Constructori și referințe

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 cuvintelor-cheie **this**, **final** și **static**
- însușirea noțiunii de obiect **imutabil**

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 inițializarea câmpurilor neinițializate, astfel:

- referințele la obiecte se inițializează cu `null`
- variabilele de tip numeric se inițializează cu
- 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();
    }
}
```

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 **referințe** 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);

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.

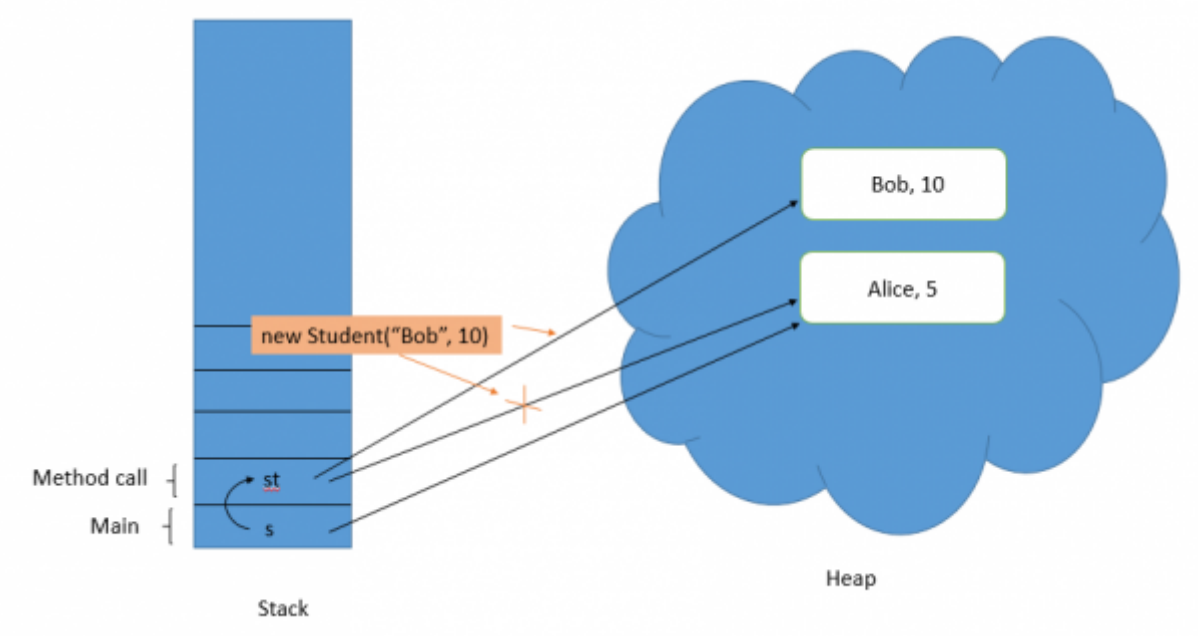
Transferul parametrilor la apelul de funcții este foarte important de înțeles. Astfel:

- 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 obicetului, 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.

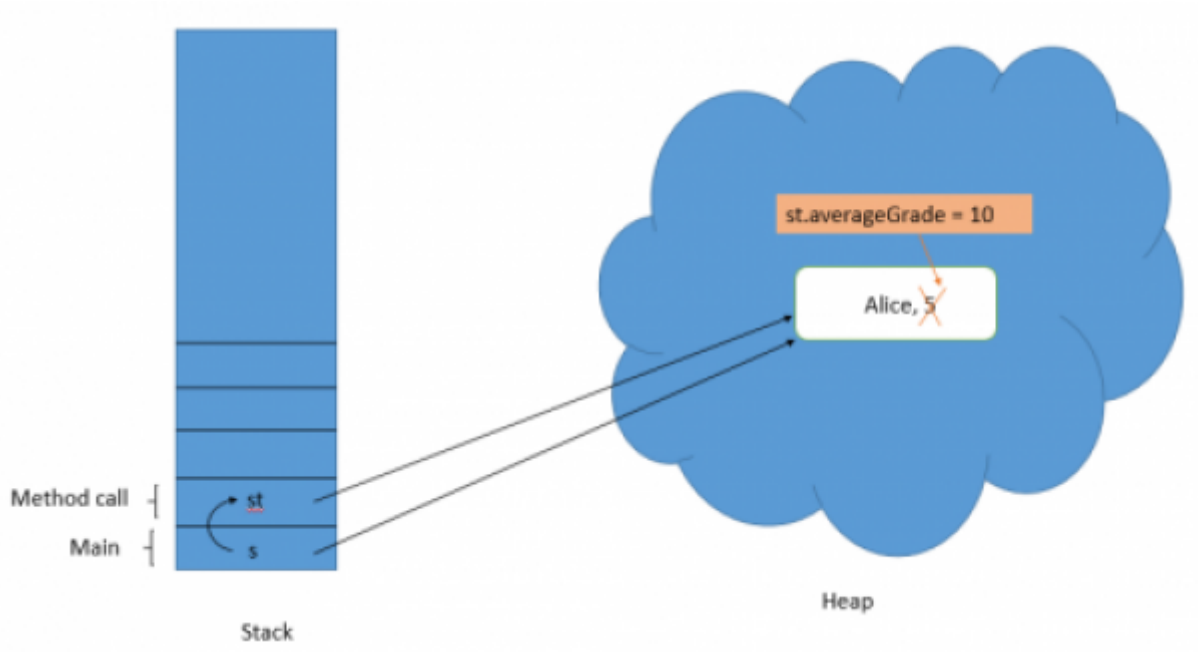
TestParams.java

```
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 `diferenta` 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)

Iată un exemplu în care vom extinde clasa `Student` pentru a cunoaște grupa din care face parte:

Group.java

```
class Group {

    private int numberStudents;
    private Student[] students;

    Group () {
        numberStudents = ;
        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;
    }
}
```

Cuvântul-cheie "final". Obiecte immutable

Variabilele declarate cu atributul `final` pot fi inițializate **o singură dată**. Observăm că astfel unei variabile de tip referință care are atributul `final` îi poate fi asignată o singură valoare (variabila poate puncta către un singur obiect). O încercare nouă de asignare a unei astfel de variabile va avea ca efect generarea unei erori la compilare.

Totuși, obiectul către care punctează o astfel de variabilă poate fi modificat intern, prin apeluri de metode sau acces la câmpuri.

Exemplu:

```
class Student {

    private final Group group;                      // a student can change the group he was assigned in
    private static final int UNIVERSITY_CODE = 15; // declaration of an int constant

    public Student(Group group) {
        // reference initialization; any other attempt to initialize it will be an error
        this.group = group;
    }
}
```

Dacă toate atributele unui obiect admit o unică inițializare, spunem că obiectul respectiv este `immutable`, în sensul că *nu putem schimba obiectul in sine (informatia pe care o stocheaza, de exemplu), ci doar referinta catre un alt obiect.*. Exemple de astfel de obiecte sunt instanțele claselor `String` și `Integer`. Odată create, prelucrările asupra lor (ex.: `toUpperCase()`) se fac prin **instantierea de noi obiecte** și nu prin alterarea obiectelor înseși.

Exemplu:

```
String s1 = "abc";

String s2 = s.toUpperCase();     // s does not change; the method returns a reference to a new object which can be accessed using s2 variable
s = s.toUpperCase(); // s is now a reference to a new object
```

Observăm că în acest exemplu am folosit un String literal. Literalii sunt păstrați într-un `String pool` pentru a limita memoria utilizată. Asta înseamnă că dacă mai declarăm un alt literal “abc”, nu se va mai aloca memorie pentru încă un String, ci vom primi o referință către s-ul inițial. În cazul în care folosim constructorul pentru String se aloca memorie pentru obiectul respectiv și primim o referință nouă.
Pentru a evidentia concret cum functioneaza acest `String pool`, sa luam urmatorul exemplu:

```
String s1 = "a" + "bc";
```

```
String s2 = "ab" + "c";
```

În momentul în care compilatorul va încerca să aloce memorie pentru cele 2 obiecte, va observa că ele conțin, de fapt, aceeași informație. Prin urmare, va instanția un singur obiect, către care vor pointa ambele variabile, s1 și s2. Observați că această optimizare (de a reduce memoria) e posibilă datorită faptului că obiectele de tip String sunt **immutable**.

O intrebare legitimă este, așadar, cum putem compara două String-uri (ținând cont de faptul că avem referințele către ele, cum am arătat mai sus). Să urmărim codul de mai jos:

```
String a = "abc";
String b = "abc";
System.out.println(a == b);  // True

String c = new String("abc");
String d = new String("abc");
System.out.println(c == d);  // False
```

Operatorul "==" compară *referințele*. Dacă am fi vrut să comparăm șirurile în sine am fi folosit metoda `equals`. Același lucru este valabil și pentru oricare alt tip referință: operatorul "==" testează egalitatea *referințelor* (i.e. dacă cei doi operanzi sunt de fapt același obiect).

Dacă vrem să testăm “egalitatea” a două obiecte, se apelează metoda: `public boolean equals(Object obj)`.

Reţineţi semnătura acestei metode!

Cuvântul-cheie "static"

După cum am putut observa până acum, de fiecare dată când cream o instanță a unei clase, valorile câmpurilor din cadrul instanței sunt unice pentru aceasta și pot fi utilizate fără pericolul ca instaţierile următoare să le modifice în mod implicit.

Să exemplificăm aceasta:

```
Student instance1 = new Student("Alice", 7);
Student instance2 = new Student("Bob", 6);
```

În urma acestor apeluri, `instance1` și `instance2` vor funcționa ca entități independente una de cealaltă, astfel că modificarea câmpului `nume` din `instance1` nu va avea nici un efect implicit și automat în `instance2`. Există însă posibilitatea ca uneori, anumite câmpuri din cadrul unei clase să aibă valori independente de instanțele acelei clase (cum este cazul câmpului `UNIVERSITY_CODE`), astfel că acestea nu trebuie memorate separat pentru fiecare instanță.

Aceste câmpuri se declară cu atributul **static** și au o locație unică în memorie, care nu depinde de obiectele create din clasa respectivă.

Pentru a accesa un câmp static al unei clase (presupunând că acesta nu are specificatorul `private`),

se face referire la clasa din care provine, nu la vreo instanță. Același mecanism este disponibil și în cazul metodelor, așa cum putem vedea în continuare:

```
class ClassWithStatics {

    static String className = "Class With Static Members";
    private static boolean hasStaticFields = true;

    public static boolean getStaticFields() {
         return hasStaticFields;
    }
}

class Test {

    public static void main(String[] args) {
        System.out.println(ClassWithStatics.className);
        System.out.println(ClassWithStatics.getStaticFields());
    }
}
```

Exerciții

Exercițiile se rezolvă în ordine.

1. (**1p**) 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)`.
2. (**2p**) 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 reprezentând colțurile. 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.
     - La final, afișați coordonatele poligonului utilizând metodă de afișare a clasei `Point`.
3. (**1p**) Testați diferența de viteză introdusă de crearea de noi obiecte. Măsurați diferența între folosirea `new Integer(2+3)` și 2+3, după modelul de mai jos:

   ```
   private long run() {
       long start = System.nanoTime();
   ```

```
    f(); // function which run time we want to measure
    return System.nanoTime() - start;
}
```

4. (**1p**) Testați diferența de memorie utilizată între a folosi String literal și String obținut prin constructor. Construiți un vector de String. Umpleți acest vector în primă fază cu literal-ul “abc” și măsurați memoria utilizată. Apoi, umpleți vectorul cu new String(“abc”) și măsurați memoria utilizată. Măsurarea memoriei utilizate se poate face folosind următoarea metodă:

```
public void showUsedMemory() {
    long usedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
    System.out.println(usedMem);
}
```

5. (**2p**) Să se implementeze o clasă RandomStringGenerator ce generează un String de o anumită lungime fixată, conținând caractere alese aleator dintr-un alfabet. Această clasă o să conțină următoarele:
   - un constructor care primește lungimea șirului și alfabetul sub formă de String. Exemplu de utilizare:
     ```
     myGenerator = new RandomStringGenerator(5, "abcdef");
     ```
   - o metodă next() care va returna un nou String random folosind lungimea și alfabetul primite de constructor.
     - Pentru a construi String-ul, este utilă reprezentarea sub formă de șir de caractere char[] (char array). Pentru a construi un String pornind de la un char array procedăm ca în exemplul următor:
       ```
       char[] a = {'a','b','c'};
       String s = new String(a);
       ```
     - Conversia unui String la char array se face apelând metoda toCharArray() a String-ului de convertit.
   - Pentru a genera valori aleatoare din domeniul [0, N - 1], utilizați:
     ```
     import java.util.Random;
     Random generator = new Random();
     int value = generator.nextInt(N);
     ```

6. (**3p**) Să se implementeze o clasă PasswordMaker ce generează, folosind RandomStringGenerator, o parolă pornind de la datele unei persoane. Această clasă o să conțină următoarele:
   - o constantă MAGIC_NUMBER având orice valoare doriți
   - un String constant MAGIC_STRING, lung de minim 20 de caractere, generat random
   - un constructor care primește: un String numit firstName, un String numit lastName și un int numit age
   - o metodă getPassword() care va returna parola
     - Parola se construiește concatenand următoarele șiruri:
       - șirul format din ultimele (age % 3) litere din firstName
       - un șir random de lungime MAGIC_NUMBER, generat cu

`RandomStringGenerator` și cu un alfabet obținut din 10 caractere obținute random din MAGIC_STRING
- și șirul format prin conversia la String a numărului (age + lungimea lui lastName).

◦ Pentru subșiruri și alte metode utile consultați documentația clasei String

Resurse

- PDF laborator

Referințe

- Constructors in Java
- Java pass by reference or pass by value
- Using the "this" Keyword
- Final Keyword in Java
- Static Keyword explained
- String Class