

Clase si obiecte (specificatori de acces, constructori, referințe si model de memorie)

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2021



Computer Science
& Engineering
Department



FACULTATEA DE
AUTOMATICA ȘI
CALCULATOARE



Universitatea
Politehnica
București

Subiecte de discutie

- Clase si obiecte: ce sunt si cum arata?
(specificatori de acces, constructori, getters, setters, this)
- Alocarea memoriei
(referinte, pointeri, alocarea memoriei si modelul de memorie)

Clase si obiecte: ce sunt si cum arata?

Programare Orientată-Obiect vs Programare Procedurală

Programare Procedurală	Programare Orientată-Obiect
Procedură: O listă de instrucțiuni care îi spun computerului ce să facă pas cu pas	Obiect: componentă a programului care știe cum să desfășoare anumite acțiuni și cum să interacționeze cu alte elemente din program
<code>printf("Hello World!\n")</code>	<code>System.out.println("Hello World!")</code>
în C, printf este o funcție	în Java, System.out este un obiect, println este metoda sa

Programare Orientată-Obiect vs Programare Procedurală

Programare Procedurală

Procedură: O listă de instrucțiuni care îi spun computerului ce să facă pas cu pas

- tipuri de date primitive
- tipuri de date compuse (a.k.a. composite data type, a.k.a. record, e.g. in C: struct si array)

Programare Orientată-Obiect

Obiect: componentă a programului care știe cum să desfășoare anumite acțiuni și cum să interacționeze cu alte elemente din program

- tipuri de date primitive
- tipuri de date compuse (struct si array in C)
- clase (tipuri ce compun date si actiuni)

Exercitiu: despre struct in C

Exercitiu: despre struct in C

Obiectele sunt instante ale claselor

```
|| int x;           // tip de date primitiu
|| MyClass myobject; // tip de date compus, clasa
```

Obiectele sunt instante ale claselor

```
|| int x;           // tip de date primitiu  
|| MyClass myobject; // tip de date compus, clasa
```

Clasele sunt tipuri de date compuse, ce contin membrii (aka argumente):

- date sub forma de **campuri** (adica variabile)
- actiuni sub forma de **metode** (adica functii)

Obiectele sunt instante ale claselor

```
|| int x;           // tip de date primitiu  
|| MyClass myobject; // tip de date compus, clasa
```

Clasele sunt tipuri de date compuse, ce contin membrii (aka argumente):

- date sub forma de **campuri** (adica variabile)
- actiuni sub forma de **metode** (adica functii)
 - constructori
 - getters si setters (accessors / properties)
 - alte metode

Obiectele sunt instante ale claselor

```
public class Punct2D {  
    private int x;  
    private int y;  
  
    public Punct2D () {  
        this.x = 0;  
        this.y = 0;  
    }  
  
    public Punct2D (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX () {  
        return x;  
    }  
  
    public void setX (int x) {  
        this.x = x;  
    }  
}
```

Specificatori de acces

- public - permite acces complet din exteriorul clasei curente
- private - limitează accesul doar în cadrul clasei curente
- protected - limitează accesul doar în cadrul clasei curente, al altor clase din același pachet și al tuturor descendenților ei (conceptul de descendență sau de moștenire va fi explicitat cursul următor)
- (default) - în cazul în care nu este utilizat explicit nici unul din specificatorii de acces de mai sus, accesul este permis doar în cadrul pachetului (package private). Atenție, nu confundați specificatorul default (lipsa unui specificator explicit) cu vreunul din ceilalți specificatori!

Obiectele sunt instante ale claselor

```
class Punct2D {  
    private:  
        int x;  
        int y;  
  
    public:  
        Punct2D () {  
            this->x = 0;  
            this->y = 0;  
        }  
  
        Punct2D (int x, int y) {  
            this->x = x;  
            this->y = y;  
        }  
  
        int getX() {  
            return x;  
        }  
  
        void setX(int x) {  
            this->x = x;  
        }  
}
```

Specificatori de acces: exemplu

```
class GeneratorNume {  
    private String[] vocabulary = {new String("Ana"), new  
        String("Mihai"), new String("Robert")};  
    private Random randomGenerator = new Random();  
  
    public String genNume() {  
        return vocabulary[randomGenerator.nextInt(3)];  
    }  
}  
...  
GeneratorNume gn = new GeneratorNume();  
System.out.println(gn.genNume());
```

Cuvantul cheie this

Cuvantul cheie **this** se refera la obiectul (instanta clasei) din care se apeleaza functia respectiva.

- dezambiguizare intre variabile locale si parametrii
- claritatea codului
- apelul unui constructor
- valoare de return pentru orice functie

Constructori: Default Constructor

Default constructor (no-arg constructor): provides the default values to the object like 0, null etc. depending on the type

```
public class Magazin {  
    private String brand;  
  
    public Magazin() {  
        brand = "de inchiriat";  
    }  
}  
...  
Magazin item = new Magazin();
```

Constructori: Parameterized Constructor

Parameterized constructor: provide different values to the distinct objects

```
public class Magazin {  
    private String brand;  
  
    public Magazin(String brand) {  
        this.brand = brand;  
    }  
    ...  
    Magazin item = new Magazin("IKEA");
```

Constructori: Constructor Overloading

A class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type

```
public class Magazin {  
    private String brand;  
    private Integer suprafata;  
  
    public Magazin(Integer suprafata) {  
        this.suprafata = suprafata;  
        this.brand = "de inchiriat";  
    }  
  
    public Magazin(Integer suprafata, String brand) {  
        this.suprafata = suprafata;  
        this.brand = brand;  
    }  
}
```

Constructori: Copy Constructor

In Java, there is no Copy Constructor per-se, but there are many ways to copy the values of one object into another:

- By constructor
- By assigning the values of one object into another
- By `clone()` method of Object class

```
public class Training {  
    private int pushups;  
    private int crunches;  
  
    public Training(Training otherTraining) {  
        this.pushups = otherTraining.pushups;  
        this.crunches = otherTraining.crunches;  
    }  
}
```

Destructori

In C++, se apeleaza la iesirea din scope a variabilei:

```
class Catalog {  
    private:  
        int* note;  
  
    public:  
        Catalog(int count = 20) {  
            note = new int[count];  
        }  
  
        ~Catalog() {  
            delete [] note;  
        }  
};  
  
int main(){  
    Catalog* grupa321CD = new Catalog(30); // -> constructor  
    ...  
    delete grupa321CD; // -> destructor  
}
```

Destructori

In C++, se apeleaza la iesirea din scope a variabilei:

```
class Catalog {  
    private:  
        int* note;  
  
    public:  
        Catalog(int count = 20) {  
            note = new int[count];  
        }  
  
        ~Catalog() {  
            delete [] note;  
        }  
};  
  
int main(){  
    Catalog* grupa321CD = new Catalog(30); // -> constructor  
    ...  
    //delete grupa321CD;  
}
```

// -> destructor

Destructori

In Java exista functia **finalize**, dar nu stim cand este apelata pentru ca de gestiunea memoriei se ocupa Garbage Collector

Accessors (Setters and Getters)

Campurile expuse prin intermediul functiilor de tip Setter-Getter se numesc proprietati.

De ce sa folosim Setter/Getter in loc de camp direct?

Getter și Setter: motive

Pot exista nivele de acces diferite pentru setter și getter (sau chiar unul să nu existe)

```
public class Employee {  
    private int salary;  
  
    public int getSalary() {  
        return salary;  
    }  
    protected void setSalary(int salary) {  
        this.salary = salary;  
    }  
}
```

Getter și Setter: motive

Se poate ascunde reprezentarea internă:

```
public class Employee {  
    private String street;  
    private int number;  
    private String city;  
  
    public int getAddress() {  
        return street+", "+number+", "+city;  
    }  
}
```

Getter și Setter: motive

Se pot face validări:

```
public class Employee {  
    private String email;  
  
    public void setEmail(String email) {  
        if (!EmailChecker.isValid(email)) {  
            System.out.println(email);  
        } else {  
            this.email = email;  
        }  
    }  
}
```

Getter și Setter: motive

Se pot face conversii:

```
public class Employee {  
    private float height;  
  
    public void setHeight(float height, String measure)  
    {  
        if (!measure.equals("m")) {  
            this.height = height / 0.3048;  
        } else {  
            this.height = height;  
        }  
    }  
}
```

Getter și Setter vs Boilerplate Code

In C#:

```
public abstract class Foo {  
    public virtual string Hello { get; set; }  
}
```

In Java:

[Project Lombok](#)

Alocarea memoriei

Pointeri vs Referințe în C/C++

```
|| int x = 13, y = 14;  
|| int *px = &x, &rx = x;
```

Pointeri vs Referințe în C/C++

```
|| int x = 13, y = 14;
|| int *px = &x, &rx = x;
|| int *px, &rx;           // eroare: referintele sunt
                           initialize la creare
```

Pointeri vs Referințe în C/C++

```
|| int x = 13, y = 14;
|| int *px = &x, &rx = x;

|| int *px, &rx;           // eroare: referintele sunt
||                      initialize la creare

|| px = &y;   rx = y;       // nu se modifica referinta, ci
||                      variabila referita
```

Pointeri vs Referințe în C/C++

```
|| int x = 13, y = 14;
|| int *px = &x, &rx = x;

|| int *px, &rx;           // eroare: referintele sunt
||                      initialize la creare

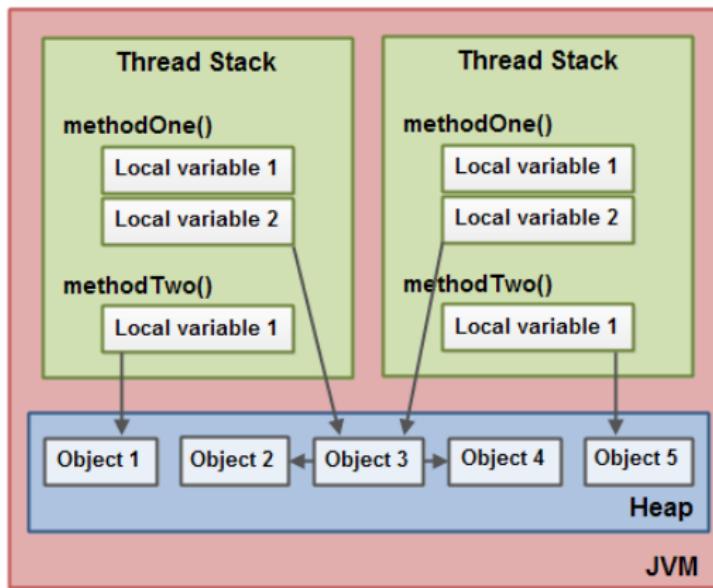
|| px = &y;   rx = y;       // nu se modifica referinta, ci
||                      variabila referita

|| px++;   rx++;           // nu se modifica referinta, ci
||                      variabila referita
```

Pointeri vs Referințe în Java

Exercitiu: În Java avem doar referințe, nu avem pointeri

Modelul de memorie al JVM



▶ Java Memory Model on Jenkov.com

Tipuri de date primitive: alocarea memoriei

În Java, variabile primitive pot fi locale unei metode sau membrii unui obiect

```
public class Album {  
    public static void main(String[] args) {  
        int price;  
    }  
}
```

```
public class Album {  
    int price;  
    public static void main(String[] args) {  
    }  
}
```

Tipuri de date primitive: valori default

variabilele locale nu sunt initializate

```
public class Album {  
    public static void main(String[] args) {  
        int price;  
        price++; // eroare de compilare  
    }  
}
```

Tipuri de date primitive: valori default

variabilele locale nu sunt initializate

```
public class Album {  
    public static void main(String[] args) {  
        int price;  
        price++; // eroare de compilare  
    }  
}
```

membrii claselor sunt initializați cu 0 sau null, în funcție de tip

```
public class Album {  
    int price;  
    public static void main(String[] args) {  
        price++; // bad programming style  
    }  
}
```

▶ Primitive Data Types doc

Alocarea memoriei

La declararea fără initializare se creează o referință nulă:

```
|| Magazin spatiuDeInchiriat;
```

Se alocă spațiu pe heap și se apelează un constructor la initializare:

```
|| Magazin magazinSport = new Magazin("Articole Sportive");
```

Alocarea memoriei

Două referințe la același obiect: se modifică obiectul

```
public Arena {  
    public int seats;  
    Arena(int seats) {  
        this.seats = seats;  
    }  
}  
  
Arena arenaNationala = new Arena(55000);  
stadionulNational = arenaNationala;  
stadionulNational.seats += 600;  
System.out.println(arenaNationala.seats); // 55000 ?  
      55600
```

Comparări

`==` tests for reference equality (whether they are the same object)
`.equals()` tests for value equality (whether they are logically "equal")

```
String name1 = new String("Michael"), name2 = name1,  
       name3 = new String("Michael");  
System.out.println(name1==name2);  
System.out.println(name1==name3);  
System.out.println(name1.equals(name3));
```

Comparății

Credeti ca ati intelese?

```
public class MyProgram {

    public static void main(String args[])
    {
        Integer a = Integer.valueOf(1);
        Integer b = Integer.valueOf(1);
        System.out.println(a==b);

        Integer x = Integer.valueOf(10001);
        Integer y = Integer.valueOf(10001);
        System.out.println(x==y);
    }

}
```

Transferul parametrilor

Transferul parametrilor la apelul funcțiilor este crucial pentru o funcționare corectă:

- variabile de tip primitiv
 - se transferă prin **copiere** pe stivă
 - orice modificare din functie a valorii variabilei NU VA FI VIZIBILA
- obiecte
 - se transferă prin **referinta** pe stivă
 - orice modificare din functie a referintei obiectului (e.g. `p = new Player()`) NU VA FI VIZIBILA

Garbage Collector:

- Când un obiect nu mai este folosit, Garbage Collector revendică spațiul său de memorie de pe Heap pentru a fi refolosit

[▶ Java Garbage Collection Basics - Oracle](#)

Reading Assignments

- [Lab 02: Constructori si Referințe](#)
- [Java Memory Model on Jenkov.com](#)
- [Rule of three, of five, of zero](#)

inca o data...

Tipuri de date

Tipuri de date:

- primitive
- compuse
- omogene

Tipuri de date

Tipuri de date in C/C++:

- primitive
 - int, char, float, double, void
 - signed, unsigned

Tipuri de date

Tipuri de date in C/C++:

- primitive
 - int, char, float, double, void
 - signed, unsigned
- compuse
 - struct din C
 - struct, class din C++

Tipuri de date

Tipuri de date in C/C++:

- primitive
 - int, char, float, double, void
 - signed, unsigned
- compuse
 - struct din C
 - struct, class din C++
- omogene
 - vectori (a.k.a. arrays)
 - structuri de date (implementate manual sau din STL)

Tipuri de date

Tipuri de date in Java:

- primitive:

```
||     float pret = 12.5;
```

- compuse - class:

```
||     Song song1 = new Song("Bucovina", "Mestecanis");
```

- omogene:

```
||     Song playlist[] = new Song[]{  
||         new Song("Subcarpati", "Balada Romanului"),  
||         new Song("Subcarpati", "Frunzulita, iarba deasa")  
||     };
```

Agregare vs moștenire, upcasting, downcasting, overriding, overloading, super

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2021



Computer Science
& Engineering
Department

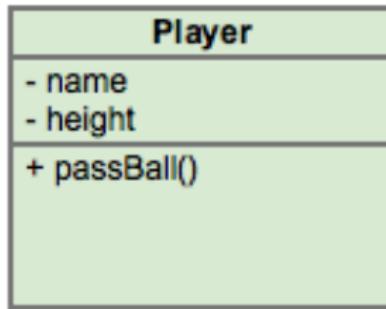


FACULTATEA DE
AUTOMATICA ȘI
CALCULATOARE



POLITEHNICA
Universitatea
București

Curs 2: ce contine o clasa



Precizări diagrame

- UML (Unified Modeling Language) este un standard pentru modelarea sistemelor software
- "+" e pentru membrii publici, "#" protected, "-" private, "~" default (package)
- Asocierea, agregarea, compunerea și moștenirea se deduc din tipul săgetii folosite între componente
- Nu este indicată includerea getterilor și setterilor în diagrame (pot ajunge foarte mari)
- Metodele moștenite apar în diagramele claselor derivate doar în cazul în care sunt suprascrisse

Relații între clase

- Asociere (*association*)
- Agregare (*aggregation*)
- Compunere (*composition*)
- Moștenire (*inheritance*)

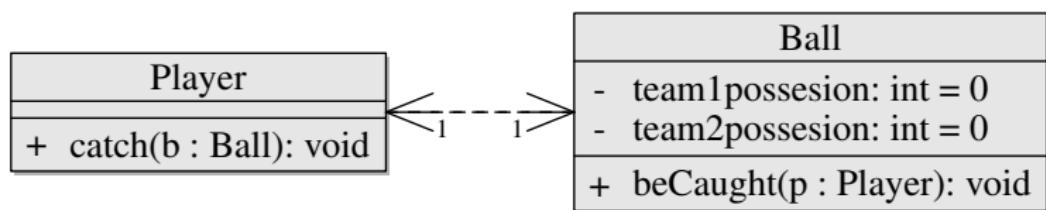
Asocierea

- Reprezintă o relație între clase private ca entități independente (...is using...)
- Fiecare entitate are propriul ciclu de viață
- Relația poate fi unu-la-unu, unu-la-mulți, mulți-la-unu, mulți-la-mulți

Asocierea: exemplu

```
public class Player {  
    public void catch(Ball b) {  
        b.beCaught(this);  
    }  
}  
  
public class Ball {  
    private team1Possession = 0, team2Possession = 0;  
    public void beCaught(Player p) {  
        if (p.getTeam() == team1) {  
            team1Possession++;  
        } else {  
            team2Possession++;  
        }  
    }  
}
```

Asocierea: UML



Agregarea

Agregare

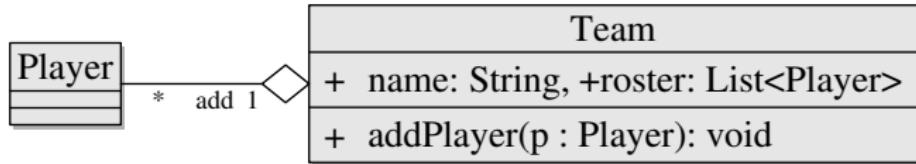
Agregarea

- o clasă conține o referință către o altă clasă (...has a...)
- relația HAS—A
- este unidirecțională (*one way association*)
- *weak association* - obiectul container poate exista și fără obiectele aggregate

Agregare: exemplu

```
public class Player {  
    ...  
}  
  
public class Team {  
    public String name;  
    public List<Player> roster;  
    ...  
    public addPlayer(Player p) {  
        roster.add(p);  
    }  
}
```

Agregare: UML



Compunerea

Compunere

Compunerea

- *strong association* - clasele sunt dependente una de alta și nu pot exista una fără cealaltă
- numită și "Death relationship"¹

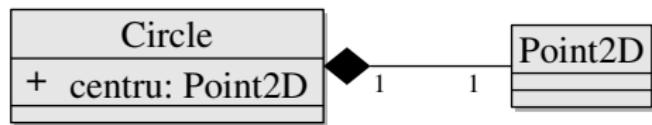
¹tutorial bun despre tipurile de asociere:

<http://www.codeproject.com/Articles/330447/Understanding-Association-Aggregation-and-Composit>

Componere: exemplu

```
public class Point2D {  
    ...  
}  
  
public class Circle {  
    public Point2D centru;  
    public Float raza;  
}
```

Componere: UML



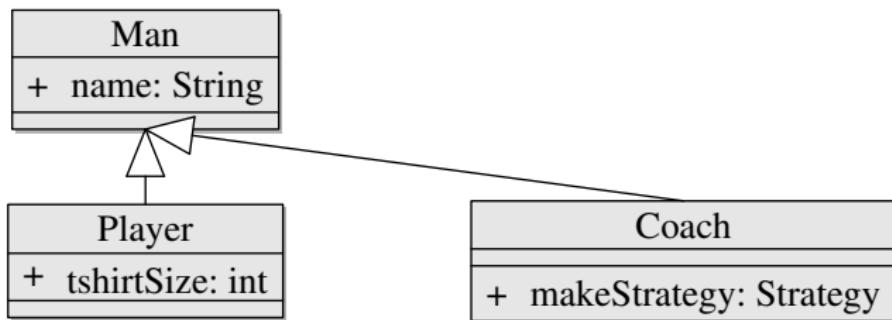
Moștenirea

- Permite unei clase să fie subclasa unei alte clase, superclasa (...is a...)
- În Java o clasă poate extinde doar o singură clasă (revenim cand o sa vorbim de Moștenire multipla data viitoare)
- O subclasă moștenește toate câmpurile și metodele public și protected ale superclasei
- Constructorii sunt apelați ierarhic: constructorul subclasei apelează constructorul superclasei
 - nu se apelează explicit dacă constructorul superclasei nu are parametri sau este 'default'
 - se apelează cu 'super' dacă clasa are doar constructori cu parametri

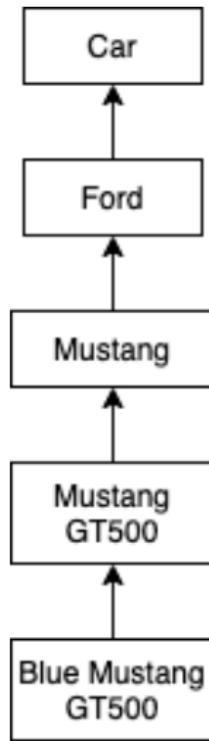
Moștenire: exemplu

```
public class Man {  
    public String name;  
}  
  
public class Player extends Man {  
    public int tshirtNo;  
}  
  
public class Coach extends Man {  
    public Strategy makeStrategy() {  
        ...  
    }  
}
```

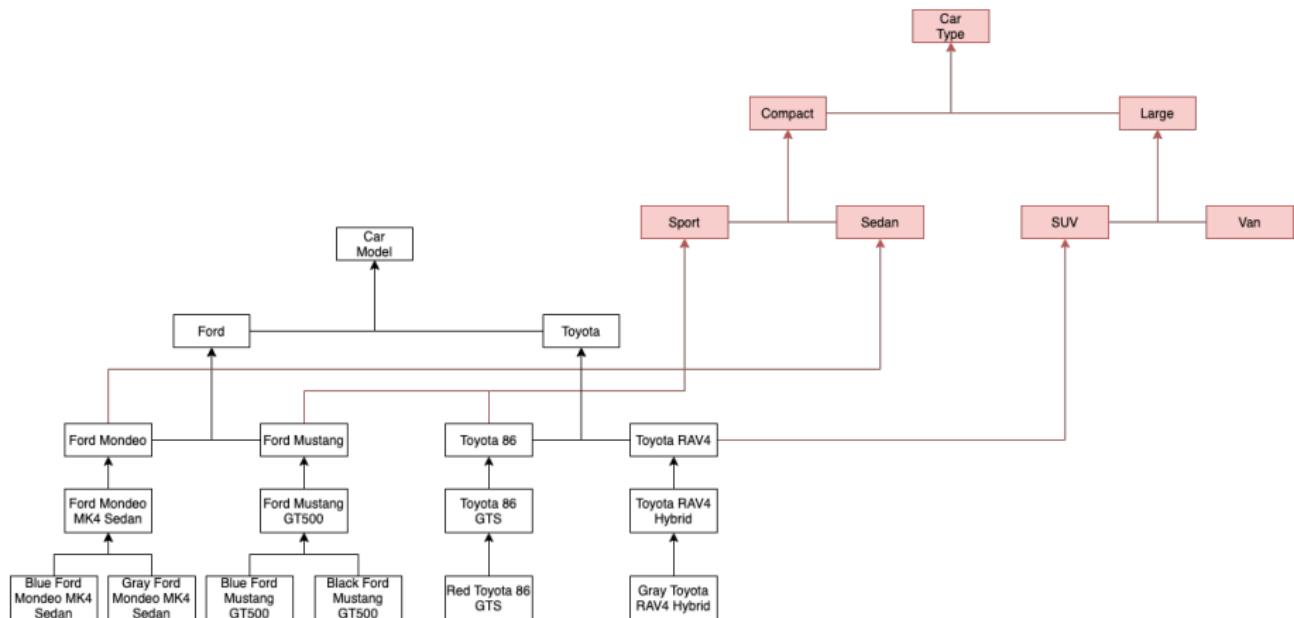
Moștenire: UML



Moștenire: UML



Moștenire: UML



Overriding (suprascrierea)

O clasă poate suprascrie orice metodă dintr-o superclasă a sa prin definirea unei metode cu aceeași semnătură (inclusiv constructori).

Overriding (suprascrierea)

O clasă poate suprascrie orice metodă dintr-o superclasă a sa prin definirea unei metode cu aceeași semnătură (inclusiv constructori).

În metoda din subclasă se poate folosi cuvantul cheie `super` pentru a apela metoda suprascrisă din superclasă:

```
public class Man {  
    public void train(Gym gym) {  
        gym.liftWeights();  
    }  
  
    public class Player extends Man {  
        public void train(Gym gym) {  
            super.train(gym);  
            gym.cardio();  
        }  
    }  
}
```

Overriding (suprascrierea)

Dar atentie la folosirea acestui mecanism pentru a nu ajunge in situatii nedorite:

```
public class Man {  
    public void train(Gym gym) {  
        gym.enter();  
        gym.liftWeights();  
        gym.exit();  
    }  
  
    public class Player extends Man {  
        public void train(Gym gym) {  
            super(gym);  
            gym.cardio();  
        }  
    }  
}
```

Overriding și prezența constructorilor

Următorul cod compilează sau nu?

```
public class Man {  
    protected String name;  
    public Man(String name){  
        this.name = name;  
    }  
  
    public class Coach extends Man {  
    }
```

Overriding și prezența constructorilor

Soluția 1: constructor default în superclasă

```
public class Man {  
    protected String name;  
    public Man() {  
        this.name = "anonim";  
    }  
    public Man(String name){  
        this.name = name;  
    }  
}  
  
public class Coach extends Man {  
}
```

Overriding și prezența constructorilor

Solutia 2: suprascriem constructorul cu parametrii în subclasă

```
public class Man {  
    protected String name;  
    public Man(String name){  
        this.name = name;  
    }  
}  
  
public class Coach extends Man {  
    public Coach(String name){  
        super(name);  
    }  
}
```

Overriding: exemplu `toString`

Implicit, orice clasă moștenește **java.lang.Object** deci are metodele: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Overriding: exemplu `toString`

Implicit, orice clasă moștenește `java.lang.Object` deci are metodele: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`
Dar ar trebui suprascrise pentru a face ceva relevant pentru clasa respectivă.

Overriding: exemplu `toString`

Implicit, orice clasă moștenește `java.lang.Object` deci are metodele: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`
Dar ar trebui suprascrise pentru a face ceva relevant pentru clasa respectivă.

```
public class Man {  
    public String name;  
}  
  
public class Player extends Man {  
    public int tshirtNo;  
    public String toString() {  
        return "("+tshirtNo+") "+name;  
    }  
}
```

Overloading (supraîncarcarea): nu confundați cu Overriding

O clasă poate avea oricâte metode cu același nume, dacă listele de parametrii sunt diferite.

Overloading (supraîncarcarea): nu confundați cu Overriding

O clasă poate avea oricâte metode cu același nume, dacă listele de parametrii sunt diferite.

```
public class Adder {  
    public Integer sum(Integer a, Integer b) {  
        return a+b;  
    }  
  
    public Integer sum(List<Integer> numbers) {  
        Integer result = 0;  
        for (Integer number : numbers) {  
            result += number;  
        }  
        return result;  
    }  
}
```

Suprascriere și supraîncărcare operatori

Operator overloading and overriding are not supported in Java.

```
class Point{
    public double x;
    public double y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }

    public Point add(Point other){
        this.x += other.x;
        this.y += other.y;
        return this;
    }
}

Point a = new Point(1,1);
Point b = new Point(2,2);
a.add(b);
```

Upcast

Casting to a supertype. Se întâmplă automat.

```
public class Man {  
    protected String name;  
    public String getName() {  
        return name;  
    }  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
    }  
}  
  
public class Coach extends Man {  
    public String getName() {  
        return "coach "+name;  
    }  
}  
  
Man mike = new Man("Mike");  
Man charlie = new Coach("Charlie"); // automat  
mike.greet(charlie);
```

Tip declarat vs instantiat

```
public class Man {  
    protected String name;  
    public String getName() {  
        return name;  
    }  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
    }  
}  
  
public class Coach extends Man {  
    public String getName() {  
        return "coach "+name;  
    }  
}  
  
Man mike = new Man("Mike");  
Man charlie = new Coach("Charlie"); // runtime coach  
mike.greet(charlie);
```

Downcast

Casting to a subtype. Trebuie făcut manual.

```
public class Man {  
    protected String name;  
    public String getName() {  
        return name;  
    }  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
        System.out.println(((Coach)otherGuy).giveSignature());  
    }  
}  
  
public class Coach extends Man {  
    public String getName() { return "coach "+name; }  
    public String giveSignature() { return "signature"; }  
}  
  
Man mike = new Man("Mike");  
Man charlie = new Coach("Charlie");  
mike.greet(charlie);
```

Downcast

ClassCastException la rulare

```
public class Man {  
    protected String name;  
    public String getName() {  
        return name;  
    }  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
        System.out.println(((Coach)otherGuy).giveSignature());  
    }  
}  
  
public class Coach extends Man {  
    public String getName() { return "coach "+name; }  
    public String giveSignature() { return "signature"; }  
}  
  
Man mike = new Man("Mike");  
Man charlie = new Man("Charlie");  
mike.greet(charlie);
```

Downcast

Ca să nu riscăm ClassCastException (eroare la rulare) putem folosi instanceof:

```
public class Man {  
    ...  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
        if (otherGuy instanceof Coach)  
            System.out.println(((Coach)otherGuy).giveSignature());  
    }  
}  
...  
  
Man mike = new Man("Mike");  
Man charlie = new Man("Charlie");  
mike.greet(charlie);
```

Downcast

Ca să nu riscăm ClassCastException (eroare la rulare) putem folosi supraîncărcarea:

```
public class Man {  
    ...  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
    }  
    public void greet(Coach otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
        System.out.println(((Coach)otherGuy).giveSignature());  
    }  
}  
...  
  
Man mike = new Man("Mike");  
Coach charlie = new Coach("Charlie");  
mike.greet(charlie);
```

Revenim la ideea asta cand discutam despre Visitor.



Tipul declarat vs instantiat

Ca să nu riscăm ClassCastException (eroare la rulare) putem folosi supraîncărcarea:

```
public class Man {  
    ...  
    public void greet(Man otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
    }  
    public void greet(Coach otherGuy) {  
        System.out.println("Hi "+otherGuy.getName());  
        System.out.println(((Coach)otherGuy).giveSignature());  
    }  
}  
  
...  
  
Man mike = new Man("Mike");  
Man charlie = new Coach("Charlie"); // atentie la tip!  
mike.greet(charlie);
```

Reading Assignments

- Lab 03: agregare și moștenire
- Understanding Association, Aggregation, and Composition
- UML Association vs Aggregation vs Composition
- Generate UML class diagrams from IntelliJ IDEA
- Generate UML Class Diagram from Java Project

Incapsulare, imutabilitate si final, static vs instantе, Singleton

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

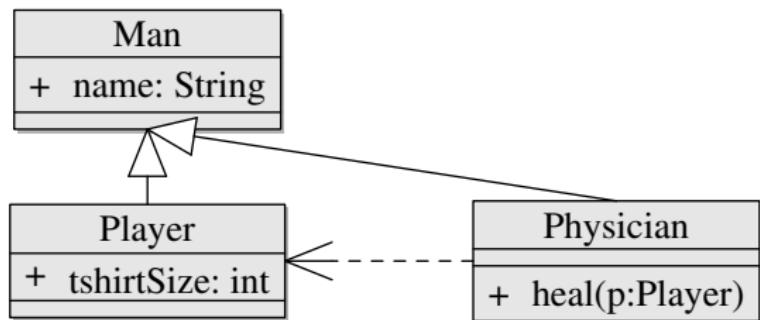
OOP, 2021



Curs 2: ce contine o clasa

Man
+ name: String

Curs 3: relatia dintre clase



Static vs instante

static

static desemnează membrii și metode independente de instanțele clasei

```
public class BasketBall {  
    public static int size = 12;  
    private String color;  
  
    Ball(String color) {  
        this.color = color;  
    }  
}
```

static vs non-static

ceea ce se petrece într-o funcție statică ar trebui să poată funcționa chiar dacă obiectul a fost instantiat sau nu

static vs non-static

ceea ce se petrece într-o funcție statică ar trebui să poată funcționa chiar dacă obiectul a fost instantiat sau nu

```
public class TestApp {  
    int count;  
    public static void main(String[] args){  
        count++;  
    }  
}
```

static vs non-static

ceea ce se petrece într-o funcție statică ar trebui să poată funcționa chiar dacă obiectul a fost instantiat sau nu

```
public class TestApp {  
    int count;  
    public static void main(String[] args){  
        count++;  
    }  
}
```

Primim eroare la compilare:

"Cannot make a static reference to the non-static field count"

Exercitiu: Cannot make a static reference to the non-static field count

Exercitiu: Cannot make a static reference to the non-static field count

static vs non-static

Dacă count chiar nu este static, o posibilă rezolvare ar fi:

```
public class TestApp {  
    int count;  
    public static void main(String[] args){  
        TestApp app = new TestApp();  
        app.incrementCount();  
    }  
    public void incrementCount() {  
        count++;  
    }  
}
```

Imutabilitate si final

Date primitive constante

În Java, cuvântul cheie final

```
public class Album {  
  
    public final int tracks = 15;  
    public final int releaseYear;  
  
    Album(int releaseYear) {  
        this.releaseYear = releaseYear;  
    }  
}
```

Obiecte constante

```
public class Boxer {  
    public final int birthYear;  
    public String name;  
  
    Boxer(String name, int birthYear) {  
        this.name = name;  
        this.birthYear = birthYear;  
    }  
}  
...  
final Boxer champion = new Boxer("Cassius Clay", 1942);
```

Obiecte constante vs campuri

Variabila referință final permite modificarea internă a obiectului către care indică:

```
public class Boxer {  
    public final int birthYear;  
    public String name;  
  
    Boxer(String name, int birthYear) {  
        this.name = name;  
        this.birthYear = birthYear;  
    }  
}  
...  
final Boxer champion = new Boxer("Cassius Clay", 1942);  
champion.name = "Muhammad Ali"; // ok
```

Obiecte constante vs campuri

O încercare nouă de a asigna o variabilă final rezultă în eroare de compilare

```
public class Boxer {  
    public final int birthYear;  
    public String name;  
  
    Player(String name, int birthYear) {  
        this.name = name;  
        this.birthYear = birthYear;  
    }  
}  
...  
final Boxer champion = new Boxer("Cassius Clay", 1942);  
champion = new Boxer("Joe Frazier", 1952); // eroare  
champion.birthYear = 2016; // eroare
```

Immutable objects

Immutable objects: toate atributele unui obiect admit o unică initializare:

```
public class Ball {  
    public final String color;  
    public Ball(String color) {  
        this.color = color;  
    }  
    Ball basketball = new Ball("brown");
```

Immutable objects

Immutable objects: toate atributele unui obiect admit o unică initializare:

```
public class Ball {  
    public final String color;  
    public Ball(String color) {  
        this.color = color;  
    }  
    Ball basketball = new Ball("brown");
```

Așa sunt, spre exemplu, obiectele de tip String sau Integer

Vectori constanti

- Nu folosiți `final` pentru array-uri
 - array-ul e `final`, însă obiectele din el pot fi modificate :)
 - folosiți colecții nemodificabile¹

¹exemplu Chapter 4, pg 70 in Joshua Bloch. 2008. Effective Java (2nd Edition) (The Java Series) (2 ed.). Prentice Hall

Clase care nu pot fi mostenite

Puteți folosi final pentru clase care doriti sa nu poata fi mostenite

```
public final class Animal {  
    ...  
}  
public class Dog extends Animal {  
    ...  
}
```

error: cannot inherit from final Animal

Immutable vs composition

Puteti marca drept final obiectele folosite in relatia de Componere

```
public class Punct2D {  
    ...  
}  
public class Circle {  
    public final Punct2D centru;  
    ...  
}
```

Singleton

Ce sunt Design Patterns?

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- Christopher Alexander

Ce sunt Design Patterns?

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- Christopher Alexander

e.g. Courtyard

Clasificare Design Patterns

- *creational patterns*: definesc mecanisme de creare a obiectelor (GoF)
- *structural patterns*: definesc relații între entități (GoF)
- *behavioral patterns*: definesc comunicarea între entități (GoF)
- *concurrency patterns*: definesc mecanisme utile în programarea pe paralela și distribuită (sincronizare etc.)
- *architectural patterns*: descriu structura întregului sistem (e.g. MVC, multi-tier etc.)

GoF Design Patterns

E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)
"Design Patterns: Elements of Reusable Object-Oriented Software"

Creational	Structural	Behavioral
Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Visitor

Description

Singleton ensures a class only has one instance, and provides a global point of access to it.

Problem that the pattern solves

- uneori este nevoie sa avem o singura instanta a unei clase

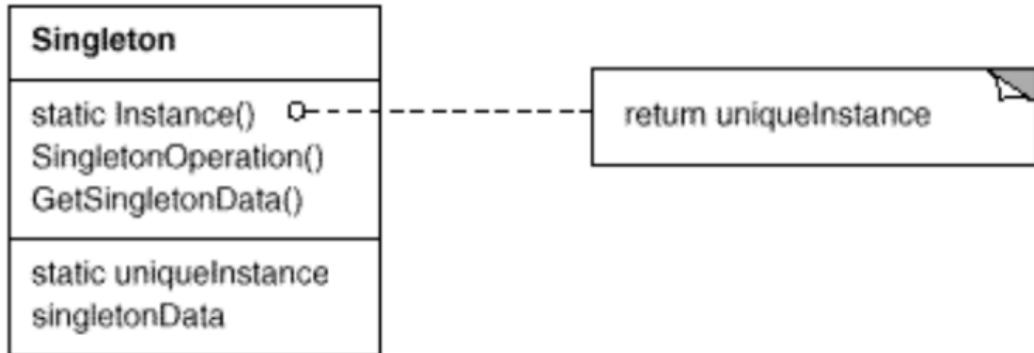
Problem that the pattern solves

- uneori este nevoie sa avem o singura instanta a unei clase
- ar trebui sa ne putem razgandi relativ usor

Problem that the pattern solves

- uneori este nevoie sa avem o singura instanta a unei clase
- ar trebui sa ne putem razgandi relativ usor
- trebuie sa fie clar pentru oricine foloseste codul nostru cum poate obtine acea instantă

Solution Structure



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

Exercitiu: Singleton

Exercitiu: Singleton

Implementation Examples

```
public class ClassicSingleton {  
  
    private static ClassicSingleton instance = null;  
    private ClassicSingleton() {  
        // Exists only to defeat instantiation.  
    }  
  
    public static ClassicSingleton getInstance() {  
        if(instance == null) {  
            instance = new ClassicSingleton();  
        }  
        return instance;  
    }  
}  
  
ClassicSingleton c = ClassicSingleton.getInstance();
```

Solution Details

- clasa incapsuleaza unica sa instanta si gestioneaza accesul la ea

Solution Details

- clasa incapsuleaza unica sa instanta si gestioneaza accesul la ea
- exista alternativa de a folosi metode statice, dar este mai rigida (in C++ nu pot fi extinse, in general mai greu sa ne razgandim)

Consequences

- acces controlat la unica instantă

Consequences

- acces controlat la unica instantă
- evita poluarea cu variabile globale

Consequences

- acces controlat la unica instantă
- evita poluarea cu variabile globale
- clasa poate fi rafinata simplu prin extindere

Consequences

- acces controlat la unica instantă
- evita poluarea cu variabile globale
- clasa poate fi rafinata simplu prin extindere
- e usor sa ne razgandim si sa permitem mai multe instante, chiar putem pune o limita a numarului de instante

Consequences

- acces controlat la unica instantă
- evita poluarea cu variabile globale
- clasa poate fi rafinata simplu prin extindere
- e usor sa ne razgandim si sa permitem mai multe instante, chiar putem pune o limita a numarului de instante
- mai flexibila decat alternativa de a folosi metode statice

Încapsulare

Definition

Încapsularea este proprietatea claselor de obiecte de:

- a grupa datele și metodele aplicabile asupra datelor
- a proteja accesul la acestea față de utilizarea eronată

Încapsulare

Cum contribuie urmatoarele elemente la incapsulare?

- ce poate contine o clasa
- specificalor de acces
- setter si getter

Încapsulare

```
class GeneratorNume {  
    private String[] vocabulary = {new String("Ana"), new  
        String("Mihai"), new String("Robert")};  
    private Random randomGenerator = new Random();  
  
    public String genNume() {  
        return vocabulary[randomGenerator.nextInt(3)];  
    }  
}  
...  
GeneratorNume gn = new GeneratorNume();  
System.out.println(gn.genNume());
```

Încapsulare vs Mostenire

Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation". Inheritance and composition each have their advantages and disadvantages.

- E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four),
"Design Patterns" (Introduction)

Exercitiu: Inheritance vs Composition

Exercitiu: Inheritance vs Composition

Reading material

- Lab 4: Static, Final, Singleton Design Pattern
- Inheritance vs Compositions, Introduction, "Design Patterns", E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)
- Delegation, Introduction, "Design Patterns", E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)

Clase Abstracte si Interfete

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

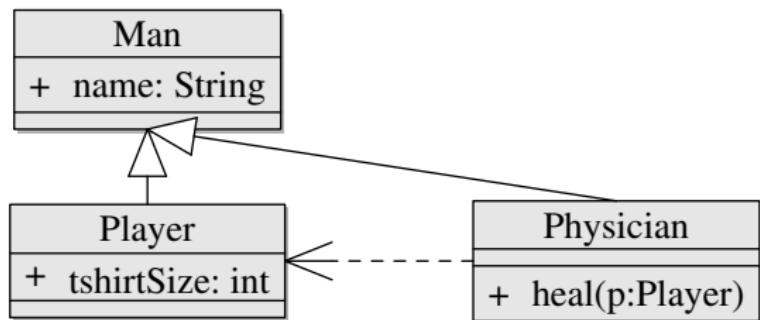
OOP, 2021



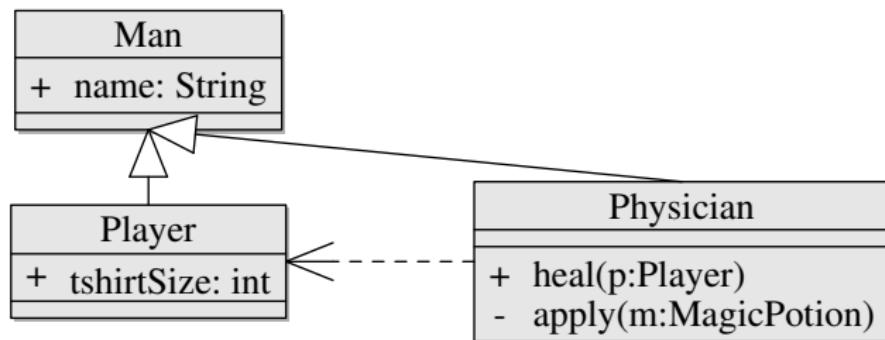
Curs 2: ce contine o clasa

Man
+ name: String

Curs 3: relatia dintre clase

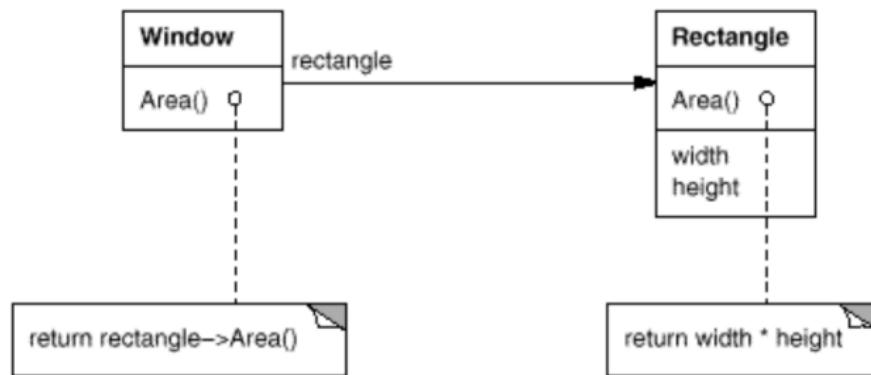


Curs 4: incapsulare, static, final



Delegation example

Pentru a putea schimba implementarea la run-time, avem nevoie de interfețe



Delegation, Introduction, "Design Patterns", E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)

Interfete

Exemplu de interfata cu utilizatorul



Ce este o interfata?

Ganditi-va la o aplicatie care monitorizeaza activitatea angajatilor unei firme

```
public interface Payee {  
    int computeMoney(Month paymentMonth);  
}  
  
public class Engineer implements Payee {  
    private Contract contract;  
    public int computeMoney(Month paymentMonth) {  
        if (daysWorked + paidLeave == paymentMonth.  
            getWorkingDays())  
            return contract.getSalary();  
    }  
}  
  
public class SalesRep implements Payee {  
    private Contract contract;  
    private int newCustomers;  
    public int computeMoney(Month paymentMonth) {  
        return contract.getSalary() + newCustomers * contract.  
            getNewCustomerBonus();  
    }  
}
```

Ce este o interfata?

Ganditi-vă la o aplicație care monitorizează activitatea angajaților unei firme

Din punct de vedere payroll, aplicația va funcționa luna de luna, indiferent cătă și ce fel de angajați are firma.

```
public class FinancialManager {  
    private BankAccount ba;  
    private List<Payee> payroll;  
    public executePayroll(Month paymentMonth) {  
        for (Payee payee : payroll) {  
            ba.pay(payee.computeMoney(paymentMonth));  
        }  
    }  
}
```

Ce este o interfata?

Definition

Interfata este un tip de date care reprezinta, in forma sa de baza, o grupare de semnaturi de metode (fara implementari) unite de un scop comun.

Interfata este un contract care defineste interactiunea a doua componente software. Se foloseste la:

- schimbarea dinamica a componentelor

Ce este o interfata?

Ganditi-vă ca vi se cere extinderea aplicatiei pentru un spital

```
public interface Payee {  
    int computeMoney(Month paymentMonth);  
}  
  
public class Nurse implements Payee {  
    private Contract contract;  
    public int computeMoney(Month paymentMonth) {  
        if (daysWorked + paidLeave == paymentMonth.  
            getWorkingDays())  
            return contract.getSalary();  
    }  
}  
  
public class Doctor implements Payee {  
    private Contract contract;  
    public int computeMoney(Month paymentMonth) {  
        return contract.getSalary() + guardShifts * contract.  
            getGuardShiftBonus();  
    }  
}
```

Ce este o interfata?

Ganditi-vă ca vi se cere extinderea aplicatiei pentru un spital
Din punct de vedere payroll, aplicatia ramane neschimbata

```
public class FinancialManager {  
    private BankAccount ba;  
    private List<Payee> payroll;  
    public executePayroll(Month paymentMonth) {  
        for (Payee payee : payroll) {  
            ba.pay(payee.computeMoney(paymentMonth));  
        }  
    }  
}
```

Ce este o interfata?

Definition

Interfata este un tip de date care reprezinta, in forma sa de baza, o grupare de semnaturi de metode (fara implementari) unite de un scop comun.

Interfata este un contract care defineste interacțiunea a doua componente software. Se foloseste la:

- schimbarea dinamica a componentelor
- decuplarea si reutilizarea componentelor

Ce poate contine o interfață?

- semnaturi de metode

```
public interface Payee {  
    int calcMoney(Month paymentMonth);  
}
```

Toate sunt public în mod automat (nu se trec specificalor de acces)

Ce poate contine o interfață?

- semnaturi de metode
- metode statice (inclusiv implementări)
- constante (static final)

```
public interface Payee {  
    static final String PAYER = "SuperStar Basket Club";  
    static final Bank PAYING_BANK = "Rich Swiss Bank"  
    static String printPayerData() {  
        System.out.println(PAYING_BANK+" on the behalf of "+  
                           PAYER);  
    }  
}
```

Toate sunt public în mod automat (nu se trec specifătorii de acces)

Ce poate contine o interfață?

- semnaturi de metode
- metode statice (inclusiv implementări)
- constante (static final)
- metode default (inclusiv implementări) din Java 8, implicatii asupra mostenirii multiple

```
public interface Payee {  
    default Contract signContract() {  
        return new Contract("Payer Inc.");  
    }  
}
```

Toate sunt public în mod automat (nu se trec specificalor de acces)

Interfete celebre: List

```
List<CEO> topCEOs = new ArrayList<CEO>();  
topCEOs.add(new CEO("Steve Jobs"));  
topCEOs.add(new CEO("Bill Gates"));  
topCEOs.add(new CEO("Jeff Bezos"));  
topCEOs.add(new CEO("Elon Musk"));
```

Interfete celebre: Comparable

```
public class CEO implements Comparable {
    public int netWorth;
    public int compareTo(CEO otherCEO) {
        return netWorth - otherCEO.netWorth();
    }
}
.
.
Collections.sort(topCEOs);
```

Evolving interfaces

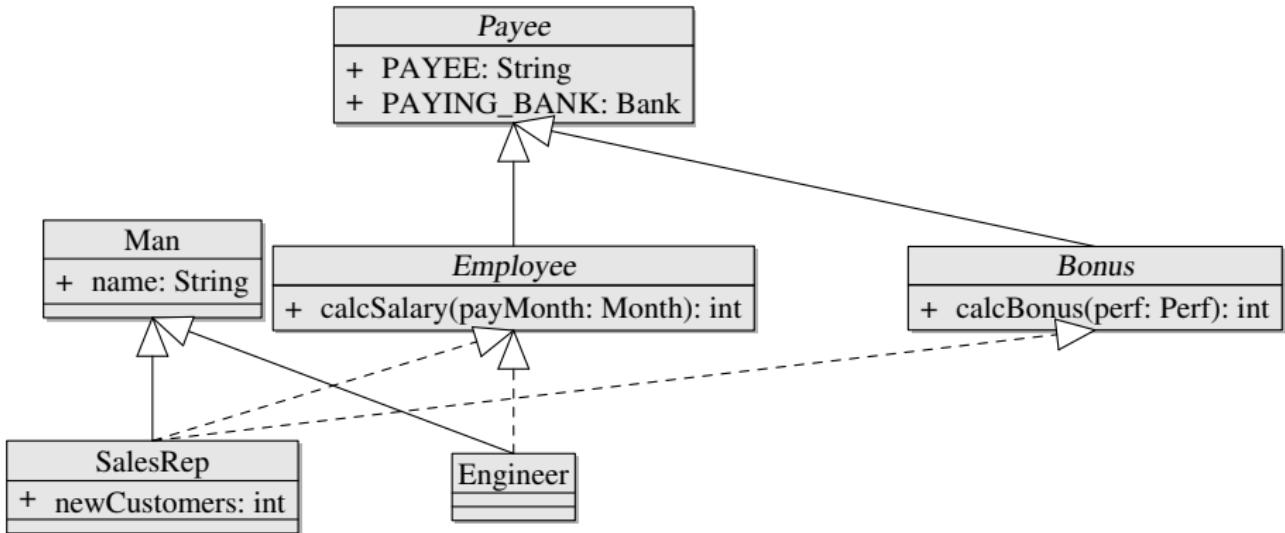
```
public interface Payee {  
    static final String PAYER = "SuperStar Basket Club";  
    static final Bank PAYING_BANK = "Rich Swiss Bank"  
    static String printPayerData() {  
        System.out.println(PAYING_BANK+" on the behalf of "+  
                           PAYER);  
    }  
    default Contract signContract() {  
        return new Contract(PAYER);  
    }  
}  
  
public interface Employee extends Payee {  
    int computeSalary(Month paymentMonth);  
}  
  
public interface Bonus extends Payee {  
    int computeBonus(Performance performance);  
}
```

Evolving interfaces

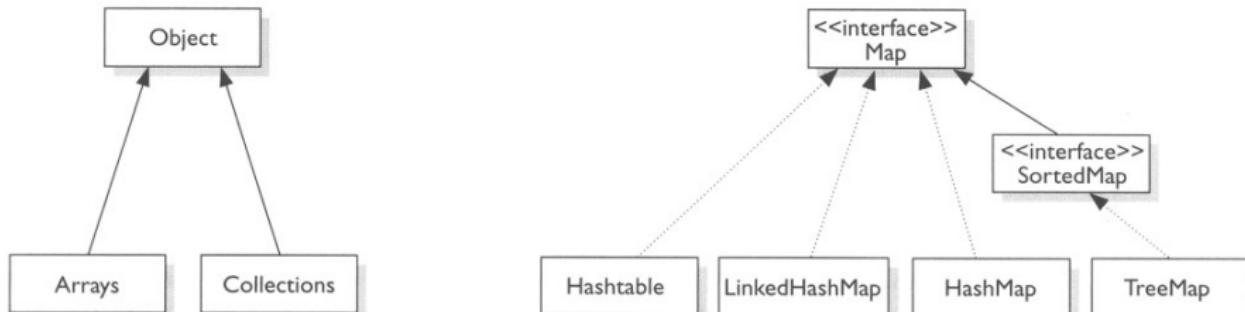
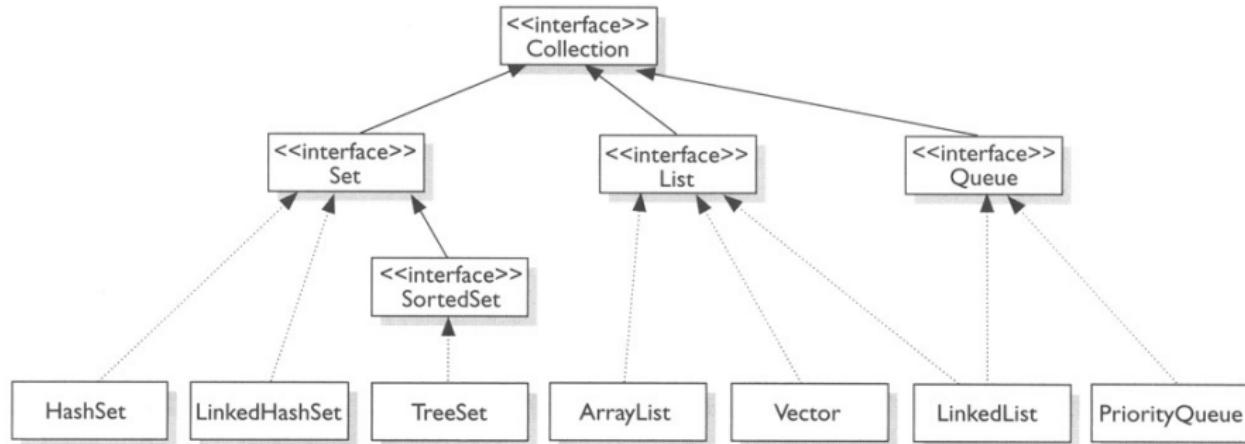
```
public class SalesRep extends Man implements Employee, Bonus
{
    ...
}

public class Engineer extends Man implements Employee {
    ...
}
```

Evolving interfaces



Interfete celebre: o familie de colectii



Relatii intre interfete si intre interfete si clase

- o interfata poate extinde mai multe interfete (mostenire)
- o interfata poate fi implementata de mai multe clase
- o interfata nu poate fi instantiata
- o clasa poate implementa mai multe interfete

Clase Abstracte

Nevoia unor entitati intermediare

Payee:

- signContract este valabil pentru toti cei care sunt platiti de companie, deci poate fi implementat din start

Nevoia unor entitati intermediare

Payee:

- signContract este valabil pentru toti cei care sunt platiti de companie, deci poate fi implementat din start
- calcMoney este variabil in functie de cei care extind Payee (Employee, Bonus) deci poate fi declarat abstract si lasat spre implementare acestora

Definition

Clasa abstracta este o clasa declarata ca abstracta prin utilizarea cuvantului cheie "abstract". Desi nu este obligatoriu, acest lucru are sens sa se intampla daca cel putin una din metodele sale este abstracta (adica declarata fara implementare). Reciproca insa este obligatorie: daca o clasa are cel putin o metoda abstracta, toata clasa trebuie marcata ca abstracta.

Ce poate contine o clasa abstracta

- orice poate contine o clasa obisnuită, doar că este marcată ca abstractă (de fapt o clasa obisnuită poate fi marcată ca abstractă fără vreun motiv anume)
- elegant ar fi ca cel puțin una din metodele unei clase abstracte să fie abstractă (adică semnatura să aibă cuvântul cheie `abstract` și să nu aibă implementare)

Specificatorii de acces se pastrează ca la o clasa obisnuită.

De ce sa folosim o clasa abstracta?

Clasa abstracta ofera o cale de mijloc intre clase si interfete, oferind posibilitatea de a crea clase incomplete, care vin cu un contract pentru a fi complet implementate de clasele ce le mostenesc:

- cand definim intr-un mod corect ceea ce au in comun clasele derive

De ce sa folosim o clasa abstracta?

Clasa abstracta ofera o cale de mijloc intre clase si interfete, oferind posibilitatea de a crea clase incomplete, care vin cu un contract pentru a fi complet implementate de clasele ce le mostenesc:

- cand definim intr-un mod corect ceea ce au in comun clasele derivele
- cand dorim sa oferim o implementare paritala unei clase, ca sa nu duplicam cod (desi din Java 8 acest lucru se poate face si cu interfete si metode default)

Famous example: AbstractList

► [AbstractList reference](#)

Relatii intre clase abstracte, interfete si clase

- o clasa poate extinde o clasa abstracta daca ii implementeaza toate metodele abstracte
- daca o clasa extinde o clasa abstracta si nu ii implementeaza toate metodele abstracte, atunci, la randul sau, va fi marcata ca abstracta

Interfete vs Clase Abstracte

	Interfata	Clasa Abstracta
Se poate instantia	nu	nu
Metode	cu sau fara implementare	cu sau fara implementare
Membrii	static final	de orice fel
Specificatori acces	public (automat)	public, private, protected, default
Relatia cu alte clase	o clasa poate implementa oricate interfete	o clasa poate extinde o singura clasa abstracta

Cand sa folosim Interfete si cand Clase Abstracte?

Use abstract classes when:

- sharing code among several closely related classes, or
- classes that extend the abstract class have many common methods or fields, or require access modifiers other than public, or
- non-static or non-final fields are needed (methods that can access and modify the state of the object to which they belong)

Cand sa folosim Interfete si cand Clase Abstracte?

Use abstract classes when:

- sharing code among several closely related classes, or
- classes that extend the abstract class have many common methods or fields, or require access modifiers other than public, or
- non-static or non-final fields are needed (methods that can access and modify the state of the object to which they belong)

Use interfaces when:

- unrelated classes would implement the interface, or
- specifying the behavior of a particular data type, but not concerned about who implements its behavior, or
- multiple inheritance of type is needed

Pentru mai multe detalii

[▶ Tutorialul despre Clase si Metode Abstracte](#)

Mostenirea multipla

Multiple inheritance

Definition

Mostenirea multipla este o functionalitate oferita de unele limbaje de programare prin care o clasa derivata preia caracteristicile mai multor clase parinte.

Multiple inheritance

Definition

Mostenirea multipla este o functionalitate oferita de unele limbaje de programare prin care o clasa derivata preia caracteristicile mai multor clase parinte.

In Java:

- o clasa poate extinde o singura alta clasa
- o clasa poate implementa mai multe interfete

Definition

Mostenirea multipla este o functionalitate oferita de unele limbaje de programare prin care o clasa derivata preia caracteristicile mai multor clase parinte.

In Java:

- o clasa poate extinde o singura alta clasa
- o clasa poate implementa mai multe interfete

Din Java 8:

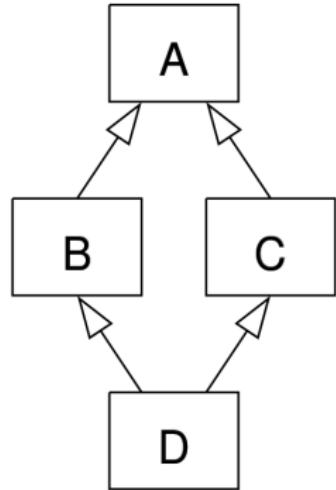
- o interfata poate contine metode default
- astfel, se poate implementa Mostenirea Multipla printr-o clasa care implementeaza mai multe interfete cu metode default

Problema diamant

Problema diamant ("diamond problem", aka "deadly diamond of death")

Problema diamant

Problema diamant ("diamond problem", aka "deadly diamond of death") se refera la ambiguitatea care apare atunci cand doua clase B si C mostenesc o clasa A, iar o alta clasa D mosteneste atat B, cat si C.



Daca exista o metoda in A, pe care o suprascriu atat B, cat si C, dar D nu o suprascrie, atunci D pe care din cele doua variante o va mosteni?

Problema diamant in Java 8 și inheritance prin interfaces

Programatorului îi este solicitat să rezolve ambiguitatea apelând
B.super.m(...) sau C.super.m(...)

► What about the diamond problem? on lambdafaqs.org

Tipuri de mostenire multiplă

- of State: not a problem, interface do not have fields
- of Implementation: default methods
- of Type: using an interface as a type

▶ Multiple Inheritance of State, Implementation, and Type

Mostenire multipla: interfete vs clase

Care este diferența dintre:

- mostenirea multipla în sensul clasic (oferita de limbaje precum C++) în care o clasa poate mosteni mai multe clase
- mostenirea multipla obtinuta prin workaround-ul din Java8+ (o clasa poate implementa mai multe interfete, iar acestea pot mosteni o singura interfata)

Mostenire multipla: interfete vs clase

Care este diferența dintre:

- mostenirea multipla în sensul clasic (oferita de limbaje precum C++) în care o clasa poate mosteni mai multe clase
- mostenirea multipla obtinuta prin workaround-ul din Java8+ (o clasa poate implementa mai multe interfete, iar acestea pot mosteni o singura interfata)

În mostenirea multipla din Java

- se mostenesc implementari, nu și stare (campuri)
- nu se mostenesc constructori

▶ Multiple Inheritance of State, Implementation, and Type

Factory

Description

Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method defines an interface for creating an object, but lets subclasses decide which class to instantiate.

Description

Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method defines an interface for creating an object, but lets subclasses decide which class to instantiate.

AbstractFactory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. A concrete factory is often a Singleton.

Description

	AbstractFactory	Builder
what?	instances of related classes	complex object, step by step
object available	immediately	as a last step

Problem that the pattern solves

- vrem sa generam instante de clase inrudite (mostenesc o interfata comuna sau extind o clasa abstracta comună)

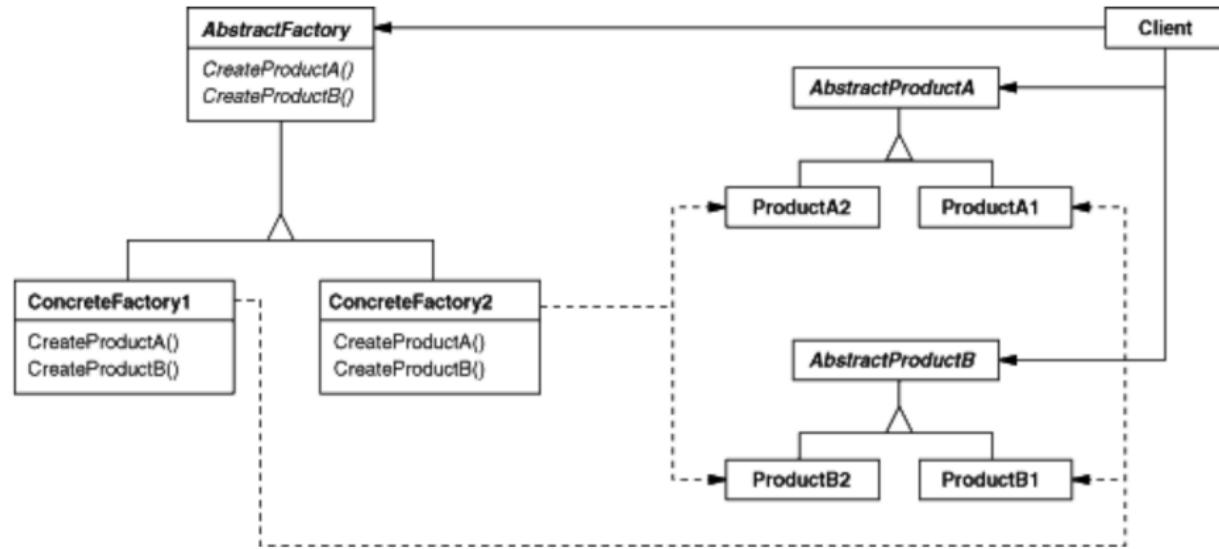
Problem that the pattern solves

- vrem sa generam instante de clase inrudite (mostenesc o interfata comuna sau extind o clasa abstracta comună)
- clasa care solicita instantele nu are nevoie sa stie toate implementarile posibile, ci doar caracteristicile obiectului ce trebuie implementat

Problem that the pattern solves

- vrem sa generam instante de clase inrudite (mostenesc o interfata comuna sau extind o clasa abstracta comună)
- clasa care solicita instantele nu are nevoie sa stie toate implementarile posibile, ci doar caracteristicile obiectului ce trebuie implementat
- vrem sa constrangem clientul sa foloseasca obiectele inrudite impreuna sau sa ii ascundem implementarile

Solution Structure



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

- Products like soups, salads, main dishes etc.
- Factories like ItalianRestaurant, AsianRestaurant etc.

Solution Details

- de cele mai multe ori este suficienta o singura ConcreteFactory (dar sunt situatii in care putem avea mai multe)

Solution Details

- de cele mai multe ori este suficienta o singura ConcreteFactory (dar sunt situatii in care putem avea mai multe)
- crearea concreta a obiectelor apartine ConcreteFactory

Implementation Examples

Urmariti acest [▶ exemplu de Factory in C#](#)

- Cautati care sunt clasele Factory in [▶ Java API](#)
- Urmariti [▶ folosirea StringBuilder](#) si sesizati diferențele dintre Factory si Builder
- Ar trebui sa puteti intelege tot ce scrie in [▶ acest articol](#)
- Ce e prea mult strica: [▶ articol foarte interesant de la unul dintre fondatorii StackOverflow](#)

Consequences

- izoleaza clientul de implementarea concreta a claselor

Consequences

- izoleaza clientul de implementarea concreta a claselor
- se poate schimba familia de produse usor (prin inlocuirea cu un alt Factory), dar adaugarea unui nou tip de produs este mai dificila din perspectiva clientului

Consequences

- izoleaza clientul de implementarea concreta a claselor
- se poate schimba familia de produse usor (prin inlocuirea cu un alt Factory), dar adaugarea unui nou tip de produs este mai dificila din perspectiva clientului
- promoveaza consistenta intre produse

Reading material

- [Lab 5: Clase abstracte si interfete](#)
- [difference between an Interface and an Abstract class?](#)
- [10 Abstract Class and Interface Interview Questions Answers in Java](#)
- [Why I hate frameworks, Benji Smith](#)

Clase Interne

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

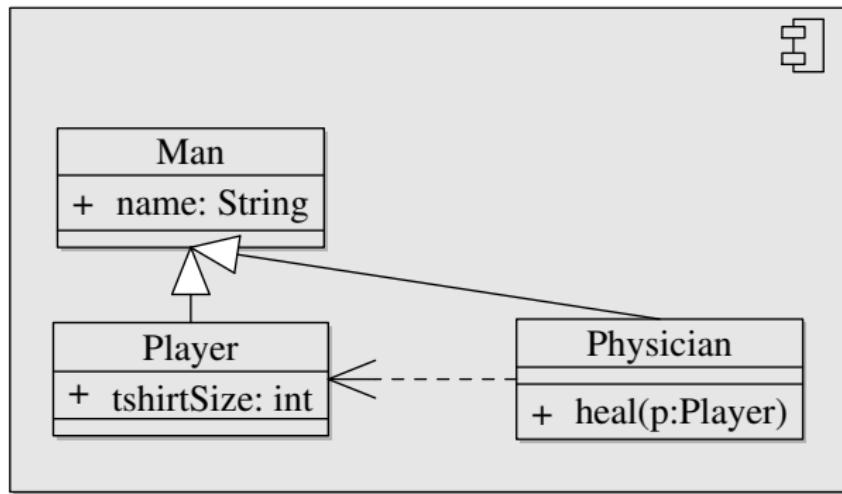
OOP, 2020



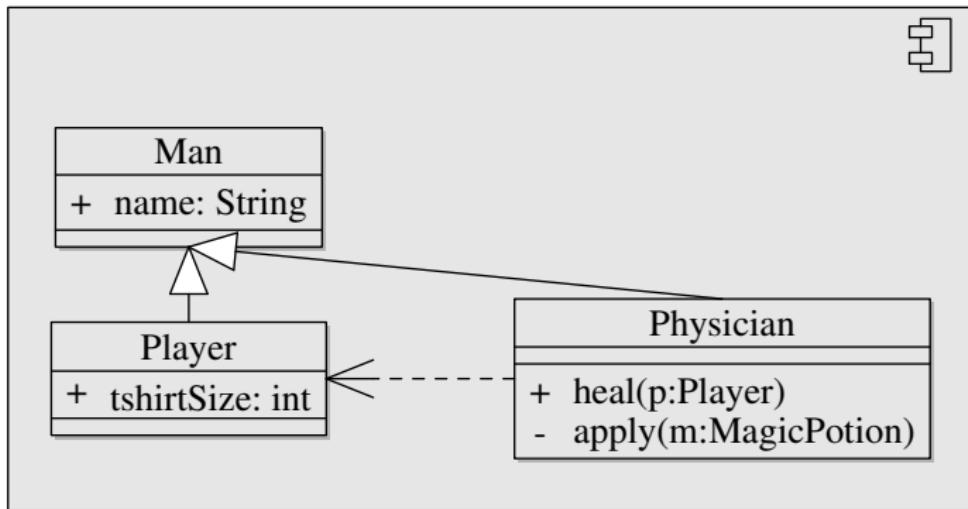
Curs 2: ce contine o clasa

Man
+ name: String

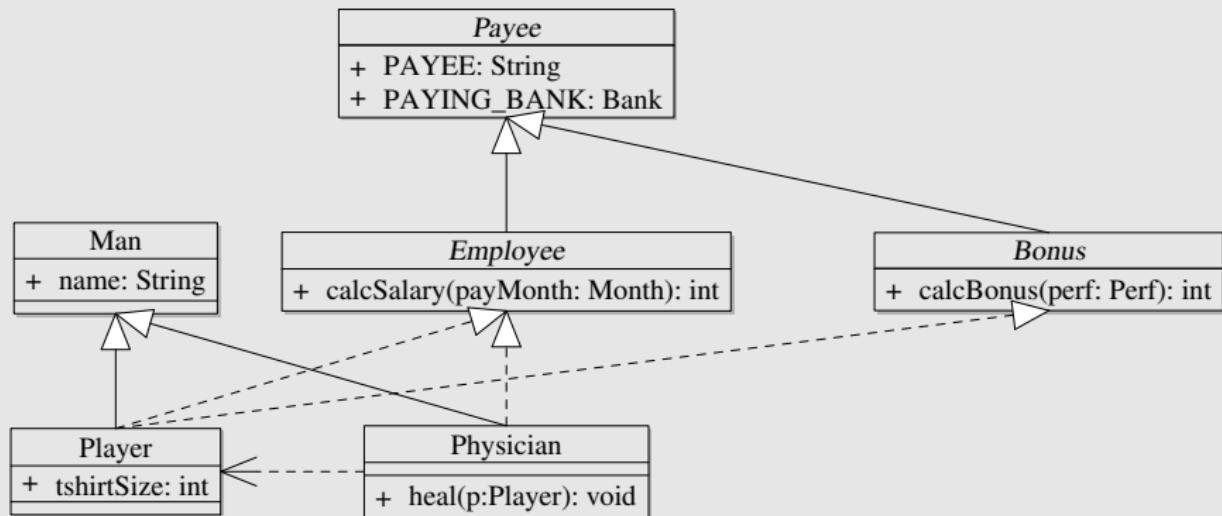
Curs 3: relatia dintre clase



Curs 4: incapsulare, static si final



Curs 5: clase abstracte si interfete



Clase interne (Nested Classes)

Apar situatii in care:

- o clasa este folosita intr-un singur loc in program (in interiorul altrei clase)
- o parte a continutului unei clase ar fi mai clar definita drept o alta clasa (dar are inca nevoie de acces la clasa parinte)

Ce este o clasa internă?

O clasa internă (Nested Class) se defineste în interiorul unei alte clase, denumita clasa externă (Outer Class),

Ce este o clasa internă?

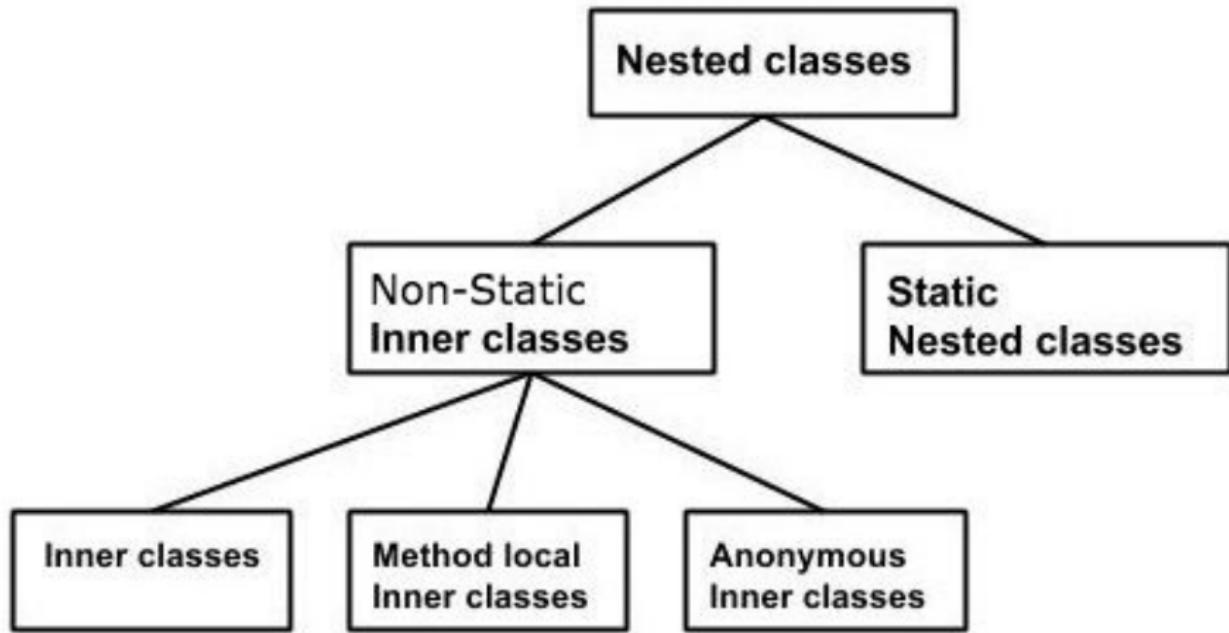
O clasa internă (Nested Class) se defineste în interiorul unei alte clase, denumita clasa externă (Outer Class), în diferite forme (ca membru, în interiorul unei metode, etc.).

Ce este o clasa internă?

O clasa internă (Nested Class) se defineste în interiorul unei alte clase, denumita clasa externă (Outer Class), în diferite forme (ca membru, în interiorul unei metode, etc.).

O clasa care nu este definită în interiorul unei alte clase se numește Top-Level Class.

Tipuri de clase interne



▶ [Inner Classes Tutorial](#)

Inner Classes (clase interne normale)

Poate fi vizibila in exterior, caz in care poate fi accesata prin intermediul unei instante a clasei externe (ori ca rezultat al unei metode ori prin instantiere)

```
public class Person {  
    public class Account {  
        public String bank;  
        public String iban;  
    }  
    public Account getAccount() {  
        Account c = new Account("SwissBank", "CH23...");  
        return c;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Person pl = new Person();  
        Person.Account c1 = pl.getAccount();  
        Person.Account c2 = pl.new Account("USBank", "US23...");  
    }  
}
```

Inner Classes (clase interne normale)

Putem sa accesam instantia clasei externe care a creat-o

```
public class Person {  
    private name;  
    public class Account {  
        public String bank;  
        public String iban;  
        public String toString {  
            return Person.this.name+'s account: '+iban;  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Person pl = new Person("Mike");  
        Person.Account c = pl.new Account("USBank", "US23.."  
            );  
        System.out.println(c); // ce se afiseaza?  
    }  
}
```

Inner Classes (clase interne normale)

Poate fi 'private', caz in care implementarea este ascunsa, dar interfata poate fi publica

```
interface BankAccount {  
}  
  
public class Person {  
    private class Account implements BankAccount {  
        public String bank;  
        public String iban;  
    }  
    public BankAccount getAccount() {  
        Account c = new Account("SwissBank", "CH23...");  
        return c;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Person pl = new Person();  
        BankAccount c3 = pl.getAccount(); // ok  
    }  
}
```

Method Local Inner Classes (clase interne in metode si blocuri)

Vizibilitatea clasei se limiteaza la metoda, respectiv bloc (desi codul binar se creeaza oricum). Pentru a putea folosi variabilele si parametrii metodei, clasa trebuie sa fie final.

```
public class TestMethodLocalInnerClass{
    public static void main(String[] args){
        class Greeter implements Runnable{
            private final String _greeted;
            public Greeter(String greeted){
                super();
                _greeted = greeted;
            }
            public void run(){
                System.out.printf("Hello %s!\n", _greeted);
            }
        }

        new Greeter("world").run();
        new Greeter("dog").run();
    }
}
```

Anonymous Inner Classes (clase anonte)

Daca o clasa este instantiata intr-un singur loc si apoi folosita (prin upcasting) doar prin intermediul clasei de baza sau al interfetei, numele sau nu mai este important.

```
new Thread( new Runnable() {
    Override
    public void run() {
        // do magic on separate thread
    }
}).start();
```

► More about Runnable

Anonymous Inner Classes (clase anomite)

Daca o clasa este instantiata intr-un singur loc si apoi folosita (prin upcasting) doar prin intermediul clasei de baza sau al interfetei, numele sau nu mai este important.

```
public class MyClass extends Applet {  
    ...  
    someObject.addMouseListener(new MouseAdapter() {  
        public void mouseClicked(MouseEvent e) {  
            ... //Event listener implementation goes here  
            ...  
        }  
    }) ;  
    ...  
}
```

► More about Event Listeners

Anonymous Inner Classes (clase anonte)

Limitari:

- O clasa anonima poate sa extinda o singura clasa sau sa implementeze o singura interfata
- O clasa anonima nu are constructori (nu are nume, nu am sti cum se cheama constructorul); in mod implicit se invoca constructorul din clasa de baza care se potriveste cel mai bine cu parametrii folositi la initializare (sau constructorul default al clasei de baza daca nu sunt folositi parametrii)

Static Nested Classes (clase interne statice)

O clasa interna poate fi un membru static al clasei externe:

- nu va avea acces la instancele clasei externe (`Outer.this`), ci functioneaza ca o alta clasa top-level
- are sens sa fie interna pentru ca este folosita doar prin clasa externa (packaging convenience)
- nu avem nevoie de o instanta a clasei externe pentru a crea o instanta a clasei interne

|| [LinkedList.Entry](#) , [Map.Entry](#)

▶ How to use it

Exercitiu: Iterarea unui Map

Exercitiu: Iterarea unui Map

Hint: folositi Map.Entry

- Lab 6: Clase Interne
- More about Event Listeners

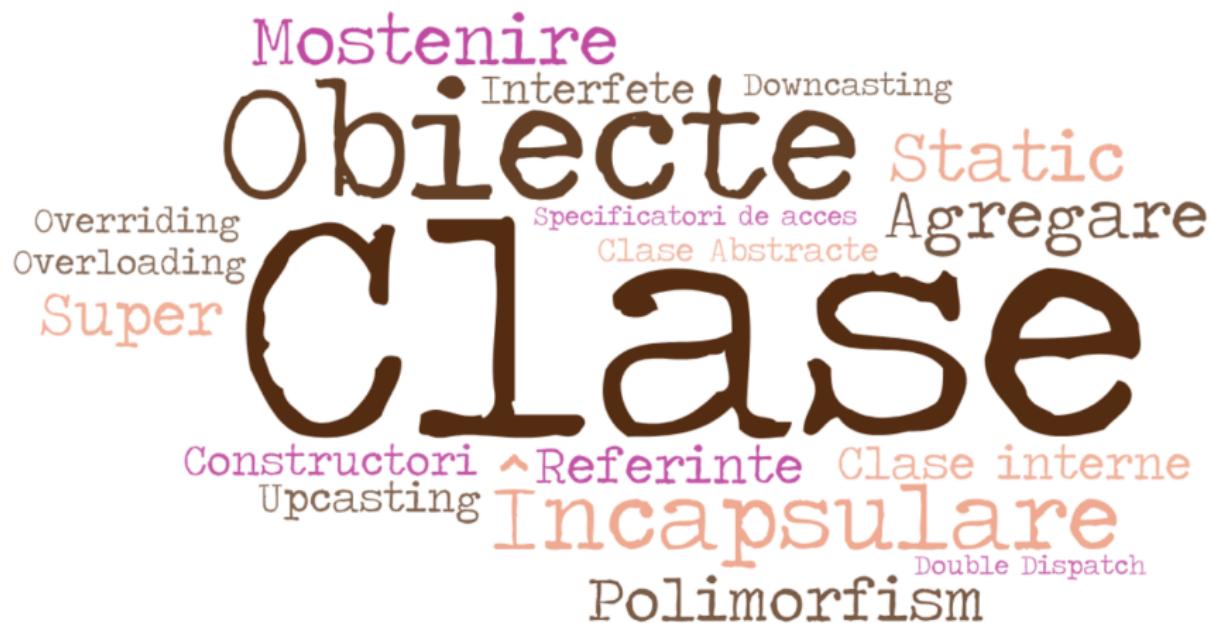
Polimorfism, Double Dispatch, Visitor

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020





Polimorfism



Definitia polimorfismului

Definition

Polimorfismul este caracteristica unei entitati de a avea forme diferite (structural, comportamental etc.) in contexte diferite.

Spre exemplu, in OOP, obiectele au aceasta caracteristica

Polimorfism in OOP

```
class Vehicle
interface Diesel
interface Electric
interface PublicTransport
class Hybrid extends Vehicle implements Electric, Diesel
class Bus extends Vehicle implements Diesel, PublicTransport
```

- A Hybrid IS-A Vehicle
- A Hybrid IS-A Electric
- A Hybrid IS-A Diesel
- A Hybrid IS-A Hybrid
- A Hybrid IS-A Object

Polimorfism in OOP

```
class Vehicle
interface Diesel
interface Electric
interface PublicTransport
class Hybrid extends Vehicle implements Electric, Diesel
class Bus extends Vehicle implements Diesel, PublicTransport
```

- A Bus IS-A Vehicle
- A Bus IS-A Diesel
- A Bus IS-A PublicTransport
- A Bus IS-A Bus
- A Bus IS-A Object

Polimorfism in OOP

```
class Vehicle
interface Diesel
interface Electric
interface PublicTransport
class Hybrid extends Vehicle implements Electric, Diesel
class Bus extends Vehicle implements Diesel, PublicTransport
```

Daca avem acelasi membru / metoda in mai multe "fatete" ale unui obiect, care se foloseste?

Polimorfism in OOP

- static polymorphism
- dynamic polymorphism
- parametric polymorphism

Static / Ad-hoc polymorphism (overloading)

Ad-hoc polymorphism (...) allows a polymorphic value to exhibit different behaviors when “viewed” at different types. The most common example of ad-hoc polymorphism is overloading, which associates a single function symbol with many implementations; the compiler (or the runtime system, depending on whether overloading resolution is static or dynamic) chooses an appropriate implementation for each application of the function, based on the types of the arguments.

B. Pierce, "Types and Programming Languages", MIT Press

Static / Ad-hoc polymorphism (overloading)

Static = compile time

```
1 + 2 = 3                                // integer addition
3 + 0.1415 = 3.1415                      // float addition
[1,2,3] + [4,5] = [1,2,3,4,5]            // list concatenation
"bab" + "oon" = "baboon"                  // string concatenation
red + yellow = orange                     // custom class operator+
```

Dynamic polymorphism (overriding)

Late binding, or dynamic binding is a computer programming mechanism in which the method being called upon an object or the function being called with arguments is looked up by name at runtime

```
public class Bike {  
    public void show() { System.out.println("bike"); }  
}  
public class Motorcycle extends Bike {  
    public void show() { System.out.println("motor"); }  
}  
...  
    public static void main(String args []){  
        Bike obj=new Motorcycle();  
        obj.show();  
    }  
}
```

private, final and static methods and variables uses static binding

Functii virtuale

In C++, se va face dynamic binding (la run-time) doar pentru functiile marcate ca fiind virtuale

Daca o functie este supraincarcata fara a fi marcata drept virtuala, se va face static binding (la compilare)

► Despre functii virtuale in C++

Dynamic polymorphism (overriding)

Atentie! dynamic binding se refera la metode, nu la campuri

```
public class Bike {  
    public int topSpeed = 30;  
}  
public class Motorcycle extends Bike {  
    public int topSpeed = 150;  
}  
...  
    public static void main(String args []){  
        Bike obj=new Motorcycle();  
        System.out.println(obj.topSpeed);  
    }
```

La afisare va fi folosit campul din referinta (in acest caz superclasa).

▶ Why are instance variables of a superclass not overridden in Inheritance ?

▶ Performance

Dynamic polymorphism (overriding)

Atentie! dynamic binding se refera la metode, nu la campuri

```
public class Bike {  
    public int topSpeed = 30;  
}  
  
public class Motorcycle extends Bike {  
    // public int topSpeed = 150;  
    public Motorcycle() {  
        topSpeed = 150;  
    }  
}  
...  
    public static void main(String args []){  
        Bike obj=new Motorcycle();  
        System.out.println(obj.topSpeed);  
    }  
}
```

Daca subclasa nu defineste campul cu acelasi nume ca in superclasa, in constructor se va folosi campul din superclasa (mai putin daca acesta este private)

Dynamic polymorphism (overriding)

Atentie! Overloading se rezolva la compile-time (fiecare metoda capata un nume unic) - vezi Static Polymorphism

```
public class Road extends Path {}
public class Motorcycle{
    public void ride(Road r) {
        System.out.println("motor on road");
    }
    public void ride(Path p) {
        System.out.println("motor on path");
    }
}
...
public static void main(String args[]){
    Motorcycle m = new Motorcycle();
    Path r = new Road();
    m.ride(r);
}
```

Double dispatch

The problem with Single Dispatch

```
public class Path {}
public class Road extends Path {}
public class Bike {
    public void ride(Road r) { sout("bike on road"); }
    public void ride(Path p) { sout("bike on path"); }
}
public class Motorcycle extends Bike{
    public void ride(Road r) { sout("motor on road"); }
    public void ride(Path p) { sout("motor on path"); }
}
...
public static void main(String args[]){
    Bike b = new Bike(); Bike m = new Motorcycle();
    Path p = new Path(); Path r = new Road();
    b.ride(p); b.ride(r); m.ride(p); m.ride(r);
}
```

Double dispatch

Mixing the second Dispatch on top of Single Dispatch

```
public class Path {  
    public void echo(Bike b){ b.ride(this); }  
}  
public class Road extends Path {  
    public void echo(Bike b){ b.ride(this); }  
}  
public class Bike {  
    public void ride(Road r) { sout("bike on road"); }  
    public void ride(Path p) { sout("bike on path"); }  
}  
public class Motorcycle extends Bike{  
    public void ride(Road r) { sout("motor on road"); }  
    public void ride(Path p) { sout("motor on path"); }  
}  
public static void main(String args[]){  
    Bike b = new Bike(); Bike m = new Motorcycle();  
    Path p = new Path(); Path r = new Road();  
    // b.ride(p); b.ride(r); m.ride(p); m.ride(r);  
    p.echo(b); r.echo(b); p.echo(m); r.echo(m);
```

Parametric polymorphism / Generics

Parametric polymorphism (...), allows a single piece of code to be typed "generically", using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same. (...)

B. Pierce, "Types and Programming Languages", MIT Press

Parametric polymorphism / Generics

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();  
List<Integer> grades = new ArrayList<Integer>();
```

Parametric polymorphism / Generics

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();
List<Integer> grades = new ArrayList<Integer>();

Collections.sort(team);
Collections.sort(grades);
```

Visitor

GoF Design Patterns

E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)
"Design Patterns: Elements of Reusable Object-Oriented Software"

Creational	Structural	Behavioral
Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Visitor

Description

Visitor represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Problem that the pattern solves

- o structura contine obiecte de clase variante, asupra carora vrem sa aplicam operatii ce depind de clasele lor concrete

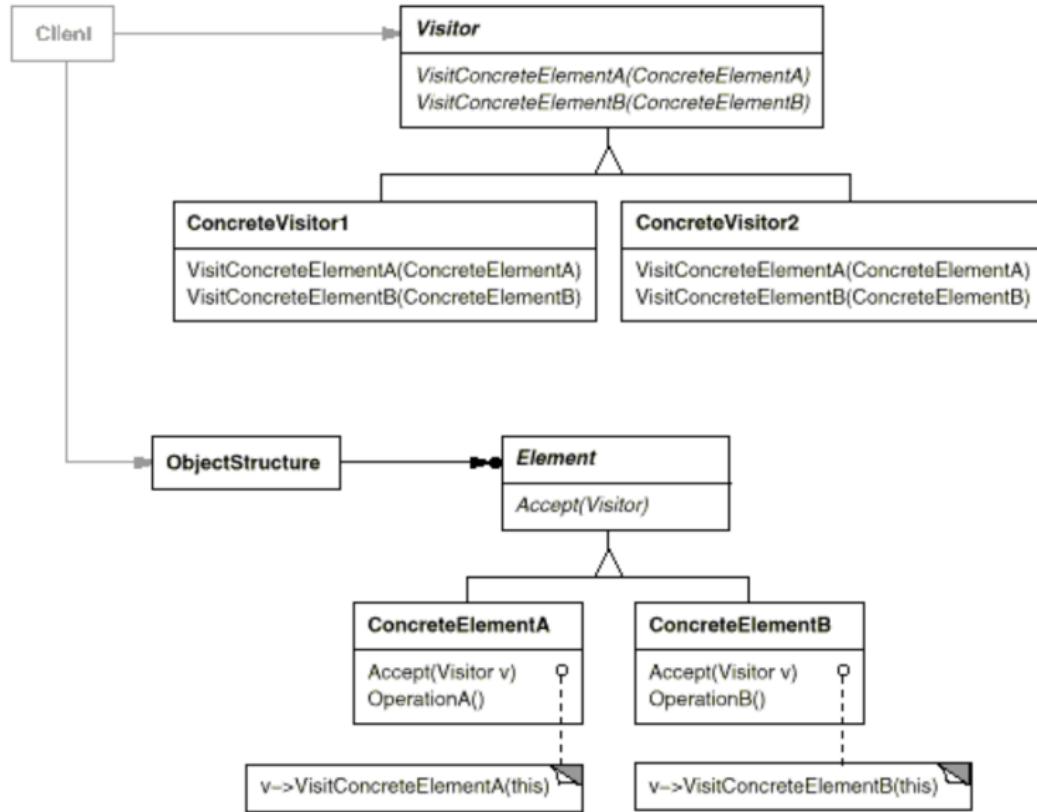
Problem that the pattern solves

- o structura contine obiecte de clase variante, asupra carora vrem sa aplicam operatii ce depind de clasele lor concrete
- operatiile pe care vrem sa le aplicam sunt diferite, fara legatura intre ele si ar trebui sa fie usor extensibile

Problem that the pattern solves

- o structura contine obiecte de clase variante, asupra carora vrem sa aplicam operatii ce depind de clasele lor concrete
- operatiile pe care vrem sa le aplicam sunt diferite, fara legatura intre ele si ar trebui sa fie usor extensibile
- clasele ce definesc structura de obiecte se schimba rar, dar vrem sa definim operatii noi usor (daca si clasele ce definesc structura de obiecte se schimba des, poate e mai bine sa definim operatiile in ele)

Solution Structure



Solution Details

Un client trebuie sa instantieze cate un vizitator concret pentru fiecare operatie si sa traverseze structura de obiecte vizitand fiecare obiect cu fiecare vizitator concret.

Solution Details

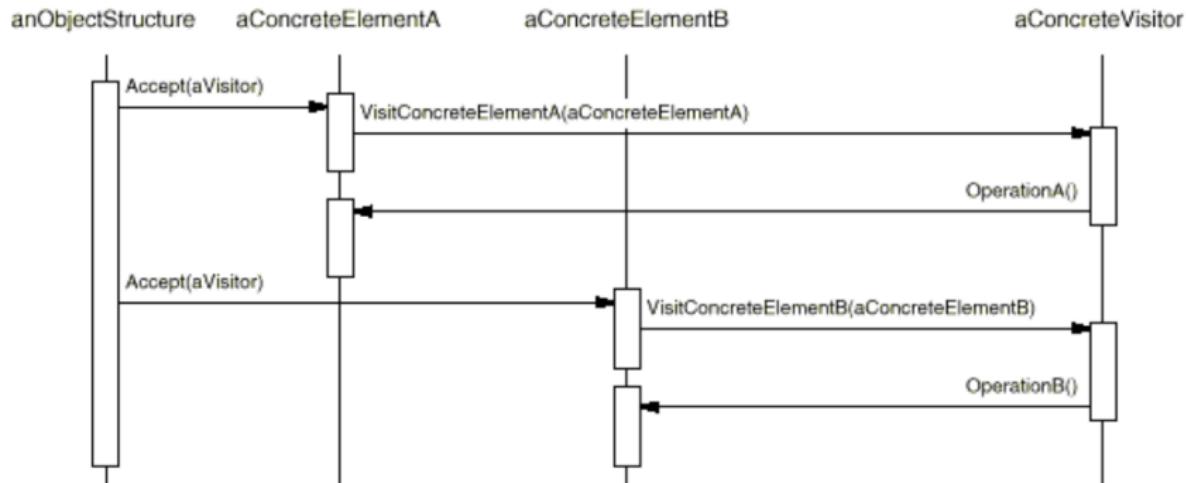
Un client trebuie sa instantieze cate un vizitator concret pentru fiecare operatie si sa traverseze structura de obiecte vizitand fiecare obiect cu fiecare vizitator concret.

Cum se face traversarea?

- de catre structura de obiecte
- de catre vizitator
- intr-un obiect iterator separat

Solution Details

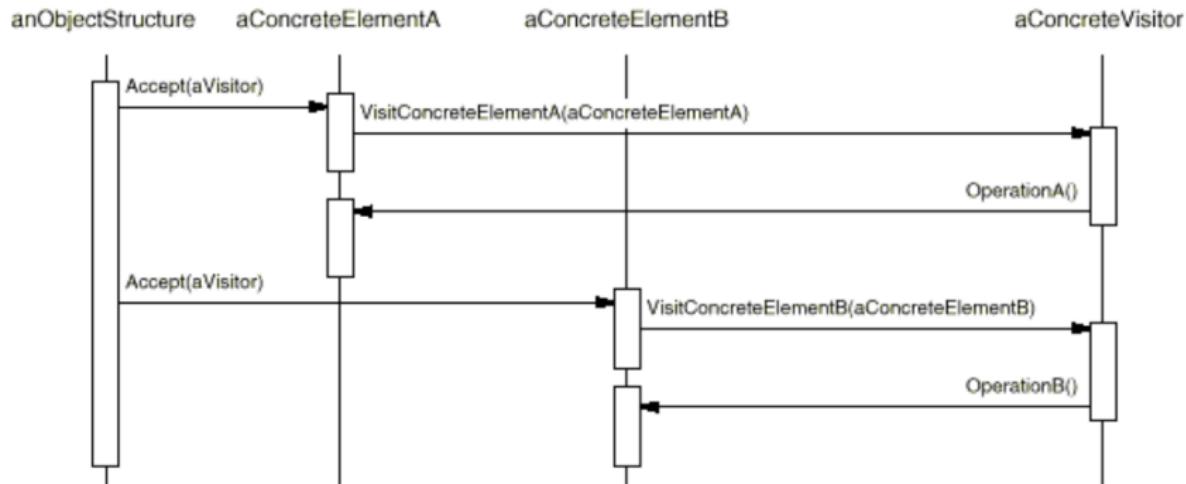
Vizitatorul concret ofera un context in care sa se desfasoare operatii, acumuleaza stare si poate apela metode din obiecte:



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

Solution Details

Vizitatorul concret ofera un context in care sa se desfaseoare operatii, acumuleaza stare si poate apela metode din obiecte:



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

Exista un echilibru care variaza de la o solutie la alta intre ce se desfaseoara in vizitator si ce este expus de obiect prin metode

Solution Details

Double Dispatch inseamna ca o actiune (precum Accept) depinde de tipul a doua elemente (in cazul Accept depinde de tipul vizitatorului si al obiectului vizitat).

Solution Details

Double Dispatch inseamna ca o actiune (precum Accept) depinde de tipul a doua elemente (in cazul Accept depinde de tipul vizitatorului si al obiectului vizitat).

	ConcreteVisitor1	ConcreteVisitor2
ConcreteElementA	VisitConcreteElementA	VisitConcreteElementA
ConcreteElementB	VisitConcreteElementB	VisitConcreteElementB

Exercitii

- Incercati sa adaugati o noua clasa de vizitator
- Incercati sa adaugati o noua clasa de obiecte vizitate

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecarei clase de obiecte vizitate)

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecarei clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecarei clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila
- vizitarea structurii de obiecte vizitate se poate face in diferite moduri, dar in forma cea mai pura de Visitor, spre deosebire de Iterator, obiectele vizitate nu trebuie neaparat sa instantieze dintr-o ierarhie de clase

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecarei clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila
- vizitarea structurii de obiecte vizitate se poate face in diferite moduri, dar in forma cea mai pura de Visitor, spre deosebire de Iterator, obiectele vizitate nu trebuie neaparat sa instantieze dintr-o ierarhie de clase
- vizitatorul poate acumula stare

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecarei clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila
- vizitarea structurii de obiecte vizitate se poate face in diferite moduri, dar in forma cea mai pura de Visitor, spre deosebire de Iterator, obiectele vizitate nu trebuie neaparat sa instantieze dintr-o ierarhie de clase
- vizitatorul poate acumula stare
- implementarea de operatii in obiectele vizitate care sa permita vizitatorului sa isi faca treaba poate compromite incapsularea in clasele de obiecte vizitate

- Lab 7: Overriding, Overloading and Visitor pattern
- Scurt tutorial despre Double Dispatch
- Despre functii virtuale in C++

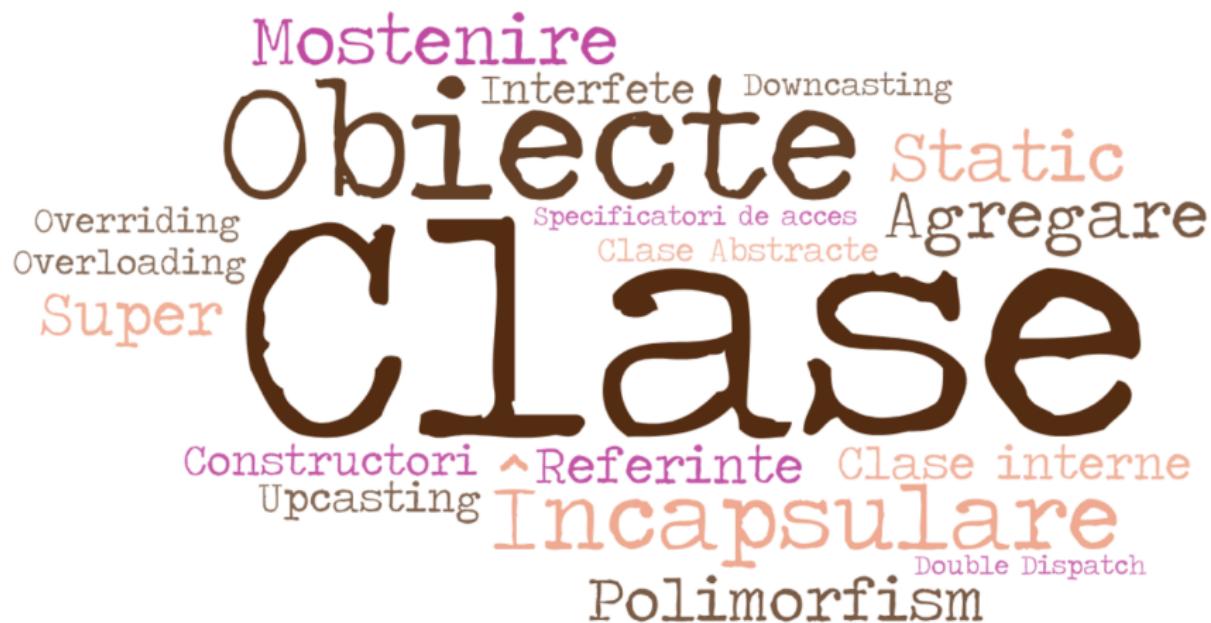
Genericitate (Generics / Templates)

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020





Genericitate

Nevoia pentru genericitate

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();  
List<Integer> grades = new ArrayList<Integer>();
```

Nevoia pentru genericitate

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();
List<Integer> grades = new ArrayList<Integer>();

Collections.sort(team);
Collections.sort(grades);
```

Nevoia pentru genericitate

daca as folosi un tip de baza cat mai general (void in C, Object in Java)
pot aparea erori la rulare si nici codul nu este foarte clar:

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
myIntList.add(new Player()); // which is bad
Integer x = (Integer) myIntList.iterator().next();
```

Nevoia pentru genericitate

daca as folosi un tip de baza cat mai general (void in C, Object in Java) pot aparea erori la rulare si nici codul nu este foarte clar:

```
List myIntList = new LinkedList();  
myIntList.add(new Integer(0));  
myIntList.add(new Player());  
Integer x = (Integer) myIntList.iterator().next();
```

```
List<Integer> myIntList = new LinkedList<Integer>();  
myIntList.add(new Integer(0));  
myIntList.add(new Player()); //Eroare!  
Integer x = myIntList.iterator().next(); // nici nu mai e  
nevoie de cast
```

Nevoia pentru genericitate

pentru metode, daca as supraincarca metoda cu fiecare tip de date, as duplica cod:

```
public class Collections {
    static boolean replaceAll(List<Integer> list,
        Integer oldVal, Integer newVal) {
        ... // o implementare
    }
    static boolean replaceAll(List<Player> list,
        Player oldVal, Player newVal) {
        ... // o alta implementare
    }
}
```

Nevoia pentru genericitate

pentru metode, daca as supraincarca metoda cu fiecare tip de date, as duplica cod:

```
public class Collections {
    static boolean replaceAll(List<Integer> list,
    Integer oldVal, Integer newVal) {
        ... // o implementare
    }

    static boolean replaceAll(List<Player> list,
    Player oldVal, Player newVal) {
        ... // o alta implementare
    }

    static <T>
    boolean replaceAll(List<T> list, T oldVal, T newVal) {
        ... // o singura implementare
    }
}
```

Definitia genericitatii

Genericitatea este un mecanism prin care tipurile folosite in definirea claselor / interfetelor si metodelor sa fie parametrizate.

Definitia genericitatii

Genericitatea este un mecanism prin care tipurile folosite in definirea claselor / interfetelor si metodelor sa fie parametrizate.

Acest mecanism se intalneste, ca principiu, in mai multe limbaje OOP: template in C++, generics in Java, C# si Objective-C.

Parametric polymorphism / Generics

Parametric polymorphism (...), allows a single piece of code to be typed "generically", using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same. (...)

B. Pierce, "Types and Programming Languages", MIT Press

Why is C++ said not to support parametric polymorphism?

Tipuri formale

La definire se foloseste tipul formal (e.g. T mai jos)

```
public class Erasure<T> {
    private T obj;
    Erasure(T o) { obj = o; }
    T getObj() { return obj; }
}
...
public static void main(String[] args) {
    Erasure<Integer> test = new Erasure<Integer>(10);
    System.out.println(test.getObj());
}
```

Type Erasure

La compilare se produce un singur cod pentru o clasa, tipul formal este substituit cu Object (alte limbaje implementeaza genericitatea diferit)

```
public class Erasure<T> {
    private T obj;
    Erasure(T o) { obj = o; }
    T getobj() { return obj; }
}
...
public static void main(String[] args) {
    Erasure<Integer> test = new Erasure<Integer>(10);
    System.out.println(test.getObj());
}
```

Incercati sa decompilati o clasa cu javap (folosind -c)

Type Erasure

Restrictii in definirea claselor generice:

- Cannot Create Instances of Type Parameters
- Cannot Create Arrays of Parameterized Types
- Restrictii

```
public class Erasure<T> {  
    private T obj;  
    Erasure() { obj = new T(); } // does not compile  
}  
  
public class GenericsErasure<T> {  
    private T[] objs;  
    Erasure() { objs = new T[10]; } // does not compile  
}
```

Se poate totusi prin Reflection

Bridge methods

Cand o clasa extinde o clasa generica sau implementeaza o interfata generica:

```
public static class A<T> {
    public T getT(T args) {
        return args;
    }
}
public static class B extends A<String> {
    public String getT(String args) {
        return args;
    }
}
...
A a = new B();
a.getT(new Object()); // ClassCastException la runtime
pentru ca...
```

Bridge methods

compilatorul produce o metoda sintetica (care nu apare in cod si nu poate fi apelata explicit):

```
public class B extends A<java.lang.String> {
    ...
    public java.lang.String getT(java.lang.String);
        Code:
            0: aload_1
            1: areturn
    public java.lang.Object getT(java.lang.Object);
        Code:
            0: aload_0
            1: aload_1
            2: checkcast      #2                      // class java/
                lang/String
            5: invokevirtual #3                      // Method getT:
                Ljava/lang/String;)Ljava/lang/String;
            8: areturn
}
```

Type Erasure

Atentie insa! la folosirea unei clase generice:

```
List<String> list = new ArrayList<String>();  
list.add("foo");  
String x = list.get(0);
```

se va substitui cu

```
List list = new ArrayList();  
list.add("foo");  
String x = (String) list.get(0);
```

Tipuri formale

Pot avea mai multe tipuri formale intr-o definitie si pot avea tipuri formale imbricate:

```
public interface Map<K,V>{
    static interface Map.Entry<K,V>;
    Set<Map.Entry<K,V>> entrySet();
    ...
}
```

Tipuri formale

Some things may make you frown:

```
public interface Map<K,V>{
    static interface Map.Entry<K,V>;
    Set<Map.Entry<K,V>> entrySet();
    V get(Object key); // desi in C# este 'V Get(K k);'
    ...
}
```

Genericitatea în subtipuri

Este List de Strings o List de Objects?

```
|| List<String> ls = new ArrayList<String>();  
|| List<Object> lo = ls;
```

Genericitatea in subtipuri

Nu, pentru ca ar duce la erori:

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls;      // eroare de compilare  
lo.add(new Object());  
String s = ls.get(0);
```

Wildcards

Ce fac daca vreau sa definesc o metoda care sa ia ca parametru lista de orice fel de elemente? (nu pot folosi List<Object>)

```
void printList(List<?> l) {  
    for (Object e : l) {          // e ok  
        System.out.println(e);  
    }  
}
```

Wildcards

Ce fac daca vreau sa definesc o metoda care sa ia ca parametru lista de orice fel de elemente? (nu pot folosi List<Object>)

```
void printList(List<?> l) {  
    for (Object e : l) {          // e ok  
        System.out.println(e);  
    }  
    l.add(new Object()); // eroare de compilare  
}
```

Bounded Type Parameters and Wildcards

Ce fac daca vreau sa limitez listele pe care le pot primi ca parametru la liste de elemente de un anumit fel?

```
void sayHello(List<? extends Man> l) {  
    for (Object e : l) {  
        System.out.println("Hello "+e.getName());  
    }  
}
```

Bounded Type Parameters and Wildcards

Ce fac daca vreau sa limitez listele pe care le pot primi ca parametru la liste de elemente de un anumit fel?

```
void sayHello(List<? extends Man> l) {  
    for (Object e : l) {  
        System.out.println("Hello " + e.getName());  
    }  
}
```

Tipurile formale si wildcards pot fi upper bounded (? extends T) si lower bounded (? super T):

```
public class Collections {  
    static <T extends Comparable<? super T>>  
        void sort(List<T> list) {  
            ... // o singura implementare  
        }  
}
```

Bounded Type Parameters and Wildcards

Ce fac daca vreau sa limitez listele pe care le pot primi ca parametru la liste de elemente de un anumit fel?

```
void sayHello(List<? extends Man> l) {  
    for (Object e : l) {  
        System.out.println("Hello " + e.getName());  
    }  
}
```

Tipurile formale si wildcards pot fi upper bounded (? extends T) si lower bounded (? super T):

```
public class Collections {  
    static <T extends Comparable<? super T>>  
        void sort(List<T> list) {  
            ... // o singura implementare  
        }  
}
```

▶ Explicatie pe StackOverflow

Type Erasure

Daca tipul formal este upper bounded, va fi substituit cu upper bound

```
public class Erasure<T extends Number> {
    private T obj;
    Erasure(T o) { obj = o; }
    T getobj() { return obj; }
}
...
public static void main(String[] args) {
    Erasure<Integer> test = new Erasure<Integer>(10);
    System.out.println(test.getObj());
}
```

Incercați să decompilați o clasa cu javap

Metode generice

Metodele generice permit folosirea de parametrii formali pentru a exprima dependenta intre parametrii si/sau intre parametrii si rezultat:

```
// Metoda corecta
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

Metode generice

Metodele generice permit folosirea de parametrii formali pentru a exprima dependenta intre parametrii si/sau intre parametrii si rezultat:

```
// Metoda corecta
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

```
// Metoda incorecta: de ce?
static void arrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o);
    }
}
```

Metode generice

Metodele generice permit folosirea de parametrii formali pentru a exprima dependenta intre parametrii si/sau intre parametrii si rezultat:

```
// Metoda corecta
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

```
// Metoda incorecta: de ce?
static void arrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // eroare de compilare
    }
}
```

Metode generice

La folosire, compilatorul deduce automat tipul formal

```
static <T> void arrayToCollection(T[] a, Collection<T> c) {  
    for (T o : a) {  
        c.add(o); // Correct  
    }  
  
    String[] sa = new String[100];  
    Collection<String> cs = new ArrayList<String>();  
  
    // T inferred to be String  
    arrayToCollection(sa, cs);
```

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    <T> T[] toArray(T[] a); // programmer controls the type  
}
```

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    <T> T[] toArray(T[] a); // programmer controls the type  
}  
  
List<Integer> myList = new ArrayList<Integer>();  
Integer[] myArray=myList.toArray(new Integer[myList.size()]);  
Number[] myArray=myList.toArray(new Number[myList.size()]);  
Object[] myArray=myList.toArray(new Object[myList.size()]);
```

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    <T> T[] toArray(T[] a); // programmer controls the type  
}
```

De ce nu `<E> E[] toArray(E[] a);`?

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    <T> T[] toArray(T[] a); // programmer controls the type  
}
```

De ce nu `<E> E[] toArray(E[] a);?`

pentru ca vreau sa pot scoate `Number[]` din `List<Integer>`

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    <T> T[] toArray(T[] a); // programmer controls the type  
}
```

De ce nu $\langle T \text{ super } E \rangle$ `toArray(T[] a)`?

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    <T> T[] toArray(T[] a); // programmer controls the type  
}
```

De ce nu $\langle T \text{ super } E \rangle$ $T[]$ $\text{toArray}(T[] a)$?

pentru ca as putea incerca sa scot un `Integer[]` din `List<Number>`

Limitari pentru generics

Why isn't `Collection.remove(Object o)` generic?

Josh Bloch and Bill Pugh (some of the guys who worked on generification) refer to this issue in:

▶ Java Puzzlers IV: The Phantom Reference Menace, Attack of the Clone, and Revenge of The Shift.

- Lab 8: Colectii
- Lab 9: Genericitate
- Why is C++ said not to support parametric polymorphism?

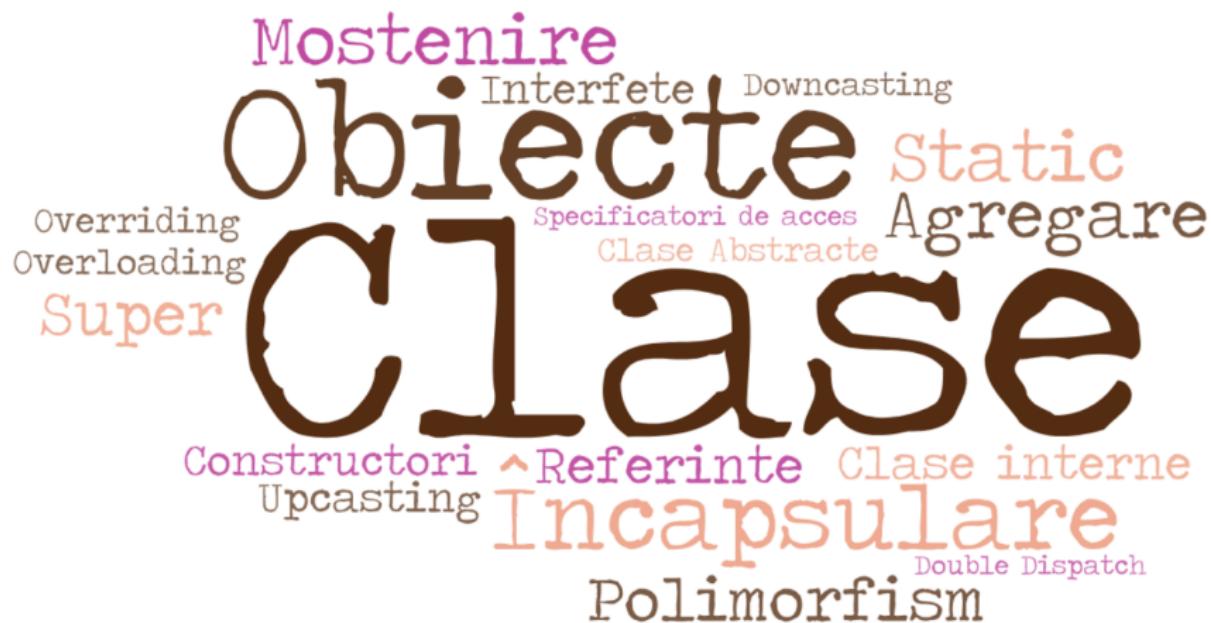
Object-Oriented Design Principles

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020





SOLID

Five design principles synthesized by Robert C. Martin (Uncle Bob):

- SRP: Single Responsibility Principle
- OCP: Open-Closed Principle
- LSP: Liskov Substitution Principle
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

Single-responsibility principle

Definition

A class should have only one responsibility, meaning that a class should have one and only one reason to change.

► Robert C. Martin "The Single-Responsibility Principle"

e.g. a Rectangle class should not contain both draw and area functions

Single-responsibility principle



Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

▶ SOLID principles explained with motivational posters

Open-closed principle

Definition

You should be able to extend a classes behavior, without modifying it.

► Robert C. Martin "The Open-Closed Principle", C++ Report, January 1996

e.g. having an `AbstractShape`, with a generic `draw` method, allows us to add new shapes

Do not confuse the "extend" mentioned by the principle with `extends` in Java

Open-closed principle

Definition

Software entities should be open for extension, but closed for modification.

Bertrand Meyer "Object-Oriented Software Construction", 1988

extending the software entities can be done if they use inheritance

Open-closed principle



Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

► SOLID principles explained with motivational posters

Liskov substitution principle

Definition

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .

► Liskov, Barbara. "Keynote address-data abstraction and hierarchy." ACM Sigplan Notices 23.5 (1988): 17-34

essentially polymorphism

Liskov substitution principle

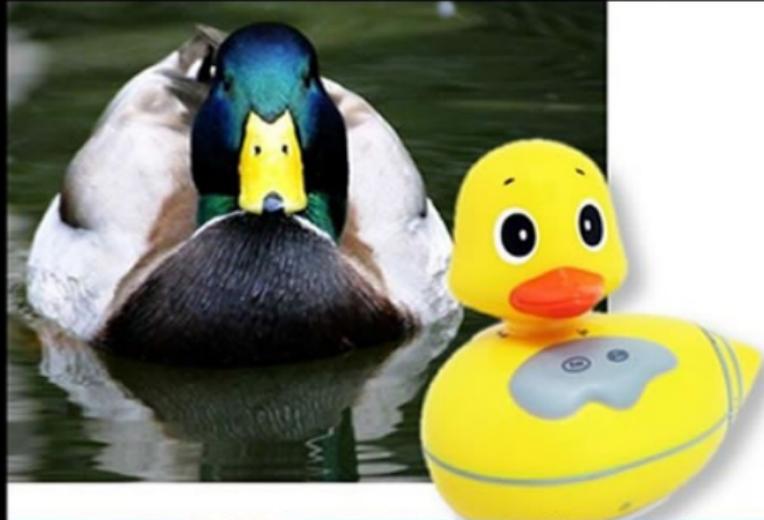
Definition

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

► Robert C. Martin "The Liskov substitution principle", C++ Report

e.g. a square might be a rectangle, but the behavior of a Square object is not consistent with the behavior of a Rectangle object

Liskov substitution principle



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

▶ SOLID principles explained with motivational posters



LSP vs OCP

Follows OCP, but not LSP:

```
public interface IPerson {}

public class Boss implements IPerson {
    public void doBossStuff() { ... }
}

public class Peon implements IPerson {
    public void doPeonStuff() { ... }
}

public class Context {
    public Collection<IPerson> getPersons() { ... }
}
```

► [LSP vs OCP on StackExchange](#)

LSP vs OCP

Follows OCP, but not LSP - problem:

```
// in some routine that needs to do stuff with
// a collection of IPerson:
Collection<IPerson> persons = context.getPersons();
for (IPerson person : persons) {
    // now we have to check the type... :-P
    if (person instanceof Boss) {
        ((Boss) person).doBossStuff();
    }
    else if (person instanceof Peon) {
        ((Peon) person).doPeonStuff();
    }
}
```

► [LSP vs OCP on StackExchange](#)

LSP vs OCP

Follows OCP, but not LSP - fix:

```
public class Boss implements IPerson {
    // we're adding this general method
    public void doStuff() {
        // that does the call instead
        this.doBossStuff();
    }
    public void doBossStuff() { ... }
}

public interface IPerson {
    // pulled up method from Boss
    public void doStuff();
}

// do the same for Peon
```

▶ [LSP vs OCP on StackExchange](#)

LSP vs OCP

Follows LSP, but not OCP:

```
public class LiskovBase {  
    public void doStuff() {  
        System.out.println("My name is Liskov");  
    }  
}  
  
public class LiskovSub extends LiskovBase {  
    public void doStuff() {  
        System.out.println("I'm a sub Liskov!");  
    }  
}  
  
public class Context {  
    private LiskovBase base;  
    public void doLiskovyStuff() {  
        base.doStuff();  
    }  
    public void setBase(LiskovBase base) {  
        this.base = base  
    }  
}
```

LSP vs OCP

Follows LSP, but not OCP - fix:

```
public class LiskovBase {  
    // the code that was duplicated is now a template method  
    public final void doStuff() {  
        System.out.println(getStuffString());  
    }  
  
    // the code that "varies" in LiskovBase and it's  
    // subclasses called by the template method above  
    // we expect it to be virtual and overridden  
    public string getStuffString() {  
        return "My name is Liskov";  
    }  
}  
  
public class LiskovSub extends LiskovBase {  
    // the actual code that varied  
    public string getStuffString() {  
        return "I'm sub Liskov!";  
    }  
}
```

Interface segregation principle

Definition

many client-specific interfaces are better than one general-purpose interface.

► Robert C. Martin "The Interface segregation principle"

e.g. having a TimedDoor class, shouldn't require all Doors to have a Timer

Interface segregation principle



Interface Segregation Principle

You want me to plug this in *where?*

► SOLID principles explained with motivational posters

Dependency Inversion Principle

Definition

High level modules should not depend upon low level modules. Both should depend upon abstractions.

► Robert C. Martin "The Dependency Inversion Principle", C++ Report

e.g. a program that copies from inputs to outputs, should not depend on the input being a keyboard and the output being a screen

e.g. a PasswordRecovery should not depend on a MySQLConnection, but on a generic DBConnectionInterface

Dependency Inversion Principle



Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

► SOLID principles explained with motivational posters

Acelasi cod, dar in limbaje diferite:

- ▶ SOLID in PHP (class inheritance, ca in Java)
- ▶ SOLID in JavaScript (prototype-based language)

Despre SOLID si alte principii de Design Orientat Obiect

- [Uncle Bob's Principles of OOD](#)
- [Principles Of Object Oriented Design](#)
- [KISS, YAGNI, DRY: 3 Principles to Simplify Your Life as a Developer](#)

Breaking the Patterns: Extending without Class Inheritance, Reflection, AntiPatterns

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică și Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020



Universitatea
Politehnica
București

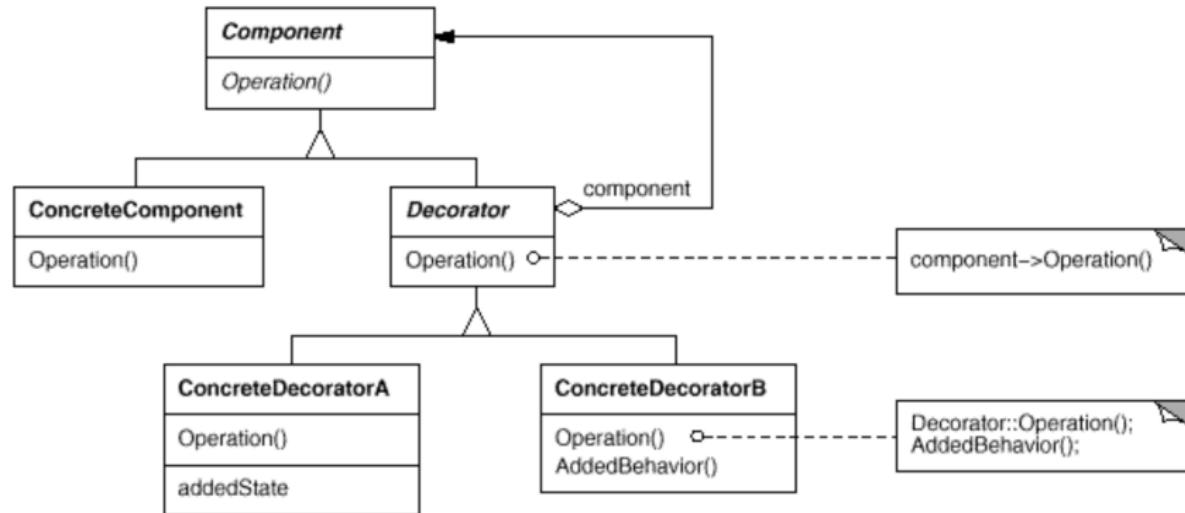
Subiecte de discutie

- Extending without Class Inheritance
- Reflection
- AntiPatterns

Extending without Class Inheritance: Decorator, Prototypal Inheritance

Decorator

Decorators provide a flexible alternative to subclassing for extending functionality (attaching additional responsibilities to an object dynamically).



Decorator

"Changing the skin of an object [...Decorator...] versus changing its guts [...Strategy...]"
- Gang of Four, Design Patterns

Prototypal Inheritance

Prototypal Inheritance objects inherit directly from other objects. Various ways to implement:

- Concatenative Inheritance
- Prototype Delegation
- Functional Inheritance

"Favor object composition over class inheritance", see the Fragile Class Problem.

► Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?

Reflection, Introspection

Introspection

In computing, **Type Introspection** is the ability of a program to examine the type or properties of an object at runtime. Some programming languages possess this capability.

In Spring, Introspection is a process of analyzing a Bean to determine its capabilities

```
final BeanWrapper wrap = new BeanWrapperImpl(AType.class);
final PropertyDescriptor[] pDescriptors =
    wrap.getPropertyDescriptors();
for (final PropertyDescriptor pDescriptor : pDescriptors) {
    logger.info(pDescriptor.getName() + ":" +
        pDescriptor.getPropertyType());
}
```

Reflection vs Introspection

Introspection should not be confused with **Reflection**, which goes a step further and is the ability for a program to manipulate the values, meta-data, properties and/or functions of an object at runtime. Some programming languages possess that capability.

▶ detalii

In general, aceasta capacitate este oferita de o suita de clase, care impreuna formeaza Reflection API, si are efecte asupra diferitelor tool-uri de compilare, rulare si debug.

In Java:

```
|| java.lang.Class, java.lang.reflect.*
```

Reflection: Class

Primul pas este obtinerea unui obiect de tip Class pentru tipul de date pentru care dorim sa facem reflection:

- daca avem la dispozitie o instantă:

```
|| Class c = "foo".getClass();
```

- daca nu avem o instantă, dar stim tipul:

```
|| Class c = java.io.PrintStream.class;
```

- Toate modalitatile de a obtine obiecte Class

Reflection: Exemple

- doresc sa apelez metoda 'met' a unui obiect oarecare 'obj', daca aceasta exista

```
Method method = obj.getClass().getMethod("met", null);  
method.invoke(obj, null);
```

- crearea de obiecte noi si schimbarea valorilor pe care le au campurile
 - Creating New Class Instances
 - si
 - Setting Field Values
- JUnit foloseste Reflection pentru a inspecta codul testelor, a observa ce metode au anotarea @Test si a le apela automat pe acestea.
- compatibilitatea unei aplicatii Android cu mai multe versiuni de Android OS
 - Backward compatibility for Android applications

Reflection: Avantaje si Dezavantaje

Avantaje

- Extensibility Features
- Backwards Compatibility for Android OS ► detalii
- Class Browsers and Visual Development Environments
- Debuggers and Test Tools

Dezavantaje

- Performance Overhead
- Security Restrictions
- Exposure of Internals

► Tutorial complet

Reflection vs Encapsulation

Exemples of uses that do not break encapsulation:

- finding out members, annotations, super-types of a class
- invoke accessible methods, modify accessible fields

Exemples of uses where it might be acceptable to break encapsulation:

- simplest way to implement a unit test (on someone else's code)
- work around a bug (in someone else's code)

► Doesn't Reflection API break the very purpose of Data encapsulation?

Reflection vs ?

Reflection vs Generics

You cannot create an instance of a type parameter.

```
public static <E> void append(List<E> list) {  
    E elem = new E(); // compile-time error  
    list.add(elem);  
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls)  
    throws Exception {  
    E elem = cls.newInstance(); // OK  
    list.add(elem);  
}
```

▶ Restrictions on Generics

Reflection vs Singleton

Using Reflection to overcome a typical implementation of Singleton.

```
Constructor[] constructors =
    Singleton.class.getDeclaredConstructors();
for (Constructor constructor : constructors)
{
    // Below code will destroy the singleton pattern
    constructor.setAccessible(true);
    instance2 = (Singleton) constructor.newInstance();
    break;
}
```

As a workaround, you can use Enums, which Java ensures are instantiated only once, but do not allow lazy initialization.

AntiPatterns

Ce sunt AntiPatterns?

AntiPatterns are

- patterns: describe general solutions to problems that occur over and over again
- bad ones: usually ineffective and risk being highly counterproductive

God Object

A **God Object** is an object that knows too much or does too much:

▶ God Object

Spaghetti Code

Spaghetti Code is object-oriented code written in procedural style, such as by creating classes whose methods are overly long and messy:

- un main de 2500 de linii de code, anyone?

▶ Spaghetti Code

Boilerplate Code

Boilerplate Code refers to sections of code that have to be included in many places with little or no alteration - e.g. classes are often provided with methods for getting and setting instance variables

"programs should be written for people to read, and only incidentally for machines to execute."

- Harold Abelson

Boilerplate Code

Boilerplate Code refers to sections of code that have to be included in many places with little or no alteration - e.g. classes are often provided with methods for getting and setting instance variables

Solutions:

- metaprogramming (which has the computer automatically write the needed boilerplate code or insert it at compile time)

```
|| public string Name { get; set; }  
|| //auto-implemented properties in C#
```

- convention over configuration (which provides good default values, reducing the need to specify program details in every project)

```
|| class ProductsController {  
||     public ActionResult Index() {...}  
|| }  
|| // when serving /products, framework uses Reflection to  
|| // create instance and call method
```

Singleton

Me: So! Have you ever heard of a book called Design Patterns?

Them: Oh, yeah, um, we had to, uh, study that back in my software engineering class. I use them all the time.

Me: Can you name any of the patterns they covered?

Them: I loved the Singleton pattern!

Me: OK. Were there any others?

Them: Uh, I think there was one called the Visitater.

Me: Oooh, that's right! The one that visits potatoes. I use it all the time.
Next!!!

I actually use this as a weeder question now. If they claim expertise at Design Patterns, and they can ONLY name the Singleton pattern, then they will ONLY work at some other company.

► Singleton Considered Stupid

Singleton

- *When used as global variables:* using global variables is an enemy of encapsulation because it becomes difficult to define preconditions for the client's object interface.
 - Imagine a couple of objects that use the same global variable; one triggers a side effect which changes the value of the global state, you no longer know the starting state when you execute the code in the other object, which will have unpredictable results. This is the reason why in Dependency Injection, objects are passed as arguments to constructors or by setters.
- *Singletons vs polymorphism:* Singletons tightly couple the code to the exact object type and remove the scope of polymorphism.
 - As a workaround, use Factory.

► When Singleton Becomes an Anti-Pattern

Factory Factory Factory

I'm currently in the planning stages of building a hosted Java web application (yes, it has to be Java, for a variety of reasons that I don't feel like going into right now). In the process, I'm evaluating a bunch of J2EE portlet-enabled JSR-compliant MVC role-based CMS web service application container frameworks.

And after spending dozens of hours reading through feature lists and documentation, I'm ready to gouge out my eyes.

.....

▶ Why I hate frameworks, Benji Smith