Laborator 1 Java - Introducere

Platforma Java

Structura unui program
- Functia “main”
- Numele canonic al unei clase

Structura unui fisier .java
- Declaratia de pachet
- Directive de import
- Decralatii/Definitii de clase
- Comentariile

Structura unei clase
- Declararea de variabile (locale sau in corpul clasei)
- Primitive si obiecte
- Declararea metodelor

Structuri de control
- Switch
- While
- Do-While
- For
- Break
- Continue
- Return

Platforma Java

Cand spunem Java nu ne referim doar la un limbaj de programare. Java este o serie de unelte care ne sunt puse la dispozitie pentru a programa un dispozitiv (nu neaparat un PC) sa faca ceva.
In multe limbaje, programele scrise sunt transformate in cod masina, adica in instructiuni specifice procesorului si masinii pe care programul a fost gandit sa lucreze. In alte limbaje, programele scrise sunt interpretate direct in timpul rularii de catre un interpretor. Limbajul Java este undeva la mijloc. Codul este scris ca text si este compilat, transformat in ceva ce numim “byte code”. Acest “byte code” este forma intermediara, pe care un “interpretor” este capabil sa il execute fie direct fie sa il transforme in cod masina inainte de executie. Acest “interpretor” este o aplicatie (pe Windows este java.exe) care este rulata ca orice alta aplicatie cu niste parametri

convenabili, care sa ii spuna unde se afla “byte code”ul pe care vrem sa il rulam. Aceasta aplicatie porneste un proces numit **“Java Virtual Machine” (JVM)**. De fapt de aici vine toata magia Java.
Fisierele de cod Java sunt simple fisiere text, desigur, cu o structura specifica limbajului. Aceste fisiere sunt transformate in niste fisiere “class”. Aceste fisiere “class” pot fi apoi incarcate si executate pe un **JVM**. Asa cum un compilator de C++ de exemplu produce un .exe care poate fi executat pe o anumita arhitectura, compilatorul de java (o sa ii spun **javac**, plecand de la numele executabilului care face aceasta treaba) produce unul (sau mai multe!) fisiere .class care pot fi executate pe o anumita masina virtuala java. Din moment ce masina este virtuala, este un program in fapt, codul pe care noi il compilam in java (acele fisiere .class) putem sa le executam direct pe un JVM sub Windows sau sub Linux (de exemplu) **fara sa recompilam** programul. Teoretic, acelasi program poate fi rulat inclusiv pe telefoane mobile, televizoare, chiar si ….masini de spalat rufe, singura conditie este ca acel dispozitiv sa poata sa porneasca o masina virtuala cu acea arhitectura. In practica insa, din motive tehnice, JVMurile care ruleaza pe PC, pe telefoane, televizoare sau masini de spalat rufe difera considerabil astfel incat rareori un program poate fi rulat direct pe oricare din aceste dispozitive.
Deja am numit doua unelte care exista in platforma java: un compilator si un executabil (care ne lanseaza JVMul). Pe langa acestea doua, mai avem si multe altele care sunt capabile sa indeplineasca diverse taskuri specifice (avansate) si o sumedenie de clase (librarii).
In concluzie, cand spunem Java, ne gandim la:

1. Un limbaj de programare
2. O platforma de executie
3. Un grup de unelte pentru diverse sarcini (compilator de exemplu, dar si multe altele)
4. O librarie de cod (cod scris de baieti destepti si cu experienta, care functioneaza bine si repede)

Cand vrem sa descarcam “Java” de fapt vrem sa descarcam unul din doua lucruri:

1. Un JRE (Java Runtime Environment) care contine masina virtuala si doar ce este absolut necesar pentru a rula un program deja compilat (adica sa poata rula fisiere .class)
2. Un JDK (Java Development Kit) care contine un JRE si multitudinea de unelte despre care am vorbit.

Structura unui program

Spre deosebire de programele in C (sau alte limbaje cu care ati lucrat pana acum), Java nu permite cod in afara unei clase. Asa cum stiti probabil, obiectele nu pot fi folosite pana nu sunt instantiate, si nu prea avem cum sa le instantiem pana nu pornim un program. Pentru a iesi din acest paradox, pornirea unui program se face folosind niste reguli mai speciale.
In primul rand, nu avem la dispozitie un executabil pe care sa il rulam, “exe”ul nostru este de fapt un fisier .class (in practica sunt de fapt o sumedenie de fisiere .class sau o arhiva de astfel de fisiere despre care voi vorbi mai tarziu). Ne reamintim astfel ca, asa cum am spus mai sus,

orice program java se porneste apeland executabilul java.exe (WINDOWS, in Unix este similar) cu parametri convenabili. De exemplu:

```
java.exe pachet.subpachet.NumeClasa parametrul1 parametrul2
```

Java.exe este un fisier pe care il gasiti in directorul “bin” din directorul unde s-a instalat JREul (sau JDKul, dupa caz).

Tot ceea ce urmeaza numelui acestui executabil nu sunt altceva decat parametrii pe care ii primeste excutabilul. Primul parametru este “numele canonic” al clasei unde se afla metoda magica **“main”**. Numele canonic este format din numele **“pachetului”** in care se afla clasa, simbolul “.” si numele clasei. Atentie, Java este case sensitive! “parametrul1” si “parametrul2” sunt trimisi apoi functiei voastre main.

Functia “main”

Functia **“main”** este de fapt foarte similara cu ceea ce voi ati facut in C/C++, singura diferenta este faptul ca trebuie declarata intr-o clasa si, pentru a fi executata de java.exe, trebuie sa aiba urmatoarea **semnatura**:

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

Mare mare atentie: “main” nu este o functie magica si este nimic special la ea! Asta inseamna ca putem avea de exemplu private main(int a) fara sa avem probleme, dar acel main nu va reprezenta un punct de intrare in program (mai corect: nu va putea fi apelata de catre java.exe). “parametrul1” si “parametrul2” (dati in linia de comanda) la java.exe, pot fi cititi in main din args[0] si args[1] respectiv.

Numele canonic al unei clase

Numele canonic al clasei este ceva spefic Java (si limbajelor similare, dar nu exista in C/C++ php etc). In Java, clasele sunt organizate in pachete, subpachete (termenul nu este folosit prea des, preferat fiind doar pachet). Conceptul poate este analog cu structura fisiere (analog clasa) si directoare (analog pachet). In fapt, aceasta structura declarativa (pachet/clasa) este vizibila direct in structura de fisiere. Adica, daca avem un fisier in care avem declarata o clasa publica “NumeClasa” care se afla in pachetul “pachet.subpachet” atunci, obligatoriu:

1. Fisierul se va numi NumeClasa.java
2. Fisierul se va afla intr-un folder numit “subpachet”
3. Folderul “subpachet” in care se afla NumeClasa.java se va afla intr-un folder “pachet”

In momentul in care un program este compilat cu **javac**, toate fisierele .java devin fisiere .class, dar structura directoarelor ramane! Deoarece in dezvoltare nu ne vom lovi aproape niciodata de nevoia de a compila manual un fisier .java, nu va voi spune cum se face asta folosind **javac** dar este bine sa stiti ca, prin intermediul altor unelte cum ar fi un **IDE(Integrated Development Environment)** sau un **build tool** se ajunge tot la executia lui **javac**, chiar daca nu va dati seama.

Revenind la java.exe, ***o greseala frecventa*** pe care o fac incepatorii este ca incearca sa ruleze fisierul .class din folderul in care acesta se afla. In momentul in care apelati java.exe (din folderul curent), pentru a rula pachet.subpachet.NumeClasa, java.exe va intra (din directorul curent) in directorul pachet, apoi in directorul subpachet si abia aici va incerca sa gaseasca

fisierul NumeClasa. Deci daca voi rulati java.exe direct din subdirectorul “subpachet” java.exe nu poate trece de pasul1 si 2 si veti primi eroare.
Informatie optionala: De fapt, aceasta procedura se repeta pentru toate locatiile definite in classpath. Classpath este un alt parametru al executiei lui java.exe si este similar lui “PATH” din sistemul de operare. Spre deosebire de PATH, in classpath putem avea si nume de fisiere “.jar” (Java ARchive) care sunt de fapt o colectie de clase organizate in foldere si subfoldere in functie de pachetele lor. Deci algoritmul descris mai sus se aplica pentru fiecare element din classpath pana cand clasa este gasita sau nu mai sunt elemente in classpath. In mod implicit, directorul curent este in classpath, deci vedeti, nu e nimic magic.
Odata pornit un program, prin apelul lui **main** de catre java.exe, in functie de ceea ce se afla in main programul poate sa se ramifice considerabil, poate sa incarce alte clase si sa apeleze codul din ele. Astfel pot fi construite aplicatii complexe.

Structura unui fisier .java

Un fisier java este structurat in felul urmator:

1. Declaratia de pachet (optional)
2. Directive de import (optional)
3. Definitii de clase

Declaratia de pachet

Pachetele au in primul rand rolul de a structura codul, de a il organiza. Este greu sa exemplifici , fara a arata o aplicatie uriasa, utilitatea pachetelor. In cadrul laboratoarelor vom incerca sa exersam plasarea claselor in pachete potrivite folosind urmatoarea regula: *clasele care indeplinesc aceeasi functie de baza sunt plasate in acelasi pachet. Daca mai multe pachete care indeplinesc functii simple sunt folosite pentru a indeplini o functie complexa vor fi toate in acelasi pachet.*
Retinem insa: daca declaratia de pachet lipseste atunci clasa definita in acel fisier este pusa in un pachet “magic” numit “default package” adica nu are niciun folder, trebuie sa fie direct in *classpath (vezi info optional mai sus)* . Folositi in toate programele voastre o declaratie de pachet, chiar daca e ceva stupid cum ar fi:

```
package pachet;
```

Directive de import

In practica, toate clasele pe care le veti folosi vor face referire la alte clase din alte pachete. De exemplu, sa spunem ca vrem sa folosim clasa noastra “NumeClasa” in o alta clasa numita “AltaClasa”. Java trebuie cumva sa stie de unde sa citeasca fisierul “NumeClasa.class” (ne reamintim de structura de directoare de pe disc) iar in cazul in care exista mai multe NumeClasa (de ce nu?) trebuie sa stie la care ne referim. Din acest motiv, o clasa trebuie folosita cu numele ei intreg (numele canonic) adica “pachet.subpachet.NumeClasa”. Exemplu:

```
pachet.subpachet.NumeClasa instanta = new pachet.subpachet.NumeClasa();
```

In cazul in care nu folosim numele canonic al clasei, complilatorul va presupune ca acea clasa se afla exact in pachetul/subpachetul curent sau in unul in pachetele/clasele importate. Deci, daca vrei ca:

```
NumeClasa instanta = new NumeClasa();
```

sa functioneze, fie clasa in care apare acest cod se alfa in pachetul “pachet.subpachet” fie avem (oricare) din urmatoarele directive de import:

```
import pachet.subpachet.NumeClasa;
import pachet.subpachet.*;
```

Atentie!

```
import pachet.*;
```

NU va functiona!

Informatie optionala: Exista si directive statice de import, puteti sa ma intrebati ce e cu ele. Vom folosi foarte des clasa “String” de exemplu, care este o clasa si nu se afla in pachetul default. Motivul pentru care compilatorul nu protesteaza este ca “String” se afla intr-un pachet pe care compilatorul il importa automat, chiar daca noi nu scriem nicio directiva de import. Directivele de import NU incarca suplimentar programul!

Decralatii/Definitii de clase

Ca regula generala, in Java, fiecare clasa este definita ca publica si intr-un fisier care ii poarta numele, in folderul corespunzator pachetului sau. Exista si exceptii, de exemplu putem avea mai multe clase non publice in acelasi fisier, care sunt vizibile doar din pachetul respectiv dar practica este puternic descurajata si voi insista foarte mult sa respectati in timpul laboratorului “regula generala” (*practic, in toata experienta mea de programare in Java nu am intalnit un caz in care sa fie nevoie de astfel de clase)*.

Comentariile

Comentariile in Java pot sa apara oriunde intr-un fisier.

```
// comment
/*
Comentariu
Pe
Mai
Multe
Linii
*/
```

Informatie optionala: Java are niste comentarii speciale numite “JavaDoc”, care sunt vizibile in IDEuri si ne vor ajuta enorm la development. Daca doriti detalii, puteti sa ma intrebati.

Structura unei clase

Clasa (publica) este definita in felul urmator:

```
public class NumeClasa
{
// corp clasa
}
```
In interiorul clasei vor fi definite campuri, metode sau alte clase (numite **clase interne** sau **inner class**).
Toate aceste 3 elemente pot fi marcate cu urmatorii modificatori:

1. public/private/protected sau niciunul din acestia (se considera tipul de protectie “default”)
2. static (optional)

Veti primi detalii suplimentare in cursurile/laboratoarele care urmeaza, mentan retinem doar structura.
Campurile sunt in fapt declaratii de variabile cu **“scope”** clasa curenta.
Metodele sunt declaratii de functii cu **“scope”** clasa curenta.
Clasele interne similar. Clasele interne vor fi un subiect complex abordat ulterior.
Informatie optionala: clasele pot contine si blocuri de cod anonime, daca doriti detalii, puteti sa ma intrebati dupa laborator.

Declararea de variabile (locale sau in corpul clasei)

Declaratiile de variabile (atat in corpul clasei cat si in functii) au urmatoarea forma, cu mentiunea ca in corpul clasei pot avea si cei doi modificatori de acces.

```
int numeIntreg; // declara o variabila de tip int
String numeString; // in Java String este o clasa aparte
NumeClasa numeObiect; // Declara un obiect de tipul NumeClasa
int[] vector; // declara un vector de intregi;
```

De retinut ca in Java nu mai avem pointeri expliciti, ca in C++. Pointerii exista insa si sunt impliciti, pentru anumite tipuri de date. In general se spune ca Java nu are pointeri, dar acest lucru este doar o aparenta datorata faptului ca managementul memoriei este automat.
Datele cu care lucreaza programul sunt de doua tipuri generale: primitive sau obiecte (in fapt, pointeri la obiecte).

Primitive si obiecte

Primitivele sunt acele tipuri referite de obicei prin cuvinte cheie (int, double, float, char, byte, long, etc). In acest caz, memoria alocata pentru ele este suficienta cat pentru a tine datele repective efective iar datele sunt stocate “in variabila”.
Obiectele sunt acele tipuri referite de obicei prin nume de clase. In acest caz, variabila este in fapt un pointer catre o locatie de memorie unde sunt stocate acele date.
Putem sa ne dam seama de tipul datelor folosind niste exemple:

```
int a = 5;
```

Declaram o variabila de tip int si ii asignam valoarea 5.

```
Integer a = new Integer(5);
```
Declaram o variabila de tip Integer si ii asignam valoarea “adresa unui nou obiect pe care il construim”. Observati cuvantul cheie “new”. Ca si in C++ (sau malloc in C), new aloca spatiu de memorie pentru stocarea unui obiect de tipul corespunzator. Deci acea atribuire este de fapt o atribuire de pointeri. Lucrurile devin mult mai evidente cand facem:
```
int[] a = new int[10];
```
In java, transferul parametrilor este de fiecare data prin valoare (adica se face o copie si modificare in interiorul functiei apelate nu este vizibila in exterior). Deci daca avem primul caz , int a = 5 si trimitem valoarea lui “a” ca parametru unei alte functii, indiferent de facem in functia respectiva, a va ramane 5 in functia noastra.
In mod similar se intampla lucrurile si cand trimitem ca parametru o variabila de tip obiect, desi am fi tentati sa spunem ca este alt tip de apel. Parametrul in acest caz este de fapt un pointer la o adresa de memorie unde e stocat obiectul respectiv, si acea adresa nu poate fi modificata, indiferent ce facem, dar obiectul de la acea adresa poate fi modificat. Sa facem urmatorul experiment.
```
public class NumeClasa
{

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

public void functieA()
{
      int a=5;
      int[] b = new int[] { 1,2,3,4,5};
      System.out.println(a);
      System.out.println(b[2]);
      functieB(a,b);
      System.out.println(a);
      System.out.println(b[2]);
}
public void functieB(int x, int[] y)
{
      x = 7;
      y[2] = 100;
}
}
```
Observam cum “a” isi pastreaza valoarea, iar “b” pare sa isi schimbe valoarea desi ambele sunt trimise functiei in mod similar. Realitatea este insa ca x este o valoare (primitiva) iar y este un pointer catre un array.

Declararea metodelor

Metodele (se mai numesc in mod “gresit” functii) sunt fragmente de cod care indeplinesc anumite functii folosind datele primite in lista de parametri formali, variabilele membru sau din alte surse. Decralarea de metoda are urmatoarele parti:

1. O lista de modificatori (optionali toti), pot fi public/protected/private, static sau abstract
2. Declaratii de generic type (*concept avansat, vom vedea ulterior*) (optional)
3. Un tip de date returnat (void daca nu returneaza nimic), obligatoriu
4. Numele functiei, obligatoriu
5. Lista parametrilor formali (obligatoriu, chiar daca e goala).
6. “;” sau “{ corp }”, adica metoda poate avea corp sau poate fi abstracta. *Vom vorbi mai tarziu despre metode abstracte, la mostenire.* In principiu metodele trebuie sa aiba corp.

Corpul functiei contine instructiunile executate la un apel al functiei.
Metodele care au return type diferit de null **trebuie** sa contina un apel de return cu tipul respectiv.

O metoda speciala este **constructorul**. Aceasta metoda este apelata automat atunci cand o clasa este instantiata (sau construita) si “returneaza” un pointer la clasa construita.
El se diferentiaza de alte metode prin faptul ca:

1. Nu poate fi static
2. Nu are return type
3. Are exact acelasi nume cu al clasei.

Practic, constructorul este apelat atunci cand facem “new” la un obiect, iar daca are nevoie de parametri poate sa ii primeasca tot atunci. Exemplu:

```
Public class Test
{
Public Test()
{
}
Public Test(int a)
{
}
Public static void main(String[] args)
{
Test testA = new Test(); // primul constructor
Test testB = new Test(5); // al doilea constructor.
}
}
```

Orice clasa are implicit un constructor public si fara argumente. In cazul in care este declarat manual un alt constructor (indiferent de forma) constructorul implicit dispare.

Structuri de control

If-else

Structura unui bloc if este:

```
if (conditie)
      Instructiune; //1
else
      Instructiune; //2
```

Conditie este o expresie care trebuie sa fie evaluata la un tip de date “boolean” (adica true sau false) si este obligatoriu.
“instructiune” este obligatoriu.
“else” este optional iar in cazul in care lipseste instructiunea lui va lipsi se asemenea.
Conditie poate fi orice espresie, adica si un apel de functie de exemplu (care returneaza boolean)
Instructiune poate fi un “bloc” adica o secventa de instructiuni inconjurate de “{“ si “}”
In cazul in care conditia este evaluata la true, se executa instructiunea 1. Altfel, daca exista, se executa instructiunea 2.

Switch

Structura:

```
Switch (variabila)
{
case constanta1:
      instructiune1;
      break;
case constanta2:
      Instructiune2;
      break;
case constanta3:
      Instructiune3;
case constanta4:
default :
      instructiune5;
      break;
}
```

Variabila este obligatorie si este comparata succesiv cu constanta1, 2, 3 si 4.
Daca este egala cu constanta1, se executa instructiunea 1. Break previne executia urmatoarelor instructiuni

Daca este egala cu constanta2, se executa instructiunea 2. Break previne executia urmatoarelor instructiuni
Daca este egala cu constanta3, se executa instructiunea 3 si, pentru ca break lipseste, toate instructiunile pana la break (adica si instructiunea 5)
Daca este egala cu constanta4, se executa toate instructiunile pana la break, adica instructiune5.
Daca nu este egal cu nici una din constante, se executa instructiunea 5.
Este obligatoriu sa avem cel putin un case. Default este optional, iar daca lipseste si valoarea nu se potriveste cu nicio constanta atunci nu se executa nimic.

While

Structura:

```
while(conditie)
      instructiune;
```

Conditie este obligatoriu si este o expresie care este de tipul boolean. Se evalueaza conditia, daca e true se executa instructiunea, daca e false se trece peste while. Se repeta procesul pana cand conditia devine false.

Do-While

Structura:

```
do
{
instructiune;
}
while(conditie);
```

Se executa instructiune apoi se evalueaza conditie. Conditie trebuie sa fie o expresie care sa fie de tipul boolean. Daca se evalueaza ca false, se trece mai departe, daca este true se executa din nou instructiunea si se reevalueaza conditia.

For

Exista doua forme ale lui for:

```
for(instructiune1;conditie;instructiune2)
      Instructiune3;
```

Unde doar conditie este obligatoriu si trebuie sa se evalueze la un boolean.
For functioneaza asa:

1. Se executa instructiune 1
2. Se evalueaza conditie, daca e true, mergi la pasul urmator, daca nu iesi din for.
3. Se executa instructiune3
4. Se executa instructiune 2
5. Sari la pasul 2.

O a doua forma este (cunoscuta si ca foreach):

```
for(Object object : iterable)
```

```
    instructiune;
```

iterable trebuie sa fie un obiect care implementeaza interfata Iterable.
Pentru fiecare valoare rezultata de iteratorul peste Iterable se va salva acea valoare in variabila locala “object” si se va executa instructiunea; Se iese din bucla atunci cand se obtine ultima valoare de la iterator.

Break

Break este folosit pentru a iesi imediat dintr-o bucla sau dintr-un switch. Executia continua dupa bucla/switch.

Continue

Continue este folosit pentru a intrerupe executia unui bloc de instructiuni dintr-o bucla si a forta reverificarea imediata a conditiei.

Return

Return este folosit pentru a intrerupe imediat executia unei functii si eventual pentru a returna o valoare. Functiile void nu au voie sa returneze valori (dar pot contine return) in timp ce functiile care nu sunt void trebuie sa contina return cu tipul respectiv; Exemple:

```
return; // pentru functiile void, optional
return 5; // pentru o functie cu return type int, obligatoriu.
```