In acest laborator vom vorbi mai pe larg despre metodele si campurile statice, despre constructori si ceea ce inlocuieste destructorii in Java.

Campurile statice

Memoria pentru un camp static este alocata atunci cand o clasa este incarcata de catre class loader. Astfel memoria nu depinde de existenta unei anumite instante a clasei respective ci reprezinta o caracteristica generica a acelui tip.
Haideti sa facem urmatorul exercitiu de logica:
Avem tipul Om.
Un camp static ar fi, de exemplu, numarul de indivizi. Observati ca nu avem nevoie sa avem efectiv un om pentru ca acea informatie sa aiba sens. Un alt exemplu, durata medie de viata. Aceste lucruri nu sunt proprietati are unei persoane ci sunt caracteristici ale speciei.
Campuri non-statice ar fi, de exemplu, varsta, numele, sexul. Toate acestea sunt proprietati ale unei persoane, ale unui individ.
La fel ca si campurile normale, campurile statice pot fi initializate pe loc.

```
public static UnObiect numeCampStatic = new UnObiect();
public static int numar = 23;
```

Metodele statice

Similar cu campurile statice, metodele statice trebuie sa poata fie executate si sa aiba sens fara ca un obiect de acel tip sa existe. Exemplul pe care o sa il implementam astazi este cel clasic, insa va propun o analogie cu situatii de care ne-am lovit in programarea de pana acum. In C/C++ ati lucrat cu functii. Multe din aceste functii nu era legate efectiv de o instanta a unui obiect (un set de date pe care le numim “state”), ele doar faceau o procesare generica a ceva si returnau rezultatul. De exemplu, vrem o functie care sa adune doua numere si sa returneze rezultatul ca String, sau o functie care sa converteasca km in mile sau o functie care sa primeasca un String ca parametru si sa returneze un Obiect, presupunand ca Stringul este un obiect serializat ca XML sau ca JSON. Astfel de functii care nu apartin unui obiect, adica nu au nevoie de o instanta, de o “stare”, ca sa fie executate sunt functii statice.
Care este primul si cel mai clar exemplu de functie statica la care va ganditi?

Referirea campurilor/metodelor statice

Un Om poate sa faca referire la un atribut al speciei. Deci daca avem o instanta de Om, o persoana, putem sa accesam o variabila a clasei Om, a speciei, sa o citim, sa o modificam. Deci:

```
public class Om
{
private static int populatie = 0;

      public void setPopulatie(int numar)
      {
      populatie = numar;
      }
}
```

este corect din punct de vedere sintactic. Campul populatie exista indiferent de numarul de instante al clasei, si in mod evident exista si daca exista o instanta a clasei Om.
Totusi, accesarea in acest mod a campului static **nu este recomandata** si trebuie sa va invatati sa nu o folositi. Atentie: ne recomandat nu inseamna neaparat gresit!
Corect ar fi fost sa spunem:
`Om.populatie = numar;`
chiar daca in interiorul functiei putem accesa populatie in mod direct.
Metodele statice sunt accesate/apelate in mod similar si cu aceleasi recomandari, dar vom observa si prezenta operatorului apel de functie (parantezele rotunde).
Referirea metodelor/campurilor ne-statice din context static nu este permisa. De exemplu:

```
public class Om
{
private int varsta;
```

```
	public static void setVarsta(int numar)
	{
	varsta = numar;
	}
}
```

este gresit. Este **absolut logic** sa fie gresit deoarece metoda SetVarsta ar trebui sa se execute indiferent daca exista sau nu vreun obiect de tip Om, caz in care spatiul de memorie pentru varsta (unde se scrie noua valoare) fie nu exista, fie sunt prea multe. Ce se intampla daca totusi stim SIGUR, ca nu exista decat un obiect?

Clasele Singleton

Exista niste tipuri speciale (prin modul in care sunt construite) care permit sa existe o singura instanta a tipului respectiv. Aceste tipuri sunt utile in primul rand in aplicatii mai mari unde avem de-a face cu obiecte “manager”. In acest laborator vom folosi un Singleton pentru a numara cate instante de clase de un anumit tip avem in program.
Cum construim un Singleton? Vrem sa avem o clasa care are toata functionalitatea accesibila fara sa avem acces la o instanta a ei si in acelasi timp sa existe o singura instanta a ei.

1. Definim o noua clasa.
2. Ne asiguram ca nu pot fi construite noi instante din afara clasei (Cum ?)
3. Adaugam o functie numita “getInstance” care sa poata fi accesata fara sa avem o instanta a clasei (pointer la instanta). Cum trebuie sa fie aceasta functie?
4. In getInstance, returnam singura instanta a clasei. Cum putem accesa o astfel de instanta? In getInstance (statica) nu avem nicio garantie ca exista o instanta. Deci trebuie sa adaugam un camp “instance”. Cum trebuie sa fie campul instance?
5. Verificam daca instance e null si daca nu este il returnam, altfel construim singura instanta a clasei si apoi o returnam.

Blocurile de initializare inline, static si non static

Fie clasa:

```
public class Clasa
{

	static
	{
	System.out.println(“static 1”);
	}
	private static int numar = 2;
	static
```

```
        {
        System.out.println(“static 2”);
        }

        {
        System.out.println(“initialize 1”);
        }
        private static String text = “abc”;
        {
        System.out.println(“initialize 2”);
        }
        public Clasa()
        {
        System.out.println(“Constructor call”)
        }
}
```

Ce se executa in acest caz si in ce ordine, atunci cand intr-un program se apeleaza “new Clasa()”:

1. Clasa este incarcata in memorie de classLoader. Asta se intampla (in cele mai multe cazuri) la intrarea in program, deci mult inainte de “new Clasa()”. Se aloca memorie pentru variabilele statice si apoi se executa “blocurile de initializare” statice, in ordinea in care apar in fisier. Deci:
    a. se afiseaza “static 1”
    b. numar primeste valoarea 2
    c. se afiseaza “static 2”
2. Se face apoi new pentru superclasa (vom reveni asupra acestui aspect dupa ce vorbim de mostenire).
3. Se executa blocurile de initializare inline non-statice in ordinea din fisier. Adica:
    a. se afiseaza “initialize 1”
    b. text primeste valoarea “abc”
    c. se afiseaza “initialize 2”
4. Se executa constructorul clasei.

Exercitiu: Verificati cele de mai sus!

Constructorul de copiere

In Java nu exista constructor de copiere. Constructorul de copiere trebuie implementat de fiecare programator acolo unde considera necesar (si unde considera ca are sens). Constructorul de copiere trebuie sa preia un obiect (o anumita stare) si sa il copieze intr-un nou obiect (cu aceeasi stare). Exemplu:

```
public class Student
```

```
{
private String nume;
private String prenume;
private Student celMaiBunPrieten;
public Student()
{}
public Student(Student original)
{
Student copie = this; // nu e necesar.
copie.nume = original.nume;
copie.prenume = original.prenume;
copie.celMaiBunPrieten = original.celMaiBunPrieten;
}
}
```

Observam ca dupa ce facem un new Student(oldStudent), putem modifica valorile campurilor noului student fara sa afectam originalul. Totusi, daca modificam campurile din celMaiBunPrieten al copiei, este afecatat si originalul. De ce?
Copia pe care o facem este numita “shallow copy” adica se copiaza doar valorile, nu se fac si copii ale acestora. Opusul unui “shallow copy” este “deep copy”:

```
public Student(Student original)
{
Student copie = this; // nu e necesar.
copie.nume = new String(original.nume);
copie.prenume = new String(original.prenume);
if (original.celMaiBunPrieten != null)
      copie.celMaiBunPrieten = new Student(original.celMaiBunPrieten);
}
```

Acum, indiferent ce facem cu copia, originalul nu va fi afectat.

Destructori in Java

Pe scurt: nu exista. Exista in schimb o functie numita “finalize” care este chemata atunci cand un obiect este efectiv distrus de GarbageCollector. Sunt foarte multe limitari legate de ce se poate face in acea functie (obiectul va fi distrus in curand), dar ideea de baza este ca acea functie nu trebuie sa existe in Java. Pur si simplu, nu va ganditi la ea deloc (la fel ca la goto in C/C++).

toString()

Orice obiect poate fi convertit intr-un String in Java. By default, obiectul este convertit folosind urmatorul format “NumeClasa@hashCode” unde hashCode este valoarea returnata de o alta functie, numita hashCode, care returneaza by default adresa obiectului. Aceasta forma urata (si inutila) poate fi modificata, redefinind metoda toString in clasa curenta. Desi acest lucru pare magic, el va avea mult mai mult sens dupa ce vorbim despre mostenire. Momentan, retineti doar ca exista acest toString() si poate fi redefinit in orice clasa.

Exercitiu

Haideti sa facem un program cu urmatoarea structura:

1. O clasa InformatiiClasa, cu campurile:
    a. String numeClasa
    b. int numarInstanteConstruite
2. O clasa NumaratorInstante, cu doua metode:
    a. instantaNoua(String numeClasa)
    b. getNumarInstante(String numeClasa)
    c. ce campuri ne trebuie?! Ce fel de clasa am vrea sa fie aceasta clasa?
3. O clasa Elev
    a. String nume, prenume;
4. O clasa Carte
    a. String titlu
5. O clasa Disciplina
    a. String nume
6. O clasa Lansator

In clasa lansator, haideti sa construim diverse clase de tip Elev, Carte, Disciplina.
Vrem apoi sa folosim metoda getNumarInstante(“Elev”) pentru a vedea cati elevi am instantiat (similar pentru celelalte 2 clase).
Ce trebuie sa facem in clasele Elev, Carte, Disciplina, ca sa putem face ceea ce ne-am propus?