Constructori, Referinte

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

Constructori

Exista uneori restrictii de integritate care trebuie indeplinite pentru crearea unui obiect. Java permite acest lucru prin existenta notiunii de **constructor**, imprumutata din C++. Astfel, la crearea unui obiect al unei clase se apeleaza automat o functie numita **constructor**. Constructorul are numele clasei, nu returneaza explicit un tip anume (nici macar `void`!) si poate avea oricati parametri. Crearea unui obiect se face cu sintaxa:

```
class MyClass {
   ...
}
...

instanceObject = new MyClass(param_1, param_2, ..., param_n); // in dreapta semnului egal avem apel la constructor.
```

Se poate uni declararea cu crearea propriu-zisa a obiectului printr-o sintaxa de tipul:

```
MyClass instanceObject = new MyClass(param_1, param_2, ..., param_n); // in dreapta semnului egal avem apel la constructor.
```

De retinut ca, in terminologia POO, obiectul creat in urma apelului unui constructor al unei clase poarta numele de **instanta** a clasei respective. Astfel, spunem ca `instanceObject` reprezinta o **instanta** a clasei `MyClass`.

Sa urmarim in continuare codul:

```
String myFirstString, mySecondString;

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

Acesta creeaza intai un obiect de tip `String` folosind constructorul fara parametru (aloca spatiu de memorie si efectueaza initializarile specificate in codul constructorului), iar apoi creeaza un alt obiect de tip `String` pe baza unui sir de caractere constant.

Clasele pe care le-am creat pana acum insa nu au avut nici un constructor. In acest caz, Java creaza automat un **constructor implicit** (in terminologia POO, **default constructor**) care face initializarea campurilor neinitializate, astfel:

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

Pentru a exemplifica acest mecanism, sa urmarim exemplul:

```
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 executiei, in consola se va afisa *"Some Class"* si nu se va genera nici o eroare la compilare, desi in clasa `SomeClass` nu am declarat explicit un constructor de forma:

```
public SomeClass() {
    ... // Eventuale initializari
}
```

Sa analizam acum un exemplu general:

`Student.java`

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

    // (1) constructor fara parametri
    public Student() {
        name = "Necunoscut";
        averageGrade = 5;
    }

    // (2) constructor cu 2 parametri, folosit daca cunoastem numele si media
    public Student(String n, int avg) {
        name = n;
        averageGrade = avg;
    }

    // (3) constructor cu un singur parametru, folosit atunci cand cunoastem doar numele studentului
    public Student(String n) {
        this(n, 5); // Astfel se va apela constructorul (2)
    }

    // (4) metoda set pentru campul name al clasei Student
    public void setName(String n) {
        name = n;
        averageGrade = 5;
    }

    // (5) metoda getter pentru campul name
    public String getName() {
        return name;
    }
}
```

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

```
Student st;
```

Crearea unui obiect student se face prin apel la unui din cei 3 constructori de mai sus:

```
st = new Student(); // apel al constructorului 1
st = new Student("Gigel", 6); //apel al constructorului 2
st = new Student("Gigel"); //apel al constructorului 3
```

Atentie! Daca intr-o clasa se definesc **doar** constructori cu parametri, constructorul default, fara parametri, **nu** va mai fi vizibil! Exemplul urmator 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) {
        Student s = new Student(); // EROARE: constructorul implicit este ascuns de constructorul cu parametri
    }
}
```

Referinte. Implicatii in transferul de parametri

Dupa cum stiti din laboratorul 1, obictele se aloca pe **Heap**. Pentru ca un obiect sa poata fi folosit, este necesara cunoasterea adresei lui. Aceasta adresa, asa cum stim din limbajul C, se retine intr-un **pointer**. Limbajul Java nu permite lucrul direct cu pointeri, deoarece s-a considerat ca aceasta facilitate introduce o complexitate prea mare de care programatorul poate fi scutit. Totusi, in Java exista notiunea de **referinte**, care inlocuiesc pointerii, oferind un mecanism de gestiune (management) transparent.

Astfel, declararea unui obiect:

```
Student st;
```

creeaza o referinta care poate indica doar catre o zona de memorie initializata cu patternul clasei `Student` fara ca memoria respectiva sa contina date utile. Astfel, daca dupa declaratie facem un acces la un camp sau apelam o functie-membru, compilatorul va semnala o eroare, deoarece referinta nu indica inca spre vreun obiect din memorie. Alocarea efectiva a memoriei si initializarea acesteia se realizeaza prin apelul constructorului impreuna cu cuvantul-cheie `new`.

Managementul transparent al pointerilor implica un proces automat de alocare si eliberare a memoriei. Eliberarea automata poarta si numele de ***Garbage Collection*** si exista o componenta separata a JRE-ului care se ocupa cu eliberarea memoriei ce nu mai este utilizata.

Un fapt ce merita discutat este semnificatia **atribuirii** de referinte. In exemplul de mai jos:

```
Student s1 = new Student("Gigel", 6);
s2 = s1;
s2.averageGrade = 10;
System.out.println(s1.averageGrade);
```

se va afisa `10`. Motivul este ca `s1` si `s2` sunt doua referinte catre **ACELASI** obiect din memorie. Orice modificare facuta asupra acestuia prin una din referintele sale va fi vizibila in urma accesului prin orice alta referinta catre el. In concluzie, atribuirea de referinte **nu** creeaza o copie a obectului, cum s-ar fi putut crede initial. Efectul este asemanator cu cel al atribuirii de pointeri in C.

Transferul parametrilor la apelul de functii este foarte importat de inteles. Astfel:

- pentru tipurile primitive se transfera prin **COPIERE** pe stiva: orice modificare in functia apelata **NU VA FI VIZIBILA** in urma apelului.
- pentru tipurile definite de utilizator si instante de clase in general, se **COPIAZA REFERINTA** pe stiva: referinta indica spre zona de memorie a obicetului, astfel ca schibmarile asupra campurilor vor fi vizibile dupa apel, dar reinstantieri (expresii de tipul: `st = new Student()`) in apelul functei si modificarile facute dupa ele, **NU VOR FI VIZIBILE** dupa apel, deoarece ele modifica o copie a referintei originale.

`TestParams.java`

```
class TestParams {
    void modificaReferinta(Student st) {
        st = new Student("Gigel", 10)
        st.averageGrade = 10;
    }

    void modificaObiect(Student s) {
        s.averageGrade = 10;
    }

    public static void main(String[] args) {
        Student s = new Student("Andrei", 5);
        modificaReferinta(s); // 1
        System.out.println(s.getName()); // 1'

        modificaObiect(s); // 2
        System.out.println(s.averageGrade); // 2'
    }
}
```

Astfel, apelul (1) nu are nici un efect in metoda `main` pentru ca metoda `modificaReferinta` are ca efect asignarea unei noi valori referintei `s`, trimisa prin valoare. Linia (1') va afisa textul: `Andrei`.

Apelul (2) al metodei `modificaObiect` are ca efect modificarea obiectului referit de `s` cu ajutorul metodei `setName` sau prin acces direct. Linia (2') va afisa textul: `10`.

Cuvantul-cheie `"this"`. Alte intrebuintari

Cuvantul cheie `this` se refera la instanta curenta a clasei si poate fi folosit de metodele (care nu sunt statice) unei clase pentru a referi obiectul curent. In general este utilizat pentru:

- a se face **diferenta** intre campuri ale obiectului curent si argumente care au acelasi nume (vezi linia (2) din exemplul urmator si laboratorul 1)
- a pasa ca **argument** unei metode (vezi linia (1) din exemplul urmator) o referinta catre obiectul curent
- a facilita apelarea **constructorilor** din alti constructori, evitandu-se astfel replicarea unor bucati de cod (vezi exemplul de la constructori)

Iata un exemplu in care vom extinde clasa `Student` pentru a cunoaste grupa din care face parte:

Grupa.java

```
class Grupa {
    private int crt;
    private Student[] studenti;

    Grupa () {
        crt = 0;
        studenti = new Student[10];
    }

    public boolean addStudent(String nume, int media) {
        if (crt < studenti.length) {
            studenti[crt++] = new Student(this, nume, media); // (1)
            return true;
        }
        return false;
    }
}
```

Student.java

```
class Student {
    private String name;
    private int averageGrade;
    private Grupa grupa;

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

Cuvantul-cheie "final". Obiecte immutable

Variabilele declarate cu atributul `final` pot fi initializate o singura data. Observam ca astfel unei variabile de tip referinta care are atributul `final` ii poate fi asignata o singura valoare (variabila poate puncta catre un singur obiect). O incercare noua de asignare a unei astfel de variabile va avea ca efect generarea unei erori la compilare. Totusi, obiectul catre care puncteaza o astfel de variabila poate fi modificat (prin apeluri de metoda sau acces la campuri). Exemplu:

```
class Student {
    private final Grupa grupa; //un student nu isi poate schimba grupa in care a fost asignat
    private static final int COD_UNIVERSITATE = 15;//declara o constanta de tip int

    public Student(Grupa grupa) {
        /* initializare referinta; orice alta initializare ulterioara a acestei variabile va fi semnalata ca
eroare */
        this.grupa = grupa;
    }
}
```

Daca toate atributele unui obiect admit o unica initializare, spunem ca obiectul respectiv este **`immutable`**, in sensul ca starea lui interna nu se poate modifica. Exemple de astfel de obiecte sunt instantele claselor **`String`** si **`Integer`**. Odata create, prelucrarile asupra lor (ex.: `toUpperCase()`) se fac prin **instantierea de noi obiecte**, si nu prin alterarea obiectelor inselor. Exemplu:

```
String s = "abc";

s.toUpperCase(); // s-ul nu se modifica. Metoda intoarce o referinta catre un nou obiect
s = s.toUpperCase(); // s este acum o referinta catre un nou obiect
```

Observam ca in acest exemplu am folosit un String literal. Literalii sunt pastrati intr-un **String pool** pentru a limita memoria utilizata. Asta inseamna ca daca mai declaram un alt literal "abc", nu se va mai aloca memorie pentru inca un String ci vom primi o referinta catre s-ul initial. In cazul in care folosim constructorul pentru String se aloca memorie pentru obiectul respectiv si primim o referinta noua. Exemplu:

```
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
```

Nu uitati ca "==" compara referintele. Daca am fi vrut sa comparam sirurile in sine am fi folosit metoda `equals`.

Cuvantul-cheie `"static"`

Dupa cum am putut observa pana acum, de fiecare data cand cream o instanta a unei clase, valorile campurilor din cadrul instantei sunt unice pentru aceasta si pot fi utilizate fara pericolul ca crearea unei alte instante sa le modifice in mod implicit. Sa exemplificam aceasta:

```
Student instance1 = new Student("Ionescu", 7);
Student instance2 = new Student("Georgescu", 6);
```

In urma acestor apeluri, `instance1` si `instance2` vor functiona ca entitati independente una de cealalta, astfel ca modificarea campului `nume` din `instance1` nu va avea nici un efect implicit si automat in `instance2`. Exista insa posibilitatea ca uneori, anumite campuri din cadrul unei clase sa aiba valori independente de instantele acelei clase (cum este cazul campului `COD_UNIVERSITATE`), astfel ca acestea nu trebuie memorate separat pentru fiecare instanta. Aceste campuri se declara cu atributul **`static`** si au o locatie unica in memorie, care nu depinde de obiectele create din clasa respectiva.

Pentru a accesa un camp static al unei clase (presupunand ca acesta nu are specificatorul `private`), se face referire la clasa din care provine, nu la vreo instanta. Acelasi mecanism este disponibil si in cazul metodelor, asa cum putem vedea in continuare:

```
class ClassWithStatics {
    static String className = "Class With Statics";
    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());
    }
}
```

Exercitii

Nota: Exercitiile se rezolva in ordine.

1. (**1p**) Sa se implementeze o clasa `Punct` care sa contina:
   - un constructor care sa primeasca cele doua numere reale (de tip `float`) ce reprezinta coordonatele.
   - o metoda `changeCoords` ce primeste doua numere reale si modifica cele doua coordonate ale punctului.
   - o functie de afisare a unui punct astfel: `"(x, y)"`.
2. (**2p**) Sa se implementeze o clasa `Poligon` cu urmatoarele:
   - un constructor care preia ca parametru un singur numar (reprezentand numarul de colturi al poligonului) si aloca spatiu pentru puncte (un punct reprezentand o instanta a clasei `Punct`).
   - un constructor care preia ca parametru un vector, cu 6 numere reale reprezentand colturile unui triunghi. Acest constructor apeleaza constructorul de la punctul de mai sus si completeaza vectorul de puncte cu cele 3 instante ale clasei `Punct` obtinute din parametrii primiti.

     La final, afisati coordonatele triunghiului utilizand metoda de afisare a clasei `Punct`.
3. (**1p**) Testati diferenta de viteza introdusa de crearea de noi obiecte. Masurati diferenta intre folosirea `new Integer(2+3)` si `2+3`, dupa modelul de mai jos:
   ```
   private long run() {
       long start = System.nanoTime();
       f(); // functia al carei timp de rulare vrem sa-l masuram
       return System.nanoTime() - start;
   }
   ```
4. (**1p**) Testati diferenta de memorie utilizata intre a folosi String literal si String obtinut prin constructor. Construiti un vector de String. Umpleti acest vector in prima faza cu literal-ul `"abc"` si masurati memoria utilizata. Apoi, umpleti vectorul cu `new String("abc")` si masurati memoria utilizata. Masurarea memoriei utilizate se poate face folosind urmatoarea metoda:
   ```
   public void showUsedMemory() {
       long usedMem = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
       System.out.println(usedMem);
   }
   ```

5. (**2p**) Sa se implementeze o clasa `RandomStringGenerator` ce genereaza un String de o anumita lungime fixata, continand caractere alese aleator dintr-un alfabet. Aceasta clasa o sa contina urmatoarele:
   - un constructor care primeste lungimea sirului si alfabetul sub forma de String. Exemplu de utilizare:
     ```
     myGenerator = new RandomStringGenerator(5, "abcdef");
     ```
   - o metoda `next()` care va returna un nou String random folosind lungimea si alfabetul primite de constructor.

     Pentru a construi String-ul, este utila reprezentarea sub forma de sir de caractere `char[]` (char array). Pentru a construi un String pornind de la un char array procedam ca in exemplul urmator:
     ```
     char[] a = {'a','b','c'};
     String s = new String(a);
     ```
     Conversia unui String la char array se face apeland metoda `toCharArray()` a String-ului de convertit.
     Pentru a genera valori aleatoare din domeniul [0, N - 1], utilizati:
     ```
     import java.util.Random;
     Random generator = new Random();
     int value = generator.nextInt(N);
     ```

6. (**2p**) Sa se implementeze o clasa `PasswordMaker` ce genereaza o parola pornind de la datele unei persoane. Aceasta clasa o sa contina urmatoarele:
   - o constanta MAGIC_NUMBER avand orice valoare doriti
   - un constructor care primeste: un String numit `firstName`, un String numit `lastName` si un int numit `age`
   - o metoda `getPassword()` care va returna parola

     Parola se construieste concatenand urmatoarele siruri: sirul format din ultimele `(age % 3)` litere din firstName, un sir random de lungime MAGIC_NUMBER (folositi `RandomStringGenerator` si ce alfabet doriti) si sirul format prin conversia la String a numarului (age + lungimea lui lastName).
     Pentru subsiruri consultati documentatia clasei `String`

7. (**1p**) Declarati o clasa `Test` cu 2 campuri de tip numar intreg, dintre care **unul static**. Scrieti un constructor care primeste 2 numere intregi si initializeaza cele 2 campuri ale clasei. Creati apoi 2 instante ale acestei clase utilizand constructorul declarat, iar la final afisati, pentru fiecare instanta, cele 2 campuri. Cum interpretati rezultatul?

- **Solutii**

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Constructori%2C_Referinte"

- Ultima modificare 10:55, 7 octombrie 2012.
- Această pagină a fost vizitată de 10.101 ori.
- Conţinutul este disponibil sub GNU Free Documentation License 1.2.
- Politica de confidenţialitate
- Despre POO
- Termeni