# Paradigma programare orientata spre obiect si masina virtuala Java

Lect. Ioan POP
Universitatea „Lucian Blaga" Sibiu

*In this paper are presented the most important aspects of an abstract machine, this time the Java Virtual Machine, and the instructions for the handling of classes, objects, methods, and fields. This manner of the computing processes implementation on a virtual machine is and could be implemented for other programming languages too, even on functional programming languages or logical programming. Implementing a virtual machine is a way of achieving the integration of a programming language in a visual working environment, in a data base working package, in knowledge basis and so on.*
***Keywords:*** *virtual machine, Java Virtual Machine, intermediate language, Oolong assembler, class, object, method.*

## Introducere

Ideea realizarii unei masini virtuale este de a crea un procesor abstract pentru a putea fi implementat pe o mare varietate de calculatoare. Odata implementata pe un computer, aceasta masina virtuala permite scrierea de programe si rularea lor pe orice masina fizica aleasa. Ne vom ocupa de o masina virtuala bazata pe conceptele programarii în limbajul Java, astfel ca ea se va numi masina virtuala Java cu acronimul JVM. De asemenea limbajul de asamblare utilizat pentru aceasta masina este Oolong, limbaj bazat pe Jasmin.

JVM în loc sa furnizeze accesul direct la memorie, la nivel de bits/bytes, trateaza memoria ca o colectie de obiecte. Toate structurile limbajului pentru JVM sunt proiectate pentru a folosi secvente mici de cod, autocontinute, numite obiecte. Obiectele folosesc elemente de baza, precum declaratii si expresii, pentru efectuarea operatiilor cerute de program. Gândim un obiect ca o entitate formata din unitati mai mici. Aceste obiecte nu sunt identice, dar au suficiente caracteristici comune pentru a fi clasate împreuna. Clasele sunt formate din obiecte.

Aceasta *paradigma* este numita programare orientata obiect. Programarea orientata obiect ofera numeroase avantaje printre care si controlul mai bun al memoriei, astfel ca fiecarui program îi este alocat accesul unei parti de memorie. Astfel este permis accesul controlat al sistemului. Dezvoltatorii JVM au creat un set de reguli prin care toate programele pot juca rolul lor. În acest fel se elimina încercarile de a strica sistemul. Setul de reguli este prins într-un algoritm numit verification, ce detecteaza care program respecta setul de reguli si care nu-l respecta. Un scop cheie al verificarii este sa detecteze programele invalide chiar înainte de a fi rulate. Numai atunci, programele luate în considerare de algoritm, vor putea fi rulate.

## Clase si obiecte

Pentru executia unui program Java, JVM furnizeaza suport pentru un numar de baza de constructii orientate obiect.

**1. Obiecte .** Un obiect este o entitate cu patru proprietati: identificatorul, clasa apartinatoare, câmpurile (un slot pentru memorarea de valori; toate obiectele din aceeasi clasa au aceleasi câmpuri; alterarea câmpurilor unui obiect nu afecteaza alt obiect), metodele invocate pe obiect (toate obiectele din aceeasi clasa au acelasi set de metode de invocat pe acestea).

O clasa se poate imagina ca un sablon (model) pentru obiecte. Atunci când un obiect este creat utilizând o clasa ca model, spunem ca am facut o instantiere din clasa. Obiectul are un slot pentru memorarea valorii fiecarui câmp nestatic din clasa. Fiecare instanta a unei clase foloseste aceleasi câmpuri, cu toate ca aceste câmpuri pot pastra valori diferite. Clasa defineste metodele care pot fi invocate pe obiect. Când una din aceste metode este invocata, clasa obiectului determina cu exac-

titate ce set de cod este executat. Exista patru feluri diferite de invocare în JVM, fiecare cu reguli diferite despre felul cum este selectat codul care este executat când o metoda este apelata. Definirea clasei determina ce câmpuri si metode pot fi accesate.

Protectia permite programatorului sa limiteze cantitatea de cod care va fi accesata acestor obiecte, facând programele mai usor de înteles si de depanat, astfel oferind un fundament pe care poate fi construita securitatea sistemului. Adaugam la cele patru proprietati descrise mai sus, a cincea numita un monitor. Monitorul este ca un lacat pe obiect: numai un fir poate avea posesia monitorului la un moment dat.

**1.1. Obiecte si referinte.** Obiectele sunt reprezentate prin referinte si sunt pastrate în Heap. Fiecare obiect este asociat cu o clasa în *Class area*. Fiecare obiect are un numar de sloturi pentru memorarea câmpurilor, astfel ca exista câte un slot pentru fiecare câmp nonstatic în clasa si câte unul pentru fiecare câmp nonstatic în superclasa etc. Heap-ul este locul unde se pastreaza obiectele(figura 1).

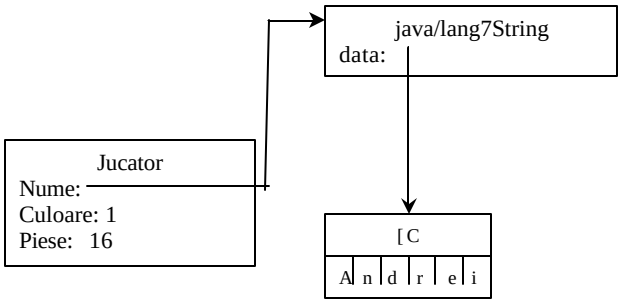


**Fig. 1.** Heap

**1.2. Numele de clasa.** Limbajul Java ne permite sa abreviem numele claselor. JVM cere ca toate clasele sa fie calificate total. Aceasta înseamna ca trebuie sa includ numele întreg al pachetului de fiecare data când mentionez clasa. În JVM clasele sunt numite ca un sir de litere, numere, (-) si (/). Diferenta fata de Java este ca aici se utilizeaza slash-ul (/) ca separator.

**1.3. Pachetele.** Pachetele sunt denumiri conventie pentru a asigura clasele cu nume unice, chiar daca ele vin din diferite grupuri. Pachetele sunt de asemenea utilizate pentru determinarea permisiunii accesului între clase; clasele care sunt în acelasi pachet au mai multe privilegii decât cele care sunt în pachete diferite. Un pachet este o colectie de clase cu acelasi nume din fata ultimului /. De exemplu pachetul din java/lang/String este java/lang/. Tabelul 1 ilustreaza cum se lucreaza cu pachetele.

**Tabelul 1.** Modul de lucru cu pachete

| Nume clasa 1 | Nume clasa 2 | Acelasi pachet? |
|---|---|---|
| *COM/company/Foo* | *COM/company/Bar* | da |
| *COM/company/Grape/Soda* | *COM/company/Cola/Soda* | nu |
| *EDU/school/Cherry/Soda* | *EDU/school/Cherry/Cola/Soda* | nu |
| *Java/lang/String* | *String* | nu |
| *Oyster* | *Clam* | poate fi |

*String*. Aceasta abreviere nu este permisa în JVM si se cere o calificare completa. Asa se elimina ambiguitatile. În general, clasele create ar putea fi numite dupa organizarea pentru lucru. Conventia stabilita pentru aceasta este sa utilizam numele domeniului, Internet, scris în revers:

*COM/sun* ≡ *SumMicrosystems*
*EDU/umd/cs* ≡ *ComputerScienceDepartment, Univ of Maryland*

Daca numele pachetului este omis, atunci clasa este considerata ca parte dintr-un pachet „nenumit”. Sistemul poate avea mai

multe pachete nenumite. Atunci când se determina accesul la câmpuri si metode, doua clase din pachetul fara nume pot sau nu fi tratate ca fiind incluse în acelasi pachet, acest lucru fiind decis de JVM la implementare.

**2. Crearea obiectelor.** Obiectele se creeaza cu instructiunea *new*:

*new ClassName* sau *new java/lang/Integer*.

Clasa nu poate fi marcata *abstract*, dar poti avea permisiunea s-o accesezi. Daca ea este marcata *public*, atunci aveti permisiunea. Altfel aveti permisiunea chiar daca nu este marcata *public*, dar clasa care contine instructiunea *new* sa fie din acelasi pachet ca si clasa care a fost instantiata.

Nu se poate utiliza obiectul pâna când nu a fost initializat. Pentru a initializa un obiect se poate invoca unul din constructorii obiectului. Constructorul este numit *<init>* si returneaza totdeauna *void*.

Exemple de constructori pentru *java/lang/Integer*:

*<init> (I)V; <init> (Ljava/lang/String;)V.* Primul initializeaza *Integer* dintr-un *int*; celalalt foloseste un *String.*

Apelul la constructor sterge referinta la obiectul care a fost construit pe stiva. Deci, exista de cele mai multe ori un *dup* între instructiunea *new* si invocarea constructorului. Constructorii sunt invocati cu *invokespecial.* În ordinea utilizarii *invokespecial*, stiva contine obiecte initializate, urmate de niste argumente la constructor. De exemplu:

```
new java/lang/Integer ; creaza un întreg Integer
dup  ; marcheaza o copie a referintei lui
bipush 27 ; pune un argument
          ; apeleaza constructorul
invokespecial java/lang/Integer/<init> (I)V
```

**3. Constructorii.** În JVM constructorii sunt metode numite *<init>*. Acestia difera fata de limbajul Java, care denumeste constructorii dupa începerea construirii clasei. Fiecare constructor trebuie sa apeleze constructorul clasei de baza undeva într-o metoda. Alternativ poti apela un alt constructor în aceeasi clasa. Compilatorul Java aplica aceasta regula prin a face un apel la *super* sau *this* ca un prim lucru în constructor. La apelul constructorului clasei de baza, se utilizeaza instructiunea *invokespecial*, care este un apel de constructor. Exemple de constructori:

```
    .class MyClass
.super MySuperClass
;un constructor fara argumente
    .method <init> ()V
    aload_o     ;încarca acesta (this)
   invokespecial MySuperClass / <init> () V
   ;apeleaza un constructor superclasa
   ;pe acesta (this)
   return       ;retur la apelant
   .end method
```

*MySuperClass* trebuie sa apeleze constructorul superclasei ei si la fel pâna ajunge la *java/lang/Object.*

Cererea pe care toate clasele trebuie s-o numeasca constructorul superclasei, ne asigura ca nimeni sa nu poata folosi o clasa înainte de a face initializarile. Acesta este cruciala pentru securitatea JVM, pentru ca unele metode depind de initializarile clasei pentru a fi in siguranta.

**4. Utilizarea câmpurilor.** Pentru a citi valoarea unui câmp dintr-un obiect se utilizeaza instructiunea *getfield*. Instructiunea *getfield* asteapta sa gaseasca un obiect de la care sa ia câmpul pe vârful stivei operand. Argumentele instructiunii *getfield* sunt clasa, numele si descriptorul câmpului. Descriptorul este necesar pentru ca algoritmul verification sa poata verifica clasa fara sa aiba de încarcat alt *classfile* pentru a determina tipul câmpului. Clasa definita în Java, pentru exemplificare:

```
Class Greeting (salutare)
{    String intro = „Hello”;  }
```

Se poate lua valoarea acestui câmp cu metoda:

```
method static use Greeating (Lgreeting;)V
aload_0 ;încarca clasa Creeting în variabila 0
;acum aduce câmpul
    getfield Greeting/intro Ljava/lang/String;
;acum sirul Hello este pe vârful stivei
```

Instructiunea *getfield* ia obiectul *Greeting* care este pe vârful stivei si încarca pe stiva valoarea câmpului obiectului, câmp numit *intro*.

Figura 2 prezinta o diagrama înaintea, iar figura 3 dupa executia lui getfield. Diferenta este ca în stiva operand referinta la obiectul *Greeting* se schimba cu referinta la obiectul *String "Hello"*.

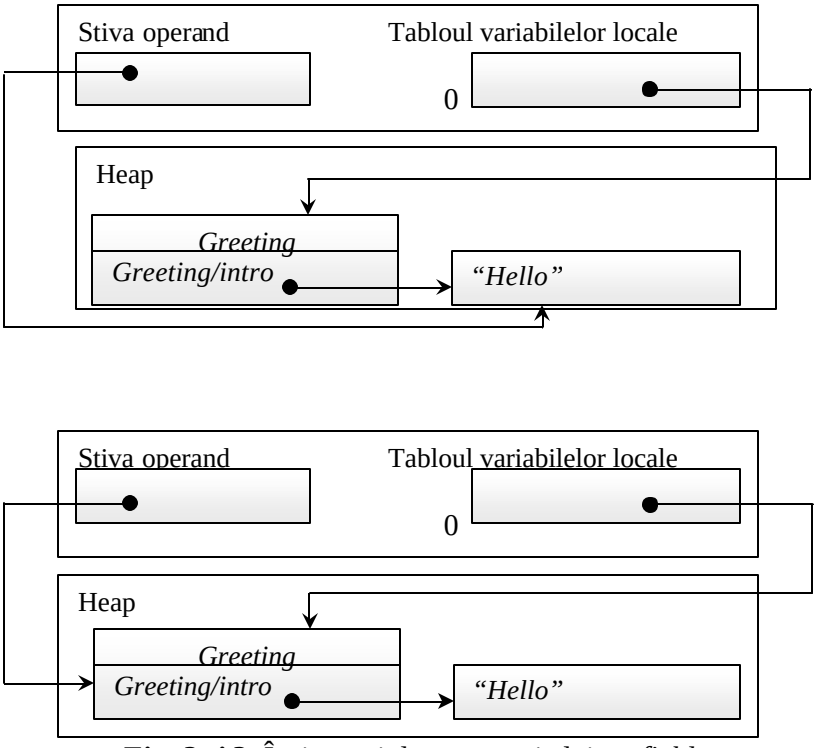


**Fig. 2 si 3.** Înaintea si dupa executia lui *getfield*

**4.1. Tipuri si getfield.** Instructiunea *getfield* cere tipul câmpului care este citit cu numele sau. Acest lucru este adevarat chiar daca câmpul contine nume din tipurile numerice. Presupunem ca avem o clasa *Point*:

```
    Class Point
    {
float x, y;
    }
```

Pentru a aduna valorile câmpurilor x si y utilizam *getfield* astfel:

```
;aceasta metoda ia un Punct si reda suma
;coordonatelor sale
    .method static xySum(LPoint;)F
    aload_0  ;încarca Punctul în variabila 0
getfield Point/x F ;încarca coordonata x pe ;stiva
    aload_0        ;încarca din nou Punctul
getfield Point/y F ;înacrca coordonata y pe ;stiva
    fadd           ;aduna coordonatele
    freturn        ;returneaza rezultatul
    .end method
```

JVM utilizeaza informatia tip în argumente ca parte a procesului de verificare. Câmpurile *x* si *y* sunt de tipul *float*. JVM trebuie sa se asigure ca acestea sunt câmpurile din *Point*, pentru a verifica daca aceste câmpuri au tipurile corecte.

**4.2. Mostenirea câmpurilor.** Atunci când o clasa are o alta clasa ca si superclasa a sa, ea mosteneste toate câmpurile nonstatice ale superclasei. Consideram o extensie la clasa *Greeting* pentru a saluta în româneste:

```
Class RomanianGreeting extends Greeting
{
    String intro = "Salut"
}
```

O instantiere a clasei *RomanianGreetin* are doua câmpuri, ambele numite *intro*. O instantiere a *RomanianGreeting* este prezentata în figura 4.

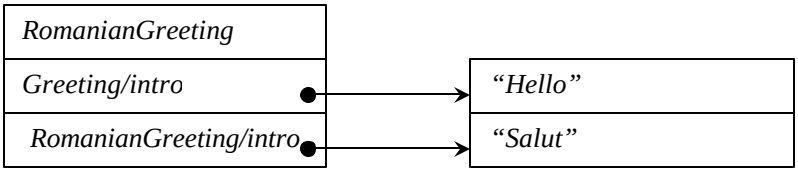


**Fig. 4.** O instantiere a obiectului *RomanianGreeting*

Numele complete ale câmpurilor sunt diferite. În Java, câmpul *intro* din engleza este ascuns înaintea versiunii din româna. În Oolong, acestea sunt ambele accesibile în mod egal:

```
.method static internationalGreetings (LRomanianGreeting;)V
aload_0 ; Aici avem un ;RomanianGreeting, în registrul 0 se pune ;"Salut"
getfield RomanianGreetin/introduction Ljava/lang/String;
    ;; restul metodei este omis
    .end method
```

La sfârsitul codului prezentat, aici vor fi doua obiecte pe stiva. Josul stivei va contine o referinta la obiectul *String "Salut"*, iar deasupra lui va fi o referinta la *String*-ul *"Hello"*.

**4.3. Modificarea valorilor câmpurilor.** Pentru a lua valoarea dintr-un câmp se utilizeaza instructiunea *getfield*, care ia un obiect pe stiva si lasa în locul sau valoarea câmpului. Corespondenta lui *getfield* pentru a schimba valoarea câmpului este instructiunea *putfield* care utilizeaza aceleasi argumente, dar asteapta deopotriva obiectul si valoarea câmpului pe stiva înainte de startul ei. De exemplu:

```
.method static make Australian (Lgreeting;) V
aload_0 ;încarca obiectul Greeting
ldc „G'Day" ; un nou intro setat si memorat
putfield Greeting/intro Ljava/lang/String;
return
```

Aceasta metoda schimba valoarea câmpului *intro* din clasa *Greeting*, daca obiectul este unul ordinar în clasa *Greeting* sau daca e unul dintr-o clasa extinsa acesta ramâne neschimbat.

Presupunem ca am definit clasa *Greeting*. Extindem aceasta clasa pentru a o folosi în limba româna:

```
Class RomanianGreeting extends Greeting
{   String intro = „Salut"   }
```

O parte de cod Java:

```
Greeting g = new RomanianGreeting();
MakeAustralian(g);
System.out.println(g.intro);
System.out.println((RomanianGreeting) g).intro;
```

Listarea arata asa:

```
G'Day
Salut
```

**5. Invocarea metodelor pe obiecte.** Invocarea unei metode dintr-un obiect se poate face în trei moduri: cu instructiunea *invokevirtual*, cu *invokespecial* sau cu *invokestatic*. Vom detalia doar modul în care lucreaza *invokevirtual*. Când se executa *invokevirtual* se creeaza un cadru stiva nou, cu un nou operand stiva si un nou tablou al variabilelor. Stiva este goala iar sloturile din tabloul de variabile locale sunt neinitializate.

Executia continua cu prima instructiune din metoda *invokevirtual*. Noua stiva si fondul de variabile sunt numai cele care pot fi accesate pâna când metoda returneaza sau trimite o exceptie. De exemplu fie clasa:

```
.class RightTriangle
.method sumOfSquares (FF)F
;variabila 0 contine obiectul RightTriangle
;variabila 1 contine primul argument (a)
;variabila 2 contine primul argument (b)
fload_1
dup
fmul ;calculeaza a^2, lasând rezultatul pe stiva
fload_2
dup
fmul ;calculeaza b^2
fadd ;aduna patratele
freturn ;returneaza rezultatul
.end method
```

Aceasta metoda primeste doua valori *float* si reda suma patratelor. Apelul metodei necesita trei lucruri pe stiva, corespunzator initializarii celor trei variabile. Primul lucru numit receptor trebuie sa fie obiectul *RightTriangle* si celelalte doua trebuie sa fie *floats* numite argumente. Acest fragment de cod apeleaza o metoda:

```
;presupunem ca în var 0 e obiectul RightTriangle
aload_0 ;încarca pe stiva obiectul RightTriangle ca receptor
    ldc 3.0.    ;încarca constanta 3.0
    ldc 4.0     ;încarca constanta 4.0
invokevirtual RightTriangle/sumOfSquares (FF)F
    ;rezultatul pe stiva este 25.0
```

În acest moment instructiunea *invokevirtual* este executata, noul cadru stiva este creat si contorul de program pointeaza la prima instructiune din metoda *sumOfSquares*. Stiva este goala. Variabila locala o contine obiectul ei. Variabilele 1 si 2 contin operanzii instructiunii invokevirtual (în acest caz variabilele *float* 3.0 si 4.0). Metoda continua executia instructiunilor pâna când ajunge la instructiunea *freturn*. La acest punct noul cadru stiva este sters si acum suntem în vârful stivei originale. Operandul instructiunii *freturn* este pus pe stiva actuala, adica în top se afla float 25.0.

Metodele statice sunt invocate diferit prin utilizarea instructiunii *invokestatic*, ele nu utilizeaza un receiver numai argumente.

De exemplu dam o alta metoda *RightTriangle* care utilizeaza apeluri atât non-statice cât si statice.

```
    .method hypotenuse (FF) F
    ;var 0 contine un obiect RightTriangle,
    ;var 1 contine (a), var 2 contine (b)
    aload_0         ;încarca obiectul
    fload_1         ;încarca a
    fload_2         ;încarca b
invokevirtual Right Triangle/sumOfSquares (FF) F
    ;calculeaza a^2+ b^2
    f2d    ;converteste float în double
    invokestatic java/lang/Math/sqrt(D) D
    ;calculeaza radicalul
    d2f    ;converteste la float
    freturn
    .end method
```

Dupa invocarea metodei *sumOfSquares* se invoca o alta metoda din clasa *Math(sqrt)* una din clasele platformei standard. Metoda *sqrt* este statica, astfel ca aceasta nu utilizeaza receiver-ul. Dar sunt necesare cele doua conversii.

Când se utilizeaza *invokevirtual*, definitia metodei care este numita în argumente nu este neaparat necesara. În schimb JVM utilizeaza o procedura dispecer virtual pentru a selecta metoda care trebuie apelata bazata pe numele ei, descriptorul metodei si tipul receptorului în runtime.

Casting-ul pentru câmpurile si metodele utilizate este asemanator cu cel din Java.

**Concluzii**

Realizarea unei masini virtuale Java care lucreaza cu obiecte, asa cum a fost prezentata mai sus, constituie un model de implementare si pentru alte limbaje de programare nu numai imperative, dar si limbaje de inteligenta artificiala. Conceptul de limbaj intermediar utilizat în crearea unei masini abstracte încadreaza foarte bine setul de instructiuni în conceptul de fisier "bytecodes", modelat pentru platformele care lucreaza în aceasta maniera.

Integrarea unei implementari într-un mediu software vizual, ca instrument de lucru pentru prelucrarea bazelor de date, a bazelor de cunostinte, îmbunatateste substantial modalitatea de a aborda probleme care implica portabilitatea, integrarea si de ce nu interoperabilitatea.

**Bibliografie**

[1]. Joshua Engel – Programming for the Java Virtual Machine, ADDISON WESLEY, 2000.

[2]. Norton P., Stanek W. – Ghid de programare Java, Teora, 1997.

[3]. Popovici D.P si Popovici I.P. C++ Tehnologia orientata spre obiecte, Teora, 2000.

[4]. Archer T., "Inside C#", Microsoft Press, 2001.

[5]. http://java.sun.com/docs

[6]. http://msdn.microsoft.com/msdnmag/netindex.asp