Laborator 4

Relațiile intre obiecte

Programarea orientata pe obiecte trebuie privita ca o rețea de entități (obiecte) independente care schimba intre ele mesaje. Fiecare din aceste obiecte are o stare interna și expune o funcționalitate externa. Obiectele comunica intre ele prin intermediul unor mesaje, semnale, reprezentate sintactic sub forma de apeluri de metode și returnare de valori sau aruncare de excepții.

Din punct de vedere ierarhic, exista doua tipuri de relații, în programarea orientata pe obiect, relațiile “Is A” (este un, moștenire) și relațiile “Has A” (are un, agregare/compunere). De exemplu:

Fie un obiect de tipul “Dacia”. Dacia are un obiect de tipul Motor, deci are un câmp intern de tipul Motor. De asemenea “Dacia” poate avea 2 obiecte de tip Reflector. Aceste relații sunt de tipul “Has A”. Observam totuși o diferența subtila intre cele doua relații. Dacia TREBUIE sa aibă un motor, în schimb POATE sa aibă și 2 reflectoare. Atunci când obiectul părinte TREBUIE sa aibă un obiect copil vorbim despre “compunere” iar atunci când POATE sa aibă vorbim despre “agregare”. Din punct de vedere al codului, atunci când avem de-a face cu agregarea, câmpul poate sa fie null, iar când avem de-a face cu compunerea câmpul nu poate sa fie null. morărească

Dacia ne referim tot la tipul “Dacia”, putem sa observam ca “Dacia” este un autoturism, adică poate sa îndeplinească toate functionalitatile unui autoturism. Este de asemenea un autovehicul, care este un vehicul, care este …… mergând tot mai sus este un “Obiect” care este cazul cel mai generic. Deci Dacia “Is A” autoturism. Autoturism “Is a” autovehicul. Observam ca pe măsura ce mergem “in sus” în ierarhia “Is A” obiectele devin tot mai generice, iar pe măsură ce mergem “în jos” obiectele devin tot mai particulare.

Aceasta este explicația “clasica” referitoare la relațiile intre obiecte. In programare efectiv însa, motivația pentru construcția acestor relații este mult mai simpla și are la baza doua idei:

1. Nu avem voie sa scriem cod duplicat (copy paste)
2. Nu vrem ca un obiect sa expună mai multa funcționalitate decât este strict necesar.

De exemplu, toate autovehiculele au un motor, iar acel motor poate fi pornit. Deci dacă am avea un câmp Motor (asa cum am spus mai sus) în clasa “Dacia” atunci ar trebui sa avem acel câmp în toate clasele care sunt autovehicul, “Ford”, “BMW”, “Renault” etc. Observam, copy paste. Deci câmpul de tipul Motor este în fapt o proprietate a autovehiculului. De asemenea, “pornesteMotor()” este o funcție pe care Dacia o are, dar ar trebui sa o aibă și toate celelalte clase, deci copy paste, și nu este bine. Atunci “pornesteMotor()” este o metoda a clasei părinte, autovehicul.
Legat de a doua idee, nu vrem ca obiectul de tip Motor sa poată fi accesat direct dintr-un autovehicul. Adica nu vrem ca sa putem sa pornim motorul dând o comanda directa motorului (apelând o funcție din Motor) ci vrem sa îl pornim dând o comanda autovehiculului. Deci câmpul Motor va fi privat și ascuns, iar procedura de pornire a motorului va fi îndeplinita de o funcție din clasa autovehicul. Prima idee este legata de moștenire, a doua, de agregare.

Redefinirea metodelor (overriding)

In general toate motoarele autovehiculelor pornesc la fel. Sa spunem însa ca un producător fabrica un model care înainte sa pornească motorul vrea sa facă niște verificări de rutina (nivelul de ulei, nivelul de lichid de frâna etc) și în funcție de asta sa pornească sau nu motorul. Deci observam ca funcția generica definita în autovehicul nu mai este suficienta, ci este nevoie de ceva particular. Aici intervine “Override” adică redefinirea unei metode în clasa copil pentru a înlocui funcționalitatea din clasa părinte (ori părinte direct ori părinte mai depărtat).

Overloading

Overloading este un concept destul de simplu: mai multe metode pot avea același nume dar lista de parametrii sa difere. In practica sunt metode diferite, pentru ca o metoda este identificata în mod unic după numele ei și lista de argumente.
Atenție! Tipul metodei sau excepțiile aruncate nu contribuie la identificarea metodei, deci nu putem avea doua metode cu return type diferit dar nume și lista de argumente identice!

Un mod simplu de a înțelege sintaxa moștenirii

Pentru a înțelege toate regulile legate de moștenire trebuie sa percepeți obiectele și relațiile dintre ele în felul următor:
Un Tip este o promisiune.
Un Obiect este ceva concret care are un Tip, adică respecta o promisiune.
Un SubTip (subclass, clasa care moștenește o alta clasa, extends) este o promisiune mai puternica.
Deci când spunem:

```
Animal animal;
```

declaram o referința la un obiect care îndeplinește promisiunea definita de tipul Animal.
Deoarece un Caine este un animal, adică un Caine îndeplinește tot ceea ce promite Animal și încă ceva în plus, avem voie sa scriem:

```
Animal animal = new Caine();
```

Putem scrie si invers?

```
Caine caine = new Animal();
```

Nu! Pentru ca este posibil ca noul obiect, de tip Animal sa nu poată sa îndeplinească ceea ce primite Caine.
Ce înseamna aceste promisiuni?
Înseamna ca ii promitem calculatorului ca atunci când va vrea sa facă “Animal.mananca()” sau “Animal.varsta = 2”, metoda “mananca” sau câmpul “varsta” chiar vor exista în obiectul respectiv (la adresa de memorie respectiva). Un Caine poate sa latre, dar dacă la adresa de memorie exista un simplu Animal, care nu are metoda “latra”, atunci calculatorul nu va putea executa acea metoda! Compilatorul de ajuta sa nu facem astfel de greșeli, verificând promisiunile la compilare.
Avem voie sa scriem?

```
Animal animal = new Caine();
Caine caine = animal;
```

Nu! Des animal este în mod evident un Caine, calculatorul este suficient de prost încât sa nu își dea seama de asta (în practica sa știți ca nu se intampla sa fie asa evident). Totuși, noi ca programatori ne dam seama și pentru a putea sa ii spunem compilatorului “ai încredere în mine, obiectul îndeplinește promisiunea respectiva” se folosește ceea ce numim Type Cast explicit.

```
Animal animal = new Caine();
Caine caine = (Caine) animal;
```

Întrebare: putem scrie asa:

```
Animal animal = new Pisica();
Caine caine = (Caine) animal;
```

Da! Vom primi însa o excepție la rulare, numita ClassCastException, care este modul calculatorului de a ne spune: m-ai mințit, de fapt obiectul nu este un Caine (nu îndeplinește promisiunile unui Caine).
Putem scrie asa:

```
Pisica animal = new Pisica();
```

```
Caine caine = (Caine) animal;
```
Nu! De data asta nu putem scrie. De ce? Pentru ca este evident pentru compilator ca o Pisica nu poate îndeplini promisiunile unui câine. Mai precis, Pisica promite un set generic de lucruri (pentru ca e un animal) și ceva specific pentru Pisica, în schimb Caine reprezinta promisiuni pentru acele lucruri generice (comune cu Pisica) dar încă ceva specific pentru Caine. Pe scurt, se promite altceva.
Exercițiu:
Dacia avem metoda:
```
public void oMetoda(Animal animal){}
```
putem sa o apelam cu ?
```
oMetoda(new Caine());
```

Clasa Object

In Java, “mama tuturor” este clasa Object. Adina orice obiect care nu e primitiva moștenește (de cele mai multe ori implicit) clasa Object. Din acest motiv, toate metodele clasei Object sunt disponibile în toate obiectele pe care noi le declaram.
Trei metode extrem de importante pentru toate obiectele sunt:

1. toString() care e folosita pentru a converti un obiect în String. Implementarea default returnează {numele canonic al clasei}@{hashCode()}
2. hashCode() care e folosita pentru a genera un hash al obiectului curent. E folosit în diverse colecții. Implementarea default returnează adresa de memorie (sau ma rog, un număr bazat pe aceasta).
3. equals(Object other) care e folosita pentru a verifica dacă un obiect este “egal” cu un altul. Implementarea default a acestuia este sa compare adresele de memorie.

Am văzut deja ca în clasa String, equals face altceva. De ce? Hint: moștenire.
Similar, toString() în clasa String, face altceva. De ce? Hint: tot moștenire.

Exercițiu

Haideți sa construim o ierarhie simpla:

1. Clasa Persoana are un nume și un prenume, și când e convertita la string este afișat “Nume”, “Prenume”
2. Clasa Profesor, care este o Persoana, are un alt câmp numit “Disciplina”, de tip string, și când e convertit la String va afișa “Nume”, “Prenume” (“Disciplina”).
3. Clasa Student, care este o Persoana, are un alt câmp numit “An” de tip int și când e convertit la String va afișa “Nume”, “Prenume” (Student an “An”).

In ce pachet punem aceste clase?
Pentru a folosi în implementarea unei metode dintr-o clasa copil implementarea din clasa părinte folosim “super” în loc instanta obiectului (sau de this). De exemplu, pentru profesor, în toString() o sa folosim:
```
return super.toString() + “(“+disciplina+”)”;
```
Pentru a face lucrurile mai grafice și a ne distanta de System.out și System.in, am pregătit un mic framework. Va rog sa descarcați următoarele doua fișiere:

https://drive.google.com/drive/folders/1z0R1dEiNFP7sudM_SWrX77NPmLklBcD7

Cu aceasta ocazie vom vedea pentru prima data și cum se utilizează o librărie externa. Am vorbit în laboratorul 1 despre “classpath” ca fiind locul unde Java va cauta clasele de care are nevoie. Pentru a putea folosi aceasta librărie, trebuie sa adaugăm ambele fișiere în classpath. Din Eclipse, dați click dreapta pe proiect și alegeți de la “Build path” opțiunea “Configure build path”. Apoi mergem la Libraries și alegem “Add External Jar” și selectam cele doua fișiere .jar pe care le-am descarcat.
Acest mic framework are nevoie ca toate obiectele ce vor fi afișate sa fie de tipul “DatabaseElement” deci trebuie sa marcam clasa “Persoana” ca extinde clasa “DatabaseElement”

Observam acum ca avem erori. DatabaseElement necesita ca o mulțime de alte metode sa fie implementate. Practic, când am stabilit ca Persoana este un DatabaseElement, am spus ca Persoana îndeplinește promisiunile obiectului DatabaseElement, iar asta înseamna ca trebuie sa aibă anumite metode definite.
O parte din aceste metode însa nu au sens pentru clasa persoana. Daca nu au sens, nu putem respecta promisiunea, și dacă nu putem respecta promisiunea trebuie sa ii spunem compilatorului ca Persoana conține promisiuni nerespectate. Acest lucru îl facem marcând clasa ca “abstract”. Dupa ce marcam clasa ca “abstract”, devine răspunderea lui Student și Profesor sa implementeze metodele respective, dar, atenție, nu este obligatoriu ca toate metodele sa fie implementate în Student/Profesor. O parte din metode POT fi implementate și în Persoana. Exact acesta este cazul nostru.
Pentru a rula aplicația, construim o clasa Launcher cu funcția main în ea.
In funcția main, declaram o variabila de tipul MainWindow și ii atribuim o noua instanta de MainWindow. Apoi pe aceasta instanta apelam metoda addElementType cu parametrul Student.class și apoi din nou cu Profesor.class. Ce este Student.class, din punct de vedere sintactic? Din punct de vedere semantic, reprezinta obiectul Class care a fost construit de classloader când a fost încărcata clasa respectiva.
Apoi chemam metoda show() din instanta lui MainWindow. Explorați aplicația. In laboratoarele următoare, vom implementa bucăți din aceasta aplicație, astfel încât veți ajunge sa o fi scris singuri pe toată.

Tema pentru acasă 1

Eliminați din classpath fisierul lab4model.jar.
In acest moment lipsește complet clasa DatabaseElement.
Descarcați codul pentru aceasta clasa.

I ce pachet trebuie pusa aceasta clasa?
Cum v-ati dat seama? Doar uitându-va în codul clasei va puteți da seama unde trebuie pusa.
Studiați implementarea din aceasta clasa

Tema pentru acasă 2

Refaceți programul astfel încât în loc de profesor, student, persoana, sa avem câine, pisica, animal. Modificați clasa DatabaseElement astfel încât sa fie afișate alte coloane în tabel.

Tema pentru acasă 3

Refaceți programul astfel încât în loc de câine, pisica, animal sa avem dacia, ford, mașina. In loc de 2 coloane o sa avem 3 coloane în tabel: Nume (mașinile noastre o sa primească un nume propriu de exemplu “Mirela”, “StrambaFiare” etc), Tip (Dacia/Ford), Capacitate cilindrica, Cod Producător.
Codul producătorului este:

1. Pentru Dacia cuvântul “DAC”urmat de capacitatea cilindrica,caracterul “-” și un număr întreg (specific fiecărui obiect)
2. Pentru Ford, cuvântul “FOR” urmat de unu număr întreg, caracterul “x” și capacitatea cilindrica.