Laboratorul 2: Constructori și referințe

Video introductiv: link [https://www.youtube.com/watch?v=Cz3sNXVrYrM&t]

Obiective

Scopul acestui laborator este familiarizarea voastră cu noțiunile de **constructori** și de **referințe** în limbajul Java.

Aspectele urmărite sunt:

- tipurile de contructori și crearea de instanţe ale claselor folosind acești constructori
- utilizarea cuvântului-cheie **this**

Constructori

Există uneori restricții de integritate care trebuie îndeplinite pentru crearea unui obiect. Java permite acest lucru prin existența noțiunii de **constructor**, împrumutată din C++. Astfel, la crearea unui obiect al unei clase se apelează automat o funcție numită **constructor**. Constructorul are numele clasei, nu returnează explicit un tip anume (nici măcar `void`) și poate avea oricâți parametri.

Crearea unui obiect se face cu sintaxa:

```
class MyClass {
   ...
}

...

MyClass instanceObject;

// constructor call
instanceObject = new MyClass(param_1, param_2, ..., param_n);
```

Se poate combina declararea unui obiect cu crearea lui propriu-zisă printr-o sintaxă de tipul:

```
// constructor call
MyClass instanceObject = new MyClass(param_1, param_2, ..., param_n);
```

De reținut că, în terminologia POO, obiectul creat în urma apelului unui constructor al unei clase poartă numele de **instanță** a clasei respective. Astfel, spunem că `instanceObject` reprezintă o `instanţă` a clasei `MyClass`.

Să urmărim în continuare codul:

```
String myFirstString, mySecondString;

myFirstString = new String();
mySecondString = "This is my second string";
```

Acesta creează întâi un obiect de tip `String` folosind constructorul fără parametru (alocă spațiu de memorie și efectuează inițializările specificate în codul constructorului), iar apoi creează un alt obiect de tip `String` pe baza unui șir de caractere constant.

Clasele pe care le-am creat până acum însă nu au avut nici un constructor. În acest caz, Java crează automat un **constructor implicit** (în terminologia POO, **default constructor**) care face iniţializarea câmpurilor neinițializate, astfel:

- referințele la obiecte se inițializează cu `null`
- variabilele de tip numeric se inițializează cu `0`
- variabilele de tip logic (`boolean`) se inițializează cu `false`

Pentru a exemplifica acest mecanism, să urmărim exemplul:

SomeClass.java

```
public class SomeClass {

    private String name = "Some Class";

    public String getName() {
        return name;
    }
}

class Test {

    public static void main(String[] args) {
        SomeClass instance = new SomeClass();
        System.out.println(instance.getName());
    }
}
```

La momentul execuției, în consolă se va afișa `"Some Class"` și nu se va genera nici o eroare la compilare, deși în clasa `SomeClass` nu am declarat explicit un constructor de forma:

```
public SomeClass() {
    ... // variables initialization
}
```

Să vedem acum un exemplu general:

Student.java

```
public class Student {

    private String name;
    public int averageGrade;

    // (1) constructor without parameters
    public Student() {
        name = "Unknown";
        averageGrade = 5;
    }

    // (2) constructor with two parameters; used to set the name and the grade
    public Student(String n, int avg) {
        name = n;
        averageGrade = avg;
    }

    // (3) constructor with one parameter; used to set only the name
    public Student(String n) {
        this(n, 5); // call the second constructor (2)
    }

    // (4) setter for the field 'name'
    public void setName(String n) {
        name = n;
```

```
    }

    // (5) getter for the field 'name'
    public String getName() {
        return name;
    }
}
```

Declararea unui obiect de tip `Student` se face astfel:

```
Student st;
```

Crearea unui obiect `Student` se face **obligatoriu** prin apel la unul din cei 3 constructori de mai sus:

```
st = new Student();           // first constructor call (1)
st = new Student("Gigel", 6); // second constructor call (2)
st = new Student("Gigel");    // third constructor call (3)
```

Atenție! Dacă într-o clasă se definesc **doar** constructori cu parametri, constructorul default, fără parametri, **nu** va mai fi vizibil! Exemplul următor va genera eroare la compilare:

```
class Student {

    private String name;
    public int averageGrade;

    public Student(String n, int avg) {
        name = n;
        averageGrade = avg;
    }

    public static void main(String[] args) {
        // ERROR: the implicit constructor is hidden by the constructor with parameters
        Student s = new Student();
    }
}
```

În Java, există conceptul de **copy constructor**, acesta reprezentând un constructor care ia ca parametru un obiect de același tip cu clasa în care se află constructorul respectiv. Cu ajutorul acestui constructor, putem să copiem obiecte, prin copierea membru cu membru în constructor.

```
public class Student {
    private String name;
    private int averageGrade;

    public Student(String name, int averageGrade) {
        this.name = name;
        this.averageGrade = averageGrade;
    }

    // copy constructor
    public Student(Student student) {
        // name este camp privat, noi il putem accesa direct (student.name)
        // deoarece ne aflam in interiorul clasei
        this.name = student.name;
        this.averageGrade = student.averageGrade;
    }
}
```

Referințe. Implicații în transferul de parametri

Obiectele se alocă pe `heap`. Pentru ca un obiect să poată fi folosit, este necesară cunoașterea adresei lui. Această adresă, așa cum știm din limbajul C, se reține într-un **pointer**.

Limbajul Java nu permite lucrul direct cu pointeri, deoarece s-a considerat că această facilitate introduce o complexitate prea mare, de care programatorul poate fi scutit. Totuși, în Java există noțiunea de **referinţe** care înlocuiesc pointerii, oferind un mecanism de gestiune transparent.

Astfel, declararea unui obiect:

```
Student st;
```

creează o referință care poate indica doar către o zonă de memorie inițializată cu patternul clasei `Student` fără ca memoria respectivă să conțină date utile. Astfel, dacă după declarație facem un acces la un câmp sau apelăm o funcție-membru, compilatorul va semnala o eroare, deoarece referința nu indică încă spre vreun obiect din memorie. Alocarea efectivă a memoriei și inițializarea acesteia se realizează prin apelul constructorului împreună cu cuvântul-cheie **new**.

Managementul transparent al pointerilor implică un proces automat de alocare și eliberare a memoriei. Eliberarea automată poartă și numele de **Garbage Collection**, iar pentru Java există o componentă separată a JRE-ului care se ocupă cu eliberarea memoriei ce nu mai este utilizată.

Un fapt ce merită discutat este semnificația **atribuirii** de referințe. În exemplul de mai jos:

```
Student s1 = new Student("Bob", 6);

Student s2 = s1;
s2.averageGrade = 10;

System.out.println(s1.averageGrade);
```

se va afișa **10**.

Motivul este că **s1** și **s2** sunt două referințe către **ACELASI** obiect din memorie. Orice modificare făcută asupra acestuia prin una din referințele sale va fi vizibilă în urma accesului prin orice altă referință către el.

În concluzie, atribuirea de referințe **nu** creează o copie a obiectului, cum s-ar fi putut crede inițial. Efectul este asemănător cu cel al atribuirii de pointeri în C.

Transferul parametrilor la apelul de funcții este foarte important de înțeles. Astfel:

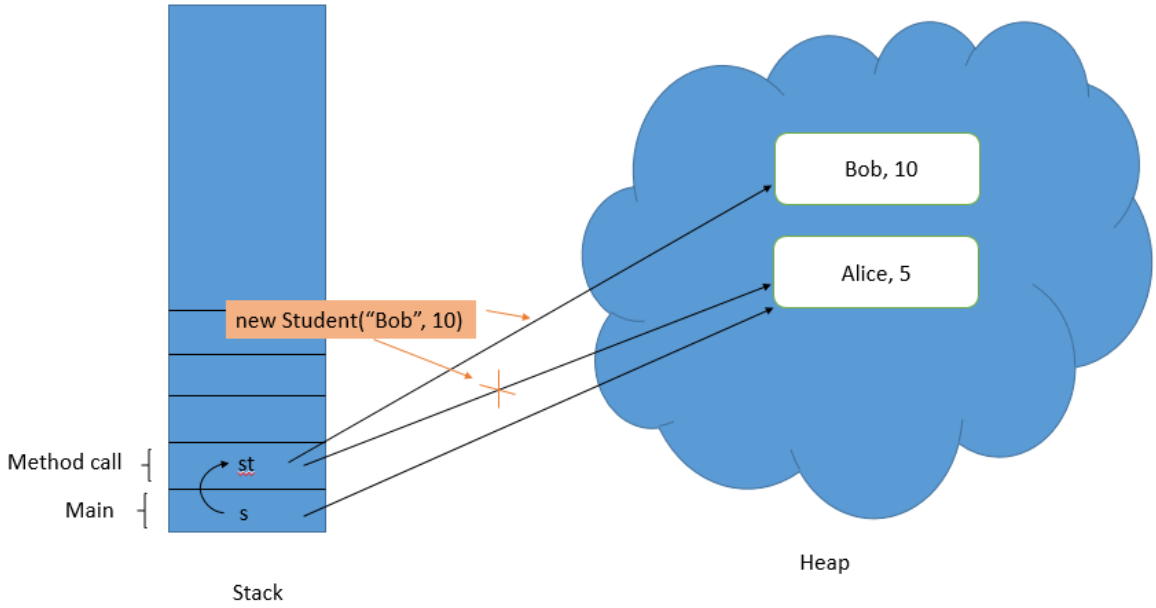
- pentru tipurile primitive se transfera prin **COPIERE** pe stivă: orice modificare în funcția apelată **NU VA FI VIZIBILĂ** în urma apelului.
- pentru tipurile definite de utilizator și instanțe de clase în general, se **COPIAZĂ REFERINȚA** pe stivă: referința indică spre zona de memorie a obiectului, astfel că schimbările asupra câmpurilor **VOR FI VIZIBILE** după apel, dar reinstanțieri (expresii de tipul: `st = new Student()`) în apelul funcției și modificările făcute după ele, **NU VOR FI VIZIBILE** după apel, deoarece ele modifică o copie a referinței originale.

TestParams.java

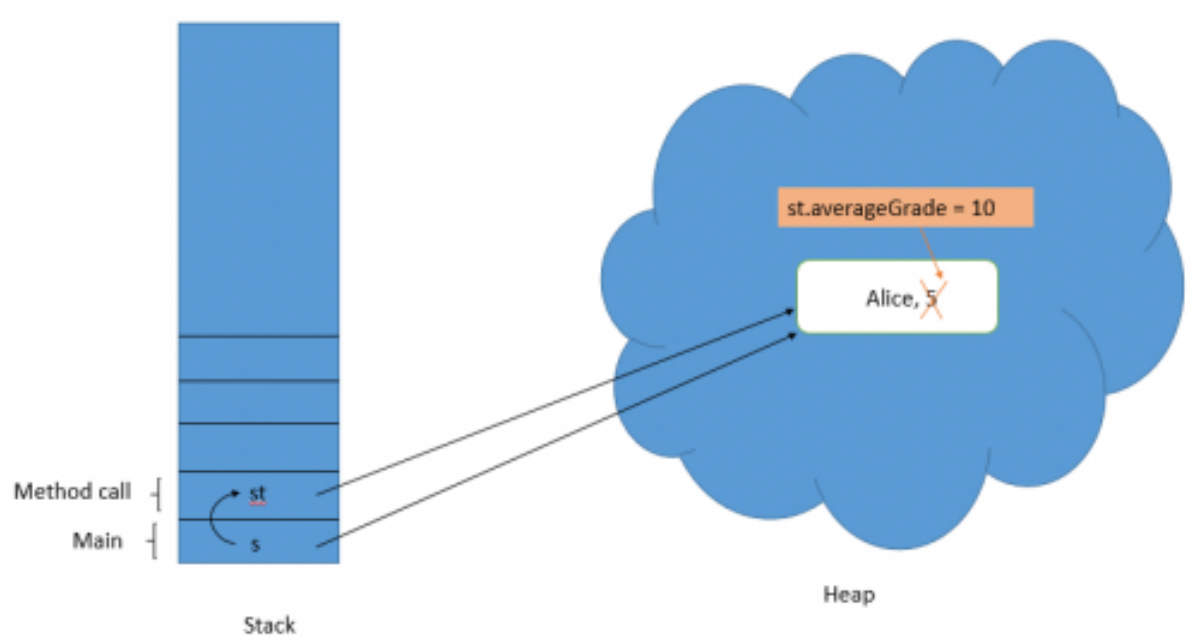
```
class TestParams {

    static void changeReference(Student st) {
        st = new Student("Bob", 10);
    }

    static void changeObject(Student st) {
        st.averageGrade = 10;
    }
```

```
    public static void main(String[] args) {
        Student s = new Student("Alice", 5);
        changeReference(s);                 // apel (1)
        System.out.println(s.getName());    // "Alice"

        changeObject(s);                    // apel (2)
        System.out.println(s.averageGrade); // "10"
    }
}
```

Astfel, apelul (1) nu are nici un efect în metoda `main` pentru că metoda `changeReference` are ca efect asignarea unei noi valori referinței `s`, copiată pe stivă. Se va afișa textul: `Alice`.



Apelul (2) metodei `changeObject` are ca efect modificarea structurii interne a obiectului referit de `s` prin schimbarea valorii atributului `averageGrade`. Se va afișa textul: `10`.



Cuvântul-cheie "this". Întrebuințări

Cuvântul cheie `this` se referă la instanța curentă a clasei și poate fi folosit de metodele, care nu sunt statice, ale unei clase pentru a referi obiectul curent. Apelurile de funcții membru din interiorul unei funcții aparținând aceleiași clase se fac direct prin nume. Apelul de funcții aparținând unui alt obiect se face prefixând apelul de funcție cu numele obiectului. Situația este aceeași pentru datele membru.

Totuși, unele funcții pot trimite un parametru cu același nume ca și un câmp membru. În aceste situații, se folosește cuvântul cheie `this` pentru *dezambiguizare*, el prefixând denumirea câmpului când se dorește utilizarea acestuia. Acest lucru este necesar pentru că în Java este comportament default ca un nume de parametru să ascundă numele unui câmp.

În general, cuvântul cheie `this` este utilizat pentru:

- a se face `diferența` între câmpuri ale obiectului curent și argumente care au același nume
- a pasa ca `argument` unei metode o referință către obiectul curent (vezi linia (1) din exemplul următor)
- a facilita apelarea `constructorilor` din alți constructori, evitându-se astfel replicarea unor bucăți de cod (vezi exemplul de la constructori)

Exemplul următor, în care vom adăuga codului anterior (reprezentat de clasa `Student`), o clasă nouă, va ilustra aceste concepte:

Group.java

```
class Group {

    private int numberStudents;
    private Student[] students;

    public Group () {
        numberStudents = 0;
        students = new Student[10];
    }

    public boolean addStudent(String name, int grade) {
        if (numberStudents < students.length) {
            students[numberStudents++] = new Student(this, name, grade); // (1)
            return true;
        }

        return false;
    }
}
```

Student.java

```
class Student {

    private String name;
    private int averageGrade;
    private Group group;

    public Student(Group group, String name, int averageGrade) {
        this.group        = group; // (2)
        this.name         = name;
        this.averageGrade = averageGrade;
    }
}
```

Metoda toString()

Cu ajutorul metodei toString() [https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Object.html#toString()], care este deja implementată în mod predefinit pentru fiecare clasă în Java, aceasta întorcând un string, putem obține o reprezentare a unui obiect ca string. În cazurile claselor implementate de utilizator, este de recomandat să implementăm (mai precis, să suprascriem - detalii despre suprascrierea metodelor în următoarele laboratoare) metoda toString() în clasele definite de către utilizator.

Un exemplu de implementare (de fapt suprascriere - mai multe detalii in laboratorului de agregare și moștenire) a metodei toString():

Student.java

```
public class Student {
    private String name;
    private int averageGrade;

    public Student(String name, int averageGrade) {
        this.name = name;
        this.averageGrade = averageGrade;
    }


    public String toString() {
        return "Nume student: " + name + "\nMedia studentului: " + averageGrade;
    }
}
```

Folosirea metodei toString():

```
Student st1 = new Student("Ilie Popescu", 5);
/*
nu este neaparat sa scriem st1.toString() la apelul metodei println
println apeleaza in mod automat metoda toString in acest caz
*/
System.out.println(st1);
/*
se va afisa urmatorul string

Nume student: Ilie Popescu
Media studentului: 5
*/
```

Exerciții

Scheletul se afla aici.

Task 1 - 3 puncte

Să se creeze o clasă numită *Complex*, care are doi membri de tip `int` (`real` și `imaginary`), care vor fi de tip `private`. Realizați următoarele subpuncte:

- să se creeze trei constructori: primul primește doi parametri de tip `int` (primul desemnează partea reală a unui număr complex, al doilea partea imaginară), al doilea nu primește niciun parametru și apelează primul constructor cu valorile 0 și 0, iar al treilea reprezinta un copy constructor, care primește ca parametru un obiect de tip Complex, care este copiat în obiectul *this*
- să se scrie metode de tip getter și setter (remember primul laborator - proprietăți), prin care putem accesa membrii privați ai clasei
- să se scrie o metodă de tip void numită `addWithComplex`, care primește ca parametru un obiect de tip Complex, prin care se adună numărul complex dat ca parametru la numărul care apelează funcția (adică la

this)

- să se scrie o metodă de tip void numită `showNumber`, prin care se afișează numărul complex astfel:
  - a + i * b, daca b >0
  - a - i * b, daca b<0
  - a, daca b = 0

Task 2 - 2 puncte

Aveți in scheletul laboratorului un cod, care conține două greșeli legate de referințe. Rolul vostru este să corectați aceste greșeli încât codul să aibă comportamentul dorit (există comentarii în cod despre modul cum ar trebui să se comporte).

Task 3 - 3 puncte

Să se implementeze o clasă *Point* care să conțină:

- un constructor care să primească cele două numere reale (de tip float) ce reprezintă coordonatele.
- o metodă `changeCoords` ce primește două numere reale și modifică cele două coordonate ale punctului.
- o funcție de afișare a unui punct astfel: (x, y). Alternativ, în loc de o metodă de afișare puteți suprascrie toString().

Să se implementeze o clasă *Polygon* cu următoarele:

- un constructor care preia ca parametru un singur număr n (reprezentând numărul de colțuri al poligonului) și alocă spațiu pentru puncte (un punct reprezentând o instanță a clasei Point).
- un constructor care preia ca parametru un vector cu 2N numere reale, adică N perechi de puncte ce reprezintă colțurile unui poligon. Acest constructor apelează constructorul de la punctul de mai sus și completează vectorul de puncte cu cele n instanțe ale clasei Point obținute din parametrii primiți.

Testați codul afișând poligonul.

Task 4 - 2 puncte

În scheletul de cod aveți definită clasa *Book*, în care trebuie să implementați metoda toString() [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Object.html#toString()], și o clasă *Main*, în care se testează metoda toString() din Book.

Referințe

- Constructors in Java [http://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html]
- Java pass by reference or pass by value [http://www.javaworld.com/article/2077424/learn-java/does-java-pass-by-reference-or-pass-by-value.html]
- Using the "this" Keyword [http://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html]
- String Class [http://docs.oracle.com/javase/7/docs/api/java/lang/String.html]