C și n discuri de diametre diferite. Inițial toate discurile sunt puse în tija A, în ordinea descrescătoare a diametrului (discul cel mai mare la bază, iar cel mai mic în vârf). Se cere să se mute discurile de pe tija A pe tija C folosind tija B ca intermediar, folosind condițiile:
 - la o manevră se mută un singur disc și anume cel din vârful unei tije;
 - nu se poate pune un disc de diametru mai mare peste unul de diametru mai mic;
 - în final, pe tija C, discurile trebuie să fie în aceeași ordine ca în starea inițială de pe tija A.

Mesajele ar trebui să fie de forma 1->3, adică a fost mutat un disc de pe tija A pe tija C. De exemplu, pentru a muta 3 discuri de pe A pe C, programul ar trebui să afișeze:

1->3

1->2

3->2
1->3
2->1
2->3
1->3

4. ARRAY-URI

Array = O colecție liniară de elemente în care fiecare element este accesibil prin intermediul unui indice.

Un array furnizează spațiu pentru valorile de tip primitiv sau referință la obiectele de un același tip. Aceste valori se numesc **elementele** array-ului. Elementele unui array sunt ordonate după unul sau mai multe criterii numite **dimensiuni**. O dimensiune a unui array este indicată printr-o pereche de []. După numărul de dimensiuni, un array este uni- sau multidimensional.

Referința la un array va fi memorată într-o **variabilă array**. Sintaxa declarării unei variabile array unidimensional este următoarea:

```
TipDeDate[] id; sau TipDeDate id[];
```

De exemplu, putem declara două variabile array `numere` și `personal` astfel:

```
int[] numere;  
Persoana personal[];
```

Nota. După executarea acestor declarații, array-urile propriu-zise nu există în memorie, ci numai variabilele array asociate.

Pentru a exista într-un program, un array trebuie să fie creat (alocat) cu **new**. Sintaxa alocării memoriei unui array este:

```
id = new tip[expresie]
```

unde *tip* este tipul de date al variabilei array cu identificatorul *id*, iar *expresie* este fie un literal întreg, fie o expresie cu valoare dreaptă un număr întreg. De exemplu,

```
numere = new int[5];  
personal = new Persoana[1000];
```

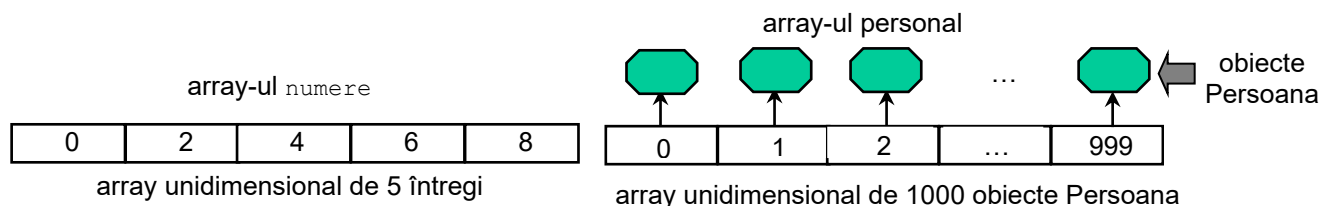
În acest moment, toate elementele array-ului `numere` sunt inițializate cu 0, iar cele ale array-ului `personal` cu valoarea `null`.

După creare, elementele array-ului se pot umple cu valori, cu observația că indicele unui array începe de la 0. Astfel un array cu 10 elemente va avea indicii de la 0 la 9. De exemplu, dacă array-ul `numere` conține primele 5 numere pare naturale, umplerea array-ului cu aceste valori se face astfel:

```
for (int i=0; i<5;i++) numere[i]=2*i;
```

În cazul array-ului de tip referință `personal`, valorile lui sunt referințe către obiecte de tip `Persoana`, care trebuia să fie create. Astfel,

```
for (int i=0; i<1000;i++) personal[i]=new Persoana();
```



Grafic, cele două array-uri arată astfel:

Prin definiție, lungimea unui array este egală cu numărul de elemente ale array-ului. Pentru a determina lungimea unui array se folosește variabila `length`. De exemplu, valoarea expresiei `personal.length` este 1000.

4.1 ÎNȚĂLIZAREA UNUI ARRAY

Am văzut că la momentul creării unui array, elementelor sale le sunt atribuite valoarea zero (tipuri primitive numerice), `\u0000` (tipul `char`), `false` (tipul `boolean`) sau `null` (tipuri referință). Elementele unui array pot fi inițializate în declararea variabilei array printr-o listă de inițializatori cuprinsă între {} ce le conține și care este atribuită variabilei array asociată. Sintaxa este următoarea:

```
TipDeDate[] id={listaDeInițializatori}
```

Aceasta este de fapt o formă prescurtată pentru

```
TipDeDate[] id=new TipDeDate[] {listaDeInițializatori};
```

unde listaInitializatori este o multime de elemente de acelasi tip cu tipul array-ului, ce sunt valorile elementelor array-ului id.

Numarul de elemente din lista de initializatori furnizeaza valoarea variabilei length pe variabila array asociata.

De exemplu, elementele array-ului indicat prin variabila numere pot fi initializate de la declararea variabilei numere astfel:

```
int[] numere={0,2,4,6,8};
```

4.2 ARRAY-URI MULTIDIMENSIONALE

Array-urile multidimensionale sunt declarate prin adaugarea dimensiunilor. De exemplu, declararea unei matrici m cu elemente de tip intreg se realizeaza astfel:

```
int[][] m;
```

Alocarea memoriei necesare memorarii matricii m se face astfel:

```
m=new [3][3];
```

In memorie, elementele lui m nu vor fi memorate ca intr-o matrice ci liniar adica, un array multidimensional este memorat ca un array de array-uri (si la randul lor pot fi de array-uri), astfel ca fiecare dimensiune este memorata ca un array diferit. De aceea, putem declara si construi array-uri multidimensionale de dimensiuni diferite. De exemplu,

```
int[][] b=new int[2][]; //b are 2 linii
```

```
b[0]=new int[3]; // prima linie are 3 coloane sau elemente
```

```
b[1]=new int[4]; // a doua linie are 4 elemente
```

Nota: Daca numarul coloanelor poate sa difere, tipul elementelor trebuie sa fie acelasi.

In cazul array-urilor multidimensionale variabila length are valori diferite. De exemplu, b.length furnizeaza numarul de linii ale lui b, adica 2, pe cand b[1].length=4.

Similar, un array multidimensional poate fi initializat si la declarare folosind o lista ce contine un numar de liste de initializatori egal cu numarul de dimensiuni ale array-ului, liste separate prin virgula. De exemplu, int[][] a={{1,0},{0,1}} memoreaza matricea unitate bidimensionala.

Exemplu. Urmatorul exemplu arata cum pot fi create, initializate si afisate diferite array-uri multidimensionale:

```
import java.util.*;
public class ArrayMultiDimensionale{
    static Random aleator= new Random();
    static int pRand(int mod) {
        return Math.abs(aleator.nextInt())%mod;
    }
    public static void main(String[] args){
        //array 3-D cu dimensiuni fixe:
        int[][][] a2 = new int[2][2][4];
        for(int i = 0; i < a2.length; i++)
            for(int j = 0; j < a2[i].length; j++)
                for(int k=0; k<a2[i][j].length;k++)
                    afiseaza("a2[" + i+"]["+j+"]["+k+"] = " + a2[i][j][k]);
        //array 3-D cu vectori de lungime diferita pentru cele trei dimensiuni
        int[][][] a3 = new int[pRand(7)[][][]];
        for(int i = 0; i<a3.length; i++){
            a3[i] = new int[pRand(5)][];
            for(int j=0; j<a3[i].length; j++)a3[i][j] = new int[pRand(5)];
        }
        for(int i=0; i<a3.length; i++)
            for(int j=0; j<a3[i].length; j++)
                for(int k=0; k<a3[i][j].length; k++) afiseaza("a3["+i+ "]["+j+"]["+k+"] = "
                                                                +a3[i][j][k]);

        //array 3x2 de obiecte Integer
        Integer[][] a4={{new Integer(1), new Integer(2)},
                        {new Integer(3), new Integer(4)},
                        {new Integer(5), new Integer(6)}};
        for(int i = 0; i < a4.length; i++) for(int j = 0; j < a4[i].length; j++)
            afiseaza("a4[" + i + "][" + j + "] = " + a4[i][j]);
```

```

}
static void afiseaza(String s){
    System.out.println(s);
}
}

```

4.3 LUCRUL CU ARRAY-URI

Cand lucram cu array-uri putem folosi urmatoarea metodă statică a clasei `System` ce permite copierea unui număr de elemente consecutive ale unui array într-un alt array:

```
static void arraycopy(arraySursă, pozSursă, arrayDest, pozDest, nrElemente)
```

În mod obișnuit, array-urile sunt folosite în probleme de sortare și de căutare. De exemplu, sortarea elementelor unui array folosind metoda `QuickSort` se face astfel în Java:

```

public class Ordonare{
    static int[] a;
    public static void initializeaza(){
        a=new int[20];
        int nr;
        for(int i=0; i<a.length;i++){
            nr=(int) (Math.random()*30);
            a[i]=nr;
        }
        afiseaza();
    }
    public static void quickSort(int p, int u){
        int i=p, j=u;
        int temp=a[(p+u)/2];
        do{
            while(a[i]<temp) i++;
            while(a[j]>temp) j--;
            if (i<j){
                int t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
            if (i<=j) {i++;j--;}
        }while (i<j);
        if(p<j) quickSort(p,j);
        if(i<u) quickSort(i,u);
    }

    public static void afiseaza(){
        for(int i=0;i<a.length;i++) System.out.print(a[i]+" ");
        System.out.println();
    }
    public static void main(String[] args){
        initializeaza();
        quickSort(0,19);
        afiseaza();
    }
}

```

Căutarea unei valori într-un array folosind metoda `Divide et Impera` se realizează astfel:

```

public class Cautare{
    static int[] a;
    public static void initializeaza(){
        a=new int[20];
        int nr;
        for(int i=0; i<a.length;i++){
            nr=(int) (Math.random()*30);
            a[i]=nr;
        }
        afiseaza();
    }
    public static boolean cauta(int val){
        int p=0, u=a.length-1, m=(p+u)/2;
        while(p<u && val!=a[m]){
            if(val<a[m]) u=m-1;
            else p=m+1;
            m=(p+u)/2;
        }
        return a[m]==val;
    }

    public static void afiseaza(){
        for(int i=0;i<a.length;i++) System.out.print(a[i]+" ");
        System.out.println();
    }

    public static void main(String[] args){
        initializeaza();
        int nr=(int) (Math.random()*30);
        if(cauta(nr)) System.out.println("A gasit="+nr);
        else System.out.println("Nu a gasit="+nr);
    }
}

```

4.4 ARRAY ÎN METODE

Daca parametrul unei metode este o variabila array, atunci transferul se face prin referinta, adica orice modificare efectuata asupra elementelor array-ului respectiv se memoreaza. Elementele unui array de tip primitiv se transmit prin valoare, exact ca niste variabile simple. De exemplu, daca in programul Ordonare anterior am fi avut metoda

```

public static void schimba(int i, int j){
    int t=i;
    i=j;
    j=t;
}

```

apelul acestei metode `schimba(a[i], a[j])` nu ar fi realizat operatia respectiva. Pentru a realiza într-adevăr schimbarea două valori transmise prin valoare, metoda `schimba()` trebuie să fie scrisă astfel:

```

public static int[] schimba(int i, int j){
    Integer n1=new Integer(i);
    Integer n2=new Integer(j);
    int[] n=new int[2];
    Integer n3=n1;
    n1=n2;
    n2=n3;
    n[0]=n1.intValue();
    n[1]=n2.intValue();
    return n;
}

```

În acest caz, metoda quickSort ar putea apela metoda schimba() astfel:

```
public static void quickSort(int p, int u){
    int i=p, j=u;
    int temp=a[(p+u)/2];
    do{
        while(a[i]<temp) i++;
        while(a[j]>temp) j--;
        if (i<j){
            int[] n=schimba(a[i],a[j]);
            a[i]=n[0];
            a[j]=n[1];
        }
        if (i<=j) {i++;j--;}
    }while (i<j);
    if(p<j) quickSort(p,j);
    if(i<u) quickSort(i,u);
}
```

Exercițiu: Magazin2.

Adăugați funcționalitatea programului Magazin pentru gestionarea titlurilor de cărți și de jocuri video, adică memorarea, găsirea și vizualizarea titlurilor:

- Copiați într-un director Magazin2 fișierele .java ale directorului Magazin1;
- Creați un tip de date Titlu cu două variabile-instanță: titlu și autor, inițializate cu "niciUnul" și implementați metodele de acces get și set pentru cele două variabile;
- Adăugați variabila titluri de tip Titlu tipului Magazin, inițializată cu null, și implementați metodele setTitluri() și getTitluri();
- Adăugați metoda vizualizeazaTitluri() tipului Magazin care parcurge array-ul de titluri și vizualizează datele titlu-autor;
- Modificați clasa TestMagazin pentru a încorpora funcționalitatea metodei vizualizeazaTitluri().

Probleme propuse

1. Să se scrie o metodă predicat (întoarce o valoare boolean):

```
public static boolean aceleașiElemente(int[] a, int[] b)
```

care verifică dacă două array-uri au elemente comune, cu același număr de apariții.

2. Să se scrie două metode ale clasei Poligon:

```
public double calculeazaPerimetru() {. . .} și
public double calculeazaArie() {. . .}
```

care calculează perimetrul și aria unui poligon. Pentru a calcula perimetrul, calculați distanța dintre punctele adiacente și adunați aceste distanțe. Aria unui poligon cu punctele $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ este:

$$|x_0y_1+x_1y_2+x_2y_3+\dots+x_{n-1}y_0-y_0x_1-y_1x_2-\dots-y_{n-1}x_0|/2$$

3. Să se scrie un program care inițializează un array de numere întregi pozitive și afișează histograma valorilor memorate în array:

Element	Valoare	Histograma
1	3	***
2	5	*****
3	9	*****
4	13	*****

5. INTRODUCERE ÎN PROGRAMAREA ORIENTATĂ SPRE OBIECTE

Dezvoltarea programelor pe obiecte

Programarea într-un **limbaj procedural** utilizează împărțirea unei probleme într-un număr de pași mai simpli în care fiecare pas reprezintă rezolvarea unei subprobleme a problemei inițiale. O aplicație scrisă într-un limbaj procedural se inspiră din modul în care calculatorul execută instrucțiunile.

O aplicație scrisă într-un **limbaj orientat spre obiecte** se concentrează asupra problemei de rezolvat utilizând o mulțime de agenți, **obiectele**, care se angajează să o rezolve colaborând între ei. Programarea într-un limbaj orientat spre obiecte înseamnă găsirea unor agenți adaptați problemei și care își împart întreaga sarcină a rezolvării, fiecare din ei specializându-se în sarcini mai mici.

Deosebirea celor două abordări bazate ambele pe principiul "divide et impera" este aceea că în timp ce abordarea procedurală identifică și rafinează funcțiile aplicației, abordarea pe obiecte identifică și rafinează conceptele problemei de rezolvat, punând instanțe ale acestor concepte să colaboreze una cu alta pe baza relațiilor semantice dintre concepte.

De exemplu, într-un sistem stereo:

- boxe se angajează să reproducă sunete de frecvență medie și înaltă,
- subwoofer se obligă să reproducă sunete de frecvență joasă,
- receptorul primește semnale radiofonice,
- cititorul de CD citește semnale audio de pe CD.

Componentele sunt proiectate pentru a interacționa între ele prin intermediul conexiunilor standard. Orice componentă poate fi schimbată cu o altă de altă marcă dar care respectă standardul legăturilor cu celelalte componente. După cum se poate vedea, fiecare obiect este specializat în furnizarea anumitor servicii legate de informațiile pe care le posedă și în acest fel servește rezolvării unor sarcini în interiorul programului.

Reflectând asupra soluției unei probleme în termeni de concepte favorizează reutilizarea obiectelor de la o aplicație la alta atunci când cele două aplicații folosesc aceleași concepte.

De aceea a programa pe obiecte înseamnă a scrie un program în care se combină cu mecanisme standard furnizate de limbajul de programare obiecte create ad hoc cu obiecte preexistente.

5.1 OBIECTE

Obiect = un mod simplificat de a identifica un lucru din lumea reală. Este o instanță a unui concept. Din punct de vedere tehnic al abordării pe obiecte, un obiect este o combinație dintre:

- o **informație de stare** reprezentată de o mulțime de **atribute** sau proprietăți ale obiectului și
- o **funcționalitate** reprezentată de un set de **operații** sau funcții asupra acestor date.

Membrii unui obiect sunt deci membrii-date și membrii-funcții. Ei pot fi proiectați ca accesibili sau nu altor obiecte. Membrii accesibili din afara obiectului formează **interfața** naturală a acestuia. Astfel, starea obiectului poate fi modificată numai prin intermediul interfeței acestuia.

Principiul care stă la baza definirii interfeței unui obiect este acela al încapsulării informațiilor ("information hiding"). Conform acestui principiu este recomandabil să ascundem în interiorul obiectului toate informațiile (date sau funcții) inutile altor obiecte și care reprezintă detalii de implementare ale obiectului și să lăsăm accesibile (vizibile) altor obiecte doar acele informații în care aceste obiecte sunt interesate.

În Figura 1 se prezintă un obiect al clasei `Punct` a cărui stare este formată din valorile atributelor `x` și `y`, atribute care reprezintă coordonatele punctului și constituie starea obiectului. Proiectarea obiectelor `Punct` a decis că aceste atribute trebuie să fie private. În acest fel, accesarea din afara obiectului a valorilor lui `x` și `y` este imposibilă. Cum aplicația are nevoie să controleze coordonatele punctelor, s-a decis ca accesul la variabilele de stare să se facă indirect, prin operațiile `read_x`, `read_y`, `change_x` și `change_y`, iar aceste operații să formeze interfața obiectelor `Punct`.

În Java:

- atributele unui obiect sunt implementate prin variabile instanță, iar
- operațiile unui obiect sunt implementate prin metode.

Deci putem spune că starea unui obiect în Java este dată de valorile variabilelor sale instanță.

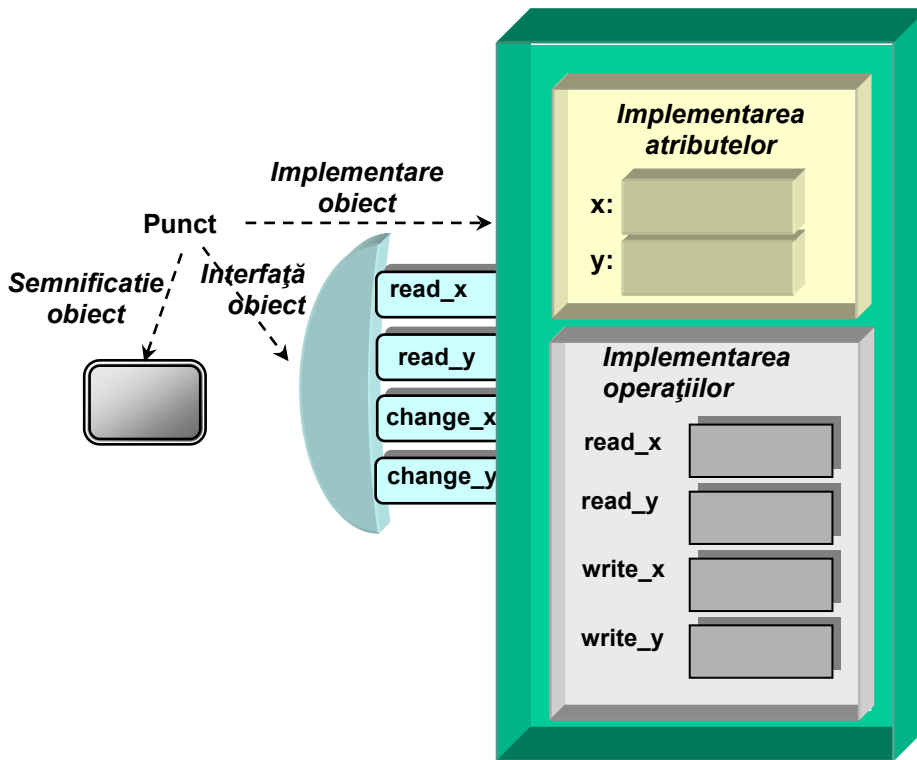


Figura 1. Puncte de vedere asupra unui obiect

5.2 CLASE DE OBIECTE

Clasă = Descriere a unei mulțimi de obiecte care au aceleași atribute, aceleași operații, aceleași relații cu alte clase și aceeași semantică.

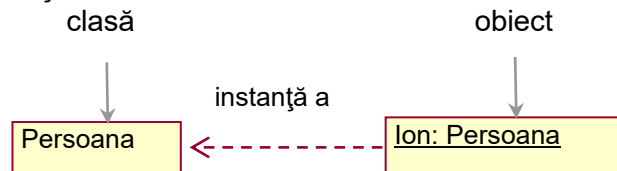


Diagrama 1. Relația de instanțiere

O clasă trebuie văzută în același timp ca:

- un **tip de date** sau un concept al aplicației,
- o **schemă** sau **model** pentru obiectele sale,
- o **fabrică** pentru obiectele sale.

În UML reprezentăm relația de instanțiere dintre o clasă și un obiect al său ca o dependență (Diagrama 1):

Clasă	Instanță
Proiectul unei case	Casa construită
Negativ fotografic	Fotografie
Rețetă culinară	Mâncare gătită după rețetă

Tabelul 1. Exemple de relații similare cu relația de instanțiere

În Tabelul 1 se prezintă câteva exemple de situații în care relațiile între concepte sunt asemănătoare relației dintre clase și instanțele lor.

5.2.1 Atribute, operații și constrângeri

Programarea pe obiecte necesită o fază preliminară de analiză a problemei de programat. În această fază obiectele sunt identificate prin conceptele pe care le reprezintă, iar colaborarea dintre

obiecte prin relațiile stabilite între concepte. Descrierea problemei rezultată din procesul de analiza ia în general o formă grafică. Limbajul grafic utilizat pentru acest tip de descrieri este UML (Unified Modeling Language). În aceste descrieri un obiect este caracterizat de clasa sa. Pentru specificarea clasei, proiectantul aplicației trebuie să specifice elementele componente ale clasei care sunt atributele, operațiile și constrângerile.

Un atribut este o abstractizare a unei proprietăți a unui obiect din lumea reală. Valorile curente ale acestor proprietăți constituie starea obiectelor clasei.

Nota. În general, stările diferențiază un obiect de un altul, deși identitatea unui obiect nu este dată de unicitatea stării sale printre stările obiectelor de același tip. Cu alte cuvinte, două obiecte pot avea exact aceeași stare, dar să nu fie identice. Unicitatea obiectului în programarea pe obiecte este asigurată de mediul de execuție.

De exemplu, un obiect *telefon celular* într-o aplicație de *e-commerce* are ca atribute posibile: *preț*, *dimensiuni*, *greutate* și *culoare*. Aceste atribute descriu celularul din punct de vedere al aplicației care ne interesează. Fiecare atribut este caracterizat de un domeniu de valori în care atributul ia valori.

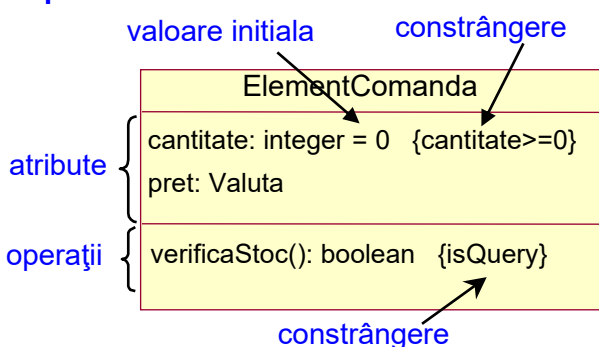
Operație = Algoritm privat al obiectului ce operează asupra atributelor sale. Se mai numește și **serviciu** (pe nivelul conceptual și al specificării) sau **metodă** (pe nivelul implementării).

Constrângeri = Condiții, cerințe sau reguli pe care trebuie să le îndeplinească un obiect.

De exemplu, un element al unei comenzi este caracterizat prin cantitate și preț. O condiție pe care o impunem este ca valoarea cantității să fie un număr pozitiv.

Deci prețul și cantitatea vor forma atributele clasei *ElementComanda*, iar condiția $\{cantitate \geq 0\}$ va fi o constrângere. Metoda *verificaStoc()* verifică dacă există în depozit cantitatea produsului respectiv pentru onorarea comenzii. Aceste elemente pot fi specificate în UML și implementate în Java. Relația între cele două reprezentări este prezentată printr-un exemplu în Figura 2.

Reprezentarea unei clase în UML



Implementarea clasei în Java

```
public class ElementComanda{
    private int cantitate;
    private Valuta pret;
    . . . .
    public boolean verificaStoc(){...}
    . . . .
}
```

Figura 2. Clasa UML și implementarea ei în Java

5.2.1. Variabile-membru ale unei clase

Într-o clasă Java, atributele obiectelor unei clase sunt definite ca **variabilă-instanță** sau **variabilă-obiect** în definiția clasei. Fiecarei variabilă-instanță i se alocă o zonă de memorie în spațiul alocat obiectului și în care se memorează valoarea variabilei. Întregul spațiu alocat unui obiect este alcătuit din totalitatea spațiilor alocate variabilelor sale instanță.

La codificarea în Java, unui atribut al clasei îi va corespunde o variabilă-instanță.

Exemplu. Fie o clasă *Avatar* într-un joc interactiv. Clasa definește o variabilă instanță *viteza* care permite "avatarurilor" (agenți care se deplasează pe ecran îndeplinind diverse roluri în joc) să se poată deplasa cu diferite viteze, după caracteristicile mediului în care se găsește. Modificând variabila instanță *viteza* a unui obiect *Avatar*, acesta se deplasează mai repede sau mai încet.

O variabilă instanță este, la nivelul obiectului, o variabilă globală cunoscută și accesibilă din toate metodele obiectului.

Variabilă de clasă este o variabilă-membru a unei clase care are un unic spațiu alocat în memorie și a cărei valoare memorată este partajată de toate obiectele clasei. O variabilă de clasă

implementeaza un atribut al clasei si nu al obiectelor clasei. Reprezinta o solutie pentru a avea, la nivelul programului, variabile globale ale tuturor obiectelor de acelasi tip.

Exemplu. În clasa `Avatar` se definește o variabilă a clasei, numită `dataOra`. Valoarea variabilei este partajată de toate obiectele `Avatar`. Dacă această variabilă ar fi fost o variabilă instanță, sincronizarea avatarilor ar fi fost mult mai dificil de realizat.

Accesul la variabilele-membru

Accesul la o variabila-instanta a unui obiect se face cu ajutorul operatorului binar `."`. Primul operand trebuie sa fie o variabila-reference sau o expresie a carei valoare sa fie de tip reference. Al doilea operand este chiar numele variabilei-instanta.

Exemplu:

```
ElementComanda ec = new ElementComanda("Computer");
ec.pret = new Valuta(147.35, "USD"); // atribuirea variabilei-instanta
                                     // pret a lui ec
ec.cantitate = 25; // atribuirea variabilei-instanta cantitate a lui ec
```

Accesul la o variabila-clasa se face tot cu ajutorul operatorului binar `."`, doar ca primul operand trebuie sa fie in acest caz numele clasei. Al doilea operand este numele variabilei-clasa.

Exemplu:

Daca in clasa `ElementComanda` ar exista o variabila-clasa care sa memoreze numarul curent al articolului cumparat pe lista comenzii respective, codul clasei ei ar trebui modificat ca mai jos:

```
public class ElementComanda{
    static int nrCurent=1;
    private int cantitate;
    private Valuta pret;
    . . . . .
    public boolean verificaStoc(){...}
    . . . . .
}
```

La momentul stabilirii produselor care intra in comanda, se poate asocia fiecarui obiect `ElementComanda` valoarea lui `nrCurent`, dupa care variabila ar trebui incrementata. Obiectul care executa aceasta operatie ar trebui sa contina urmatorul cod:

```
Comanda c = new Comanda();
. . . . .
ElementComanda ec = new ElementComanda();
. . . . .
c.add(ec, ElementComanda.nrCurent++);
```

Exista situatii in care notatia cu punct pentru a accesa o variabila-membru este inutila. Este cazul accesului din interiorul obiectului sau clasei care contine variabila-membru.

Exemplu:

Fie clasa:

```
public class ElementComanda{
    private int cantitate;
    private Valuta pret;
    . . . . .
    public boolean verificaStoc() {
        . . . . .
        return cantitate <= stoc;
    }
    . . . . .
}
```

Se observa ca metoda `verificaStoc` foloseste variabila-instanta `cantitate` fara sa specifice obiectul caruia apartine variabila si aceasta deoarece este o variabila a chiar obiectului metodei.

5.2.2 Domeniul de valabilitate și domeniul de existență

Domeniul de valabilitate (scope) = zonă de program în care un nume este recunoscut ca având o aceiași semnificație. Un domeniu de vizibilitate este și un **spatiu de nume**, adică un domeniu de identificatori unici.

De exemplu, declaratia unei clase in Java este un domeniu de valabilitate pentru identificatorii tuturor membrilor clasei. Domeniile de vizibilitate intr-un program pot fi incuibate, adica un domeniu de valabilitate poate contine unul sau mai multe domenii de valabilitate.

In Java, in afara declaratiei de clasa, exista mai multe constructii sintactice care furnizeaza un domeniu de valabilitate. De exemplu, fiecare bloc (instructiune compusa) din corpul unei metode formeaza un domeniu de valabilitate in care sunt cunoscute toate numele declarate in interiorul blocului asociate fiecare unei anumite semantici. Cum corpul metodei este el insusi un bloc in care sunt cunoscuti atat parametrii metodei cat si toate variabilele declarate in acel bloc, el reprezinta un prim domeniu de valabilitate in care pot fi incluse alte domenii de valabilitate.

Asa-numitele "reguli de contur" valabile in toate limbajele moderne, inspirate din programarea structurata, impun ca un domeniu de valabilitate sa fie inclus in cel mult un alt domeniu de valabilitate. Structura care cel mai bine descrie relatia de incuibare a domeniilor de valabilitate este arborele. De aici ideea de ierarhie de domenii de valabilitate.

Un alt exemplu de domeniu de valabilitate este instructiunea for. In header-ul instructiunii se pot declara variabile ale caror nume sunt cunoscute atat in header-ul ca si in corpul instructiunii.

Din punct de vedere al domeniului de valabilitate, variabilele pot fi:

- **globale**, daca domeniul de valabilitate se extinde la tot programul.
Nota. Calificativul de global se poate utiliza si in mod relativ si cand se ia ca referinta un domeniu de valabilitate care in interior contine alte domenii de valabilitate. Atunci, despre variabilele domeniului extern putem spune ca sunt globale domeniilor interne. In acest sens o variabila instantata a unui obiect, chiar daca este declarata privata obiectului, deci invizibila din exterior, ea este globala tuturor variabilelor locale declarate in metodele obiectului.
- **locale**, daca domeniul are o intindere limitata, cuprinsa intr-un alt domeniu de valabilitate, mai larg. Toate variabilele declarate in metodele unei clase sunt locale corpului metodei care, fiind un bloc, formeaza un domeniu de valabilitate. Trebuie precizat ca in Java declaratiile pot fi inserate oriunde intr-un bloc, nu numai la inceputul acestuia. Unica restrictie este ca in cadrul domeniului declaratia unui identificator sa preceada prima utilizare a acestuia.

Sa privim codul prezentat în Programul 1.

```
public class Mediu{
    int x=0;           // variabilă instanță, globală la nivel de obiect
    void met1() {
        int y, x=1;    // declaratie de variabile locale
        y=x;           // OK, y=1 și nu y=0
    }
    void met2() {
        int z=1;        // declaratie de variabila locala
        x=y+z;          // eroare! y nu este cunoscut in acest domeniu
    }
    void met3() {
        met1();          // apel al unei alte metode a obiectului
        int w=x*x;
        {                // incepe un domeniu de valabilitate inclus in domeniul lui met3
            x+=2;
            float w=1.2*x; // variabila locala; declaratia precede utilizarea
            x=(int) (w-2.2);
        }               // se termina domeniul de valabilitate inclus in domeniul lui met3
        x*=w;
    }
}
```

Programul 1. Clasa Mediu

Diagrama din Figura 3 numita si **diagrama de contur**, descrie relatiile intre domeniile de valabilitate din programul precedent.

Stabilit mecanismul domeniilor de valabilitate se poate enunta regula care stabileste accesibilitatea componentelor programului (variabile, tipuri, constante simbolice sau metode) a caror semantica este stabilita prin declaratii.



Figura 3. Diagrama de contur

Regula. O componenta cu nume x este accesibilă într-un punct din program aflat într-un domeniu de valabilitate y , dacă componenta a fost declarată în y cu o declarație ce precede punctul din program considerat sau, în cazul ca lipsește această declarație, dacă este vizibilă în domeniul înconjurător lui y .

Domeniul de existență (lifetime) = Perioada de timp în cursul execuției programului în care o componenta a programului există, adică are alocat un spațiu în memorie.

Nota. Deși în cazul anumitor variabile domeniul de valabilitate și cel de existență sunt strâns legate între ele prin mecanisme dictate de regulile limbajului de programare, cele două concepte sunt complet diferite între ele. Domeniul de valabilitate se referă la identificatori din program și de aceea este un concept static pentru că se referă la momentul compilării, atunci când contează identificatorii. Domeniul de existență se referă la entitățile din program prezente în memorie la execuție (variabile, obiecte, array etc.) și de aceea este un concept dinamic, specific execuției.

În exemplul de mai sus variabila întreaga w este vizibilă în tot corpul metodei met3 mai puțin în blocul interior acestuia în care cu numele w este vizibilă o variabilă float . În schimb, variabila întreaga w are domeniul de existență extins pe toată durata execuției corpului funcției, începând cu momentul executării declarației sale. Deci ea există și pe durata existenței variabilei $\text{float } w$, doar că nu este accesibilă.

Dacă pentru variabilele locale se poate stabili o relație între domeniul de valabilitate și domeniul de existență, pentru celelalte componente ale execuției, cum ar fi obiectele și array-urile, relația nu există deoarece acestea în Java au un domeniu de existență, dar nu și un domeniu de valabilitate. Accesibilitatea lor este dată în mod indirect de variabilele care le memorează adresa în spațiul Heap.

5.2.3 Metode

Reluăm pe scurt tema metodelor în Java aducând unele precizări față de introducerea făcută în capitolul precedent.

Metodele sunt folosite pentru a implementa serviciile pe care proiectantul programului le-a încredințat clasei și obiectelor sale. Aceste servicii alcatuiesc responsabilitatea clasei respective, rațiunea existenței ei.

Declarația de metodă

O metodă în Java trebuie să fie declarată în cadrul clasei careia îi aparține. O metodă este specificată printr-un header urmat de corpul metodei.

Header-ul unei metode conține:

1. un eventual modificador de vizibilitate (`private`, `public` sau `protected`),
2. un tip de date care este tipul valorii rezultatului întors de metodă. Dacă metoda nu întoarce nici un rezultat, tipul trebuie să apară oricum și el este `void`,
3. numele metodei, un identificator, și
4. lista parametrilor formali între paranteze.

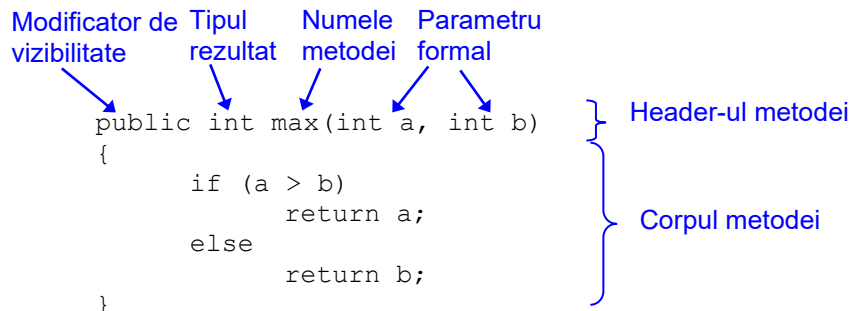
Lista parametrilor formali poate fi vidă sau poate conține unul sau mai mulți parametri separați prin virgulă.

Fiecare parametru formal este specificat prin tip și numele parametrului.

Corpul metodei este un bloc de instrucțiuni care implementează algoritmul calculului realizat de metodă. Corpul metodei reprezintă un domeniu de valabilitate în care se pot declara variabile

locale. Acest domeniu este initializat cu numele parametrilor formali care sunt cunoscuti in intreg corpul metodei. Ei se comporta ca niste variabile locale de tipul declarat in lista de parametri si initializate la fiecare apel al metodei.

Exemplu:



Metodă a instanței = Metodă care aparține fiecărui obiect al unei clase.

Metodă a clasei = Metodă care aparține unei clase și la care au acces toate obiectele sale. Este o metoda declarata cu `static`.

Comportamentul obiectelor unei clase = Acțiuni pe care obiectele unei clase sunt capabile să le îndeplinească. Acest comportament este dat de metodele non-static ale clasei.

Exemplu:

Comportamentul unui obiect `Avatar` ar putea fi următorul:

- controlează dacă se va lovi de un obstacol,
- semnalează propria poziție,
- emite sunete.

Comportamentul unei clase poate fi utilizat de alte obiecte pentru a cere obiectelor clasei:

- sa le raporteze, prelucrate sau nu, informatii prezente in variabilele-instanta ale obiectelor clasei,
- sa modifice valorile variabilelor-instanta ale obiectelor clasei.

In cursul rezolvarii propriilor sarcini, obiectele clasei pot cere si sa primească informații de la alte obiecte sau să trimită mesaje altor obiecte cerându-le, la randul lor, să execute diverse sarcini in care aceste din urma obiecte sunt specializate.

Apelul de metodă

Apel de metoda = Mecanism oferit de un limbaj de programare prin care o entitate din program (in cazul nostru un obiect) cere executia unei metode. Aceasta metoda poate sa apartina obiectului apelant (apel intern) sau, in cazul general, poate sa apartina unui alt obiect. Pentru a putea apela o metoda `M` a unui obiect `B`, un obiect `A` trebuie sa aiba acces la metoda `M`. Accesul la o metoda este stabilit de regulile de vizibilitate din Java care vor fi explicate intr-o sectiune urmatoare.

De obicei un apel de metoda are loc in momentul in care obiectul apelant executa o metoda a sa. Apelul metodei intrerupe executia metodei curente si trece controlul metodei apelate. La sfarsitul executiei metodei apelate, controlul revine la metoda intrerupta, exact in punctul de intrerupere.

Daca metoda apelata restituie (intoarce) o valoare metodei apelante, aceasta valoare poate participa la calculul unei expresii devenind un operand in acea expresie. Este cazul in care apelul de metoda apare in interiorul unei expresii sau in locul unei expresii.

Apelul unei metode poate fi vazut si ca un mesaj pe care obiectul apelant il trimite obiectului apelat. In acest sens putem spune ca obiectele comunică între ele prin intermediul metodelor.

Un obiect poate chema metodele altui obiect din mai multe motive:

- comunică o modificare a propriei stari obiectului apelat,
- avertizează un alt obiect de aparitia unui eveniment la care acel obiect probabil nu are acces,
- cere unui alt obiect să execute o anumită operație.

Apelarea unei metode se face cu ajutorul operatorului binar `.`. Primul operand al apelului trebuie sa fie o variabila-reference sau o expresie a carui valoare sa fie de tip reference catre un obiect in memorie. Al doilea operand este chiar numele metodei urmat de lista parametrilor efectivi.

De exemplu, metodele instanță ale clasei `Avatar` ar putea fi următoarele:

```

    public boolean lovesteObstacol(){...}
    public Punct semnaleazaPozitie(){...}
    public void emiteSunete(){...}

```

O metoda a unui alt obiect ar putea sa foloseasca un obiect Avatar utilizand un cod cum ar fi urmatorul:

```

    Avatar av = new Avatar();
    . . .
    if (av.lovesteObstacol()) {
        Punct p = av.semnaleazaPozitie();
        . . .
    }

```

In cazul apelurilor de metode ale clasei (este cazul metodelor declarate static), regula de construire a apelului este aceeaasi doar ca primul operand trebuie sa fie chiar numele clasei. De exemplu, in clasa Avatar ar putea fi metoda statica `creazaAvatar()`:

```

    public static Avatar creazaAvatar(){...}

```

care returneaza obiecte de tip Avatar.

Un obiect are gestioneaza obiecte Avatar ar putea folosi aceasta metoda pentru a-si crea noi obiecte Avatar:

```

    Avatar av = Avatar.creazaAvatar();

```

Nota. Ca si in cazul variabilelor-membru, apelul unei metode apartinand unui obiect din interiorul unei metode che apartine aceluiasi obiect se face fara a utiliza operatorul "." si fara a utiliza referinta la obiectul destinatie. Acest obiect este subinteles a fi obiectul curent in care ne aflam cu executia.

5.2.4 Declararea unei clase

O clasa in Java se declara utilizand un header ce contine un eventual modificador de vizibilitate, cuvantul cheie `class` si numele clasei. In corpul clasei delimitat de `{ ... }` se declara toti membrii clasei: variabile si metode.

```

public class Persoana {           // Declararea numelui clasei
    String nume;                  // Variabilă instanță de tip String
    String prenume;              // Variabilă instanță de tip String
    int varsta= 29;              // Variabilă instanță de tip int
    char sex = 'M';              // Variabilă instanță de tip char
    public int getVarsta() {      // Metodă instanță
        return varsta;
    }
    public void setNume(String n) { //Metodă instanță
        nume=n;
    }
    public void setPrenume(String p) { //Metodă instanță
        prenume=p;
    }
    public void afiseaza() {      //Metodă instanță
        System.out.println("Persoana: "+nume+" "+prenume);
    }
    public static void main(String args[]) { //Metodă clasă
        . . . . .
    }
}

```

Semnătura unei metode este, prin definiție, o combinație dintre numele metodei și tipul parametrilor în ordinea declarării lor în definirea metodei. De exemplu, semnătura metodei `setPrenume` din clasa `Persoana` este: `setPrenume + String`.

Limbajul Java admite **supraîncărcarea (overloading-ul) metodelor**: metodele dintr-o clasă pot avea același nume, este suficient sa difere prin semnătură (adică, prin tipul și/sau numărul parametrilor).

Se poate observa că tipul rezultatului nu apare în semnatura deoarece se presupune ca toate metodele supraîncărcate trebuie să aibă același tip de rezultat.

Exemplu: În clasa `String` metodele `substring()` și `indexOf()` sunt metode `overloaded`.

5.2.4. Construirea de obiecte

Constructor al unei clase = o metodă cu același nume ca și cel al clasei din care face parte și careia îi lipsește din declarație tipul rezultatului.

Un constructor descrie cum se inițializează variabilele instanță ale noului obiect.

Construcții, ca orice altă metodă, pot fi supraîncărcate. În general, o clasă poate avea mai mulți constructori. De exemplu, clasa `Persoana` ar putea avea doi constructori ca mai jos:

```
public Persoana() { //Constructor
    nume="";
    prenume="";
    varsta=0;
}
public Persoana(String n, String p){ //Constructor
    nume=n;
    prenume=p;
}
```

În acest exemplu, constructorii clasei `Persoana` sunt supraîncărcate, ei diferind prin numărul și tipul parametrilor.

Unei clase îi pot lipsi constructorii. În acest caz, compilatorul oferă automat clasei un constructor predefinit fără parametri care inițializează variabilele instanță ale clasei astfel: variabilele primitive numerice cu zero, variabilele booleene cu false și cele referință cu null.

Constructorul este chemat atunci când dorim să creăm un obiect al clasei constructorului. Pentru aceasta trebuie să folosim operatorul `new` împreună cu constructorul clasei:

```
new NumeClasa(parametri);
```

În exemplul următor se creează două obiecte `Persoana` și se memorează referințele la ele în variabile de același tip, `Persoana`:

```
Persoana amiculTau = new Persoana();
Persoana amiculMeu = new Persoana("Ionescu", "Pop");
```

Aceste variabile ar putea fi folosite pentru a apela metodele instanță. După cum s-a menționat mai sus, având o variabilă care memorează un obiect, se poate utiliza **notația cu punct** pentru apelarea unei metode instanță a unui obiect (`amiculMeu.afiseaza()`) sau pentru accesarea variabilelor instanță ale unui obiect (`amiculMeu.varsta`).

Exemplu:

```
public static void main(String args[]) {
    Persoana amiculMeu = new Persoana("Ionescu", "Pop");
    amiculMeu.varsta = 31;
    amiculMeu.afiseaza();
    Persoana amiculTau = new Persoana();
    amiculTau.setNume("Popescu");
    amiculTau.setPrenume("Ion");
    amiculTau.afiseaza();
    System.out.println("Varsta: " + amiculTau.getVarsta());
}
```

De exemplu, primul constructor al clasei `Persoana` poate fi scris așa:

```
public Persoana(String nume, String prenume){
    this.nume=nume;
    this.prenume=prenume;
}
```

this = Cuvânt cheie care se utilizează în interiorul corpului unei metode a unei instanțe de clasă și care furnizează referința la obiectul pentru care a fost apelată metoda.

Într-un constructor se poate apela un alt constructor al aceleiași clase. Astfel, al doilea constructor al clasei `Persoana` ar putea fi scris așa:

```
public Persoana() {
    this("", "");
    varsta=0;
}
```

Exemplu. Vom scrie două clase: `Punct` și `Dreptunghi` care reprezintă un punct și un dreptunghi pe ecran.

Un punct este caracterizat prin coordonatele sale `x` și `y` care sunt valori întregi. Atunci vom declara `x` și `y` ca variabile instanță ale clasei `Punct`, iar constructorului acesteia le va inițializa cu valorile dorite.

Similar, clasa `Dreptunghi` va avea ca variabile instanță: lățime și înălțime de tip `int` și originea care este de tip `Punct` și care reprezintă colțul din stânga sus. Responsabilitatea clasei `Dreptunghi` este de a muta dreptunghiuri și de a le calcula aria. Așadar, vom avea metodele instanță `muta()` și `calculeazaAria()`.

A treia clasă `TestObiecte` exemplifică, în metoda `main`, crearea și utilizarea obiectelor de tip `Punct` și `Dreptunghi`. În metoda `main` a acestei clase mai întâi creăm un punct și două dreptunghiuri, unde punctul este originea primului dreptunghi, iar cel de-al doilea are originea punctul (0,0). Apoi, afișăm caracteristicile primului dreptunghi (lățimea și înălțimea), precum și aria acestuia. Urmează schimbarea originii celui de-al doilea dreptunghi, astfel încât cele două dreptunghiuri să aibă aceeași origine și afișarea poziției acesteia. Ultimele operații efectuate este de a deplasa al doilea dreptunghi și de a afișa noile coordonate ale originii acestuia. Implementarea celor trei clase este arătată în Programul 2.

5.2.5 Modificatori de acces

Când am tratat conceptul de interfață a unui obiect, am văzut că un obiect separă două universuri: cel extern, universul tuturor obiectelor și cel intern, al său, care trebuie protejat (prin ascunderea detaliilor de implementare). Vizibilitatea pentru o clasă în Java ca și pentru membrii ei este stabilită cu ajutorul modificatorilor de acces:

- **public**. Oferă o vizibilitate globală, de oriunde din afara clasei.
- **protected**. Vizibilitate acordată doar subclasei clasei în care sunt definite elementele protejate, indiferent care sunt pachetele acestor subclase.
- **private**. Elementele private nu sunt accesibile decât în interiorul clasei în care sunt definite.
- **modificator nespecificat**. Este cunoscut și ca modificator `friendly` și oferă acces liber claselor din interiorul aceluiași pachet.

Există și alți modificatori ce pot fi incluși în declarații. Ei se referă la alte aspecte decât drepturile de acces:

- **static**. Definește membrii clasei: variabile și metode.
- **final**:
 - pentru variabile. Reprezintă variabilele ale căror valori, odată inițializate, nu se pot modifica în cursul execuției. De obicei ele trebuie inițializate în momentul declarării:


```
final String numeFirma="... SRL";
```
 - pentru metode. Astfel de metode nu pot fi redefinite în subclasele clasei din care fac parte.
 - pentru clase. O astfel de clasă nu poate fi extinsă. Este o clasă "frunză" în structura arborescentă a claselor formată pe baza relației de moștenire dintre clase.
- **synchronized**. Cere ca în orice moment al execuției cel mult un fir de execuție (thread) să poată executa acea metodă sau să acceseze acea dată sincronizată.
- **native**. Reprezintă o metodă implementată într-un fișier extern, provenind dintr-un program scris într-un alt limbaj (de exemplu, limbajul C).

abstract. Introduce o clasă sau o metodă abstractă. Dacă este vorba de o clasă abstractă, atunci ea trebuie să conțină cel puțin o metodă abstractă.

Modificatorul "static"

Metodele și variabilele care nu sunt declarate "statice" sunt metode și variabile ale instanțelor. Analog, metodele și variabilele declarate "statice" sunt metode și variabile ale claselor.

În Programul 3 sunt prezentate două exemple care folosesc variabile și metode statice.

```

public class Punct{
    public int x;
    public int y;
    public Punct(int x, int y){//constructor
        this.x = x;
        this.y = y;
    }
}
//sfarsitul clasei Punct
public class Dreptunghi{
    public int latime, inaltime;
    public Punct origine;
    public Dreptunghi(){origine=new Punct(0, 0);}//constructor (overloaded)
    public Dreptunghi(Punct p){origine=p;} //constructor (overloaded)
    public Dreptunghi(int l, int i){this(new Punct(0, 0), l, i);}//constructor
    //overloaded
    public Dreptunghi(Punct p, int l, int i){ //constructor (overloaded)
        origine=p;
        latime=l;
        inaltime=i;
    }
    public void muta(int x, int y){
        origine.x = x;
        origine.y = y;
    }
    public int calculeazaAria(){return latime*inaltime;}
}
//sfarsitul clasei Dreptunghi
public class TestObiecte{
    public static void main(String[] args) {
        //cum cream un punct si doua dreptunghiuri
        Punct primaOrigine=new Punct(23, 94);
        Dreptunghi unu=new Dreptunghi(primaOrigine,100, 200);
        Dreptunghi doi=new Dreptunghi(50, 100);
        //afisam attributele unui dreptunghi
        System.out.println("Latimea primului dreptunghi: " + unu.latime);
        System.out.println("Inaltimea lui unu: " + unu.inaltime);
        //calculam aria unui dreptunghi
        System.out.println("Aria lui unu: " + unu.calculeazaAria());
        //schimbam originea dreptunghiului
        doi.origine=primaOrigine;
        //afiseaza pozitia dreptunghiului
        System.out.println("Pozitia x a dreptunghiului doi: "+ doi.origine.x);
        System.out.println("Pozitia y dreptunghiului doi: "+ doi.origine.y);
        //mutam si apoi afisam dreptunghiul
        doi.muta(40, 72);
        System.out.println("Pozitia x a dreptunghiului doi: "+ doi.origine.x);
        System.out.println("Pozitia y dreptunghiului doi: "+ doi.origine.y);
    }
}

```

Programul 2. TestObiecte

```

class Exemplu1 {
    static String text = "Un program simplu!";
    public static void main(String[] args){
        System.out.println(text);
    }
}

class Exemplu2 {
    String text = " Un program simplu!";
    static String text2 = " Un program simplu!";
    String getText(){return text;}
    String getStaticText(){return text2;}
    public static void main(String[] args){
        Exemplu2 instanta=new Exemplu2();
        String unText=instanta.getText();
        String unTextStatic=instanta.getStaticText();
        System.out.println(unText);
        System.out.println(unTextStatic);
    }
}

```

Programul 3. Utilizarea modifierului `static`

Primul exemplu evidențiază faptul că variabilele statice pot fi folosite direct de către metodele statice ale aceleiași clase. Este cazul variabilei `text`, a cărei valoare este afișată în metoda `main` a clasei `Exemplu1`.

A doua clasă, `Exemplu2`, conține două variabile: `text` și `text2`. Prima este variabilă instanță iar a doua este declarată `static`, deci este variabilă a clasei. Valorile acestor variabile sunt returnate de către metodele instanță `getText()` și `getStaticText()` ale aceleiași clase. Aceste metode vor fi apelate din metoda statică `main`. Primul lucru pe care-l facem este să ne creăm un obiect de tip `Exemplu2` a cărui referință este memorată în variabila `instanta`. Șirurile rezultate din apelarea metodelor instanță sunt memorate în variabilele locale `unText` și `unTextStatic`, iar acestea din urmă vor fi folosite în apelarea metodei `println()`. Așadar, al doilea exemplu pune în evidență faptul că variabilele statice pot fi folosite și în interiorul metodelor instanță ale clasei în care au fost declarate.

Observație. Într-o metodă declarată cu `static` nu se poate folosi variabila `this!!!`

Membrii statici ai unei clase

Variabilele statice ale unei clase, în cazul în care nu sunt declarate `private`, pot fi accesate direct, prin intermediul fie a clasei de care aparțin, fie a unei instanțe a clasei memorată într-o variabilă referință de tipul clasei respective.

Exemplu:

Definiție

```

class Comanda {
    String denumire;
    String TVA;
    int nrComanda;
    Date data;
    static int totalComenzi = 0;
    . . .
}

```

Acces

```

Comanda noua = new Comanda("...", "...", ...);
noua.totalComenzi++;

Comanda.totalComenzi++;

```

La fel, metodele statice ale unei clase pot fi apelate direct, cu ajutorul clasei de care aparțin sau prin intermediul variabilelor referință de tipul clasei respective. Continuând exemplul anterior, adăugăm clasei `Comanda` metoda statică `anuleaza()` care anulează valoarea variabilei statice `totalComenzi`:

Definiție

Acces

```

class Comanda{
    String denumire;
    String TVA;
    int nrComanda;
    Date data;
    Comanda.anuleaza();
    static int totalComenzi = 0;
    static void anuleaza() {
        totalComenzi=0;
    }
}

Comanda noua=new Comanda(. . . . .);
noua.anuleaza();

Comanda.anuleaza();

```

5.2.5. Pachete

Unitate de compilare = Un fișier de cod sursă cu extensia **.java** care poate fi compilat. Un asemenea fișier poate conține una sau mai multe declarații de clase și interfețe. Doar una din clasele din fișier trebuie să fie declarată **public**. Numele acestei clase este și numele fișierului.

Pachet (package) = Un grup de clase și interfețe grupate sub un nume. Reprezintă un spațiu de nume pentru toți identificatorii de clase și interfețe declarați ca aparținându-i. Amintim că într-un spațiu de nume numele declarate sunt unice.

Apartenența la un pachet se declară global pentru toate numele de clase și interfețe ale unei unități de compilare. Declararea apartenenței claselor dintr-o unitate de compilare la un pachet se face utilizând cuvântul cheie **package** urmat de numele pachetului ca prima linie a unității de compilare respective:

```

package Identificator;
package modulGrafic;

```

În consecință, într-un program Java putem găsi mai multe unități de compilare declarate ca aparținând aceluiași pachet, iar pentru fiecare pachet vom avea cel puțin o unitate de compilare care să declare clase și interfețe din pachet.

În lipsa declarației de pachet, o unitate de compilare va plasa toate clasele și interfețele sale într-un pachet anonim, pachetul **predefinit**, care nu trebuie specificat dacă dorim să accesăm componentele sale.

Sintaxa unei unități de compilare ce conține clase și interfețe ce aparțin unui pachet cu nume este următoarea:

```

package NumePachet;
DeclaratiiImportPachete // declaratii ce folosesc cuvantul cheie import
DeclaratiiDeClaseSiInterfete

```

Utilizarea numelor de pachete

Un pachet poate fi declarat ca aparținând unui alt pachet. Aceasta face ca structura pachetelor în Java să fie ierarhică. În cazul unei ierarhii cu mai multe niveluri, în locul numelui simplu al pachetului va trebui să utilizăm numele complet care folosește notația cu punct ca în următorul exemplu ce identifică un pachet R:

P.Q.R

unde R este un pachet declarat ca aparținând lui Q, Q un pachet care aparține lui P, iar P este un pachet care aparține programului. Declarația incluziunii unui pachet într-un alt pachet se face simplu, specificând în declarația **package** numele complet al pachetului.

Ideea care stă la baza acestei organizări este evitarea conflictelor de nume. Într-adevăr, există clase în pachetele Java standard care au același nume. Există, de exemplu, o clasă cu numele **Object** în pachetul **java.lang** și o interfață cu același nume în pachetul **org.omg.CORBA**. Ele se pot utiliza în cadrul aceluiași program fără a intra în conflict între ele doar dacă folosim numele lor complet: **java.lang.Object** și **org.omg.CORBA.Object**. În cazul numelui **Object** se poate verifica și o altă regulă. Dacă utilizăm în programul nostru numele simplu **Object**, acesta se va referi automat la clasa **java.lang.Object** deoarece **java.lang** este unicul pachet Java standard importat automat în programul nostru. Pentru a utiliza interfața **Object** din pachetul **org.omg.CORBA** va trebui să utilizăm numele complet al clasei: **org.omg.CORBA.Object**.

Exemplu. Sa consideram o clasă `Document` care este declarată în pachetul `idd` din structura de pachete `ro.univ_ovidius` (în cele ce urmeaza se va considera în locul adresei `univ-ovidius.ro` a domeniului Internet al universitatii Ovidiu adresa `univ_ovidius.ro`). `Document` folosește o clasă `Vector` care face parte din pachetul `java.util`. În program vom scrie atunci următoarele declarații:

```
package ro.univ_ovidius.idd;
import java.util.Vector;
public class Document{
    . . .
}
```

Un program Java este în general format din conținutul mai multor pachete. Clasele și interfețele care nu sunt indicate că aparțin unui pachet, aparțin automat pachetului predefinit, anonim. Acesta, la rândul său, poate fi alcătuit din clase, interfețe și alte pachete. Într-un program complex, cu o ierarhie de pachete pe mai multe niveluri, pachetul anonim ar fi de dorit să conțină numai pachete. După cum se poate constata, structura de pachete și clase se aseamănă cu structura de directoare și fișiere dintr-un sistem de fișiere. Această asemănare nu este doar de suprafață în Java. Ea ne conduce la modul de structurare a fișierelor de bytecode `.class` pe discul calculatorului. Regula este simplă: fiecărui pachet Java trebuie să îi corespundă pe disc un director având același nume cu numele pachetului. În directorul respectiv se vor găsi fișierele `.class` ale tuturor claselor și interfețelor declarate în pachet plus directoarele pachetelor declarate în pachet. Deci dacă pachetul Java respectiv este la rândul său inclus într-un alt pachet, atunci directorul său trebuie să fie inclus în directorul pachetului-tată. Această ierarhie de directoare trebuie să aibă un director-radacină corespunzător pachetului predefinit. Pentru a putea executa un program structurat pe pachete, acest director trebuie să apară în variabila `CLASSPATH` a sistemului.

De aceea, pentru a compila cu Java SDK clasa `Document` din pachetul `idd` de mai sus se poate utiliza comanda:

```
javac -d C:\studenti\ionica Document.java
```

unde `C:\studenti\ionica` este numele unui director declarat în variabila `CLASSPATH` a sistemului. Această comandă va memora fișierul (fișierele) `.class` rezultat (e) din compilarea lui `Document.java` în directorul `C:\studenti\ionica\ro.univ_ovidius.idd`.

Nota. Pentru a reutiliza clasele Java scrise de noi în alte programe, trebuie să fim siguri că numele acestor clase, care sunt unice în orice program, sunt importate. În acest scop Sun recomandă folosirea ca nume de pachete a unor nume compuse în stilul numelor domeniile Internet. Este ideea folosită de declarația de pachet de mai sus în care se utilizează un nume de domeniu internet unic care este `univ-ovidius.ro` pentru a crea un pachet `ro.univ_ovidius.idd` unic în lume. Se observă inversarea ordinii de apariție în numele compus a numelor simple. Dacă nu aveți un domeniu Internet la dispoziție puteți utiliza cu mare probabilitate de a avea nume unice pentru clase adresa voastră de e-mail. De exemplu, prefixul `com.yahoo.popescu.ion` ar putea fi o soluție pentru pachetele de clase Java ale lui Ion Popescu. Chiar și în această situație rămâne valabilă convenția folosită de Java pentru a mapa pachetele de clase pe disc. Pentru fișierele claselor unui pachet `com.yahoo.popescu.ion.licenta`, Ion Popescu va trebui să își creeze un subdirector pe disc `com\yahoo\popescu\ion\licenta` în care să găzduiască fișierele claselor pachetului.

Dacă acest pachet va fi folosit într-un unic program, acest subdirector se poate include în directorul ce conține fișierele programului.

Dacă vom utiliza clasele pachetului `com.yahoo.popescu.ion.licenta` în mai multe programe, în loc să includem subdirectorul `com\yahoo\popescu\ion\licenta` în directoarele tuturor acestor programe, putem crea un director nou pe disc, să zicem `C:\studenti\ionica` pe care îl vom memora o singură dată. Pentru a accesa din toate programele Java clasele pachetului `com.yahoo.popescu.ion.licenta` este suficient să includem în `CLASSPATH` șirul `C:\studenti\ionica;`.

5.2.6 Structura unui program Java

Ca o concluzie la acest capitol, în **Errore. L'origine riferimento non è stata trovata.** se prezinta program `TestAngajat` alcatuit din doua clase si în care s-au evidentiat principalele constructii sintactice.

```
package ro.univ_ovidius.personal; //pachetul clasei
import java.util.*; //clasele de importat
import pachetulMeu.DataCalend;

public class TestAngajati{
    public static void main(String[] args) { //metoda main() a clasei
        Angajat[] personal = new Angajat[3];
        personal[0] = new Angajat("Paul Enescu", 350, new DataCalend(1989,10,1));
        personal[1] = new Angajat("Ion Popescu", 750, new DataCalend(1987,12,5));
        personal[2] = new Angajat("Pop Ionescu", 380, new DataCalend(1990,3,15));
        for (int i = 0; i < 3; i++) personal[i].maresteSalariu(5);
        for (int i = 0; i < 3; i++) personal[i].print();
    }
}

class Angajat{

// variabile-instanta
    private String nume;
    private double salariu;
    private DataCalend dataAngajare;

//constructorul de obiecte Angajat
    Angajat(String n,double s,DataCalend d){
        nume = n;
        salariu = s;
        dataCalendAngajare = d;
    }

// metode-instanta
    public void print() {
        System.out.println(nume + " " + salariu + " " + getAnAngajare());
    }
    public void maresteSalariu(double procent) {
        salariu *= 1 + procent / 100;
    }
    public int getAnAngajare() {
        return dataAngajare.getAn();
    }
}
```

Programul 4. TestAngajat

5.3 RELAȚII ÎNTRE CLASE ȘI OBIECTE

Prin definiție, o **relație** este o legătură logică sau fizică între două clase sau între obiecte a două clase implicate în relație. Relațiile sunt de mai multe tipuri:

1. **Generalizare/Specializare** = Relație logică între două clase, reprezentand doua concepte A si B: B este o specializare a lui A cand despre cele doua concepte A si B se poate spune "B este un A" sau "B este un tip de A" sau "B este o specie de A":
"dulapul este o mobila",
"abonamentul TV este un tip de contract",
"delfinul este o specie de mamifer" etc.
2. **Asociere** = Relație fizică între obiectele a două clase care indică faptul că instanțele unei clase "cunosc" sau sunt conectate la instanțele celeilalte clasei:
"un medic isi cunoaste pacientii" si vice versa "un pacient isi cunoaste medicii care il trateaza",

“un cont bancar isi cunoaste posesorul”,
 “un produs isi cunoaste furnizorul”,
 “un angajat isi cunoaste functia in intreprindere”,
 “un automobil Dacia este conform unei fise tehnice a modelului” etc.

3. **Agregare** = Relație fizică structurală între obiectele a două clase care indică faptul că instanțele unei clase sunt compuse logic din instanțe ale celeilalte clase.
 4. O **compoziție** este o formă mai puternică de agregare între două clase în care existența obiectelor agregate depinde de existența obiectului care le conține.
 5. **Dependență** = Relație în general de tip “utilizează” în care se reprezintă faptul că o modificare în specificația unei componente ar putea influența componenta care o utilizează, dar nu invers.
- Diagrama 2 prezintă relațiile dintre clasele: *GestiuneComenzi*, *Comanda*, *ElementComanda*, *ComandaUrgenta* și *ContBancar*, în limbajul vizual UML.

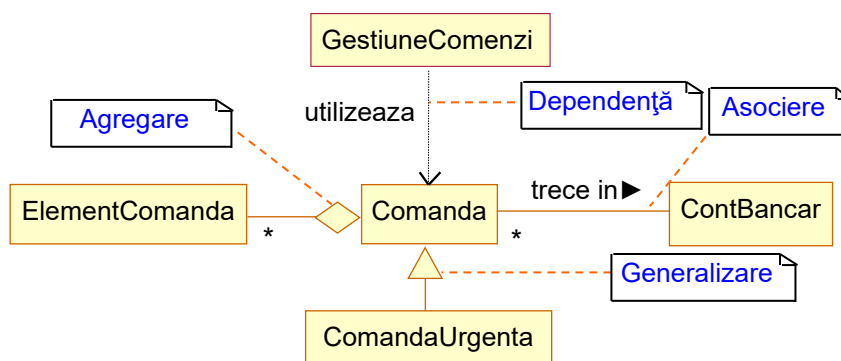


Diagrama 2. Relatii între clase în UML

Generalizare/Specializare

Este o relație cu înțelesul “. . .este o specie de. . .” între două concepte sau tipuri de date reprezentate de două clase de obiecte:

- **superclasa** sau **clasa de bază**,
- **subclasa** sau **clasa derivată**.

Exemplu.

În Diagrama 3 se prezintă relațiile de generalizare dintre clasele *Client*, *ClientPrivat* și *ClientSocietate*. Ultimele două clase reprezintă specializări ale primei clase:

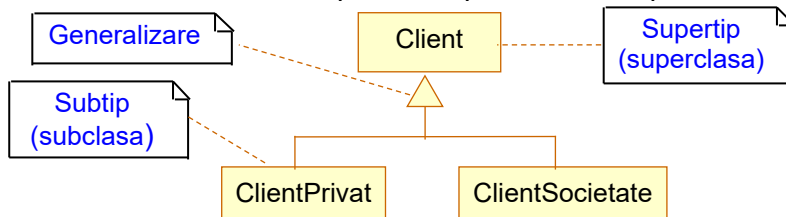


Diagrama 3. Generalizare/Specializare

Generalizare vs. moștenire

Implementarea directă a relației de generalizare/specializare se face prin intermediul **moștenirii**, un mecanism specific limbajelor orientate spre obiecte.

Moștenire = Clasa derivată **moștenește** din clasa de bază toate atributele și toate operațiile neprivate. Prin intermediul moștenirii, o clasă obține imediat funcționalitatea unei clase existente.

Exemplu.

Presupunem că avem un editor grafic care folosește clasele din Diagrama 4:

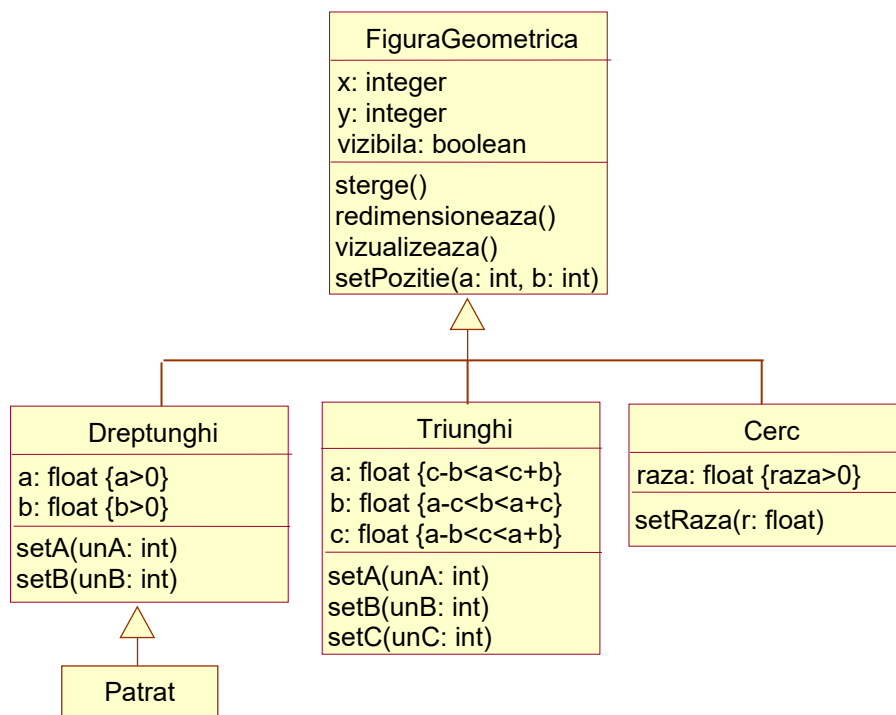
Subclasele Dreptunghi, Triunghi și Cerc moștenesc din superclasa FiguraGeometrica, atributele:

- x, y care reprezintă coordonatele punctelor ce formează un dreptunghi sau un triunghi, sau centrul cercului, dacă forma geometrică este un cerc, și
- predicatul vizibilă care determină dacă figura există sau nu în desenul construit de utilizator;

și metodele:

- *sterge()* care elimină din desen figura geometrică respectiva,
- *redimensioneaza()* care redimensioneaza figura respectiva în desen,

- vizualizeaza() care vizualizeaza figura respectiva in desen,
- setPozitie(a: int, b: int) seteaza coordonatele unui punct (adică modifică valorile atributelor x și



y).

Diagrama 4. Ierarhia de clase FiguraGeometrica

Asociere

O **asociere** între două clase este o mulțime de legături sau conexiuni între obiecte a celor două clase. O asemenea conexiune se numește *link*.

Un link se poate specifica ca o pereche de obiecte aparținând celor două clase. Ordinea în cadrul perechii, pentru a nu fixa punctul de plecare a relației poate fi dată de **roluri**. Un rol este un nume dat unuia din cei doi membrii ai perechii.

Formal, o asociere este o relație între mulțimile de obiecte ale celor două clase, adică o submulțime a produsului cartezian al obiectelor celor două clase.

Exemplu.

Fie mulțimile de entități Spital și Persoana.

Spital = { "Judetean", "Municipal", "Policlinica" }

Persoana = { "George Sarbu", "Nicu Enescu", "Ion Popescu", "Pop Ionescu" }

Una din relațiile dintre o persoană și un spital poate fi relația de *conducere*. Ea indică legătura dintre un spital și directorul său.

Ea poate fi definită ca o mulțime de perechi (spital, persoana):

conduce = { ("Judetean", "George Sarbu"), ("Municipal", "Ion Popescu"), ("Policlinica", "Pop Ionescu") }

Se vede că:

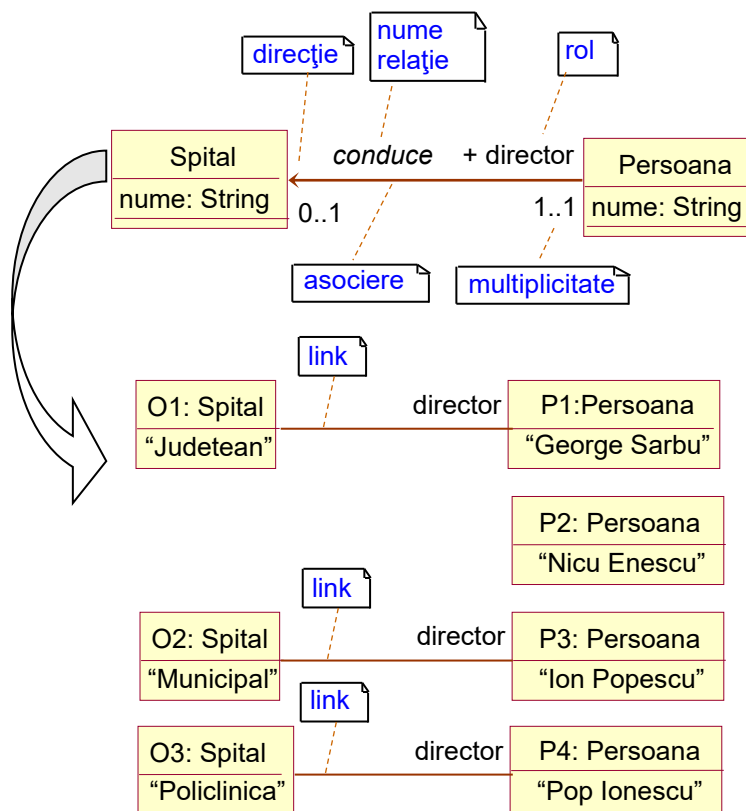
$conduce \subset (Spital \times Persoana)$

Să considerăm definiția a două proiecții pr_1 și pr_2 asociate unei relații $R \subset A \times B$.

$pr_1(R) = \{ a \in A \mid \exists b \in B, (a, b) \in R \}$

$pr_2(R) = \{ b \in B \mid \exists a \in A, (a, b) \in R \}$

Diagrama de mai jos sintetizează în UML această relație.



Utilizarea cardinalitatilor 1..1 si 0..1 indica numarul de obiecte dintr-o clasa care pot participa la conexiuni cu un acelasi obiect din cealalta clasa, deci se refera la existenta in relatia conduce a unor conditii de tipul:

1..1 $\Rightarrow pr_1(\text{conduce}) = \text{Spital}$

0..1 $\Rightarrow pr_2(\text{conduce}) \subset \text{Persoana}$

In exemplul urmator se introduce intre aceleasi clase Spital si Persoana o alta relatie, aceea de *angajare*.

$\text{angajare} \subset (\text{Spital} \times \text{Persoana})$

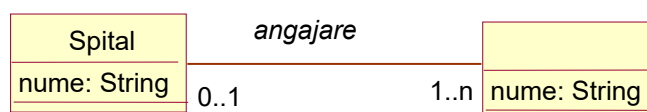
Un spital, dupa cum se stie, poate angaja mai multe persoane, de altfel nu poate functiona fara angajati. Cu alte cuvinte printre perechile (spital, persoana) va trebui să gasim cel putin una pentru fiecare spital. Adica:

$pr_1(\text{angajare}) = \text{Spital}$

O persoana poate fi sau nu angajata unui spital, iar daca este angajata nu poate fi decat a unui spital. Cardinalitatea deci este 0..1 deoarece:

$pr_2(\text{angajare}) \subset \text{Persoana}$

Am ajuns la aceleasi expresii pentru proiectii, desi cardinalitatea in partea Persoana a relatiei este diferita fata de exemplul precedent. ? Stim din pr_1 ca orice spital are angajati, dar cum am putea stii care sunt acesti angajati? Ce lipseste din modelul matematic ca sa putem captura si multiplicitatea aparitiilor de conexiuni cu un acelasi obiect?



Ar trebui sa utilizam conceptul de *corespondenta* (mapping) care ne da asocierile dintre obiectele celor doua clase. Echivalentul acestui concept in UML este jucat de catre rolul relatiei. Intr-adevar, rolul angajat reprezinta numele unei functii care pentru fiecare spital ne restituie multimea angajatilor lui.

angajat("Municipal") = { "Ana Velciu", "Simona Nae", "Maria Ionescu", . . . }

Așa cum un rol patron plasat pe partea clasei Spital este o funcție care pentru fiecare persoană restituie o mulțime de spitale la care persoana este angajată. În cazul de față aceasta este o mulțime vidă sau cel mult cu un singur spital.

patron("Maria Ionescu") = { "Municipal" }

Probleme cu clase de obiecte

1. Să se implementeze o clasă Cerc, împreună cu metodele getAria() pentru a obține aria și getCircumferinta(), care returnează circumferința. În constructor indicați raza cercului.
2. Să se implementeze o clasă Student. Fiecare student are un nume și un punctaj total al examenelor. Furnizați un constructor corespunzător și metodele: getNume() pentru obținerea numelui; adaugaExamen(int nota) pentru a aduna punctajul fiecărui examen; getTotal() pentru a obține punctajul total; getMediaNote() pentru a returna media punctajului.
3. Să se implementeze o clasă Produs. Fiecare produs are un nume și un pret. Scrieți constructorul și următoarele metode: afiseazaProdus(), getPret() și setPret(). Să se scrie un program care crează două produse și le afișează, apoi să li se reducă prețul cu 10000 lei și apoi să se afișeze din nou.
4. Să se scrie o clasă numită Rational pentru realizarea operațiilor aritmetice cu numere raționale. Folosiți variabile întregi pentru a reprezenta variabilele instanță private ale clasei: numărător și numitor. Scrieți un constructor care inițializează numărătorul cu 0, iar numitorul cu 1. Un alt constructor va stoca fracția în forma redusă (de ex, fracția $\frac{2}{4}$ ar trebui stocată în obiect ca $\frac{1}{2}$). Clasa rational va furniza următoarele metode publice ce vor implementa următoarele operații asupra numerelor raționale:
 - adunarea a două numere raționale. Rezultatul va fi un număr rațional stocat în forma redusă.
 - scăderea a două numere raționale. Rezultatul va fi un număr rațional stocat în forma redusă.
 - înmulțirea a două raționale. Rezultatul va fi un număr rațional stocat în forma redusă.
 - împărțirea a două numere raționale. Rezultatul va fi un număr rațional stocat în forma redusă.
 - afișarea unui număr rațional în forma numărator/numitor.
 - afișarea unui număr rațional în format virgulă flotantă.Să se scrie o altă clasă Test care testează funcționalitatea clasei Rational.
5. Să se scrie un program care verifică dacă data calendaristică introdusă de utilizator este corectă. Formatul datei introduse este zi/luna/an, iar anul trebuie să fie între 1901 și 2099. Se vor folosi constructori supraîncărcați.
6. Să se scrie o clasă Complex care realizează operații aritmetice cu numere complexe. Numerele complexe au forma: parteaReală+i*parteImaginara, unde $i=\sqrt{-1}$. Folosiți variabile double pentru a memora datele private ale clasei. Furnizați doi constructori, unul care inițializează variabilele clasei și un constructor care le inițializează cu valori implicite. Scrieți metode publice pentru fiecare din următoarele operații:
 - adunarea a două numere complexe. Rezultatul este un număr complex nou.
 - scăderea a două numere complexe. Rezultatul este un număr complex nou.
 - afișarea unui număr complex sub forma (a,b) unde a este partea reală, iar b partea imaginară a numărului complex.Să se scrie o altă clasă care testează funcționalitatea clasei Complex.

6. MOȘTENIRE ȘI POLIMORFISM

Moștenirea în Java este indicată prin cuvântul cheie `extends` în declarația unei clase:

```
class Identificator extends NumeSuperClasa {  
    CorpulClasei  
}
```

unde *Identificator* este subclasa, iar *NumeSuperClasa* este superclasa sa. De exemplu, clasa *Femeie* este o subclasa a clasei *Persoana*:

```
class Femeie extends Persoana{  
    ...  
}
```

În Java, clasele formează o structură de arbore în care rădăcina este clasa `Object`, a cărei definiție apare în pachetul `java.lang`. Orice clasă extinde direct (implicit sau explicit) sau indirect clasa `Object`.

O subclasă care dorește să se deosebească de superclasa sa (pentru că altfel ce rost ar avea să o introducem?) are două posibilități:

- să adauge membri noi și/sau
- să redefinească membrii superclasei.

super = Cuvânt cheie care se utilizează în interiorul corpului unei metode a unei instanțe a unei clase pentru a indica obiectul superclasei ce intră în structura instanței curente.

Nota. Un obiect al unei subclase este alocat în memorie într-o zonă care include atât spațiul alocat obiectului superclasei, cât și spațiul necesar variabilelor declarate în subclasa.

Exemplu

Reutilizând clasele `Angajat` și `TestAngajat` de mai sus, să definim o clasă nouă `Manager` ca un caz particular de `Angajat`. Știm că un manager are o secretară cu un nume cunoscut. În fine, știm că mărirea salariului unui obiect `Manager` se face ca și pentru un `Angajat` doar că la fiecare mărire i se mai adaugă un procent de 0.5% pentru fiecare an de vechime. Iată programul:

```
import java.util.*;
. . . .
public class TestAngajat{
    public static void main(String[] args) {
        Angajat[] personal= new Angajat[3];
        . . . . .
    }
}
class Angajat{
public void maresteSalariu(double procent){
    . . .
}
class Manager extends Angajat{
    public Manager(String n, double s, Date d, String nume) {
        super(n, s, d);
        numeSecretara = nume;
    }
public void maresteSalariu(double procent) {
    //adauga o prima de 1/2% pentru fiecare an
    Zi azi=new Zi();
    double prima=0.5*(azi.getAn()-anAngajare());
    super.maresteSalariu(procent + prima);
}
    public void setNumeSecretara(String n) {
        numeSecretara = n;
    }
    public String getNumeSecretara(){
        return numeSecretara;
    }
    private String numeSecretara;
}
```

S-a definit o clasa `Manager` ca subclasă a lui `Angajat`. `Manager` are o variabilă privată în plus `numeSecretara` pentru memorarea numelui secretarei. Aceasta informație în plus ne obliga să adăugăm o pereche `get/set` pentru accesul la variabila privată. Setarea numelui secretarei se face apelând o metodă inexistentă în clasa `Angajat`: `setNumeSecretara(String s)`. Metoda care restituie numele secretarei este `getNumeSecretara()`. Trebuie să rezolvăm și inițializarea unui obiect `Manager`. Pentru aceasta se definește un constructor special pentru clasa `Manager`. În constructor se inițializează mai întâi partea obiectului care corespunde unui obiect `Angajat` (apelând constructorul superclasei cu `super`) și apoi se inițializează variabila `numeSecretara` cu sirul `nume`.

În fine, mărirea salariului unui obiect `Manager` se face tot cu metoda `maresteSalariu(double)` ca și pentru un `Angajat` doar că salariul se calculează altfel: la fiecare mărire `i` se mai adăugă un procent (0.5) pentru fiecare an de vechime. Așadar, a trebuit să rescriem metoda `maresteSalariu(double)`. Dat fiind că după adăugarea premiului de vechime, unui `Manager` i se mărește salariul ca și unui `Angajat`, s-a reutilizat algoritmul `maresteSalariu()` al superclasei chemând:

```
super.maresteSalariu(procent + prima);
```

Clasa Object

După cum s-a aratat toate obiectele din Java moștenesc clasa `Object`. Orice clasă redefineste metode moștenite din clasa `Object`.

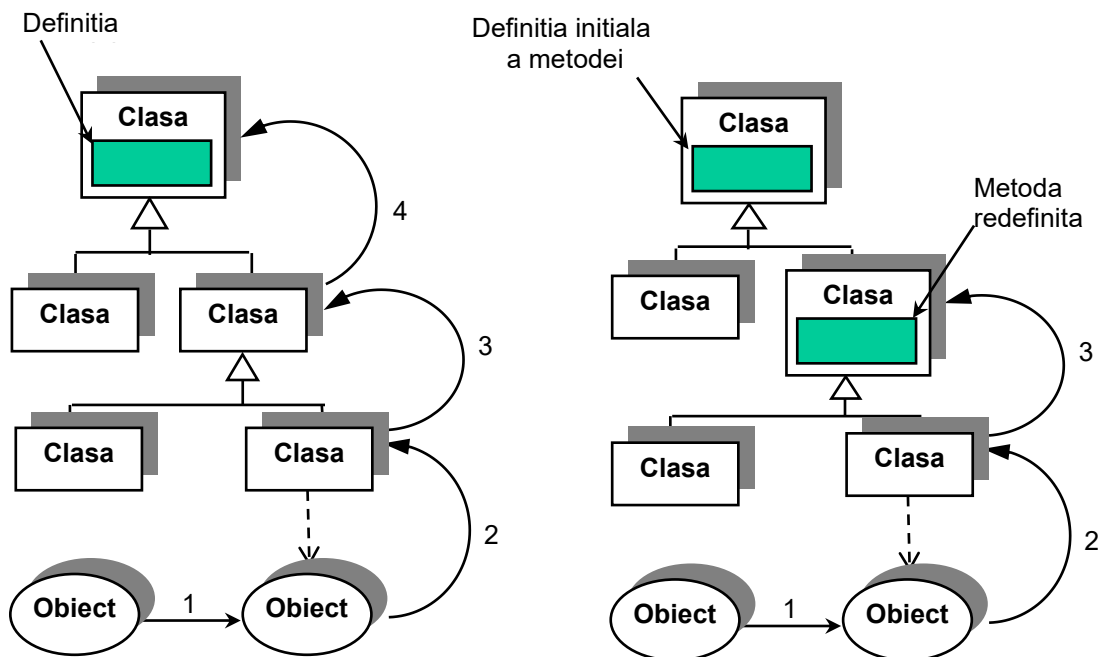
```
public class Object {
    public java.lang.Object();           // constructor
    public java.lang.String toString();
    protected native java.lang.Object clone() throws
                                                CloneNotSupportedException;
```

```

public boolean equals(java.lang.Object);
public native int hashCode();
protected void finalize() throws Throwable;
public final native java.lang.Class getClass();
public final native void notify();
public final native void notifyAll();
public final void wait() throws InterruptedException;
public final native void wait(long) throws InterruptedException;
public final native void wait(long, int) throws InterruptedException;
}

```

Mecanismul apelarii metodelor



O metodă din superclasă poate fi redefinită (suprascrisă) în subclasă cu următoarele observații:

- metoda redefinită trebuie să aibă exact aceeași semnătură;
- metoda redefinită trebuie să aibă același tip al rezultatului întors;
- dacă metoda redefinită își schimbă modificatorul de acces, acesta trebuie să acorde o vizibilitate "mai mare" față de metoda din superclasă.

Dupa cum se vede din figura de mai jos, apelul unei metode a unui obiect aparținând unei subclase, dacă metoda apelată nu este definită în subclasa, este transmis în sus în ierarhia de clase până când este găsită o prima definiție a metodei apelate.

Reutilizarea codului

Reutilizarea codului în Java oferă mai multe posibilități unui programator pentru a-și ușura munca:

1. reutilizarea codului unei clase pentru crearea de noi obiecte,
2. reutilizarea claselor pentru scrierea de clase noi utilizând mecanismul moștenirii,
3. reutilizarea obiectelor unor clase existente pentru implementarea funcționalității obiectelor unei clase noi (compoziție +delegare).

Cum prima posibilitate este evidentă pentru că nu putem programa cu obiecte dacă nu o utilizăm, celelalte două necesită o analiză atentă din partea programatorului.

Moștenirea va fi utilizată doar în cazul în care între noua clasă și o clasă din program există o relație de generalizare/specializare sintetizată prin "este o specie de ..." sau prin "este un tip de ..." (un manager este un tip de angajat, "o ființă umană este o specie de mamifer" etc.).

Când decide pentru utilizarea moștenirii, programatorul trebuie să fie atent:

- sa nu exagereze cu overriding-ul metodelor in subclase sau cu anularea membrilor superclasei. Nevoia utilizarii repetate a acestor operatii ar trebui sa ridice dubii asupra corectitudinii folosirii mostenirii;
- sa nu confunde relatia “este o tip de ...” cu relatia “**este un rol dezvoltat de ...**”. Pentru aceasta din urma nu trebuie folosita mostenirea ci delegatia cu compozitie.

Nota. Cand se utilizează moștenirea trebuie să funcționeze **principiul înlocuirii** :

Orice obiect al subclasei poate înlocui în program un obiect al superclasei fără să genereze mesaje de eroare la execuție.

Compozitie cu delegare

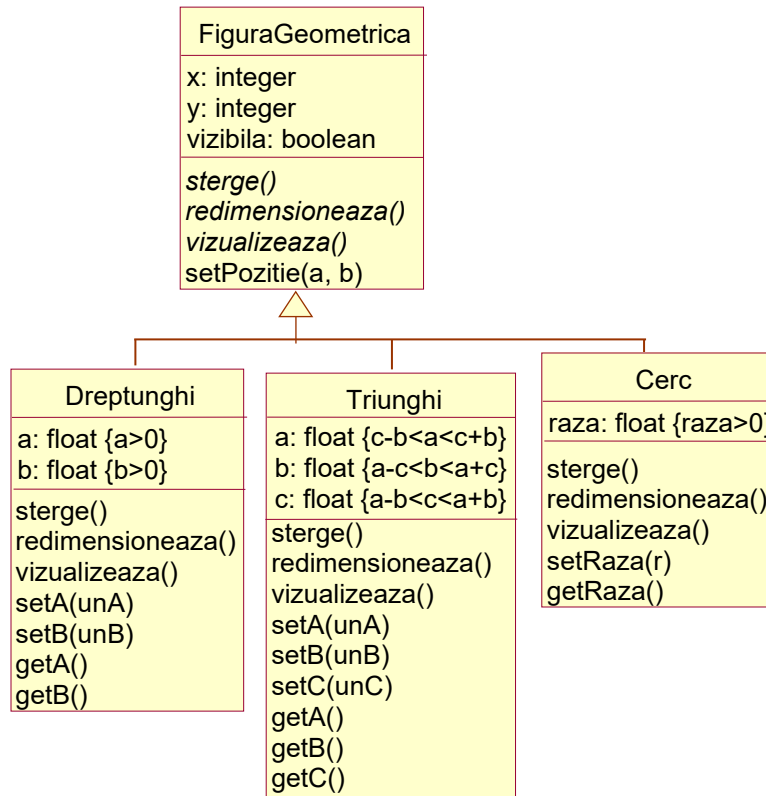


Diagrama 1. Clasa FiguraGeometrica cu subclasele sale

Presupunem că avem un editor grafic cu trei figuri geometrice la dispoziție: dreptunghi, cerc și triunghi in relatie asa cum se vede in Diagrama 1. Ni se cere să adăugăm o figură geometrică nouă: CercPatrat formată dintr-un cerc și un pătrat.

Compozitia cu delegare ne da o solutie eleganta problemei in care nu trebuie sa rescriem nici o instructiune din codul deja existent. Mai mult, mecanismul il putem utiliza pentru implementarea noii functionalitati. Solutia este sintetizata in Diagrama 2.

Codul noii clase este prezentat in Programul 1. El utilizeaza clasa FiguraGeometrica si obiectele clasele Dreptunghi si Cerc deja existente. Clasa CercPatrat extinde FiguraGeometrica si agrega un cerc si un dreptunghi. Metodele sale utilizeaza cele 2 obiecte agregate pentru a implementa functiunile impuse de FiguraGeometrica.

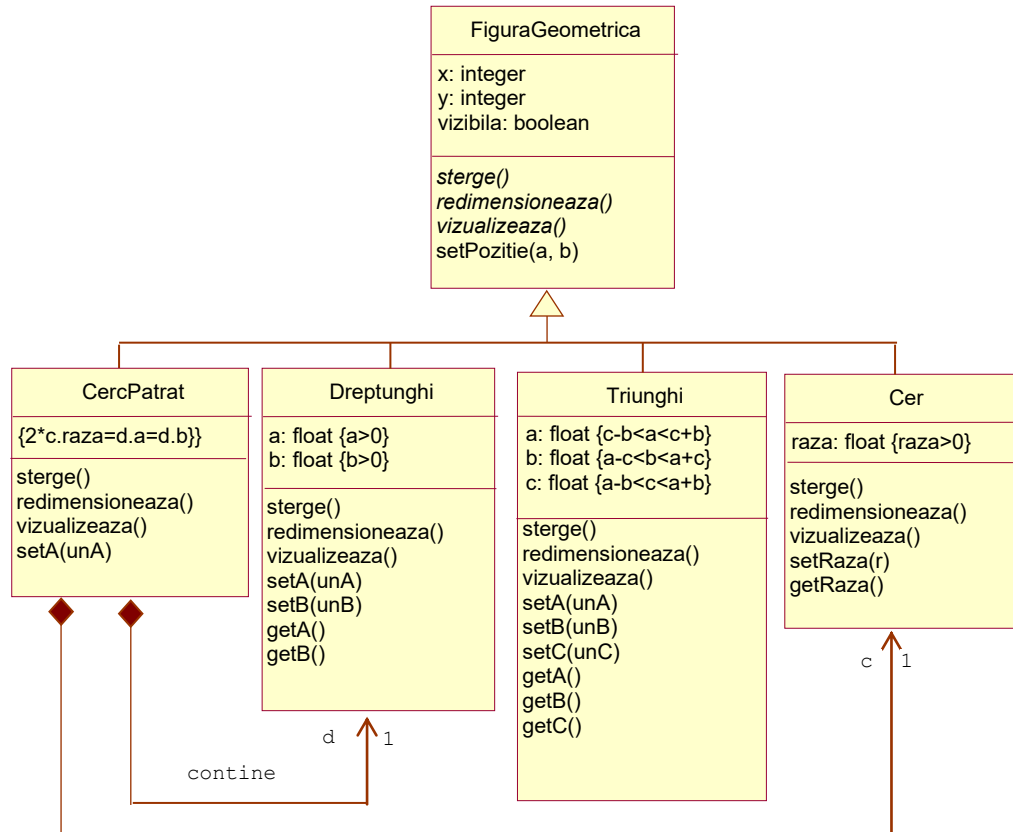


Diagrama 2. Includerea unei noi clase CercPatrat

```

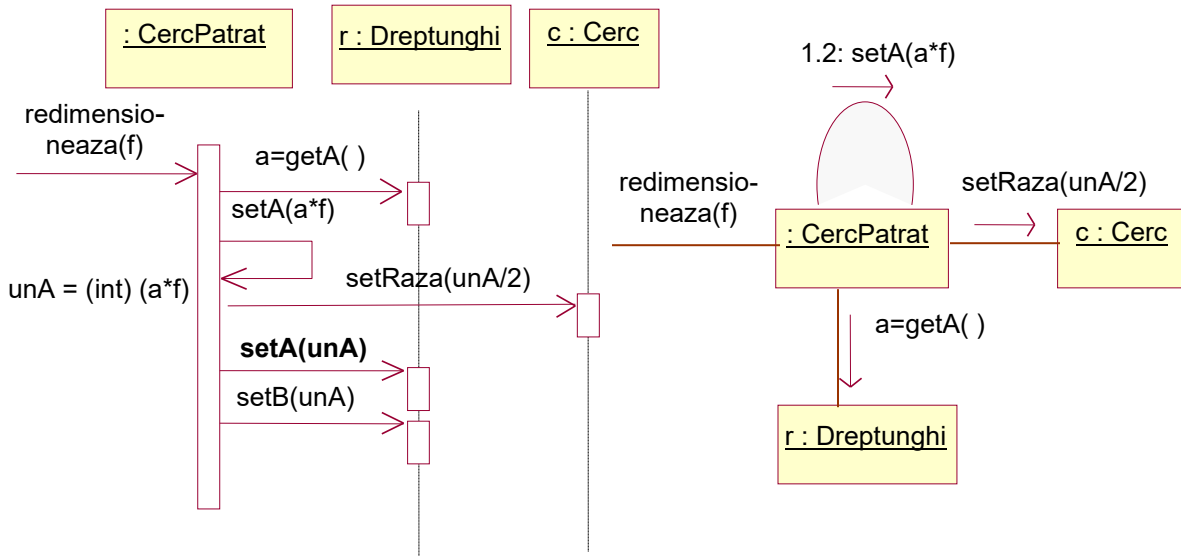
class CercPatrat extends FiguraGeometrica{
    Cerc c;
    Dreptunghi r;
    public void redimensioneaza(float f) {
        int a;
        a=r.getA();
        setA((int) (a*f));
    }

    public void vizualizeaza(){
        c.vizualizeaza();
        r.vizualizeaza();
    }
    public void sterge(){
        c.sterge();
        r.sterge();
    }
    public void setA(int a){
        c.setRaza(a/2);
        r.setA(a);
        r.setB(a);
    }
}
    
```

Programul 1. Clasa CercPatrat

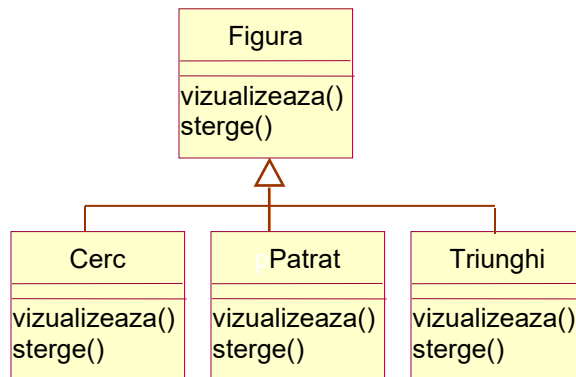
Interacțiuni între obiecte (UML)

Cele două diagrame de mai jos descriu colaborarea unui obiect CercPatrat cu obiectele Dreptunghi si Cerc r si c. Ele sunt diagrame de interactiuni UML.



Polimorfism în Java

Fie cele patru clase cu relatiile dintre ele prezentate in urmatoarea diagrama UML.



Codul Java care implementeaza aceasta diagrama este prezentat mai jos. In cod s-a adaugat o clasa `Figuri` care verifica functionarea polimorfismului in Java. Se observa ca se genereaza in maniera random 9 obiecte `Cerc`, `Dreptunghi` si `Triunghi`, promovate ca obiecte `Figura`, ceea ce este perfect corect deoarece obiectele unei subclase sunt si obiecte ale superclasei clasei respective. Lucrand cu cele 9 obiecte `Figura`, chemarea metodelor `vizualizeaza()` si `sterge()`, produce nu un efect nul cum sunt implementate in clasa `Figura` ci efectul metodelor specializate din clasele `Cerc`, `Dreptunghi` si `Triunghi`. Acesta este polimorfismul!

Intr-un limbaj de programare pe obiecte ce utilizeaza mostenirea claselor, apelul unei variabile a superclasei cu o metoda a acesteia lanseaza o metoda care este decisa la momentul executiei pe baza obiectului memorat in variabila si nu pe baza tipului variabilei utilizate in apel.

```
class Figura{ // clasa de bază
    void vizualizeaza(){}
    void sterge(){}
}
class Cerc extends Figura{ //clasa derivata
    void vizualizeaza(){ // redefinire
        System.out.println("Cerc. vizualizeaza()");
    }
}
```

```

    void sterge() { // redefinire
        System.out.println("Cerc.sterge()");
    }
}
class Patrat extends Figura { // derivata
    void vizualizeaza() { //redefinire
        System.out.println("Patrat. vizualizeaza()");
    }
    void sterge() { // redefinire
        System.out.println("Patrat.sterge()");
    }
}
class Triunghi extends Figura { // derivata
    void vizualizeaza() { // redefinire
        System.out.println("Triunghi. vizualizeaza()");
    }
    void sterge() { // redefinire
        System.out.println("Triunghi.sterge()");
    }
}
public class Figuri {
    public static Figura genereazaFigura() {
        switch((int) (Math.random() * 3)) {
            default:
                case 0: return new Cerc();
                case 1: return new Patrat();
                case 2: return new Triunghi();
        }
    }
    public static void main(String[] args) {
        Figura[] s = new Figura[9];
        for(int i = 0; i < s.length; i++)
            s[i] = genereazaFigura();
        for(int i = 0; i < s.length; i++)
            s[i].vizualizeaza();
    }
}
Iesire:
Cerc.vizualizeaza()
Triunghi.vizualizeaza()
Cerc.vizualizeaza()
Cerc.vizualizeaza()
Cerc.vizualizeaza()
Patrat.vizualizeaza()
Triunghi.vizualizeaza()
Patrat.vizualizeaza()
Patrat.vizualizeaza()

```

Overriding in Java

<pre>class Parinte{ int y=1; int oMetoda() { return y; } } class Fiu extends Parinte{ int y; int oMetoda(){ y=super.y+1; return super.oMetoda()+y; } }</pre>	<pre>class Fruct { boolean coajaComestibila=false; } class Mar extends Fruct { boolean coajaComestibila=true; public static void main(String args[]) { Mar m = new Mar(); Fruct f = m; System.out.println(f.coajaComestibila); System.out.println(m.coajaComestibila); System.out.println(((Fruct)m).coajaComestibila); } } Desire: false true false</pre>
---	---

Din exemplele de mai sus se constata ca variabilele si metodele suprascrise nu dispar din obiectele subclaselor!

Constructori și moștenirea

```
public class Mamifer{
    String numme;
    int varsta;
    char sex;

    public Mamifer(){
        varsta = 0;
    }

    public int getVarsta(){
        return varsta;
    }

    public void schimbaNume(String n){
        nume=n;
    }
}
```

```
class Persoana extends Mamifer{
    String prenume;
    public Persoana() {
        //este implicit Mamifer()
    }
    public Persoana(String n, String p){
        //este implicit super();
        nume=n;
        prenume=p;
    }
    . . . . .
}
```

Clase abstracte

Metoda abstractă = O metodă care nu definește algoritmul de implementare.

Exemplu:

```
abstract void metoda();
```

```
abstract class Message {
    public Message(String from){sender = from;}
    public abstract void play();
    public String getSender() { return sender; }
    private String sender;
}

class TextMessage extends Message {
    public TextMessage(String from, String t) {
        super(from); text = t;
    }
    public void play() {System.out.println(text);}
    private String text;
}

class VoiceMessage extends Message {
    public VoiceMessage(String from, String f) {
        super(from); filename = f;
    }
    public void play() {
        AudioPlayer ap = AudioPlayer.player;
        try {
            AudioStream as = new AudioStream(
                new FileInputStream(filename));
            ap.start(as);
        } catch(IOException e) {}
    }
    private String filename;
}
```

```
public abstract int read() throws IOException;
```

Clasa abstracta = O clasa cu cel puțin o metodă abstractă.

Exemplu: `abstract class ClasaAbstracta{ .. }`

Dezvoltarea unei aplicatii pe obiecte

Cont bancar

Se considera definirea unei clase care descrie comportamentul unui cont bancar.

1. Definirea numelui clasei: ContBancar.
2. Determinarea proprietatilor clasei, care caracterizeaza obiectele clasei. Se insereaza variabila sold de tip long.
3. Intelegerea modului in care se comporta obiectele clasei. Care sunt operatiile care se pot face cu un cont bancar?

Responsabilitatea clasei ContBancar:

Depune bani ⇒ depune

Scoate bani ⇒ scoate

Cere soldul curent ⇒ getSold

4. Definirea modurilor in care vor fi create obiecte ContBancar.

Un cont bancar poate fi creat vid ⇒ `ContBancar()`

Un cont bancar poate fi creat initializand soldul ⇒ `ContBancar(long s)`

ContBancar
- sold: long
+ depune(suma:long)
+ scoate(suma: long)
+ getSold():long
+ ContBancar()
+ ContBancar(depozit: long)

Clasa ContBancar

Responsabilitatea clasei ContBancar:

- Depune bani
- Scoate bani
- Cere soldul curent

```
public class ContBancar{
    public void depune(long suma){
        . . . . .
    }
    public void scoate(long suma){
        . . . . .
    }
    public long getSold(){
        . . . . .
    }
}
```

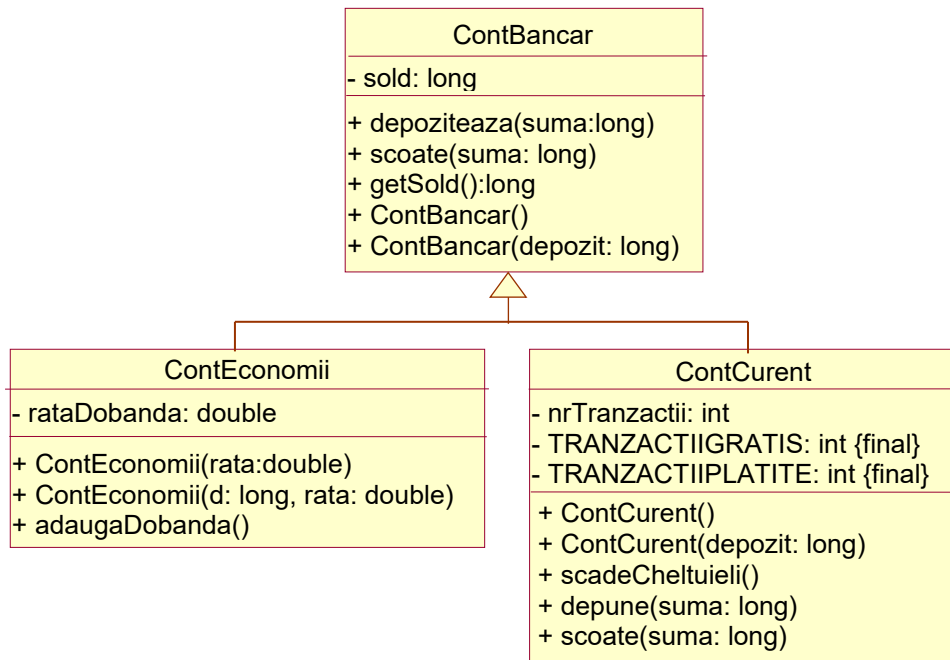
Variabila instanta: `private long sold;`

Constructori:

- Construiește un cont cu un depozit vid
- Construiește un cont cu un depozit specificat

```
public class ContBancar{
    public ContBancar() {
        sold=0;
    }
    public ContBancar(long depozit) {
        sold = depozit;
    }
    . . . . .
}
```

Exercitiu(I). Sa se implementeze clasa ContBancar prezentata anterior. Considerati diagrama de clase:

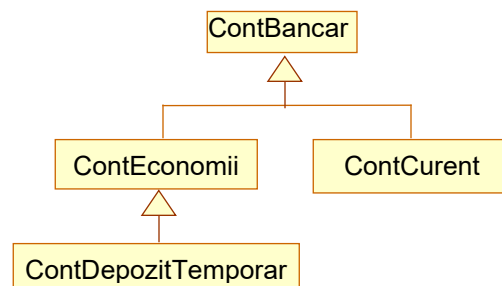


Proiectati si implementati clasele ContEconomii (rata fixa de dobanda pe depozite) si ContCurent (ofera un numar de tranzactii gratuite; daca se termina aceste tranzactii, fiecare tranzactie se plateste cu o suma fixa la sfarsitul anului).

Sa se scrie o clasa TestContBancar care creaza doua conturi de economii si un cont curent, sa se execute 2-3 tranzactii pentru fiecare cont si sa vizualizeze situatia, la inceput si dupa efectuarea tranzactiilor.

Exercitiu (II). Sa se imbogateasca ierarhia clasei ContBancar cu o noua clasa ContDepozitTemporar, un tip particular de ContEconomii. In acest cont, titularul se obliga lase banii un anumit numar de luni, probabil in schimbul unei dobanzi mai mari. Odata ce s-a indeplinit termenul de scadenta a contului, nu exista nici o penalizare pentru il prlievo. In schimb, daca suma este scoasa inainte de scadenta, ea va fi eliberata fara dobanda.

Sa se modifice clasa TestContBancar, modificand un cont de economii intr-un cont de perioada limitata (utilizati polimorfismul metodelor).



7. GESTIUNEA EXCEPȚIILOR

O **excepție** este o condiție exceptională care întrerupe execuția unui program și care poate fi:

- condiție de eroare provocată de erori interne sau lipsa resurselor mediului de execuție Java,
- situație particulară apărută în timpul execuției programului care poate fi de unul din următoarele tipuri:
 - excepții necontrolate cauzate de erori de programare: cast greșit, cast care depășește limita unui array, referință la un obiect null,
 - excepții controlate: erori de intrare/ieșire precum continuarea operației de citire după ce s-a ajuns la sfârșitul fișierului, încercarea de a deschide un URL greșit.

Când apare o excepție este important să captăm cauza ce a dus la apariția ei, împreună cu alte informații, cum ar fi valoarea variabilelor folosite, ce vor ajuta la tratarea excepției. De aceea, toate excepțiile în Java sunt reprezentate de obiecte ale subclaselor clasei `Throwable` din pachetul `java.lang`:

```
public class Throwable{
    public Throwable();
    public Throwable(String mesaj);
    public String getMessage();
    public String toString();
    public void printStackTrace();
    public void printStackTrace(java.io.PrintStream ps);
    public void printStackTrace(java.io.PrintWriter pw);
    public Throwable fillInStackTrace();
}
```

După cum observăm, clasa `Throwable` implementează mai multe metode printre care și metoda `getMessage()` care returnează un șir care descrie situația care a cauzat excepția.

De exemplu, clasa `RuntimeException` conține tipuri de excepții lansate automat de JVM. Dacă nu sunt tratate de programator, dacă apare o astfel de excepție se va apela metoda `printStackTrace()` a clasei `Throwable`. De exemplu, executarea următorului program

```
public class Exceptie{
    static void metoda1(int[] a){
        metoda2(a);
    }
    static void metoda2(int[] b){
        System.out.println(b[0]);
    }
    public static void main(String[] args){
        metoda1(null);
    }
}
```

determină afișarea mesajului:

```
java.lang.NullPointerException
    at Exceptie.metoda2(Exceptie.java:6)
    at Exceptie.metoda1(Exceptie.java:3)
    at Exceptie.main(Exceptie.java:9)
```

Mesajul afișat conține următoarele informații:

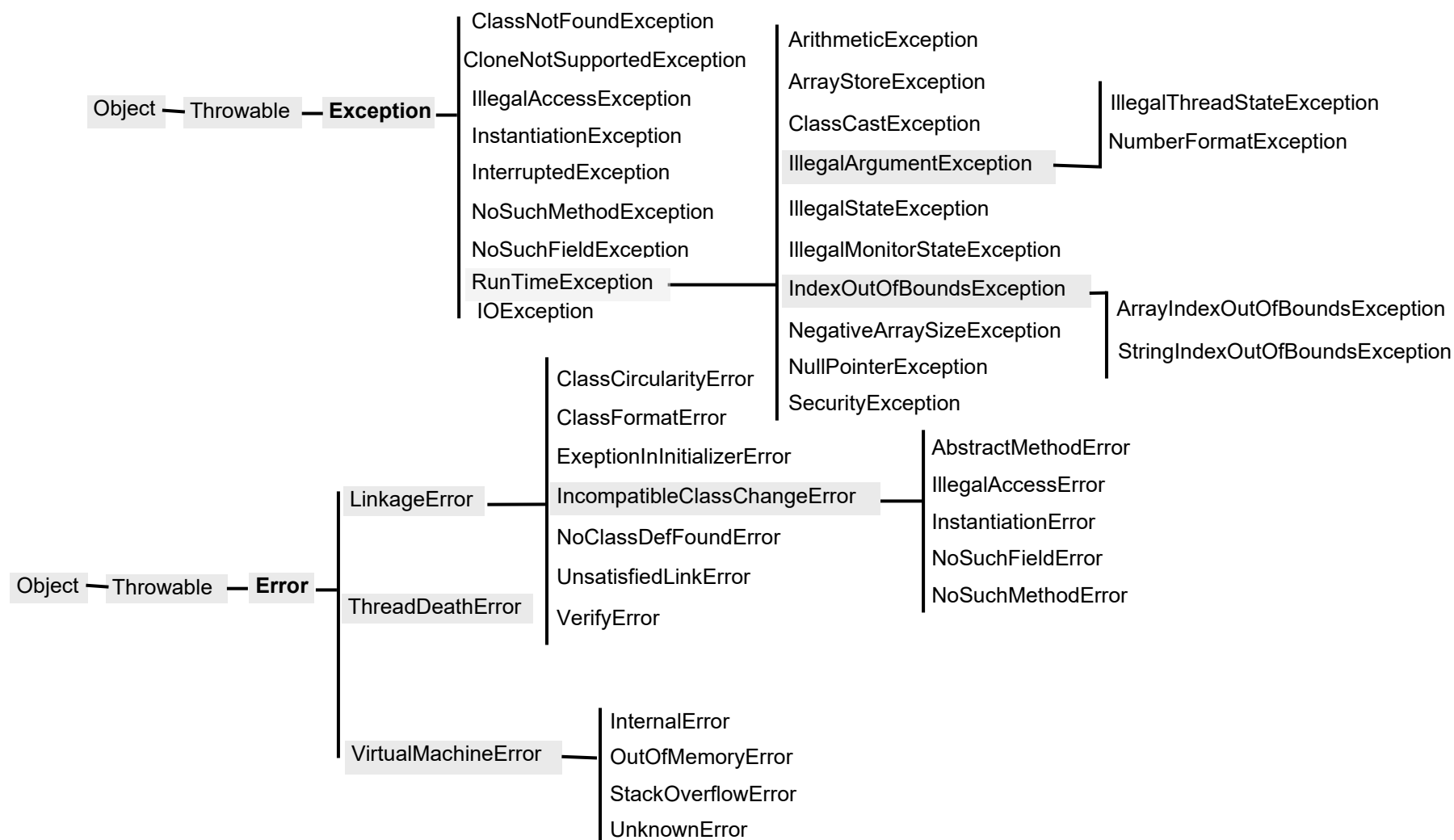
- tipul de excepție aruncat,
- stiva apelurilor metodelor, fiecare cu clasa și unitatea de compilare de care aparține, împreună cu numărul liniei pe care se găsește declarația metodei. Metoda din vârful stivei este cea care a aruncat excepția respectivă.

Toate excepțiile ce pot fi generate sunt împărțite în două clase: `Exception` și `Error`, ambele derivate din clasa `Throwable`.

Subclasele lui `Exception`, cu excepția lui `RuntimeException` și a subclaselor sale, indică situații particulare care nu sunt erori. `Error`, `RuntimeException` și subclasele lor indică condiții care în general nu sunt corecte și care de obicei generează terminarea unui program. Toate pachetele incluse în JDK generează diferite tipuri de excepții. O parte din acestea sunt prezentate în figurile de mai jos și unele din acestea sunt explicate în următorul tabel:

Excepție	Cauzată de
ArithmeticException	Erori matematice cum ar fi împărțirea la zero.
ArrayIndexOutOfBoundsException	Indicele unui array este în afara array-ului.
ArrayStoreException	Se memorează într-un array o valoare de tip diferit de tipul de date al array-ului.
ClassNotFoundException	Se încearcă folosirea unei clase pe care compilatorul nu o cunoaște. Clasa respectivă trebuie să fie importată sau scrisă de către programator.
FileNotFoundException	Se accesează un fișier care nu există în locația specificată de program.
IOException	Erori generale de I/O, cum ar fi incapacitatea de citi dintr-un fișier.
NullPointerException	Referirea unui obiect null.
NumberFormatException	O conversie eșuată între șiruri și numere.
OutOfMemoryException	Memorie insuficientă creării unui obiect nou.
SecurityException	Un applet încearcă să realizeze o acțiune care nu este permisă de browser.
StackOverflowException	Depășirea stivei de execuție.
StringIndexOutOfBoundsException	Programul încearcă să acceseze un caracter de pe o poziție care nu există în șir.

Excepții și erori



Cum utilizăm excepțiile? Există mai multe alternative pentru programator:

- corectarea problemei și apelarea metodei care a cauzat excepția;
- se încearcă repararea unei greșeli și se continuă fără să se reapeleze metoda;
- elaborarea unui rezultat alternativ raportat la cel produs în mod normal de metodă;
- se face tot ce se poate în contextul curent și se relansează aceeași excepție în contextul superior;
- se face tot ce se poate în contextul curent și se relansează o excepție diferită în contextul superior;
- se termină programul.

Tratarea excepțiilor prezintă următoarele avantaje:

- simplifică dacă schema de elaborare a excepțiilor aleasă complică lucrurile,
- face mai fiabile programul și librăriile utilizate. Aceasta e desigur o investiție bună pe termen scurt (pentru debugging) și pe termen lung (pentru robustețea programului).

Mecanismul de gestiune a excepțiilor furnizează reguli care impun cine este capabil să genereze excepții (throws) și când (throw), cine va încerca excepțiile (try..catch) și ce trebuie să facă pentru a împiedica apariția unei excepții.

Interceptarea și tratarea excepțiilor

Interceptarea și tratarea excepțiilor se realizează de către gestorii de excepții (exception handler). Când în interiorul unei metode va fi lansată o excepție sau apelul unei metode lansează o excepție, în mod normal, controlul abandonează metoda. Dacă nu se dorește ca metoda respectivă să fie abandonată, instrucțiunile ce pot genera cel puțin o excepție se includ într-un bloc try care “încearcă” să le execute. Blocul crează un mediu (spațiu de nume) în interiorul metodei. Dacă apare o excepție ca rezultat al executării uneia din instrucțiunile blocului, ea poate fi “prinsă” (catch) de gestorul excepției respective.

Gestorii excepțiilor pentru un bloc try sunt introduși cu cuvântul cheie catch și urmează blocul try. Fiecare gestor de excepții este ca o metodă care primește un unic argument care are tipul unei excepții și va fi executată în cazul în care apare o excepție de tipul respectiv. Argumentul metodei poate să nu fie utilizat în codul gestorului, iar alegerea gestorului este în întregime bazată pe tipul excepției.

Pentru același bloc try putem avea mai mulți gestori. În acest caz este importantă ordinea gestorilor, deoarece în cazul apariției unei excepții, se verifică tipul excepției cu fiecare gestor în ordinea în care apar după blocul try. Se va executa primul gestor care verifică excepția, iar restul se ignoră. În exemplul următor, acest lucru înseamnă că al doilea gestor niciodată nu va trata nici o excepție, deoarece primul este mai general și va trata și excepțiile de intrare/ieșire (deoarece `IOException` este o subclasă a lui `Exception`). Această situație va fi detectată de compilatorul Java și va raporta o eroare.

```
void metoda1() {  
    try{  
        //cod care poate produce o exceptie  
    }catch(IOException ioe){  
        //cod care gestioneaza exceptia ioe  
    }  
    catch(Exception e){  
        //cod care gestioneaza exceptia e  
    }  
}
```

Un bloc try poate fi urmat eventual și de un bloc declarat cu finally. Un bloc finally conține instrucțiuni ce vor fi executate întotdeauna, chiar dacă apare o excepție și furnizează un spațiu util de a scrie cod care să fie executat indiferent ce se întâmplă.

De exemplu, următoarea metodă gestionează interceptarea unei excepții și a unei erori și indiferent dacă acestea apar sau nu, se va executa codul din blocul finally:

```
void metoda2() {
    try{
        //cod care poate produce o exceptie
    } catch(Exception e){
        //cod care gestioneaza exceptia e
    }
    catch(Error er){
        //cod care gestioneaza eroarea er
    }
    finally{
        //cod ce va fi executat indiferent daca apar sau nu exceptia e sau eroarea er
    }
}
```

Gestorul poate prinde o excepție de tip:

- `Exception` pentru orice excepție din bibliotecile Java;
- specializat pentru un pachet Java (de exemplu `IOException`);
- definit de programator.

Declarații `throws`

Dacă o metodă nu tratează toate excepțiile ce pot apare atunci când este executat corpul său, atunci ea trebuie să declare în header-ul său toate tipurile de excepții care pot să apară și nu le tratează. Pentru aceasta se inserează o declarație `throws` după parametrii metodei și înainte de corpul său:

```
modificatori numeTip numeMetoda(listaParametri) throws listaNumeTip{
    //lista de instructiuni
}
```

unde *listaNumeTip::=numeTip [, numeTip]**

Excepțiile declarate cu `throws` se vor propaga metodei superioare în stivă, aceleia care a apelat-o. În acest caz, metoda apelantă trebuie să se afle în una din următoarele situații:

- metoda este apelată dintr-un bloc `try` care are gestori capabili să prindă toate tipurile de excepții ce pot fi aruncate de metoda apelată.
- metoda este apelată dintr-un bloc `try` care are gestori capabili să prindă numai unele din tipurile de excepții ce pot fi aruncate de metoda apelată. Pentru celelalte tipuri de excepții, metoda apelantă trebuie să le declare cu `throws` și ele vor fi propagate mai departe.
- metoda este apelată din afara unui bloc `try` și atunci toate tipurile de excepții ce pot fi aruncate trebuie să apară în declarația `throw` a metodei apelante.

Notă: Excepțiile de tip `Error`, `RuntimeException` și a subclaselor sale nu trebuie să fie declarate cu `throws`.

Următoarea metodă returnează un caracter citit de la tastatură:

```
public char citeșteCaracter() throws IOException{
    return (char)System.in.read();
}
```

Lansarea excepțiilor

În timpul proiectării programului sau în timpul depanării, dacă se consideră necesar ca o metodă să lanseze o excepție, se execută următorii pași:

- căutarea clasei de excepții corespunzătoare;
- dacă metoda nu tratează excepția apărută, se declară că apelul metodei va lansa o excepție (cu ajutorul lui `throws` și clasa de excepție găsită la pasul anterior);
- crearea unui obiect al acelei clase;
- lansarea obiectului creat imediat după punctul în care se verifică excepția, folosind instrucțiunea `throw`;

Exemplu. Metoda următoare citește caracter cu caracter o linie de la tastatură pe care o returnează metodei apelante care trebuie să trateze excepția de tip `IOException` sau s-o arunce mai departe metodei apelante:

```
import java.io.*;
public class Linie{
    StringBuffer sb=new StringBuffer();

    void readLine() throws IOException{
        char c;
        while(true){
            c=(char)System.in.read();
            if (c==13) throw new IOException("Caracterul citit este CR");
            sb.append(c);
        }
    }

    public static void main(String[] args){
        Linie f=new Linie();
        try{
            f.readLine();
        }catch(IOException e){System.out.println(e.getMessage());}
        System.out.println(f.sb.toString());
    }
}
```

Procesul de reportare a unei excepții în sus continuă până când excepția va fi interceptată de o metodă activă care se angajează să o rezolve sau până când ajunge în vârful stivei și aplicația se termină.

Următorul exemplu arată cum se poate citi conținutul unui fișier și să detectăm sfârșitul fișierului aruncându-se o excepție de tip `EOFException`:

```
String readData(BufferedReader in){
    StringBuffer sb=new StringBuffer();
    String s;
    try{
        while(true){
            s=in.readLine();
            if (s==null){
                in.close();
                throw new EOFException("Am ajuns la sfarsitul fisierului");
            }
            sb.append(s);
        }
    }catch(IOException ioe){
        System.out.println("Excepție de citire: "+ioe.getMessage());
    }
    finally{
        return sb.toString();
    }
}
```

Observație: Cum `EOFException` este o subclasă a lui `IOException`, metoda `readData()` tratează excepția apărută, deci ea nu va fi aruncată metodei apelante.

Important: lansând o excepție, metoda nu o returnează apelului.

Se pot crea clase de excepții noi. Trebuie să avem o clasă care extinde clasa `Exception` sau un alt tip apropiat ca semantică cu excepția noastră, în care să redefinim constructorii. De exemplu, ne definim o clasă excepție numită `FileFormatException`:

```
public class FileFormatException extends IOException{
    public FileFormatException() {}
    public FileFormatException(String mesaj){
        super(mesaj);
    }
}
```

Atunci, metoda `readData()` de mai sus ar putea fi rescrisă astfel:

```
String readData(BufferedReader in){
    StringBuffer sb=new StringBuffer();
    String s;
    try{
        while(true){
            s=in.readLine();
            if (s==null){
                in.close();
                throw new FileFormatException("Am ajuns la sfarsitul fisierului");
            }
            sb.append(s);
        }
    }catch(IOException ioe){
        System.out.println("Exceptie de citire: "+ioe.getMessage());
    }
    finally{
        return sb.toString();
    }
}
```

O excepție poate fi rearuncată și dintr-un bloc `catch`. Atunci putem avea mai multe metode din lanțul de metode active care să trateze excepția, poate fiecare din acestea îndeplinindu-si partea sa din restaurarea stării programului astfel încât acesta din urmă să-și poată continua execuția după ce excepția a fost tratată.

Probleme propuse

1. Să se verifice dacă un șir de la tastatură citit de la tastatură și verifică dacă este un număr întreg sau real, sau este un șir alfanumeric.
2. Să se scrie clasa `OutOfRangeException` care este un tip de excepție utilizator. Acest tip de excepție va fi aruncat de o metodă a unei alte clase care verifică dacă o valoare citită de la tastatură se află într-un interval ale cărui capete sunt specificate de program.

8. INPUT/OUTPUT ÎN JAVA

Pachetul *java.io*

Furnizează suport pentru citirea și scrierea datelor de la și la dispozitive diferite.

Conține 4 categorii de clase:

- clase de fluxuri de intrare și clase de citire
- clase de fluxuri de ieșire și clase de scriere
- clase de gestiune a fișierelor
- clasa `StreamTokenizer`

1. **Clase de fluxuri de octeți:** `BufferedInputStream`, `DataInputStream`, `FileInputStream` și `StringBufferInputStream`. Derivate din clasa abstractă `InputStream`.

Clase de citire a caracterelor Unicode: `Reader`, `BufferedReader`, `FileReader` și `StringReader` sunt derivate din clasa abstractă `Reader`.

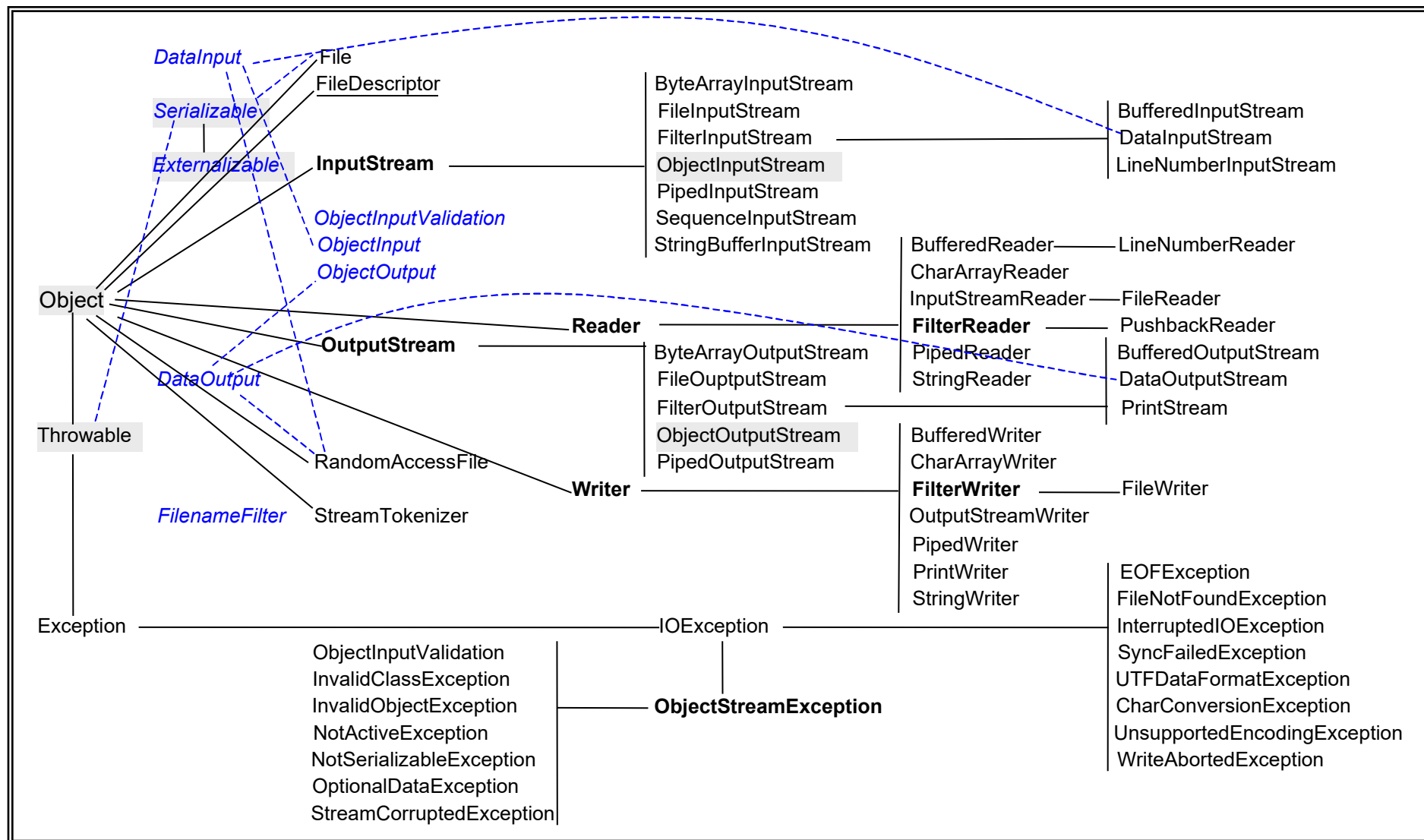
2. **Clase de fluxuri de ieșire:** `OutputStream`, `FileOutputStream`, `BufferedOutputStream`, `PrintStream` și `StringBufferOutputStream`. Sunt derivate din clasa `OutputStream`.

Clase de scriere a caracterelor în Unicode: `Writer`, `BufferedWriter`, `FileWriter` și `PrintWriter`. Sunt derivate de clasa abstractă `Writer`.

3. Claselor `FileInputStream` și `FileOutputStream` se adaugă alte două clase destinate gestiunii fișierelor: `File` și `RandomAccessFile`.

4. Clasa `StreamTokenizer` permite convertirea unui flux de intrare a datelor într-un flux de token-i și conține metode pentru definirea sintaxei token-ilor.

Ierarhia claselor I/O



Clasa InputStream

InputStream = clasă abstractă din care derivează toate clasele fluxului de intrare.

Metodele clasei InputStream sunt prezentate în următorul tabel:

Semnătura metodei	Semnificație –Valoarea returnată
<code>abstract int read() throws IOException</code>	returnează un octet citit de pe fluxul de intrare. Dacă se detectează sfârșitul fluxului atunci metoda întoarce -1.
<code>int read(byte[] b) throws IOException</code>	citește octeți de pe fluxul de intrare și îi memorează în array-ul b. Dacă b este mai mic decât numărul de octeți citați, se lansează o excepție. Metoda returnează numărul de octeți citați efectiv.
<code>int read(byte[] b, int poz, int nr) throws IOException</code>	citește maxim nr octeți de pe fluxul de intrare și îi memorează în array-ul b, de la poziția poz. Dacă b este mai mic decât numărul de octeți citați, se lansează o excepție. Metoda returnează numărul de octeți citați efectiv.
<code>long skip(long n) throws IOException</code>	ignoră maxim n octeți din fluxul de intrare și întoarce numărul de octeți cu care s-a avansat în flux.
<code>int available() throws IOException</code>	returnează numărul de octeți disponibili pentru citire fără blocare.
<code>void mark(int poz)</code>	marchează poziția poz în flux pentru ca ulterior să se poată reveni la ea. Nu toate fluxurile de intrare oferă această operație.
<code>void reset() throws IOException</code>	repoziționează fluxul la ultima poziție marcată.
<code>boolean markSupported()</code>	întoarce true dacă fluxul suportă operația de marcare și revenire la poziția marcată.
<code>void close() throws IOException</code>	închide fluxul de intrare și eliberează resursa alocată.

Clasa Reader

Aceeași structură are clasa abstractă Reader, specializată în citirea caracterelor. De aceea, sintaxa metodelor `read()` a fost modificată astfel:

Semnătura metodei	Semnificație –Valoarea returnată
<code>int read() throws IOException</code>	citește un caracter de pe fluxul de intrare și returnează codul său Unicode. Dacă se detectează sfârșitul fluxului atunci metoda întoarce -1.
<code>int read(char[] b) throws IOException</code>	citește caractere de pe fluxul de intrare și îi memorează în array-ul b. Dacă b este mai mic decât numărul de caractere citite, se lansează o excepție. Metoda returnează numărul de caractere citite efectiv.
<code>abstract int read(char[] b, int poz, int nr) throws IOException</code>	citește maxim nr caractere de pe fluxul de intrare și îi memorează în array-ul b, de la poziția poz. Dacă b este mai mic decât numărul de caractere citite, se lansează o excepție. Metoda returnează numărul de caractere citite efectiv.

`System.in` = variabilă statică a clasei `System` care memorează o instanță a clasei `InputStream`. De obicei, această instanță reprezintă fluxul standard de intrare (tastatura), dar poate fi setată cu `System.setIn(InputStream)` astfel:

```
FileInputStream fi = new FileInputStream("input.txt");
System.setIn(fi);
```

Crearea unui flux de date de intrare

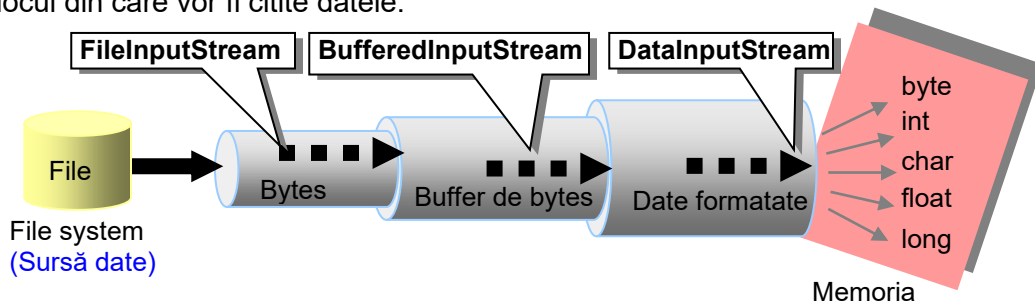
Crearea unui flux de date de intrare respectă modelul pipeline Unix: face parte dintr-o sursă de date cu interfața `InputStream` și se crează succesiv cu filtri ai aceleași interfețe.

Caracteristicile citirii:

- este un flux continuu.
- când nu mai sunt date disponibile, instanța clasei se blochează în așteptarea ca datele să devină disponibile.

Pentru a citi date în Java procedura este întotdeauna aceeași:

1. Se crează o instanță a unei clase de flux de intrare.
2. Se indică locul din care vor fi citite datele.



Exemplu. Metoda următoare citește un caracter de la tastatură și-l returnează:

```
public char citeste(){
    char c='0';
    try{
        c=(char)System.in.read();
        System.in.skip(2);//nu se citește caracterul generat de apăsarea lui Enter
    }catch(IOException e){System.out.println(e.getMessage());}
    return c;
}
```

Clase de flux de octeți de intrare

1. `DataInputStream (InputStream in)`:
 - utilizată pentru citirea datelor primitive (`int`, `char`, `long` etc.);
 - moștenește atât interfața `DataInput` cât și clasa abstractă `FilterInputStream`;
 - la metodele furnizate de `InputStream` se adaugă metodele interfeței `DataInput` `readxxx()` (`xxx` = `Boolean`, `Byte`, `Short`, `UnsignedShort`, `Char`, `Int`, `Long`, `Float` sau `Double`) care citesc un număr de octeți din fluxul de intrare necesar pentru a obține o valoare de tip primitiv specificat în numele metodei. De exemplu, metoda `readInt()` citește patru octeți din fluxul de intrare și întoarce o valoare de tip `int`.
2. `BufferedInputStream (InputStream in)`:
 - furnizează un flux de intrare buffer-izat permițând astfel o citire mult mai rapidă;
 - include numai metodele furnizate de `InputStream`;
 - are unele variabile membru: `byte buf[]`, `int count`, `int pos`, `int markpos` și `int marklimit`, utile dacă se extinde clasa `BufferedStream`.
3. `FileInputStream (String nume)`:
 - execută operații de intrare simple asupra fișierelor (pentru operații mai avansate se utilizează `RandomAccessFile`).
 - construiește fluxul și cu un obiect `File` sau `FileDescriptor`.
 - funcționează ca un `InputStream`, dar operează asupra unui fișier.
4. `PushBackInputStream (InputStream in)`

- construiește un flux cu un caracter lookahead.
- are o metodă unread(int ch).
- funcționează ca InputStream.

Clasa abstractă Reader are subclase asemănătoare cu cele ale clasei abstracte InputStream: BufferedReader, FileReader, StringReader, PushBackReader. BufferedReader are o metoda String readLine(). În plus, există clasa InputStreamReader care face legătura dintre o clasa de tip InputStream și un Reader.

Următorul exemplu arată mai multe modalități de a face o citire folosind clasele derivate din InputStream și Reader:

```
import java.io.*;
public class FluxDeIntrareIesire{
    public static void main(String[] args) {
        try {
            //1. Citirea unei linii de la tastatura
            BufferedReader in1 = new BufferedReader( new InputStreamReader(System.in));
            String s=in1.readLine();
            in1.close();
            //2. Citirea dintr-un fisier
            BufferedReader in2 = new BufferedReader( new FileReader("intrare.txt"));
            String s1, s2 = new String();
            while((s1 = in2.readLine()) != null) s2 += s1 + "\n";
            in2.close();
            //3. Citirea din memorie
            StringReader in3 = new StringReader(s2);
            int c;
            while((c = in3.read() ) != -1) System.out.print((char)c);
            in3.close();
            //4. Citirea din memorie formatata
            DataInputStream in4 = new DataInputStream(
                new FileInputStream("intrare.txt"));
            while(true) System.out.print((char)in4.readByte());
        } catch (EOFException e){
            System.out.println("Am ajuns la sfarsitul fisierului");
        } catch (FileNotFoundException e){
            System.out.println("Fisier inexistent");
        } catch (IOException e){
            System.out.println("Exceptie IO");
        }
    }
}
```

Clasa OutputStream

OutputStream = clasă abstractă din care derivează toate clasele fluxului de ieșire.

Metodele clasei OutputStream sunt prezentate în următorul tabel:

Semnătura metodei	Semnificație –Valoarea returnată
abstract void write(int n) throws IOException	scrie valoarea lui n pe fluxul de ieșire.
void write(byte[] b) throws IOException	scrie pe fluxul de ieșire octeții din array-ul b
void write(byte[] b, int poz, int nr) throws IOException	scrie nr octeți pe fluxul de ieșire începând de la poziția poz a array-ului b.
void flush() throws IOException	golește fluxul de ieșire
void close() throws IOException	închide fluxul de intrare și eliberează resursa alocată.

Clasa Writer

Clasa `Writer` specializată în scrierea fluxurilor de caractere are o ierarhie similară cu ierarhia lui `OutputStream`. Semnătura metodelor `write()` se modifică și apar două forme noi ale acestei metode:

Semnătura metodei	Semnificație –Valoarea returnată
<code>void write(int n) throws IOException</code>	scrie valoarea lui n pe fluxul de ieșire.
<code>void write(char[] b) throws IOException</code>	scrie pe fluxul de ieșire caracterele din array-ul b
<code>abstract void write(char[] b, int poz, int nr) throws IOException</code>	scrie nr caractere pe fluxul de ieșire începând de la poziția poz a array-ului b.
<code>public void write(String str) throws IOException</code>	scrie șirul str pe fluxul de ieșire.
<code>public void write(String str, int poz, int nr) throws IOException</code>	scrie pe fluxul de ieșire subșirul de lungime nr pe care-l extrage de la poziția poz a șirului str.

`System.out` = variabilă statică a clasei `System` care memorează o instanță a clasei `PrintStream` ce reprezintă de obicei ecranul monitorului. Este setată cu `System.setOut(OutputStream):`

```
FileOutputStream fs = new FileOutputStream("log.txt");
```

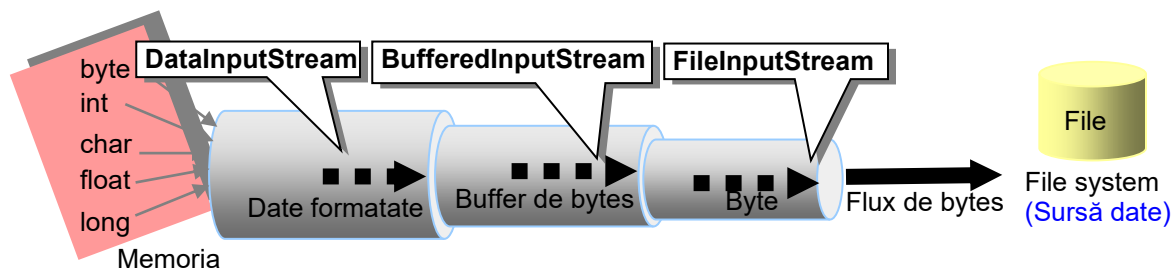
```
System.setOut(new printStream(fs));
```

Caracteristicile scrierii:

- este un flux continuu.
- instanța clasei de flux trimite date cât timp acestea sunt disponibile.

Procedura de scriere a datelor cu o clasă de fluxuri de ieșire este întotdeauna aceeași:

1. Se crează o instanță a clasei fluxului de ieșire specificând fluxul final, la destinație. Instanța se poate crea plecând de la destinație și compunând fluxul din obiecte-flux încuibate, deoarece obiectul obținut are caracteristicile dorite: buffer-izat, în stare să scrie date primitive etc.
2. Se indică locul unde vor fi scrise datele.



Clase de flux de caractere în ieșire

1. `PrintWriter`

- derivează din `Writer`
- afișează date-text (șiruri sau numere).
- metodele sale, altele decât cele furnizate de `Write`, include: `print(tip)` și `println(tip)` unde `tip` ia una din valorile: `Object`, `String`, `boolean`, `char`, `char[]`, `int`, `long`, `float` și `double`.

```
PrintWriter out= PrintWriter(new FileWriter("personale.txt"), true);
```

2. `BufferedWriter`

- derivează din `Writer`,
- furnizează un flux de ieșire buffer-izat,
- constructorul are ca argument un obiect `Writer`,
- nu conține alte metode în afară de cele furnizate de `Writer`,
- are două variabile membru utile: `byte buf[]` și `int count`.

3. `FileWriter`

- execută operații simple de ieșire de caractere pe fișiere (pentru operații mai avansate pe fișiere se utilizează `RandomAccessFile`),
- construiește un obiect numai cu un șir, un obiect `File` sau un `FileDescriptor`,
- funcționează ca `Write`, dar operează pe fișiere.

Scrierea cu clasele *OutputStream* și *Writer*

Primul exemplu din această subsecțiune arată cum se poate determina numărul de linii ale unui fișier și scrierea fiecărei linii citite într-un alt fișier, împreună cu numărul liniei curente.

Observație. Pentru a afla numărul liniei curente vom folosi un cititor buffer-izat ce păstrează numărul liniei citite. Adică vom folosi o instanță a clasei `LineNumberReader`, subclasă a lui `BufferedReader`.

```
try{
    LineNumberReader lnr = new LineNumberReader(new FileReader("intrare.txt"));
    PrintWriter out=new PrintWriter(new BufferedWriter(
                                                new FileWriter("iesire.txt")));

    String s="";
    while((s = lnr.readLine()) != null) out.println("Linia " +lnr.getLineNumber()+
                                                    " "+ s);

    lnr.close();
    out.close();
}catch(EOFException e){System.out.println("Sfarsitul fisierului");}
catch(IOException ioe){System.out.println("Eroare de intrare/iesire");}
```

Următorul exemplu arată cum scriem și citim date formate într-un/dintr-un fișier. Vom folosi instanțe ale claselor `DataOutputStream` și `DataInputStream` pentru a scrie numărul 3.14159 și șirul "Valoarea lui PI":

```
try{
    DataOutputStream out=new DataOutputStream(new BufferedOutputStream(
                                                new FileOutputStream("Data.txt")));

    out.writeDouble(3.14159);
    out.writeBytes("Valoarea lui PI");
    out.close();

    DataInputStream in = new DataInputStream(new BufferedInputStream(
                                                new FileInputStream("Data.txt")));

    BufferedReader inbr = new BufferedReader(new InputStreamReader(in));
    System.out.println(in.readDouble());
    System.out.println(inbr.readLine());
}catch(EOFException e) {System.out.println("Sfarsitul fluxului");}
catch(IOException ioe){System.out.println("Eroare de intrare/iesire");}
```

Clasa următoare furnizează două metode care citesc, respectiv scriu date formate pe linii de lungime fixă:

```
import java.io.*;
public class DataIO{
public static String citesteSirFix(int size, DataInput in)throws IOException{
    StringBuffer buffer=new StringBuffer(size);
    int i=0;
    boolean more=true;
    while(more&& i<size){
        char c=in.readChar();
        i++;
        if(c==0) more=false;
        else buffer.append(c);
    }
    in.skipBytes(2*(size-i));
    return buffer.toString();
}

public static void scrieSirFix(String s, int size, DataOutput out) throws
                                                                    IOException{
    for(int i=0;i<size;i++){
        char c=0;
        if(i<s.length()) c=s.charAt(i);
```

```

        out.writeChar(c);
    }
}

```

Această clasă poate fi folosită de exemplu, pentru a scrie un program care salvează într-un fișier datele (id, nume, salariu) a trei angajați ai unei companii și apoi le afișează în ordinea inversă scrierii lor:

```

import java.io.*;

class Angajat{
    public static final int SIZE_NUME=40;
    public static final int RECORD_SIZE=2*SIZE_NUME+4+8;

    private String nume;
    private int id;
    private double salariu;

    public Angajat(){}

    public Angajat(int id, String nume, double salariu){
        this.id=id;
        this.nume=nume;
        this.salariu=salariu;
    }

    public void writeData(DataOutput out)throws IOException{
        out.writeInt(id);
        DataIO.scrieSirFix(nume, SIZE_NUME,out);
        out.writeDouble(salariu);
    }

    public void readData(DataInput in)throws IOException{
        id=in.readInt();
        nume=DataIO.citesteSirFix(SIZE_NUME,in);
        salariu=in.readDouble();
    }

    public String toString(){
        return id+" "+nume+" "+salariu;
    }
}

public class TestAngajat{
    public static void main(String[] arg){
        Angajat[] a=new Angajat[3];
        a[0]=new Angajat(1, "popescu ion", 45.899);
        a[1]=new Angajat(2, "ionescu pop", 435.877);
        a[2]=new Angajat(2, "marinescu marin", 235.877);

        try{
            DataOutputStream out=new DataOutputStream(new
                                                                    FileOutputStream("angajat.dat"));

            a[0].writeData(out);
            a[1].writeData(out);
            a[2].writeData(out);
            out.close();
        }catch(IOException ioe){}
    }
}

```

```

try{
    RandomAccessFile in=new RandomAccessFile("angajat.dat", "r");
    int n=(int) (in.length()/Angajat.RECORD_SIZE);
    System.out.println("n="+n);
    Angajat[] an=new Angajat[n];

    for(int i=n-1; i>=0;i--){
        an[i]=new Angajat();
        in.seek(i*Angajat.RECORD_SIZE);
        an[i].readData(in);
        System.out.println(an[i]);
    }
} catch (IOException e){System.out.println(e.getMessage());}
}

```

Clasa File

Clasa `File` prezintă o vedere abstractă a numelor fișierelor și a directoarelor, precum și locațiilor unde se găsesc acestea într-un sistem de fișiere. Locația unui fișier reprezintă calea în sistemul de fișiere unde este memorat acesta. Calea poate fi absolută sau relativă la directorul în care se găsesc class-urile programului. Separatorul unei căi depinde de sistemul de operare pe care rulează programul și anume:

- "\" pentru sistemele Windows
- "/" pentru sistemele Unix.

Acesta este memorat în constantele `String separator` și `char separatorChar`, văzut ca un `String`, respectiv ca un caracter.

Constructori:

`File(String cale)`//crează un obiect `File` ce conține o cale absolută sau relativă și care poate conține numele fișierului

`File(String cale, String nume)`//crează un obiect `File` ce conține separat numele fișierului și calea directorului în care se află fișierul

`File(File dir, String nume)`//calea directorului părinte este dat ca un obiect `File`

Metodele mai importante ale clasei `File` sunt prezentate în următorul tabel:

Semnătura metodei	Semnificație –Valoarea returnată
-------------------	----------------------------------

Semnătura metodei	Semnificație –Valoarea returnată
String getName()	returnează numele fișierului sau directorului curent
String getPath()	returnează calea unde se află fișierul sau directorul curent pe disc
String getAbsolutePath()	returnează calea absolută unde se află fișierul sau directorul curent pe disc
String getParent()	întoarce directorul părinte a directorului sau în care se găsește fișierul curent
boolean exists()	returnează true dacă fișierul indicat in constructor există pe disc.
boolean canWrite()	returnează true dacă fișierul respectiv poate fi modificat
boolean canRead()	returnează true dacă fișierul respectiv poate fi citit
boolean isFile()	returnează true dacă șirul indicat ca fișier există pe disc și este un fișier
boolean isDirectory()	returnează true dacă șirul indicat ca director există pe disc și este un director
long lastModified()	returnează data și ora ultimei modificări
long length()	returnează mărimea fișierului, în biți
boolean mkdir()	returnează true dacă a fost creat directorul specificat in cale
boolean mkdirs()	returnează true dacă au fost create directoarele specificate in cale
boolean renameTo(File dest)	redenumeste fișierul cu curent cu numele dest și returnează true
boolean delete()	șterge fișierul sau directorul curent
String[] list()	întoarce într-un array de șiruri numele fișierelor și directoarelor din directorul curent.
String[] list(FilenameFilter filtru)	întoarce într-un array de șiruri numele fișierelor și directoarelor din directorul curent care îndeplinesc criteriul filtru

De exemplu, pentru a crea un filtru trebuie să scriem o clasă care implementează interfața FilenameFilter. De exemplu, următoarea clasa filtrează fișierele dintr-un director și le acceptă numai pe cele care sunt programe Java:

```
import java.io.*;

public class Filtru implements FilenameFilter{

    public boolean accept(File dir, String nume){
        String sufix="";
        int i=nume.lastIndexOf('.');
        if(i>0 && i<nume.length()-1) sufix=nume.substring(i+1);
        return sufix.equals("java");
    }

    public String getDescription(){
        return "fisiere java (*.java)";
    }
}
```

Următorul exemplu citește numele unui fișier/director de la tastatură și-l scrie informații despre el, cum ar fi: dacă este fișier sau director, locația sa pe disc, data ultimei modificări, etc, în fișierul analiza.txt:

```

import java.io.*;
public class Fisier{
    BufferedReader br;
    PrintWriter pw;

    public Fisier(){
        try{
            br=new BufferedReader(new InputStreamReader(System.in));
            pw=new PrintWriter(new FileWriter("analiza.txt"), true);
            scrie(analizeaza(citeste()));
        }catch(IOException ioe){}
    }

    public String citeste()throws IOException{
        System.out.println("Dati numele fisierului/directorului");
        return br.readLine();
    }

    public String analizeaza(String sir){
        File nume=new File(sir);
        StringBuffer buffer=new StringBuffer();
        if(nume.exists()){
            buffer.append(nume.getName()+" exista\r\n");
            if(nume.isFile()) buffer.append("este un fisier\r\n");
            if(nume.isDirectory())buffer.append("este un director\r\n");
            if(nume.isAbsolute())buffer.append("este o cale absoluta\r\n");
            else buffer.append("nu este o cale absoluta\r\n");
            buffer.append("data ultimei modificari "+nume.lastModified()+"\r\n");
            buffer.append("lungimea "+nume.length()+"\r\n");
            buffer.append("calea "+nume.getPath()+"\r\n");
            buffer.append("calea absoluta "+nume.getAbsolutePath()+"\r\n");
            buffer.append("parinte "+nume.getParent()+"\r\n");

            if(nume.isFile()){
                buffer.append("Continutul fisierului\r\n");
                try{
                    RandomAccessFile in=new RandomAccessFile(nume, "r");
                    String s="";
                    while((s=in.readLine())!=null) buffer.append(s+"\r\n");
                    in.close();
                }catch(IOException ioe){}
            }
            else
                if(nume.isDirectory()){
                    buffer.append("Continutul directorului\r\n");
                    String[] dir=nume.list();
                    for(int i=0;i<dir.length; i++)
                        buffer.append(dir[i)+"\r\n");
                }
        }
        else
            buffer.append("acest nume nu exista");
        return buffer.toString();
    }

    public void scrie(String sir) throws IOException {
        pw.print(sir);
        pw.close();
    }

    public static void main(String[] a){

```

```
        new Fisier();  
    }  
}
```

Programul 1. Fisier: Lucrul cu clasa File

Clasa StreamTokenizer

Ca și StringTokenizer, această clasă imparte un flux în token-uri, dar spre deosebire de prima clasă, diferențiază între tipuri de token-uri: șiruri, numere, sfârșitul unei linii, sfârșitul fișierului, etc.

Constructorul clasei:

```
public StreamTokenizer(Reader r) //creaza un tokenizer care imparte fluxul de intrare r într-un șir de token-uri.
```

Pentru a ști dacă s-a ajuns la sfârșitul liniei curente se folosește constanta:

```
public static final int TT_EOL //token-ul citit conține sfârșitul liniei curente (“\r” sau “\n”)
dacă acesta este tratat ca un token, adică a fost apelată metoda void
eolIsSignificant(boolean val) cu val=true.
```

iar pentru a trata sfârșitul fluxului de intrare folosim constanta

```
public static final int TT_EOF //token-ul citit conține sfârșitul fluxului de intrare
```

Deosebirea față de clasa StringTokenizer este că această clasă diferențiază token-ii obținuți din fragmentare. Astfel, clasa oferă următoarele constante:

```
public static final int TT_NUMBER //token-ul citit este un număr
```

```
public static final int TT_WORD //token-ul citit este un cuvânt
```

Pentru a obține numărul, respectiv cuvântul citit se folosesc următoarele variabile instanță:

```
public double nval //conține valoarea numărului întors ca token
```

```
public String sval //conține caracterele cuvântului întors ca token
```

Pentru a obține un token din fluxul analizat se apelează metoda instanță:

```
public int nextToken()
```

care returnează tipul token-ului citit ca un număr întreg. Acesta este memorat după apelul metodei și în variabila instanță `public int ttype`.

Clasa oferă și alte metode utile în analizarea unui flux de intrare specializat:

```
public void parseNumbers() //sunt analizate numerele și caracterele ‘.’ și ‘-’.
```

```
public int lineno() //întoarce numărul liniei curente.
```

```
public void ordinaryChar(int ch) //tratează caracterul dat la apelul metodei ca unul
oarecare, fără să mai verifice tipul sau să-l întoarce ca un token de tip TT_WORD.
```

```
public void ordinaryChars(int min, int max) //toate caracterele ce au codul Unicode în
intervalul [min, max] sunt tratate ca oarecare.
```

```
public void commentChar(int ch) //caracterul ch indică începutul unui comentariu. În acest
caz, token-ii fluxului de intrare aflați până la sfârșitul liniei vor fi ignorați.
```

```
public void quoteChar(int ch) //caracterele găsite între două apariții a lui ch sunt tratate ca un
literal String și întoarse ca un token de tip TT_WORD. Dacă a doua apariție a caracterului ch nu
este găsită până la sfârșitul liniei, se iau caracterele până la acesta. Dacă ch apare o singură dată
în fluxul de intrare, se iau caracterele de la ch până la sfârșitul fișierului.
```

```
public void whitespaceChars(int min, int max) // toate caracterele ce au codul Unicode în
intervalul [min, max] sunt tratate ca spații.
```

```
public void wordChars(int min, int max) // toate caracterele ce au codul Unicode în intervalul
[min, max] sunt tratate ca parte din cuvinte.
```

```
public void pushBack() //token-ul curent va fi obținut și la următorul apel al metodei nextToken()
```

Exemplu. Să scriem un program care citește conținutul fișierului “intrare.txt” și afișează tipul fiecărui token și valoarea sa, precum și numărul de cuvinte, respectiv numere, din fișierul respectiv.

```
import java.io.*;
public class Tokenizer{
    private StreamTokenizer st;
    private int w, n;

    public Tokenizer(){
        String linie;
        try{
            st=new StreamTokenizer(new BufferedReader(new FileReader("intrare.txt")));
        }catch(FileNotFoundException nf){System.out.println(nf.getMessage());}
```

```

        catch(IOException io){System.out.println(io.getMessage());}
    }

    public void imparte(){
        if(st==null) System.exit(0);
        try{
            while(st.nextToken()!=StreamTokenizer.TT_EOF){
                switch(st.ttype){
                    case StreamTokenizer.TT_WORD: System.out.print("cuvant= "+st.sval+" ");
                                                    w++; break;
                    case StreamTokenizer.TT_NUMBER: System.out.print("numar= "+st.nval+" ");
                                                    n++; break;
                }
                System.out.println();
            }
            System.out.println("Total cuvinte= "+w);
            System.out.println("Total numere= "+n);
        }catch(IOException io){System.out.println(io.getMessage());}
    }

    public static void main(String[] args){
        Tokenizer t=new Tokenizer();
        t.imparte();
    }
}

```

Probleme propuse

1. Sa se scrie un program care citeste doua siruri de la tastatura si verifica daca sunt egale lexicografic. In caz negativ, sa se afiseze sirul care este mai mic.
2. Cititi de la tastatura un sir de numere intregi pozitive pana la introducerea numarului 0. Introduceti aceste numere intr-un array si apoi scrieti numerele ordonate crescator intr-un fisier "numere.txt".
3. Construiti o clasa Student. Fiecare student are attributele: nume, prenume, facultate si medieGenerala. MediaGenerala este media studentului din anul trecut. In fisierul "studenti.txt" se afla informatiile referitoare la studentii unei universitati in formatul:
Nume_Prenume_Facultate_MedieGenerala

Construiti un array de obiecte Student, cu informatiile din fisierul de intrare. Apoi afisati studentii in ordinea descrescatoare a mediilor generale.

4. Sa se scrie un program care translateaza frazele scrise in limba romana intr-un alt limbaj numit «pig Latin». Pentru a traduce fiecare cuvnt din romaneste in pig Latin se plaseaza prima litera la sfarsitul cuvntului si apoi se adauga sirul «ei». Astfel cuvntul «noi»devine «oinei». Presupunem ca cuvintele romanesti sunt separate printr-un singur spatiu si sunt memorate in fisierul «intrare.txt». Fiecare fraza este scrisa pe o singura linie si se termina cu un punct. Frazele obtinute prin traducere sunt memorate intr-un alt fisier «traducere.txt».
5. Presupunem ca avem un fisier care contine enuntul problemei anterioare. Sa se scrie un program care citeste continutul fisierului si afiseaza numai cuvintele care contin subsirul "ra".
6. Scrieti un program care citeste cinci linii de text de la tastatura si scrie intr-un fisier numarul de aparitii in text a fiecarei litere a alfabetului.

Exp.

a 0

b 2

c 1

.
.
.
z 0

7. Scrieti un program care citește conținutul unui fisier și scrie într-un alt fisier numărul cuvintelor formate dintr-o literă, din 2 litere, etc. Exp:

Lungimea cuvântului	Apariții
1	0
2	1
3	4