

Ingineria Sistemelor de Programare

Agregarea si Mostenirea

mihai.hulea@aut.utcluj.ro

2019

Compozitia si agregarea

- Relatia dintre obiecte raspunde afirmativ la intrebarea “are un/are o”
- Exemple:
 - Telefonul are o baterie ? – “DA”
 - Masina are un motor ? “DA”
 - Studentul are o persoana ? “NU” \ “Studentul este un tip de persoana ?” DA -
> se foloseste mostenirea
- Diferenta dintre agregare si compozitie este la nivel conceptual si nu la nivel sintactic

Compozitia

- Obiectele componente isi incheie ciclul de viata impreuna cu obiectele care il compun.

Compozitia

```
class TV {  
    Display d;  
    PowerSource pw;  
}
```

```
class Display {  
    int size;  
}
```

```
class PowerSource {  
    int watt;  
}
```

Compozitia

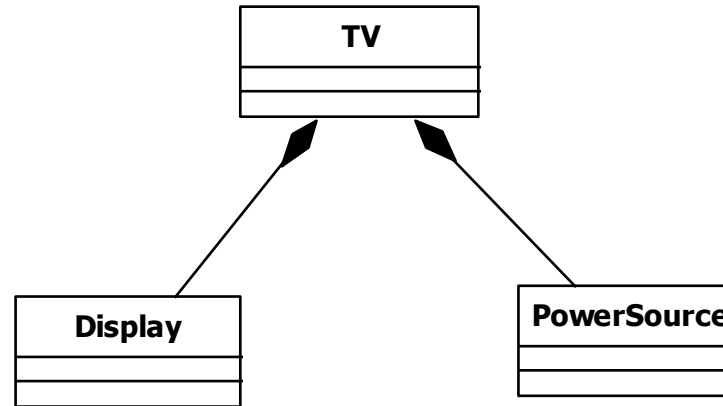
- In mod usual in procesul de compozitie clasa compusa are rolul de a instantia obiectele componente

Compozitia

```
class TV {  
    Display d;  
    PowerSource pw;  
  
    TV(){  
        d = new Display();  
        pw = new PowerSource();  
    }  
}
```

```
class Display {  
    int size;  
}
```

```
class PowerSource {  
    int watt;  
}
```



Agregarea

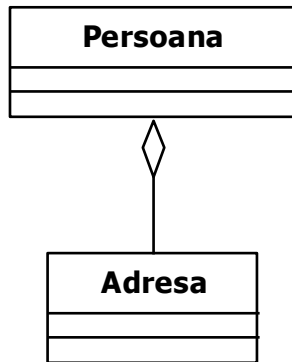
- Obiectele component pot exista si dupa ce obiectul din care fac parte isi incheie ciclul de viata

Agregarea

```
class Adresa {  
    String strada;  
}
```

```
class Persoana{  
    String nume;  
    Adresa adresa;  
}
```


Agregarea



```
class Adresa {String strada;
    Adresa(String str){
        strada = str;
    }
}
```

```
class Persoana{
    String nume;
    Adresa adresa;
    Persoana(String n, Adresa a){
        nume = n;
        adresa = a;
    }
}
```

```
public class Test{
    public static void main(String[] args){
        Adresa a = new Adresa("V.
        Alecsandri");
        Persoana p = new Persoana("Alex",a);
    }
}
```

Ce este mostenirea

Mecanismul de construire a unei clase plecand de la o clasa existanta prin adaugare de attribute si comportamente noi sau prin modificarea comportamentului existent.

Avantajele mostenirii

- Reutilizare codului
 - Comportamentul definit in clasa de baza este automat mostenit in clasa derivata
 - Proprietatile definite in clasa de baza sunt mostenite in clasa derivata
 - O subclasa trebuie sa implementeze doar diferentele fata de clasa de baza

Cum se implementeaza

- Se utilizeaza cuvantul cheie **extends**

```
public class Persoana {  
    String nume;  
    int varsta;  
  
    Persoana(){  
        System.out.println("Constructor persoana.");  
    }  
}
```



Clasa de baza

```
class Student extends Persoana {  
  
    Student(){  
        System.out.println("Constructor student.");  
    }  
}
```



Clasa derivata

Cand se utilizeaza

- Cand relatia dintre cele doua clase raspunde afirmativ la intrebarea *“Este un tip de ?”*
 - Studentul este un tip (un caz particular) de persoana ? DA => putem utiliza mostenirea
 - Adresa este un tip de persoana ? NU => nu putem utiliza mostenirea

Ce putem face intr-o clasa derivata

- Subclasa mosteneste toti membri de tip “public” si “protected”
- Daca subclasa este in acelasi pachet mosteneste si membri cu acces “package”

Ce putem face intr-o clasa derivata

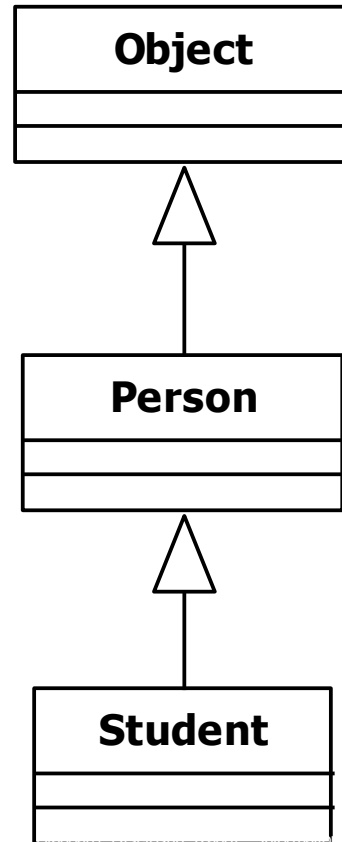
- Membrii mosteniti pot fi accesati si folositi in mod direct
- Putem declara attribute cu acelasi nume => nu este recomandat pt ca se ascund membrii din clasa de baza
- Putem declara o metoda in clasa derivata cu acelasi nume si attribute cu cea din clasa de baza => suprascriere (eng. Overwrite)
- Putem declara noi attribute si metode

Clasa Object

- Clasa de baza a tuturor claselor in Java
- Toate clasele mostenesc clasa Object
- Definieste comportamentul comun tuturor claselor:
 - getClass()
 - equals()
 - toString()
 - ...

Clasa Object

- Ierarhia completa pentru exemplul anterior:



Apelarea Constructorului

```
public class Persoana {  
    String nume;  
    int varsta;
```

```
        Persoana(){  
            System.out.println("Constructor persoana.");  
        }  
    }  
    2  
    3  
    class Student extends Persoana {  
        Student(){  
            System.out.println("Constructor student.");  
        }  
    }  
    1  
    4  
    ...  
    Student s = new Student();
```

The diagram illustrates the execution flow of constructors. It shows two classes: Persoana and Student. Persoana has a constructor Persoana() that prints "Constructor persoana.". Student extends Persoana and has its own constructor Student() that prints "Constructor student.". The code also shows a Student object s being created with new Student(). Blue arrows and numbers indicate the sequence of calls: 1. The new Student() call calls the Student() constructor. 2. The Student() constructor calls the Persoana() constructor. 3. The Student() constructor also references the Student class declaration. 4. The new Student() call also references the Student() constructor.

Cuvantul cheie “super”

- O subclasa poate apela explicit constructorul de baza utilizand “super()”
- Se va executa constructorul care corespunde ca numar si tip de argumente
- Instructiunea “super()” trebuie sa fie prima in constructor

Cuvantul cheie “super”

```
public class Persoana {
    String nume;
    int varsta;

    Persoana(){
        System.out.println("Constructor persoana.");
    }

    Persoana(String nume, int varsta){
        this.nume = nume;
        this.varsta = varsta;
    }
}

class Student extends Persoana {

    Student(){
        super("Alin",1);
        System.out.println("Constructor student.");
    }
}
```

Cuvantul cheie “super”

- O alta utilizare a lui “super” este pentru referirea unui atribut sau a unei metode din clasa de baza (in mod similar cu “this”)

...

```
super.metoda(arg1,arg2);
```

...

Suprascrierea metodelor

- O clasa derivata poate modifica comportamentul unei metode dintr-o clasa de baza, suprascriind (eng. overwrite) acea metoda.
- **Atentie!** In cazul metodelor statice aceasta operatie se numeste ascundere (eng. Hiding method).
- Metoda derivata trebuie sa aiba acelasi tip de return, numar si tip de parametri.

Suprascrierea metodelor

- Specificatorii de acces in metodele suprascrise pot extinde accesul dar nu il pot restriction – de exemplu o metoda protected poate fi facuta publica dar nu privata.

Polimorfismul

- Tipuri de polimorfism
 - Polimorfism dinamic (eng. dynamic / runtime polymorphism)
 - Polimorfism static (eng. static / compile polymorphism)
- Comportamentul prin care se descoperă în mod dinamic în timpul rulării tipul concret al unui obiect din cadrul unei ierarhii de tipuri se numește polimorfism dinamic => suprascrierea metodelor (eng. Overwriting).
- Polimorfismul dinamic permite tratarea tuturor obiectelor dintr-o ierarhie de clase prin intermediul tipului lor de baza.
- Identificarea în momentul compilării a metodei ce trebuie apelate se numește polimorfism static => supraincercarea metodelor (eng. Overloading)

Polimorfismul

...

```
Persoana p1 = new Student("Dan", 21, "UCTN");
```

```
Persoana p2 = new ErasmusStudent("Dan", 21, "UCTN","UBB");
```

```
p1.showPersonDetails(); // call Student showPersonDetails
```

```
p2.showPersonDetails(); // call ErasmusStudent showPersonDetails
```

...

Polimorfismul

- Pentru metodele statice polimorfismul de tip runtime nu se aplica => daca tratam un obiect ca si tipul de baza, apelarea unei metode statice va duce la apelul metodei din clasa tipului de baza si nu la apelul metodei din cadrul tipului concret din care face parte obiectul

Conversii de tip

- Cand un obiect este construit, spunem despre acesta ca este de tipul clasei din care a fost creat. In acelasi timp acel obiect este de tipul tuturor claselor mostenite
- Exemplu:
 - `Student s1 = new Student()`
 - `s1` este de tip `Student`
 - `s1` este de tip `Persoana`
 - `s1` este de tip `Object`

Conversii de tip

- Un obiect poate fi manipulat (utilizat) prin intermediul tipului sau concret sau prin intermediul tipului de baza
- Manipularea unui obiect prin tipul de baza se numeste **conversie de tip implicita**

...

```
Student s = new Student(...);
```

```
Person p = s; // conversie de tip implicita
```

```
Object o = s; // conversie de tip implicita
```

...

Conversii de tip

- Un obiect definit ca si tip de baza poate fi tratat prin tipul derivate prin intermediul **conversiei de tip explicitie**

...

```
Person p1 = new Student(...);
```

```
Student s1 = (Student)p1;
```

...

Conversii de tip

- Conversia de tip explicita poate genera erori la runtime

...

```
Person p1 = Student();
```

```
ErasmusStudent e1 = (ErasmusStudent) p1;
```

...

```
Exception in thread "main" java.lang.ClassCastException: v3.Student  
cannot be cast to v3.ErasmusStudent
```

```
at v3.Test.main(Test.java:14)
```

Conversii de tip

- Pentru a valida la runtime tipul concret al unui obiecte se poate utiliza instructiunea “instanceof”

```
if (p1 instanceof ErasmusStudent) {  
    ErasmusStudent ss = (ErasmusStudent) p1;  
    ss.showPersonDetails();  
}
```

Utilizati instanceof cu atentie

Effective C++, by Scott Meyers:

"Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself.

Exemplu polimorfism

```
class Bird {
    public void move(){
        System.out.println("The bird is moving.");
    }
}

class Penguin extends Bird{
    public void move(){
        System.out.println("The PENGUIN is swiming.");
    }
}

class Goose extends Bird{
    public void move(){
        System.out.println("The GOOSE is flying.");
    }
}
```

Exemplu polimorfism

```
public class BirdController{
    Bird[] birds = new Bird[3];
    BirdController(){
        birds[0] = createBird();
        birds[1] = createBird();
        birds[2] = createBird();
    }
    public void relocateBirds(){
        for(int i=0;i<birds.length;i++)
            birds[i].move();
    }

    private Bird createBird(){
        int i = (int)(Math.random()*10);
        if(i<5)
            return new Penguin();
        else
            return new Goose();
    }

    public static void main(String [] args){
        BirdController bc = new BirdController();
        bc.relocateBirds();
    }
}
```

Clase finale

- O clasa finala nu poate fi extinsa
- Declararea unei clase finale

```
public final class Persoana{  
    ...  
}
```

- Exemplu: clasa String nu poate fi extinsa

Metode finale

- O metoda finala nu poate fi suprascrisa
- Declararea unei metode finale:

```
public final void afiseaza(){  
    ...  
}
```

- Metodele statice sunt automat finale !

Atribute finale

- Nu pot fi modificate
- Pot fi initializate o singura data

```
public class MyClass {  
    private final int myField = 3;  
    public MyClass() {  
        ...  
    }  
}
```

```
public class MyClass {  
    private final int myField;  
    public MyClass() {  
        ...  
        myField = 3;  
        ...  
    }  
}
```

Variabile de metoda finale

- Variabilele nu vor putea fi modificate in interiorul metodei

```
public void foo(final Class1 c1, final Class c2){  
}
```

Beneficiile cuvântului cheie 'final'

- Ajuta la îmbunătățirea performanței – JVM poate stoca în cache variabilele finale utilizate frecvent
- Variabilele finale pot fi utilizate în aplicații multi-thread fără a implementa mecanisme de sincronizare
- Cuvântul final permite JVM să optimizeze metodele, variabilele și clasele