Laboratorul 1: Java basics

Video introductiv: link [https://www.youtube.com/watch?v=gRyTzrvgvYo&t=79s&ab_channel=EchipaPOOCACD-ACSUPB]

Slideuri folosite în video: PDF

Obiective

Scopul acestui laborator este familiarizarea studenților cu noțiunile de bază ale programării în Java.

Aspectele urmărite sunt:

- organizarea unui proiect Java
- familiarizarea cu IDE-ul
- definirea noțiunilor de clasă, câmpuri, proprietăți, metode, specificatori de acces
- folosirea unor tipuri de date

De ce este POO o paradigmă bună?

Succesul acestei paradigme se poate datora și faptului că multe dintre cele mai folosite limbaje se întamplă să fie limbaje orientate pe obiect, cum ar fi:

Android	**iOS**	**Web**
Java	Swift	JavaScript
C++	Objective-C	Python
Kotlin		PHP
		Ruby

Spre deosebire de **programarea procedurală**, care folosește o listă de instrucțiuni pentru a spune computerului pas cu pas ce să facă, **programarea orientată-obiect** se folosește de componente din program care știu să desfășoare anumite acțiuni și să interacționeze cu celelalte.

Astfel, **POO** oferă **modularitate** și **ordine** programului, **reușind să modeleze situații din viața reală** - fiind **mai avansată** decât programarea procedurală, care reflectă un mod simplu, direct de a rezolva problema.

De ce să folosim POO?

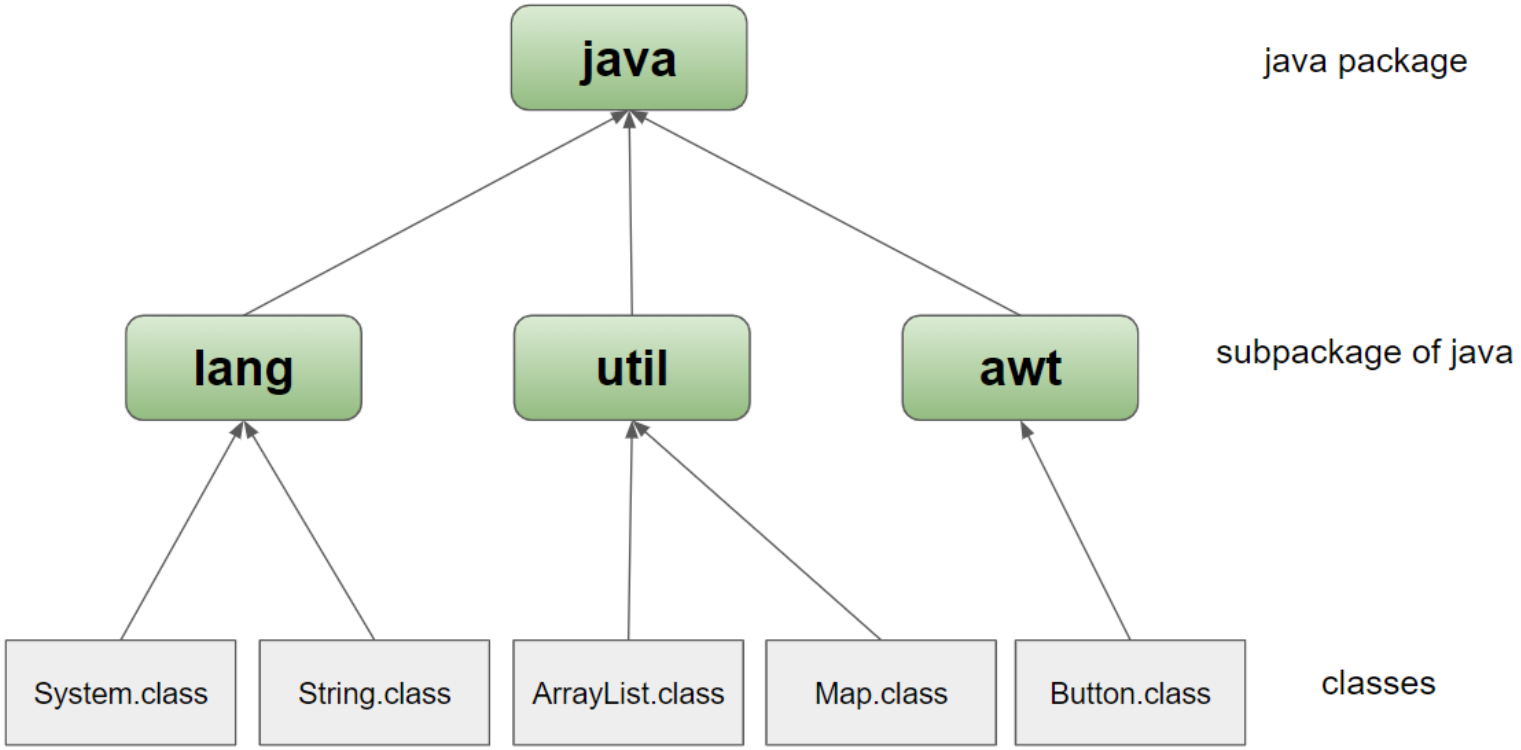
- Scrierea de cod complex devine clară și cu minim de erori **(readability)**
- Împărțirea codului între membrii echipei și urmărirea progresului **(transparency)**

- Cod ușor de extins și reparat **(extensibility)**
- Devine clar ce cod este testat automat și ce cod nu este testat **(testability)**
- Refolosim soluții la probleme comune și uneori chiar cod **(reusability)**
- Ușurința în crearea - uneori automată - a documentației **(documentation)**

Organizarea unui proiect Java

În cadrul acestui laborator veți avea de ales între a folosi IntelliJ IDEA [https://www.jetbrains.com/idea/] (tutorial instalare) folosind contul de mail de la facultate (tutorial activare) sau Eclipse [https://eclipse.org/] (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 nu 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/interfață 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 mai multe informații despre cum funcționează mașina virtuală de Java (JVM) precum și o aprofundare în POO și Java, consultați acest link.

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.

```
boolean isValid = true;

char nameInitial = 'L';

byte hexCode = (byte)0xdeadbeef;

short age = 23;

int credit = -1;

long userId = 169234;

float percentage = 0.42;

double money = 99999;
```

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

```
 new Integer(0);
```

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`.

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

```
Integer myZero = new Integer(0);
```

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";
```

Aceasta 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:

Tip primitiv	Definitie
private	limitează accesul doar în cadrul clasei curente
default	accesul este permis doar în cadrul *pachetului* (package private)
protected	accesul este permis doar în cadrul pachetului si în clasele ce moștenesc clasa curentă
public	permite acces complet

Atenţie, nu confundaţi specificatorul default (lipsa unui specificator explicit) cu `protected`

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 modificării lor doar prin intermediul metodelor.

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

Încapsularea 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).

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 [http://www.oracle.com/technetwork/java/codeconventions-135099.html]

Arrays

Vectorii sunt utilizați pentru a stoca mai multe valori într-o singură variabilă. Un vector este de fapt o matrice (array) unidimensională.

Exemplu definire vector de String-uri cu valorile deja cunoscute

```
String[] cars = { "Volvo", "BMW", "Ford" };
```

Exemplu creare și populare vector cu valori de la 1 la 20

```
int[] intArray = new int[20];
for (int i = 0; i < intArray.length; i++) {
        intArray[i] = i + 1;
}
```

- Înainte să populăm vectorul, trebuie declarat (**int[] intArray**) și alocată memorie pentru 20 elemente de tip int (**new int[20]**).
- Pentru a accesa lungimea vectorului, folosim câmpul **length** păstrat în vector.
- Indexarea elementelor într-un array începe de la 0.

Beginner mistakes

Aceste tipuri de greșeli sunt normale la început și ele sunt exemplificate aici pentru a vă fi mai ușor pentru a identifica soluții în caz că aveți erori sau greșeli de tipul celor exemplificate mai jos în cadrul codului vostru.

Scrierea instrucțiunilor în clase

Spre deosebire de C, în Java instrucțiunile trebuie scrise în cadrul metodelor (funcțiilor), iar metodele trebuie implementate în clase.

Exemple de greșeli:

- exemplul 1:

```
public class SomeClass {
    // nu putem să scriem instrucțiuni în afara metodelor, în cadrul unei clase
    System.out.println("Nu scriem println in afara metodelor");
    // dar o putem face în blocuri de cod, o să vedem mai târziu despre acest lucru

    // de asemenea, nu putem să scriem instrucțiuni de tip loop (for / while) în afara metodelor, în cadrul unei clase
    for (int i = 0; i < 10; i++) {
        System.out.println("Nu scriem for in afara metodelor");
    }
}
```

- exemplul 2:

```
int a = 100;
// nu putem declara variabile în afara claselor

System.out.println("Nu scriem println in afara claselor");
// nu putem folosi instrucțiuni în afara claselor

for (int i = 0; i < 10; i++) {
    System.out.println("Nu scriem for in afara claselor");
}
// nu putem folosi structuri de tip for / while în afara claselor

void doStuff() {
    System.out.println("Nu scriem metode in afara claselor");
}
// nu putem scrie metode în afara claselor

public class SomeClass {
}
```

Cum ar trebui să fie corect:

```
public class MyClass {
    int a = 100; // a este un membru al clasei MyClass, putem face direct initializarea aici

    void doStuff() {
        System.out.println("folosim println in metoda");
        a++;
        for (int i = 0; i < 10; i++) {
            a += 2;
            System.out.println("punem for in metoda " + a);
        }
    }
}
```

Semnătura main-ului

În Java, funcția main are următoarea semnătură: `public static void main(String[] args)`, care este echivalent cu `int main (int argc, char* args[])` din C/C++. În Java, funcția `main` nu returnează nimic (este `void`) și este o metodă statică (o să vedem despre

`static` în următoarele laboratoare). Este important să păstrăm această semnătură pentru ca să fie recunoscut drept entry point-ul programului.

Instanțierea obiectelor și popularea câmpurilor unui obiect

O greșeală frecventă este atunci când se accesează câmpul unui obiect, însă acel obiect nu este instanțiat, fapt ce duce la apariția unei erori (excepția `NullPointerException`, care este echivalent cu segmentation fault din C). Când vrem să folosim un obiect, mai întâi il instanțiem (adică îl creăm cu ajutorul keyword-ului `new` - echivalent cu `malloc` din C), apoi după instanțiere putem să îi populăm câmpurile cu informații.

Exemplu de greșeală:

```
Student st; // fără instanțiere, are valoare default null
st.name = "Costică";
// va fi aruncată excepția NullPointerException, având în vedere că nu are sens să lucrăm cu ceva ce nu există (adică obiectul st, care nu a fost creat)
```

Exemplu corect:

```
Student st = new Student(); // am creat obiectul, putem să ne folosim de el
st.name = "Costică";
```

Folosirea array-urilor

În Java, putem folosi array-uri similar ca în C. Pentru a crea un array de 10 numere întregi, declarăm în felul următor în Java: `int[] arr = new int[10];`, `arr` având o dimensiune fixă de 10 elemente.

Când construim un array de obiecte, trebuie să avem grijă ca obiectele din array să fie instanțiate, altfel vor putea apărea erori (de regulă, excepția `NullPointerException`).

Exemple de greșeli:

```
Student[] arr = new Student(3); // aici este greșit, dimensiunea array-ului e declarată cu paranteze drepte
// Student arr[3]; // nici așa nu merge declarat un array în Java, așa se poate în C

arr[0].name = "Bibi";
// aici vom avea NullPointerException, deoarece nu există obiectul de tip Student din poziția 0, acesta nefiind instanțiat
```

Exemplu corect:

```
Student[] arr = new Student[3]; // array de 3 obiecte de tip Student
arr[0] = new Student(); // mai întâi punem un obiect Student în poziția 0, apoi atribuim valori obiectului din poziția respectivă
arr[0].name = "Andreea";

arr[1] = new Student();
arr[1].name = "Maria";
```

```
arr[2] = new Student();
arr[2].name = "Daniela";
```

Este important să alocăm cât avem nevoie, nu este recomandat să alocăm memorie pentru 1000 de elemente și să populăm array-ul doar cu 50 de elemente. În plus, pot apărea probleme (excepția NullPointerException, atunci când încercăm să accesăm un element care nu a fost creat / instanțiat).

Pentru a evita probleme de acest tip, putem să folosim colecții, despre care vom vorbi în laboratoarele următoare.

Exemplu de greșeală:

```
// dorim să avem un array care conține două obiecte de tip Student

Student[] students = new Student[100]; // greșit, aici trebuia 2 în loc de 100, alocăm cât este nevoie
students[0] = new Student();
students[0].name = "Miruna";

students[1] = new Student();
students[1].name = "Elena";

for (int i = 0; i < students.length; i++) {
    System.out.println(students[i].name); // la i = 2 o să crape codul, se va arunca excepția NullPointerException
}
```

Exemplu corect:

```
// dorim să avem un array care conține două obiecte de tip Student
Student[] students = new Student[2]; // acum este corect
students[0] = new Student();
students[0].name = "Miruna";

students[1] = new Student();
students[1].name = "Elena";

for (int i = 0; i < students.length; i++) {
    System.out.println(students[i].name); // aici nu o să avem erori
}
```

JavaDoc

Ce este?

JavaDoc reprezintă o specificație care explică scopul sau înțelesul elementului căruia îi este atașat. Acesta se poate atașa fie unei clase, fie unei metode. Codul pe care îl scriem nu este complet dacă nu are acest tip de documentație, deoarece, cu toate că următoarea persoană poate să își dea seama ce face o bucată de cod, aceasta nu o să aibă nicio informație legată de utilitate sau despre direcția de dezvoltare din viitor. Fără o astfel de documentație un programator nu poate lua decizii informate despre cum să interacționeze cu codul.

Cum trebuie să fie această documentație?

Atunci când scriem documentația trebuie să ținem cont de 3 aspecte. Aceasta trebuie să fie **clară**, **completă** și **concisă**.

Structura

Un JavaDoc trebuie să fie ușor de citit și astfel recomandăm următoarea structură:

- o propoziție care sumarizează ce presupune clasa / metoda
- informații despre parametrii de intrare ale unei metode și ce întoarce metoda
- dacă există referințe utile pentru clasă / metodă

Această structură este o sugestie care trebuie adaptată în funcție de unde este folosită.

Block tags

Block tag	Descriere	Exemplu
`@param NumeParametru`	Ne oferă informații legate de parametrii metodelor. Dacă anumite valori nu sunt acceptate drept argument (ex. null), acestea trebuie menționate în documentație.	`@param start` începutul intervalului de căutare
`{@link}`	Este utilizat pentru a face o legătură cu o componentă deja existentă printr-un link. Este folosit pentru a fixa o referință.	Extinde funcționalitatea `{@link metodaMeaSuper(String)}` pentru a fi utilizată pe date de tip Float.
`{@code}`	Folosit pentru a referi părți de cod fără a fi formatată precum text HTML.	`{@code HashList}` reprezintă o structură de date unde datele sunt de tipul cheie-valoare.

Exemplu de JavaDoc în cod:

```
/**
* Returns an image that represents a solved sudoku.
* This method always returns immediately, whether or not the
* image exists. It is a similar implementation to {@link solveTetris}
* located in the same suite of games.
*
* @param  path an absolute path to the location of the starting image
* @param  name the name of the image that represents the solved sudoku
*/
public Image solveSudoku(String path, String name) {
    try {
        return solve();
    } catch (Exception e) {
        return null;
    }
}
```

Coding style for beginners

De ce să respectăm un coding style?

Un aspect foarte important în momentul în care trebuie să scrieți cod în Java este legat de modul în care scrieți, mai exact de organizarea codului în interiorul unor clase cu funcționalități bine definite. Poate cel mai important motiv al respectării acestor reguli este faptul că vă va fi de ajutor în momentele în care faceți debugging sau testing pe o sursă. Primii pași pentru a avea un cod cât mai lizibil și ușor de urmărit sunt următorii:

- Dați nume intuitive variabilelor și metodelor folosite (de ex: nu folosiți nume simpliste, precum

  ```
  int a, b, x, y;
  ```

  deoarece la un moment dat veți uita ce rol au acestea și ce reprezintă 😃)

- Metodele folosite trebuie să exprime o funcționalitate bine definită și **numai una**
- Implicit, metodele trebuie să nu aibă multe linii de cod (vezi Rule of 30 [https://dzone.com/articles/rule-30-%E2%80%93-when-method-class-or]) pentru a se putea asigura testarea unei singure funcționalități mai târziu

Exemplu

Vom prelua exemplul anterior, Veterinary Report, pe care îl vom „altera” în felul următor:

```
public class VeterinaryReport {
    int a;
    int b;

    public int foo() {
        return a + b;
    }

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

- Observăm că unui utilizator îi va fi imposibil să se folosească de functionalitățile clasei noastre mai departe deoarece denumirile pe care le au atât variabilele, cât și metodele îl împiedică să înțeleagă logica din spatele codului.
- Evitați întotdeauna să folosiți acest mod de redactare oferind variabilelor o denumire sugestivă!
- Mai multe detalii privind Coding Style-ul, precum și documentația necesară le găsiți pe pagina dedicată [https://ocw.cs.pub.ro/courses/poo-ca-cd/administrativ/coding_style_ide].

Nice to know

- Fiecare versiune de Java aduce și concepte noi, dar codul rămâne cross-compatible (cod scris pt Java 8 va compila si rula cu Java 12). Dintre lucrurile adaugate in versiuni recente avem declararea variabilelor locale folosind `var`, ceea ce face implicit inferarea tipului. Codul devine astfel mai ușor de urmărit (Detalii și exemple [https://dzone.com/articles/finally-java-10-has-var-to-declare-local-variables])

```
var labString = "lab 1" // type String
```

- Când folosiți biblioteci third-party o să observați că pachetele au denumiri de forma *com.organizationname.libname*. Acesta este modul recomandat pentru a le denumi, vedeți mai multe detalii pe pagina oficială [https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html]. Există situații în care acest stil este obligatoriu, de exemplu o aplicație Android nu poate fi publicată pe Play Store dacă numele pachetelor sale nu respectă aceasta convenție.

Summary

- Codul Java este scris în interiorul claselor, enum-urilor și interfețelor (lab interfete)
- Un fișier de tip java poate conține mai multe clase
- Numele singurei clase publice dintr-un fișier trebuie să fie același cu numele fișierului (fără extensia *.java*)
- Fișierele sunt grupate în pachete
- În interiorul unei clase declarăm
  - variabile
  - metode
  - alte clase (lab clase interne)
- **Clasele și metodele ar trebui să aibă o singură responsabilitate. Evitați metodele lungi și clase cu mai mult de un rol. Nu puneți toată logica în metoda *main*. Metoda main ar trebui să conțină doar câteva linii cu instațieri si apeluri de metodele specializate.**
- În declarația claselor, câmpurilor, metodelor putem adăuga specificatori de acces sau diverse keywords pe care le vom studia în următoarele laboratoare
  - specificatori de access: public, private, default, protected
- Java are coding style-ul său și este importantă respectarea lui, altfel proiectele, mai ales cele cu mulți dezoltatori, ar arăta haotic.
- Tipuri de date primitive: int, long, boolean, float, double, byte, char, short. Câmpurile primitive se inițializează automat la instanțierea unei clase cu valori default: e.g. 0 pentru int.
- Clasa String oferă multe metode pentru manipularea șirurilor de caractere.

Exerciții

În cadrul laboratorului și la teme vom lucra cu ultima versiune stabila [https://www.oracle.com/java/technologies/javase-downloads.html] de Java. Când consultați documentația uitați-vă la cea pentru această versiune.

Prerequisites

- Java JDK 17 instalat (laboratorul merge și pe versiuni mai vechi, însă este ok să aveți ultima versiune stabilă pentru teme)
- IDE (Intellij, Eclipse) instalat
- Verificat din linia de comandă versiunea de java:
  - `javac -version` - comanda `javac` este folosită pentru compilare
  - `java -version` - comanda `java` este folosită pentru rulare

Task 0 (0p)

1. Intrați pe link-ul de Github Classroom aferent slotului de laborator, dat de către asistent, clonați repository-ul și deschideți proiectul din repository în IntelliJ.

Task 1 (3p)

1. Creați pachetul `lab1`,unde adăugați codul din secțiunea Exemplu de implementare. Rulați codul din IDE.
2. Folosind linia de comandă, compilați și rulați codul din exemplu
3. Mutați codul într-un pachet `task1`, creat în pachetul `lab1`. Folosiți-vă de IDE, de exemplu Refactor → Move pentru IntelliJ. Observați ce s-a schimbat în fiecare fișier mutat.

Task 2 (5p)

Creați un pachet `task2` (sau alt nume pe care îl doriți să îl folosiți). În el creați clasele:

- `Student` cu proprietățile: `name` (String), `year` (Integer)
- `Course`
  - cu proprietățile: `title` (String), `description` (String), `students` (array de clase `Student` - exemplu arrays).
  - cu metoda: `filterYear` care întoarce o listă de studenți care sunt într-un an dat ca parametru.
- Nu folosiți vreun modificator de acces pentru variabile (aka "nu puneți nimic în fața lor în afară de tip")
- `Test` cu metoda `main`. La rulare, ca argument [https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html] în linia de comandă se va da un integer reprezentând anul în care este un student
  1. creați un obiect `Course` și 3-4 obiecte `Student`. Populați obiectul Course.
  2. afișați toți studenții din anul dat ca parametru. **Hint:** folositi `Arrays.toString(listStudenti)`. In clasa Student folositi IDE-ul pentru a genera metoda toString (pt Intellij Code→Generate…)
  3. rulați atât din IDE (modificati run configuration) cât și din linie de comandă
- Opțional, în loc de arrays pentru `filterYear` puteți să folosiți și obiecte de tip List, e.g. ArrayList [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/ArrayList.html] (exemplu [https://www.w3schools.com/java/java_arraylist.asp])

Task 3 (1p)

1. Creați două obiecte `Student` cu aceleași date în ele. Afișați rezultatul folosirii `equals()` între ele. Discutați cu asistentul despre ce observați și pentru a vă explica mai multe despre această metodă.
    - documentație equals [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)]

Metoda `equals` este folosită în Java pentru a compara dacă două obiecte sunt egale în ceea ce privește informațiile încapsulate în acestea. Mai precis, se compară referințele celor două obiecte. Dacă acestea indică spre aceeași zonă de memorie atunci `equals` va returna `true`, altfel va returna `false`. Veți învăța în laboratorul 3 mai multe despre cum se folosește această funcție pentru a verifica egalitatea dintre două obiecte.

Exemplu de folosire:

```
if (obj1.equals(obj2)) {
    // do stuff
}
```

Task 4 (1p)

1. Adăugați modificatorul de acces 'private' tuturor variabilelor claselor Student și Course (e.g. `private String name;`)
2. Rezolvați erorile de compilare adăugând metode getter și setter acestor variabile.
    a. Ce ați făcut acum se numește *încapsulare (encapsulation)* și este unul din principiile de bază din programarea orientată pe obiecte. Prin această restricționare protejați accesarea și modificarea variabilelor.

Pentru a vă eficientiza timpul, folosiți IDE-ul (IntelliJ) pentru a generarea aceste metode: Code → Generate… → Getters and Setters

Resurse și linkuri utile

- Exerciții din alți ani
- POO și Java
- Organizarea surselor și modificatori de acces