Lecţia 1

Introducere în programarea orientată pe obiecte

1.1 Primii paşi în programarea orientată pe obiecte

Totul a început datorită necesităţii disperate de a organiza mai bine programele. Într-o viziune primitivă programele sunt văzute ca fiind "făcute" din proceduri şi funcţii, fiecare implementând o bucăţică din funcţionalitatea programului (altfel spus, un algoritm) şi făcând uz de un fond global comun de date. Prin intermediul apelurilor de proceduri şi funcţii, aceşti algoritmi sunt combinaţi în vederea obţinerii funcţionalităţii dorite din partea programului.

O lungă perioadă de timp programatorii au scris programe având în minte această viziune de organizare. Ei bine, odată ce programele au început să devină tot mai mari şi mai complexe, această modalitate de organizare a început să-şi arate deficienţele. Astfel, în loc să se ocupe de implementarea propriu-zisă a unor noi bucăţi de funcţionalitate necesare extinderii unui program, programatorii au început să petreacă tot mai mult timp încercând să înţeleagă diverse proceduri şi funcţii pentru a putea combina noile bucăţi de funcţionalitate cu cele deja existente. Cum întelegerea subprogramelor devenea din ce în ce mai grea datorită modificărilor succesive aduse lor, adesea de programatori diferiţi, aceste programe au început să arate în scurt timp ca nişte "saci cu foarte multe petice". Cel mai grav, s-a ajuns în multe situaţii ca nimeni să nu mai ştie rolul unui "petec" iar uşoare încercări de modificare a lor ducea la "spargerea sacului", adică la un comportament anormal al programului. Încet încet, programul scăpa de sub control, nimeni nu îl mai putea extinde şi în scurt timp nici un client nu mai era interesat în a-l achiziţiona din moment ce nu mai corespundea noilor sale necesităţi.

În vederea limitării acestor situaţii neplăcute s-au dezvoltat diferite modalităţi de organizare a programelor cunostute sub numele de stiluri arhitecturale. Stilul arhitectural descris anterior a fost numit "Main program and subroutines". Există descrise multe

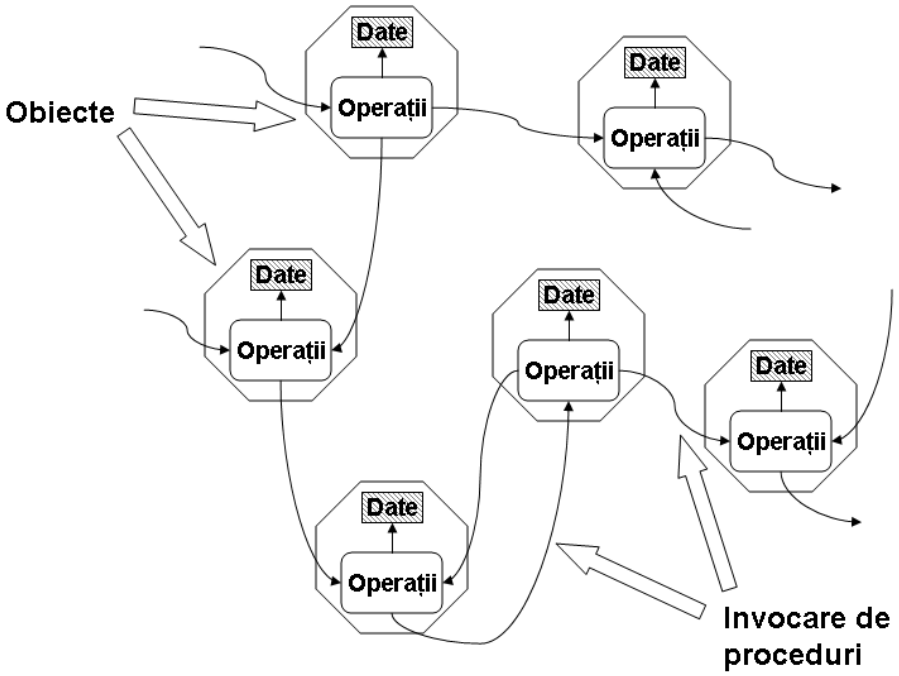

Figura 1.1: Organizarea orientată pe obiecte a unui sistem software.

alte stiluri în literatura de specialitate. Pe parcursul acestei cărţi noi ne vom concentra asupra stilului arhitectural denumit object-oriented. În viziunea acestui stil de organizare, un program este "făcut" din obiecte, la fel ca lumea reală. **Un obiect trebuie să "grupeze" împreună un set de date şi un set de operaţii primitive singurele care ştiu manipula respectivele date**. Obiectele cooperează între ele în vederea obţinerii funcţionalităţii dorite din partea programului prin intermediul apelurilor la operaţiile primitive corespunzătoare fiecărui obiect. Figura 1.1 surpinde aspectele legate de organizarea orientată pe obiecte a unui program.

Atenţie Este total împotriva spiritului de organizare orientat pe obiecte ca un obiect să facă uz de datele unui alt obiect. Singura formă de cooperare permisă între obiecte trebuie realizată prin intermediul apelului la operaţii primitive. Şi apropo, o operaţie a cărei singur rol e de a returna o dată a obiectului asociat NU e în general o operaţie primitivă.

În acest context putem da o definiţie parţială a programării orientate pe obiecte.

Definiţie 1 *Programarea orientată pe obiecte este o metodă de implementare a programelor în care acestea sunt organizate ca şi colecţii de obiecte care cooperează între ele.*

Ei bine, acum ştim parţial ce înseamnă programarea orientată pe obiecte. Dar putem deja să scriem programe conform programării orientate pe obiecte? Răspunsul este categoric NU!!! Cum "descompunem" un program în obiectele sale componente? Cum stabilim operaţiile primitive asociate unui obiect? Cum implementăm operaţiile primitive asociate unui obiect? Înainte de toate, pentru a putea răspunde la aceste întrebări, trebuie să înţelegem anumite concepte care stau la baza organizării orientate pe obiecte. Fără o bună înţelegere a lor nu vom şti niciodată să programăm corect orientat pe obiecte. Nici măcar dacă ştim la perfecţie limbajul de programare Java!

Important

În urma discuţiei de mai sus anumite persoane ar putea ajunge la anumite concluzii pripite. *"Tot ce am învăţat despre programare până acum este inutil."* Este total fals. Tendinţa aproape instinctuală a unei persoane, când are de scris un program care să rezolve o anumită problemă, este de a identifica o serie de algoritmi necesari rezolvării unor părţi din problema iniţială, de a implementa aceşti algoritmi ca subprograme şi de a combina aceste subprograme în vederea rezolvării întregii probleme. Prin urmare, pare natural să organizezi un program aşa cum am amintit la începutul acestui capitol, dar în acelaşi timp nu este bine să procedăm aşa. Ei bine, nu este întru-totul adevărat. Acest mod de organizare este bun dacă e folosit unde trebuie. Nu e bine să organizăm astfel întregul program, dar, după cum vom vedea mai târziu, obiectele sunt organizate astfel în intimitatea lor.

"Limbajul de programare C este depăşit şi nu are sens să-l mai studiez şi nici să-l mai utilizez." Este mai mult decât fals. Utilizarea filosofiei organizării orientate pe obiecte a unui program nu depinde absolut de utilizarea unui limbaj de programare orientat pe obiecte cum sunt, de exemplu, Java şi C++. Este adevărat că limbajul C nu deţine anumite mecanisme absolut necesare programării orientate pe obiecte, dar nu este imposibil să implementăm un program în C şi în acelaşi timp să-l organizăm orientat pe obiecte. În astfel de situaţii se vorbeşte despre programare cu tipuri de date abstracte sau despre programare bazată pe obiecte. În capitolele următoare vom arăta care sunt diferenţele.

1.1.1 Abstractizarea

Lumea care ne înconjoară este făcută din obiecte. Zi de zi interacţionăm cu obiecte pentru a duce la îndeplinire anumite activităţi. Unele obiecte sunt mai simple, altele sunt mai complexe. Dar poate cineva să ne spună în cele mai mici detalii cum funcţionează fiecare obiect cu care interacţionează el sau ea zilnic? Răspunsul e simplu: nu! Motivul ar putea fi şocant pentru unii: pentru că în general ne interesează ceea ce fac obiectele şi nu cum fac. Cu alte cuvinte, simplificăm sau abstractizăm obiectele reţinând doar caracteristicile lor esenţiale. În cazul programării orientate pe obiecte aceste caracteristici se referă exclusiv la comportamentul observabil al unui obiect.

Mulţi dintre noi interacţionează zilnic cu obiectul televizor pentru a urmări diferite programe Tv. Dar oare câţi dintre noi pot să ne spună în cele mai mici detalii cum funcţionează televizorul de acasă? Evident foarte puţini. Motivul este că pe majoritatea dintre noi ne interesează doar să utilizăm televizorul pentru a urmări programe Tv. Cum anume ajunge imaginea din studioul unei televiziuni pe ecranul televizorului îi interesează doar pe cei care fac televizoare.

Dar sunt aceste caracteristici esenţiale aceleaşi pentru oricare persoană care interacţionează cu un obiect particular? Evident nu. Depinde de la persoană la persoană.

Pentru a achita valoarea unei călătorii cu autobusul, un pasager utilizează un compostor de bilete pentru a-şi perfora biletul de călătorie. Din perspectiva firmei care pune la dispoziţie serviciul de transport, obiectul compostor poate fi utilizat şi pentru a determina numărul de călători dintr-o zi şi/sau intervalul orar cel mai aglomerat, lucruri care nu sunt importante din perspectiva pasagerului.

Definiţie 2 *O abstracţiune (rezultatul procesului de abstractizare) denotă caracteristicile esenţiale ale unui obiect care-l disting de orice alt obiect definind astfel graniţele conceptuale ale obiectului relativ la punctul de vedere din care este văzut.*

O aceeaşi abstracţiune poate corespunde mai multor obiecte din lumea care ne înconjoară dacă ele sunt identice din perspectiva din care sunt privite. De exemplu, dacă pe o persoană o interesează cât e ora la un moment dat poate folosi la fel de bine un ceas mecanic sau un telefon mobil cu ceas. Dacă altcineva vrea însă să dea un telefon evident că acestor obiecte nu le poate corespunde aceeaşi abstracţiune. O aceeaşi abstracţiune poate corespunde în acest caz mai multor feluri de telefoane (mobile sau fixe).

1.1.2 Interfaţa

După cum reiese din secţiunea anterioară abstracţiunea scoate în evidenţă comportamentul observabil al unui obiect din punctul de vedere al unui utilizator. Cu alte cuvinte, abstracţiunea focalizează atenţia asupra serviciilor pe care le pune la dispoziţie obiectul spre a fi utilizate de alte obiecte. Aceste servicii nu sunt altceva decât operaţiile primitive ale obiectului iar ansamblul tuturor operaţiilor primitive se numeşte *interfaţa* obiectului respectiv. Un nume utilizat pentru identificarea unei anumite interfeţe se numeşte *tip* (mai general tip de date abstract).

Să revenim la exemplul cu ceasul mecanic şi telefonul mobil. Din perspectiva unei persoane care vrea să ştie cât e ceasul ambele obiecte au aceeaşi interfaţă care pune la dispoziţie operaţia prin care omul sau, în general, un alt obiect poate afla ora exactă. Din punctul lui de vedere obiectele sunt de acelaşi tip. Este clar că pentru o persoană care vrea să dea un telefon obiectele au interfeţe diferite şi deci tipuri diferite. Să ne gândim acum la compostorul de bilete. Pare ciudat, dar acelaşi obiect compostor are două tipuri diferite. Aceasta pentru că diferă perspectiva din care este privit de clienţi, adică de către călător, respectiv de către managerul firmei de transport.

1.1.3 Ascunderea informaţiei

Abstractizarea este utilă pentru că focalizează atenţia doar asupra caracteristicilor comportamentale esenţiale ale unui obiect din perspectiva unui utilizator, permiţând astfel identificarea interfeţei obiectului. Dar, după cum am spus la începutul acestei expuneri, un obiect grupează un set de date şi un set de operaţii primitive, singurele care ştiu manipula aceste date. Îl interesează pe utilizator aceste date în mod direct? Mai mult, operaţiile primitive trebuie să aibă o implementare. Interfaţa ne spune doar care sunt aceste operaţii şi nimic altceva. Îl interesează pe utilizator modul lor de implementare în mod direct?

Aici intervine aşa numitul *principiu al ascunderii informaţiei*. O discuţie completă despre acest principiu este după părerea noastră cu mult dincolo de scopul acestei cărţi. Drept urmare, ne vom rezuma a spune că acest principiu este utilizat când se decide să se ascundă ceva neesenţial despre un obiect oarecare de orice utilizator potenţial. Ei bine, în contextul programării orientate pe obiecte acest principiu este utilizat când spunem că reprezentarea datelor unui obiect şi implementarea operaţiilor sale primitive trebuie ascunse de orice client al obiectului. Cum anume realizăm această ascundere vom vedea în următorul paragraf.

În urma abstractizării televizorului de acasă, un utilizator ar putea ajunge la concluzia că interfaţa unui astfel de obiect conţine următoarele operaţii: comutăPornitOprit, măreşteVolumul, micşorează Volumul, comutăPeProgramulAnterior, comutăPeProgramulUrmător. Ei bine, toate aceste operaţii primitive au o implementare undeva înăuntrul cutiei televizorului. Folosind principiul ascunderii informaţiei constructorul televizorului a hotărât ca toate detaliile de implementare să fie ascunse utilizatorului televizorului. Atenţie, doar a hotărât că trebuie ascunse. Faptul că ele au fost ascunse folosind o cutie nu are importanţă din punctul de vedere al principiului ascunderii informaţiei.

1.1.4 Încapsularea

În programarea orientată pe obiecte, abstractizarea ajută la determinarea serviciilor furnizate de un obiect. Prin utilizarea principiului ascunderii informaţiei se spune că datele şi implementarea operaţiilor unui obiect trebuie ascunse de orice client potenţial al obiectului. Cu alte cuvinte, obiectul nu trebuie să spună nimic despre datele şi implementarea operaţiilor sale. Încapsularea vine să completeze cele două noţiuni, reprezentând mecanismul necesar punerii împreună a interfeţei unui obiect cu o anumită implementare a acestei interfeţe. Mecanismul permite în acelaşi timp ascunderea implementării de orice posibil client al respectivei interfeţe, făcând astfel posibilă aplicarea principiului ascunderii informaţiei.

Definiţie 3 *Încapsularea este procesul de compartimentare a elementelor unei abstracţiuni care constituie structura şi comportamentul său. Încapsularea serveşte la separarea interfeţei unei abstracţiuni şi a implementării sale.*

Atenţie

Este total împotriva noţiunii de programare orientată pe obiecte ca un client să poată accesa datele unui obiect sau să ştie detaliile de implementare ale unei operaţii primitive. Aceste informaţii trebuie ascunse. Încapsularea permite să se ascundă implementarea unui obiect, dar nu de la sine. De obicei, programatorul este cel care trebuie să spună explicit atunci când defineşte un obiect ce este ascuns şi ce nu. Cu excepţia motivelor bine întemeiate, toate datele unui obiect trebuie ascunse explicit de către programator. Promisiuni de genul "promit să nu mă uit niciodată la datele tale" NU se acceptă deoarece mai devreme sau mai târziu cineva tot le va încălca.

Să revenim la exemplul anterior despre televizor. Clientul este interesat doar de interfaţa televizorului prin intermediul căreia poate controla acest obiect. Folosind principiul ascunderii informaţiei, constructorul televizorului decide să ascundă detaliile de implementare ale acestei interfeţe. Pentru a realiza acest lucru el foloseşte de obicei o cutie. Încapsularea este procesul prin care televizorul în ansamblul său este intodus în respectiva cutie. Ea va ascunde tot ce ţine de detaliile de funcţionare ale televizorului şi lasă accesibil doar ce ţine de interfaţa televizorului.

1.2 Primii paşi în Java

Dacă în prima parte a acestei lecţii am introdus primele elemente legate de programarea orientată pe obiecte, în această secţiune vom prezenta pe scurt câteva noţiuni de bază necesare scrierii unui program în limbajul Java.

1.2.1 Comentariile

Comentariile reprezintă porţiuni de cod sursă care sunt ignorate de către compilator şi sunt utilizate cu precădere pentru documentarea codului sursă. Modul de marcare a comentariilor este prezentat în exemplul de mai jos.

```
/*Exemplu de comentariu pe o linie*/

/*Exemplu de comentariu pe
    mai multe linii*/

//Alt exemplu de comentariu.
```

În cazul primului tip de comentariu, tot ce este cuprins între /* şi */ este ignorat de către compilator. În cazul celui de-al doilea tip, tot ce urmează după // până la terminarea liniei este ignorat.

1.2.2 Identificatori, tipuri primitive şi declararea de variabile

Identificatorul este un nume asociat unei entităţi dintr-un program. Numele unei variabile, a unui parametru sau a unei funcţii (în Java metode) reprezintă identificatori. Principial, în Java un identificator poate începe cu _ sau cu o literă şi poate conţine litere, cifre şi caracterul _ [1]. Există un număr de identificatori care sunt rezervaţi deoarece ei reprezintă cuvinte cheie ale limbajului Java, de exemplu cuvintele cheie *main* sau *class*. Nu vom furniza aici o listă a tuturor acestor cuvinte cheie deoarece ele vor fi învăţate pe parcurs. Momentan ne vom opri doar asupra cuvintelor cheie corespunzătoare tipurilor primitive. În tabelele următoare sunt prezentate tipurile primitive definite în limbajul de programare Java.

Tip	Număr de biţi utilizaţi	Domeniul valoric
byte	8	$-128, 127$
short	16	$-32768, 32767$
int	32	$-2147483648, 2147483647$
long	64	$-9223372036854775808, 9223372036854775807$

Tabelul 1.1: TIPURI NUMERICE ÎNTREGI.

Este important de amintit aici şi tipul *String* asociat şirurilor de caractere. După cum vom vedea ceva mai târziu, acest tip nu este unul primitiv. Limbajul Java tratează într-un mod special acest tip pentru a face mai uşoară munca programatorilor (şirurile de caractere se utilizează des în programe). Din acest motiv el se aseamănă cu tipurile primitive şi poate fi amintit aici.

[1] În Java, un identificator poate începe şi conţine caractere Unicode corespunzătoare unui simbol de monedă.

Tip	Număr de biţi utilizaţi	Domeniul valoric
float	32	$\pm 1.4E-45, \pm 3.4028235E+38$
double	64	$\pm 4.9E-324, \pm 1.7976931348623157E+308$

Tabelul 1.2: Tipuri numerice în virgulă flotantă.

Tip	Număr de biţi utilizaţi	Exemple de valori
char	16	• 'a' - caracterul a • '\n' - caracterul linie nouă • '\u3C00' - caracterul π specificat în hexazecimal
boolean	nespecificat	Sunt posibile doar valorile logice date de cuvintele cheie *true* şi *false*

Tabelul 1.3: Alte tipuri primitive.

În continuare, vom arăta prin intermediul unui exemplu modul în care se declară variabile în Java, mod ce nu diferă fundamental de declararea variabilelor în limbajul C.

```
//Declararea unei variabile are forma: Tip NumeVariabila = Initializare
char unCaracter,altCaracter = 'A';
int i;
String sir = "Acesta e un sir de caractere";
```

1.2.3 Expresii şi operatori

În paragraful anterior am văzut care sunt tipurile primitive ale limbajului Java şi modul în care putem declara variabile de diferite tipuri. Variabilele împreună cu literalii (constante de genul 'A' sau "Un sir de caractere") sunt cele mai simple expresii posibile, denumite şi expresii primare. Expresii mai complicate pot fi create combinând expresiile primare cu ajutorul *operatorilor*. Câţiva dintre operatorii Java sunt prezentaţi în Tabelul 1.4.

Există două noţiuni importante care trebuie amintite în contextul operatorilor. Prima este *precedenţa* (P) operatorilor care dictează ordinea în care se vor efectua operaţiile. Ca şi exemplu, operatorul de înmulţire are o precedenţă mai ridicată decât operatorul de adunare şi prin urmare înmulţirea se realizează înainte de adunare. Precedenţa

implicită a operatorilor poate fi modificată prin utilizarea parantezelor.

```
int a,b,c;

//Operatorul * are o precedenta mai ridicata decat + deci se executa primul
a * b + c

//Operatorul * are o precedenta mai ridicata decat + dar din cauza
//parantezelor care prezinta explicit ordinea operatiilor adunarea
//se va executa prima
a * (b + c)
```

A doua noţiune importantă legată de operatori constă în *asociativitatea* (A) lor. Ea specifică ordinea în care se execută operaţiile atunci când o expresie utilizează mai mulţi operatori de aceeaşi precedenţă. Un operator poate fi asociativ la stanga sau asociativ la dreapta. În primul caz expresia se evaluează de la stânga la dreapta, iar în al doilea caz de la dreapta la stânga. Evident, utilizarea parantezelor poate modifica ordinea de evaluare implicită.

```
int a,b,c;

//Operatorul - este asociativ la stanga, deci prima data se executa
//scaderea iar apoi adunarea
a - b + c

//Operatorul - este asociativ la stanga, dar parantezele spun ca prima data
//se executa adunarea apoi scaderea
a - (b + c)

//Operatorii += si = sunt asociativi la dreapta, deci prima data se da
//valoarea 5 lui b dupa care se da valoarea a + 5 lui a
a+= b = 5
```

1.2.4 Tipărirea la ieşirea standard

După cum am mai spus anterior, un program organizat în spiritul stilului orientat pe obiecte este alcătuit din obiecte. Prin urmare, pentru a afişa o valoare pe ecranul calculatorului, avem nevoie de un obiect care ştie face acest lucru. Limbajul Java pune automat la dispoziţie un astfel de obiect, accesibil prin "variabila" *out*. El are o operaţie primitivă [2] (mai exact metodă) denumită *println* care ştie să afişeze parametrul dat, indiferent de tipul său, la ieşirea standard. În exemplul următor se arată modul în care se realizează afişarea pe ecran dintr-un program.

[2] De fapt are mai multe dar pentru moment nu ne interesează acest lucru.

P	A	Operator	Tipul operanzilor	Efectul execuţiei
15	S	++, --	variabilă de tip numeric /char	Post incrementare/decrementare
14	D	++, --	variabilă de tip numeric /char	Pre incrementare/decrementare
14	D	+, -	numeric/char	Plus/Minus unar
14	D	~	întreg/char	Negare pe biţi
14	D	!	boolean	Negare logică
12	S	*, /, %	numeric/char, numeric /char	Înmulţire/împărţire/rest
11	S	+,-	numeric/char, numeric /char	Adunare/scădere
11	S	+	String, orice tip	Concatenare de şiruri de caractere
10	S	<<	întreg/char, întreg/char	Deplasare stânga pe biţi
10	S	>>	întreg/char, întreg/char	Deplasare dreapta pe biţi cu inserţie de semn
10	S	>>>	întreg/char, întreg/char	Deplasare dreapta pe biţi cu inserţie de 0
9	S	<, <=	numeric/char, numeric /char	Mai mic/mai mic sau egal
9	S	>, >=	numeric/char, numeric /char	Mai mare/mai mare sau egal
8	S	==	orice tip , orice tip	Egalitate valorică
8	S	!=	orice tip, orice tip	Inegalitate valorică
7	S	&	întreg/char, întreg/char	Şi pe biţi
7	S	&	boolean, boolean	Şi logic
6	S	^	întreg/char, întreg/char	Sau exclusiv pe biţi
6	S	^	boolean, boolean	Sau exclusiv logic
5	S	|	întreg/char, întreg/char	Sau pe biţi
5	S	|	boolean, boolean	Sau logic
4	S	&&	boolean, boolean	Şi logic
3	S	||	boolean, boolean	Sau logic
1	D	=	variabilă de orice tip, tipul variabilei	Atribuire

Tabelul 1.4: Un subset de operatori Java.

```
//Afisarea unui sir de caractere
System.out.println("Acest mesaj va fi afisat pe ecranul calculatorului!");

//Afisarea valorii unei variabile
System.out.println(i);
```

Atenţie Prezenţa lui *System* înaintea obiectului *out* este obligatorie. Motivul îl vom vedea mai târziu.

1.2.5 Instrucţiuni de control

Instrucţiunile de control sunt necesare pentru a permite unui program să ia anumite decizii la execuţia sa. Principalele instrucţiuni de decizie sunt *if* şi *switch*. Buclele de program se realizează prin instrucţiunile *while*, *do* şi *for*. Deoarece toate aceste instrucţiuni sunt asemănătoare celor din C nu vom insista asupra lor, rezumându-ne în a prezenta mai jos forma lor generală.

```
if (ExpresieLogica)
    //Instructiunea se executa daca ExpresieLogica este adevarata
    //Daca sunt mai multe instructiuni ele trebuie cuprinse intre { si }
else {
    //Instructiuni
    //Ramura else poate sa lipseasca daca nu e necesara
}

switch (Expresie) {
    //Expresie trebuie sa fie de tip char, byte, short sau int
    case ExpresieConstanta1:
        //Instructiuni ce se executa cand Expresie ia valoarea lui
        //ExpresieConstanta1
            break;
    case ExpresieConstanta2:
        //Instructiuni ce se executa cand Expresie ia valoarea lui
        //ExpresieConstanta2
            break;
    case ExpresieConstanta3:
        //Instructiuni ce se executa cand Expresie ia valoarea lui
        //ExpresieConstanta3
            break;
    default:
        //Instructiuni ce se executa cand Expresie ia o valoare diferita
        //de oricare ExpresieConstanta. Ramura default poate lipsi
}
```

```
while(ExpresieLogica) {
    //Instructiuni ce se executa atata timp cat ExpresieLogica este
    //adevarata
}

do {
    //Instructiuni ce se repeta atata timp cat ExpresieLogica e adevarata
    //Ele se executa cel putin o data pentru ca ExpresieLogica este testata
    //la sfarsitul buclei
} while(ExpresieLogica);

for (Initializare;ExpresieLogica;Incrementare) {
    //Instructiuni ce se repeta atata timp cat ExpresieLogica e adevarata
    //Inaintea primei iteratii se executa Initializare
    //Dupa fiecare iteratie se executa Incrementare
}
```

La fel ca în limbajul C, există de asemenea instrucţiunile *continue* şi *break*. Instrucţiunea *continue* trebuie să apară în interiorul unui ciclu, iar efectul său constă în trecerea imediată la execuţia următoarei iteraţii din bucla imediat înconjurătoare instrucţiunii *continue*. Tot în interiorul ciclurilor poate apare şi instrucţiunea *break*. Efectul ei constă în terminarea imediată a ciclului imediat înconjurător instrucţiunii *break*. În plus, instrucţiunea *break* poate apare şi în corpul unuei instrucţiuni *switch*, mai exact pe una dintre posibilele ramuri *case* ale instrucţiunii. Execuţia unui astfel de *break* conduce la terminarea execuţiei instrucţiunii *switch*. Dacă pe o ramură *case* nu apare instrucţiunea *break* atunci la terminarea execuţiei instrucţiunilor respectivei ramuri se va continua cu execuţia instrucţiunilor ramurii următoare (evident dacă există una). Situaţia este exemplificată mai jos.

```
char c;
//Instructiuni
switch(c) {
    case '1': System.out.println("Unu ");break;
    case '2': System.out.println("Doi ");
    case '3': System.out.println("Trei");
}
//Daca c este caracterul '1' pe ecran se va tipari
//     Unu
//Daca c este caracterul '2' pe ecran se va tipari
//     Doi
//     Trei
//Daca c este caracterul '3' pe ecran se va tipari
//     Trei
```

1.2.6 Primul program

În acest paragraf vom vedea un prim program Java şi modul în care îl lansăm în execuţie.

```
class PrimulProgram {

    public static void main(String argv[]) {
        System.out.println("Hello world!");
    }
}
```

La fel ca şi în limbajul de programare C, execuţia unui program Java începe în funcţia (în Java metoda) *main*. Singurul parametru al acestei metode este un tablou de şiruri de caractere prin intermediul căruia se transmit parametri din linia de comandă. Metoda nu returnează nici o valoare, motiv pentru care se specifică tipul *void* ca tip returnat.

După cum vom vedea în lucrarea următoare, în Java orice metodă trebuie să fie inclusă într-o *clasă*, în acest caz clasa *PrimulProgram*. Tot acolo vom înţelege şi rolul cuvintelor cheie *public* şi *static*. Deocamdată nu vom vorbi despre ele.

Pentru a rula programul, acesta trebuie mai întâi compilat. Să presupunem că programul prezentat mai sus se află într-un fişier denumit *PrimulProgram.java*. Pentru a-l compila se foloseşte comanda:

```
javac PrimulProgram.java
```

Rezultatul va fi un fişier cu numele *PrimulProgram.class* care conţine, printre altele, codul executabil al metodei *main*.

Important Să presupunem că programul de mai sus s-ar fi aflat în fişierul *Program.java*. În acest caz comanda de compilare ar fi fost *javac Program.java*, dar rezultatul compilării ar fi fost tot *PrimulProgram.class* deoarece numele fişierului rezultat se obţine din numele clasei compilate şi nu din numele fişierului ce conţine codul sursă.

Pentru a se lansa în execuţie programul se utilizează comanda:

```
java PrimulProgram
```

Ca urmare a acestei comenzi, maşina virtuală Java va căuta în fişierul *PrimulProgram.class* codul metodei *main* după care va trece la execuţia sa. În acest caz se va afişa pe ecran mesajul “Hello world!”.

Atenţie Dacă clasa dată ca parametru maşinii virtuale Java nu conţine metoda *main* se va semnala o eroare.

1.3 Exerciţii

1. Compilaţi şi lansaţi în execuţie programul "Hello World!" dat ca exemplu în Secţiunea 1.2.6.
2. Scrieţi un program Java care iniţializează două variablile întregi cu două valori constante oarecare. În continuare, programul va determina variabila ce conţine valoarea maximă şi va tipări conţinutul ei pe ecran.
3. Scrieţi un program Java care afişează pe ecran numerele impare şi suma numerelor pare cuprinse în intervalul 1-100 inclusiv.

Sfat Pentru a rezolva exerciţiile 2 şi 3 folosiţi doar variabile locale metodei *main*. În Java nu există variabile globale ca în Pascal sau C.

Bibliografie

1. Edward V. Berard, *Abstraction, Encapsulation, and Information Hiding*, http://www.itmweb.com/essay550.htm, 2002.
2. Grady Booch, *Object-Oriented Analysis And Design With Applications*, Second Edition, Addison Wesley, 1997.
3. David Flanagan, *Java In A Nutshell. A Desktop Quick Reference*, Third Edition, O'Reilly, 1999.
4. Mary Shaw, David Garlan, *Software Architecture. Perspectives On An Emerging Discipline*, Prentice Hall, 1996.