Java basics

- Responsabil: Laurențiu Stamate
- Data publicării: 02.10.2017
- Data ultimei modificări: 07.10.2017

Organizarea unui proiect Java

În cadrul acestui laborator veți avea de ales între a folosi IntelliJ IDEA (tutorial instalare) folosind contul de mail de la facultate (tutorial activare) sau Eclipse (tutorial instalare). Primul pas este să vă familiarizați cu structura unui proiect și a unui fișier sursă Java.

Înainte de a începe orice implementare, trebuie să vă gândiți cum grupați logica întregului program pe unități. Elementele care se regăsesc în același grup trebuie să fie **conectate în mod logic**, pentru o ușoară implementare și înțelegere ulterioară a codului. În cazul Java, aceste grupuri logice se numesc **pachete** și se reflectă pe disc conform ierarhiei din cadrul proiectului. Pachetele pot conține atât alte pachete, cât și fișiere sursă.

Următorul pas este delimitarea entităților din cadrul unui grup, pe baza unor trăsături individuale. În cazul nostru, aceste entități vor fi clasele. Pentru a crea o clasă, trebuie mai întâi să creăm un fișier aparținând proiectului nostru și unui pachet (dacă este cazul și proiectul este prea simplu pentru a-l împărți în pachete). În cadrul acestui fișier definim una sau mai multe clase, conform urmatoarelor reguli:

- dacă dorim ca această clasă să fie vizibilă din întreg proiectul, îi vom pune specificatorul **public** (vom vorbi despre specificatori de acces mai în detaliu în cele ce urmează); acest lucru implică însă 2 restricții:
  - fișierul și clasa publică trebuie să aibă același nume
  - nu poate exista o altă clasă/interfată publică în același fișier (vom vedea în laboratoarele următoare ce sunt interfețele)
- pot exista mai multe clase în același fișier sursă, cu condiția ca **maxim una** să fie publică

Pentru un exemplu de creare a unui proiect, adăugare de pachete și fișiere sursă, consultați acest link pentru IntelliJ Idea si acest link pentru Eclipse.

Tipuri primitive

Conform POO, **orice este un obiect**, însă din motive de performanță, Java suportă și tipurile de bază, care nu sunt clase. Aceste tipuri, numite **tipuri primitive**, corespund tipurilor de date cel mai des folosite și sunt prezentate mai jos, alături de spațiul ocupat și intervalul corespunzător de valori:

Tip primitiv	Dimensiune	Minim	Maxim
`boolean`	<dimensiune variabilă>	—	—
`char`	16	Unicode 0	Unicode $2^{16} - 1$
`byte`	8	-128	127
`short`	16	-2^{15}	$2^{15} - 1$

Tip primitiv	Dimensiune	Minim	Maxim
`int`	32	-2^{31}	$2^{31} - 1$
`long`	64	-2^{63}	$2^{63} - 1$
`float`	32	IEEE754	IEEE754
`double`	64	IEEE754	IEEE754

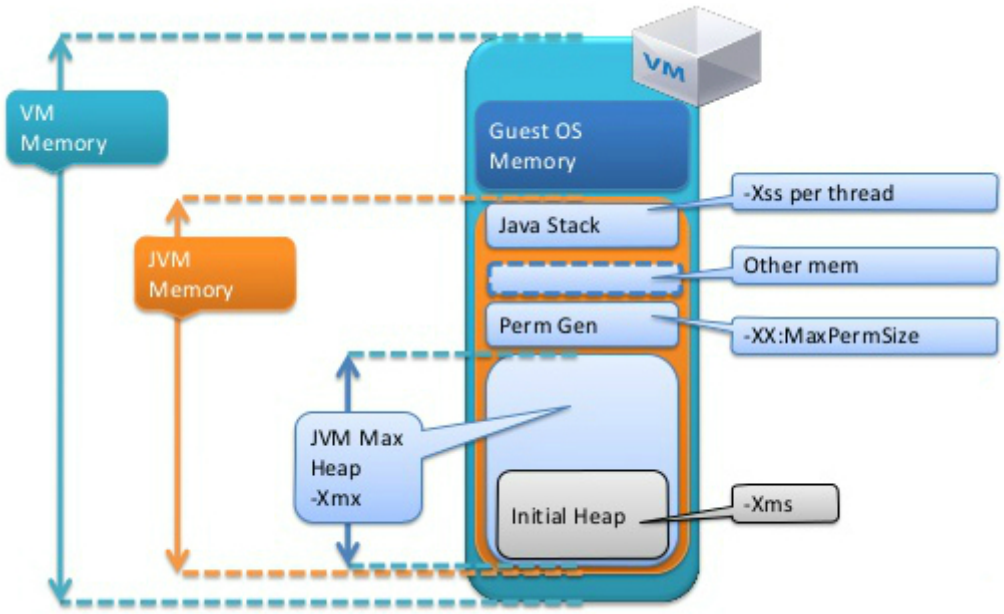
Tab. 0: Tipuri primitive de date

Tipurile de date primitive sunt asemănătoare celor din C. Variabilele (locale metodelor, a nu se confunda cu membri claselor) având tipuri primitive sunt alocate pe stivă (exact ca în C), în contrast cu instanțele claselor, care sunt alocate pe heap (remember stack vs. heap).

Se observă că:

- Java **nu** posedă tipuri `unsigned`
- tipul `char` este pe 16 biți și întrebuințează **Unicode**
- tipul `boolean` nu are o dimensiune fixă
- deși valorile posibile sunt doar 0 și 1, ocupând 1 bit, acestui bit i se adaugă un header dependent de mașină, după care acesta se completează cu biți până se ajunge la un număr multiplu de 8
- pentru mai multe detalii: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
- `void` **nu este tip in Java**, este doar un cuvânt cheie pentru cazurile în care dorim să indicăm că ceea ce se returnează este **nimic**
- din documentația 'Java Language specification 7': “ Note that the Java programming language does not allow a “cast to void” - void is not a type ”

Segmentele de memorie inițiale, precum și parametri de rulare sunt ilustrați mai jos.



Exemplu de declarație:

```
int i, j;
boolean k;
```

Celelalte tipuri existente sunt **clase**. Instanțele claselor sunt tipuri **referință**. Acest lucru înseamnă că în spate, mașina virtuală Java lucrează cu pointeri la obiecte și noi folosim pointeri impliciți către zone de memorie alocate pentru obiectele utilizate. Acest lucru va avea consecințe mai târziu, când vom studia transferul parametrilor în Java.

Care este diferența dintre referințe (Java) și pointeri (C) ?

- referințele pot ascunde un anumit număr de niveluri de indirectare deasupra pointerilor
- din cauza faptului că nu putem accesa direct memoria, **nu putem face operații aritmetice cu referințele**, spre deosebire de pointeri
- datorită libertății oferite de pointerii din C, putem face cast oricărui pointer pentru a da un alt sens zonei respective de memorie; acest lucru nu este posibil in cazul referințelor, se poate face cast doar la un alt tip al unui obiect care este deja încapsulat în obiectul curent (mai multe detalii când vom discuta despre moștenire)

Acum e de reținut faptul că în Java **nu** există pointeri expliciți. Această facilitate a fost considerată generatoare de erori și nu a fost implementată. Acest lucru nu limitează capacitățile platformei.

Clase

Clasele reprezintă tipuri de date definite de utilizator sau deja existente în sistem (din `class library` - set de biblioteci dinamice oferite pentru a asigura portabilitatea, eliminând dependența de sistemul pe care rulează programul). O clasă poate conține:

- **membri** (`variabile membru` (**câmpuri**) și `proprietăți`, care definesc starea obiectului)
- **metode** (`functii membru`, ce reprezintă operații asupra stării).

Prin instanțierea unei clase se înțelege crearea unui obiect care corespunde unui șablon definit. În cazul general, acest lucru se realizează prin intermediul cuvântului cheie new.

Biblioteca Java oferă clase **wrapper** (“ambalaj”) pentru fiecare tip primitiv. Avem astfel clasele `Char`, `Integer`, `Float` etc. Un exemplu de instanțiere este următorul:

```
new Integer();
```

Procesul de inițializare implică: declarare, instanțiere și atribuire. Un exemplu de inițializare este următorul:

```
Integer myZero = new Integer();
```

Un alt exemplu de clasă predefinita este clasa `String`. Ea se poate instanția astfel (**nu** este necesară utilizarea new):

```
String s1, s2;

s1 = "My first string";
s2 = "My second string";
```

Această este varianta preferată pentru instanțierea string-urilor. De remarcat că și varianta următoare este corectă, dar **ineficientă**, din motive ce vor fi explicate ulterior.

```
s = new String("str");
```

Câmpuri (membri)

Un câmp este un obiect având tipul unei clase sau o variabilă de tip primitiv. Dacă este un obiect atunci trebuie inițializat înainte de a fi folosit (folosind cuvântul cheie new).

```
class DataOnly {
    int i;
    float f;
    boolean b;
    String s;
}
```

Declarăm un obiect de tip `DataOnly` și îl inițializăm:

```
DataOnly d = new DataOnly();

// set the field i to the value 1
d.i = 1;

// use that value
System.out.println("Field i of the object d is " + d.i);
```

Observăm că pentru a utiliza un câmp/funcție membru dintr-o funcție care nu aparține clasei respective, folosim **sintaxa**:

```
classInstance.memberName
```

Proprietăți

O proprietate este un câmp (membru) căruia i se atașează două metode ce îi pot expune sau modifica starea. Aceste doua metode se numesc `getter` si `setter`.

```
class PropertiesExample {
    String myString;

    String getMyString() {
        return myString;
    }

    void setMyString(String s) {
        myString = s;
    }
```

```
}
```

Declarăm un obiect de tip `PropertiesExample` și îi inițializăm membrul `myString` astfel:

```
PropertiesExample pe = new PropertiesExample();

pe.setMyString("This is my string!");

System.out.println(pe.getMyString());
```

Metode (funcții membru)

Putem modifica programul anterior astfel:

```
String s1, s2;

s1 = "My first string";
s2 = "My second string";

System.out.println(s1.length());
System.out.println(s2.length());
```

Va fi afișată lungimea în caractere a șirului respectiv. Se observă că pentru a aplica o funcție a unui obiect, se folosește sintaxa:

```
classInstance.methodName(param1, param2, ..., paramN);
```

Funcțiile membru se declară asemănător cu funcțiile din C.

Specificatori de acces

În limbajul Java (şi în majoritatea limbajelor de programare de tipul OOP), orice clasă, atribut sau metodă posedă un **specificator de acces**, al cărui rol este de a restricţiona accesul la entitatea respectivă, din perspectiva altor clase. Există specificatorii:

- **`public`** - permite acces complet din exteriorul clasei curente
- **`private`** - limitează accesul doar în cadrul clasei curente
- **`protected`** - limitează accesul doar în cadrul pachetului, clasei curente si a celor care o mostenesc (conceptul de *descendenţă* sau de *moştenire* va fi explicat mai târziu)
- **(default)** - în cazul în care nu este utilizat explicit nici unul din specificatorii de acces de mai sus, accesul este permis doar în cadrul *pachetului* (package private). Atenţie, nu confundaţi specificatorul default (lipsa unui specificator explicit) cu `protected`!

Utilizarea specificatorilor contribuie la realizarea ***încapsulării***.

Încapsularea se referă la acumularea atributelor şi metodelor caracteristice unei anumite categorii de obiecte într-o clasă. *Pe de altă parte, acest concept denotă şi ascunderea informaţiei de stare internă a unui obiect, reprezentată de atributele acestuia, alături de valorile aferente, şi asigurarea comunicării strict prin intermediul metodelor* (*interfata* clasei).

Acest lucru conduce la izolarea modului de implementare a unei clase (atributele acesteia şi cum sunt manipulate) de utilizarea acesteia. Utilizatorii unei clase pot conta pe funcţionalitatea expusă de aceasta, **indiferent de implementarea ei internă** (chiar şi dacă se poate modifica în timp). Dacă utilizatorii ar avea acces la modul efectiv de implementare a unei clase, ar fi imposibilă modificarea implementării ei (necesitate care apare des în practică) fără un impact lateral asupra utilizatorului.

Exemplu de implementare

Clasa `VeterinaryReport` este o versiune micșorată a clasei care permite unui veterinar să țină evidența animalelor tratate, pe categorii (câini/pisici):

```
public class VeterinaryReport {
    int dogs;
    int cats;

    public int getAnimalsCount() {
        return dogs + cats;
    }

    public void displayStatistics() {
        System.out.println("Total number of animals is " +
getAnimalsCount());
    }
}
```

Clasa `VeterinaryTest` ne permite să testăm funcționalitatea oferită de clasa anterioară.

```
public class VeterinaryTest {
    public static void main(String[] args) {
        VeterinaryReport vr = new VeterinaryReport();

        vr.cats = 99;
        vr.dogs = 199;

        vr.displayStatistics();
        System.out.println("The class method says there are " +
vr.getAnimalsCount() + " animals");
    }
}
```

Observații:

- Dacă *nu inițializăm* valorile câmpurilor explicit, mașina virtuală va seta toate *referințele* (vom discuta mai mult despre ele în laboratorul următor) la `null` și *tipurile primitive* la 0 (pentru tipul `boolean` la `false`).
- În Java *fișierul trebuie să aibă numele clasei (publice) care e conținută în el*. Cel mai simplu și mai facil din punctul de vedere al organizării codului este de a avea fiecare clasă în propriul fișier. În cazul nostru, `VeterinaryReport.java` și `VeterinaryTest.java`.

Obiectele au fiecare propriul spațiu de date: fiecare câmp are o valoare separată pentru fiecare obiect creat. Codul următor arată această situație:

```
 public class VeterinaryTest {
     public static void main(String[] args) {
         VeterinaryReport vr = new VeterinaryReport();
         VeterinaryReport vr2 = new VeterinaryReport();

         vr.cats = 99;
         vr.dogs = 199;
         vr2.dogs = 2;

         vr.displayStatistics();
         vr2.displayStatistics();

         System.out.println("The first class method says there are " +
vr.getAnimalsCount() + " animals");
         System.out.println("The second class method says there are " +
vr2.getAnimalsCount() + " animals");
     }
 }
```

Se observă că:

- Deși câmpul `dogs` aparținând obiectului `vr2` a fost actualizat la valoarea 2, câmpul `dogs` al obiectului `vr1` a rămas la valoarea inițiala (199). **Fiecare obiect are spațiul lui pentru date!**
- System.out.println(...) este metoda utilizată pentru a afișa la consola standard de ieșire
- Operatorul '+' este utilizat pentru a concatena două șiruri
- Având în vedere cele două observații anterioare, observăm că noi am afișat cu succes șirul “The first/second class method says there are ” + vr.getAnimalsCount() + “ animals”, deși metoda getAnimalsCount() întoarce un întreg. Acest lucru este posibil deoarece se apelează implicit o metodă care convertește numărul întors de metodă în șir de caractere. Apelul acestei metode implicite atunci când chemăm `System.out.println` se întâmplă pentru orice obiect și primitivă, nu doar pentru întregi.

Având în vedere că au fost oferite exemple de cod în acest laborator, puteți observa că se respectă un anumit stil de a scrie codul in Java. Acest coding style face parte din informațiile transmise in

cadrul acestei materii și trebuie să încercați să îl urmați încă din primele laboratoare, devenind un criteriu obligatoriu ulterior în corectarea temelor. Puteți gasi documentația oficială pe site-ul Oracle. Pentru început, încercați să urmați regulile de denumire:
http://www.oracle.com/technetwork/java/codeconventions-135099.html

Exerciții

În cadrul laboratorului și la teme vom lucra cu Java 8. Când consultați documentația uitați-vă la cea pentru această versiune.

1. (**1p**) Numere complexe
    1. Creați un proiect nou cu numele `ComplexNumber`.
    2. Creați clasa `ComplexNumber.java`. Aceasta va avea două campuri: `re` si `im`, ambele de tip `float`.
2. (**1p**) Operații
    1. Creați clasa `Operations.java`. Această clasă va implementa operațiile de adunare și înmulțire pentru numere complexe. Definiți în clasă `Operations` câte o metodă pentru fiecare operație;
    2. (**0.5p**) Testați metodele implementate.
3. (**3p**) Biblioteca
    1. Creați un proiect nou cu numele `Bookstore`.
    2. Creați clasa `Book.java` cu următoarele atribute: `title`, `author`, `publisher`, `pageCount`.
    3. Creați clasa `BookstoreTest.java`, pentru a testa viitoarele funcționalități ale bibliotecii. Completați această clasă, așa cum considerați necesar. Apoi, creați un obiect de tip carte și setați atributele introducând date de la tastatură. Pentru această folosiți clasa `Scanner`:

```
Scanner s = new Scanner(System.in);
String title = s.nextLine();
```

4. Verificați ca numărul de pagini introdus să fie diferit de zero. Puteți consulta documentația claselor `String` și `Integer`.
5. (**3p**) Verificări cărți
    1. Creați o clasă nouă, `BookstoreCheck`, ce va conține două metode, cu câte 2 parametri de tipul `Book`.
        - Prima metodă va verifica dacă o carte este în dublu exemplar, caz în care va întoarce adevărat.
        - A doua metodă va verifica care carte este mai groasă decât altă, și va întoarce cartea mai groasă.
    2. Testați aceste doua metode.
6. (**2p**) Formatare afișare
    - Puteți consulta documentația clasei `String`
    - Afișați informația despre o carte în felul următor:

```
BOOK_TITLE: [insert_book_title]
```

```
BOOK_AUTHOR: [insert_book_author]
BOOK_PUBLISHER: [insert_book_publisher]
```

Cu următoarele precizări:

- Titlul cărții va fi scris în întregime cu **majuscule**
- Numele editurii va fi scris în întregime cu **minuscule**
- Numele autorului rămânând **neschimbat**

- PDF laborator
- **Soluție**