Metode Avansate de Programare
Clase, moștenire și interfețe

Arthur Molnar
arthur.molnar@ubbcluj.ro

Universitatea Babeș-Bolyai

2023

Privire de ansamblu

Concepte de bază în POO - Recapitulare

- **Clasă**: reprezintă un tip de dată.
- **Obiect**: o instanță a unei clase.
- **Încapsulare**: gruparea datelor și funcțiilor înrudite sub forma claselor/obiectelor și definirea unei interfețe pentru accesarea acestora.
- **Abstractizare**: separarea *specificației* unui obiect de *implementarea* acestuia.
- **Moștenire**: o relație între clase de tipul *"este un" (eng. "is a")* ce facilitează scrierea de cod flexibil, reutilizabil.
- **Polimorfism**: permite adoptarea comportamentului corect în timpul rulării programului pe baza tipului actual al obiectului pe care se apelează metodele.

Declararea unei clase Java

```
[public] [abstract] [final] class ClassName
{
    [member data (attributes) declarations]
    [method declarations and implementations]
    [nested classes]
}
```

- Dacă clasa `ClassName` este publică atunci trebuie să se afle în fișierul `ClassName.java`.
- Un fișier `.java` poate include mai multe definiții de clase, însă una singură poate fi publică.
- Recapitulare C++
  - De regulă avem două fișiere, unul pentru interfață și unul pentru implementare (`.h` și `.cpp`).
  - Metodele publice sunt declarate în fișierul `.h` dar sunt implementate de regulă în `.cpp`.

Declararea datelor membru I

```
[public] [abstract] [final] class ClassName
{
    [access_modifier] [abstract] [static] [final]
                                Type name [=initial_value];
}
```

- **access_modifier** poate fi: private, protected, public
- Modificatorul de acces implicit (eng. default) (dacă nu se specifică unul din cei de mai sus) este privat la nivel de pachet.

Declararea datelor membru II

```
public class Doctor {
    private String name;
    private String speciality;
    private double salary;

    // ...
}

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

    //...
}
```

Declararea datelor membru III

```
public class Circle {
    double radius;
    Point centre;
    static final double PI = 3.14;

    //...
}
```

Inițializarea datelor membru

- La momentul declarării:

```
private double radius = 0;
```

- În constructorul clasei
- Într-un bloc de inițializare:

```
public class Rational
{
    private int numerator;
    private int denominator;

    {
        numerator = 0
        denominator = 1;
    }
}
```

- Datele membru care nu sunt inițializate explicit, sunt inițializate cu valoarea implicită a tipului lor.

Constructori

- Un constructor este apelat de fiecare dată când se creează o instanță a unei clase.
- Memoria necesară noului obiect este alocată automat în momentul apelării constructorului.
- Constructorul are același nume ca și clasa, poate avea modificatori de acces, dar nu poate avea un tip returnat.
- Dacă la definirea unei clase nu se declară cel puțin un constructor, compilatorul generează unul implicit, care nu are parametri de intrare și care este `public`.

```
[...] class ClassName
{
    [access_modifier] ClassName([parameter_list])
    {
        // constructor body
    }
}
```

Supraîncărcarea constructorilor I

- Există posibilitatea de a avea mai mulți constructori pentru o clasă
- Diferența e făcută de numărul și/sau tipul parametrilor de intrare

```
public class Rational {
    private int numerator;
    private int denominator;

    public Rational()  {
        this.numerator = 0;
        this.denominator = 0;
    }

    public Rational(int num) {
        this.numerator = num;
        this.denominator = 1;
    }
}
```

Supraîncărcarea constructorilor II

```
public Rational(int num, int den)  {
    this.numerator = num;
    this.denominator = den;
}
```

- Un constructor poate apela un alt constructor al aceleași clase folosind cuvântul cheie `this`.
- Apelul către alt constructor trebuie să fie prima instrucțiune din constructor.

```
public Rational(int num) {
    this(num, 1);
}
```

Supraîncărcarea constructorilor III

- Nu e posibilă apelarea a doi constructori diferiți.

```
public Rational(int num) {
    this(num, 1);
    this(1, 0); // ERROR
}
```

- Nu e posibilă apelarea directă a unui constructor dintr-o altă metodă a clasei.

```
public Rational add(Rational r) {
    this(this.numerator + r.numerator,
         this.denominator + r.denominator); // ERROR
return this;
}
```

Crearea obiectelor

- La rularea programului, obiectele create folosind operatorul `new` sunt alocate pe heap.
- Programul operează cu referințe la aceste obiecte.
- `null` - valoarea implicită pentru o referință.

```
public static void main(String[] args) {
    Rational r1 = new Rational(1, 2);
    Rational r2 = new Rational(3);
    Rational r3 = null; // r3 nu are un obiect asociat
    Rational r4; // null, valoarea implicita

    r1 = r2; // r1 si r2 refera acelasi obiect
    System.out.println(r1);
}
```

Distrugerea obiectelor

- În limbajul Java nu avem destructori.
- Memoria este dealocată automat de către o componentă a JVM numită *garbage collector*.
- Componenta *garbage collector* rulează în mod constant în timpul execuției programului (aproximativ ☺)
- Obiectivul este de a elibera memoria heap prin distrugerea obiectelor la care programul nu mai are referințe.
- Există două strategii principale pentru a găsi referințele care pot fi eliberate:
  - Numărarea referințelor (eng. *reference counting*)
  - Trasabilitatea referințelor (eng. *tracing collector*)

Distrugerea obiectelor

Java Garbage Collection Basics

Descrierea mecanismului de garbage collection
https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Declararea metodelor

```
[...] class ClassName
{
    [access_modifier][method_attributes]
                Type methodName ([parameter_list])
    {
        // corpul metodei
    }
}
```

- `Type` poate fi un tip primitiv, clasă, tablou sau `void`.
- Modificatorul de acces implicit este cel privat la nivel de pachet (eng. default).

Metode cu număr variabil de parametri

- O metodă poate fi declarată pentru a putea fi apelată cu zero, unul sau mai mulți parametri (eng. variable arguments, varargs)
- Aceasta se face prin adăugarea unui parametru special, care trebuie să fie ultimul în lista parametrilor.
- O metodă poate avea un singur parametru de acest tip.

```
[access_modifier] methodName(type ... args) {
    // method body
}
```

Supraîncărcarea metodelor

- Mai multe metode pot avea același nume, dar trebuie să difere numărul/ tipul parametrilor
- Supraîncărcarea metodelor este un exemplu de polimorfism static (legare la compilare sau legare devreme) (eng. compile-time binding or early binding).
- Supraîncărcarea:
  - număr diferit de parametri;
  - tipuri de date diferite;
  - tipul returnat de metodă nu contează în cazul supraîncărcării;

Metodele `toString()` și `equals()`

- Semnăturile lor trebuie să nu fie modificate la implementare unei noi clase.
- `toString()` întoarce reprezentarea unui obiect ca șir de caractere.
- Mașina virtuală Java (JVM) apelează metoda `toString()` pentru afișarea unui obiect.
- `equals()` permite verificarea egalității logice a două obiecte (`==` testează egalitatea referințelor)

Exemplu

`lecture.examples.lecture2.Rational.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Date membru și metode statice I

- Date membru statice: o singură copie a fiecărei date membru per clasă (pentru care se alocă memorie), indiferent de numărul de obiecte instanțiate la un moment dat.

```
public class Rational {
    private int numerator;
    private int denominator;

    static double PRIMEGAME_FIRST = 17/91;
    // John H. Conway's prime producing machine
    // ...
}
```

- Pot fi referite utilizând numele clasei (de preferat) sau numele unei variabile instanță a clasei.

```
System.out.println(Rational.PRIMEGAME_FIRST);
System.out.println(r1.PRIMEGAME_FIRST);
```

Date membru și metode statice II

- Metodele statice (metodele de clasă) nu sunt specifice obiectelor, ci clasei
- Sunt partajate între toate obiectele clasei.
- O metodă statică nu are acces la date membru sau metode non statice ale clasei (nu există referința `this`).
- Metodele statice au acces la datele membru și metodele statice ale clasei.

Exemplu

`lecture.examples.lecture2.Rational.java`, clasa `MathUtils`[a]

[a] acesta este un link, toate exemplele sunt la adresa `https://github.com/cs-ubbcluj-ro/MAP`

Moștenirea

- O relație de tipul *"este"* (eng. *"is-a"* relationship).
- Structura (datele membru non-private) și comportamentul (metodele non-private) clasei de bază sunt disponibile claselor derivate.
- În Java se folosește cuvântul cheie `extends`.

```
class Base {
    // ...
}

class Derived extends Base {
    // ...
}
```

- Java nu permite moștenirea multiplă. (dar în **Python**? **C++**?)

Constructorii în moștenire

- Dacă în clasa derivată nu se apelează în mod explicit un constructor al clasei de bază, compilatorul introduce un apel către constructorul implicit (fără parametri) al clasei de bază în mod automat.
- Dacă în clasa de bază nu există un constructor implicit, constructorul clasei derivate trebuie să apeleze în mod explicit unul din constructorii existenți.
- Constructorul clasei de bază se apelează folosind cuvântul cheie `super`, acest apel fiind obligatoriu prima instrucțiune din constructorul clasei derivate.
- Cuvântul cheie `super` se folosește și pentru a referi datele membru și metodele clasei de bază, sub forma `super.member` sau `super.memberFunction()` (cât timp nu sunt `private`).

Exemplu

`lecture.examples.lecture2.Inheritance.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Clasa `java.lang.Object`

- Toate clasele Java sunt derivate din clasa `Object`
- Metodele clasei `Object`
  - `toString()` - reprezentarea sub formă de șir de caractere.
  - `equals()` - verificarea egalității logice.
  - `hashCode()` - returnează codul hash al obiectului.
  - `getClass()` - returnează clasa obiectului.
  - `clone()` - creează o clonă a obiectului (eng. shallow clone în mod implicit).
  - `finalize()` - apelat de componenta de garbage collection înainte de distrugerea obiectului.
  - ... and others.

Super-important de știut!

- Documentația clasei `java.lang.Object` este aici https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html.
- Parcurgeți secțiunea metodelor `equals()` și `hashCode()` pentru a economisi câteva ore de depanare ☺.

Suprascrierea metodelor

- Se folosește adnotarea `@Override`:
  - Transmitem compilatorului că dorim să suprascriem o metodă.
  - Face codul mai ușor de înțeles.
- Metodele statice nu pot fi suprascrise.
- Tipul returnat de o metodă suprascrisă poate fi un subtip al tipului returnat în clasa de bază (returnarea unui tip covariant, (eng. *covariant return type*).

Exemplu

`lecture.examples.lecture2.Inheritance.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa `https://github.com/cs-ubbcluj-ro/MAP`

Polimorfism

```
Animal p3 = new Penguin("white and black", 8, "Emperor");
System.out.println(p3.toString());
```

- Ce este polimorfismul? (cum funcționează în C++, Python)
- În limbajul Java toate metodele sunt în mod implicit virtuale (excepție făcând metodele statice, private, finale sau constructorii).
- De fapt, în limbajul Java nu există cuvântul rezervat `virtual`, acest comportament fiind implicit.

Cuvântul cheie `final`

- *Metode finale* - nu pot fi suprascrise în clasele derivate.
- *Clase finale* - nu pot fi create clase derivate din acestea.
- *Date membru finale* - trebuie inițializate la declarare sau în cadrul unui bloc de inițializare. Valoarea unei variabile marcate `final` nu poate fi modificată.

Exemplu

`lecture.examples.lecture2.FinalExample.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa `https://github.com/cs-ubbcluj-ro/MAP`

Clase abstracte I

- Utile pentru a defini concepte abstracte.
- O metodă este abstractă dacă este declarată dar nu este definită. O metodă abstractă se declară folosind cuvântul cheie `abstract`.

```
[access_modifier] abstract
                  Type name([parameters_list]);
```

- O clasă abstractă este o clasă ce poate conține metode abstracte (dar nu este obligatoriu).
- O clasă abstractă nu poate fi instanțiată
- Dacă o clasă are cel puțin o metodă abstractă ea trebuie declarată abstractă

Clase abstracte II

- O clasă se declară ca fiind abstractă folosind cuvântul cheie `abstract`.

```
abstract class ClassName {
    // ...
}
```

- O clasă este abstractă atâta timp cât are metode (sau metode moștenite din clasele abstract părinte) care sunt abstracte.
- Dacă o clasă derivată nu definește toate metodele abstracte moștenite din clasa de bază sau din părinții acesteia în mod tranzitiv, aceasta trebuie să fie declarată abstractă.

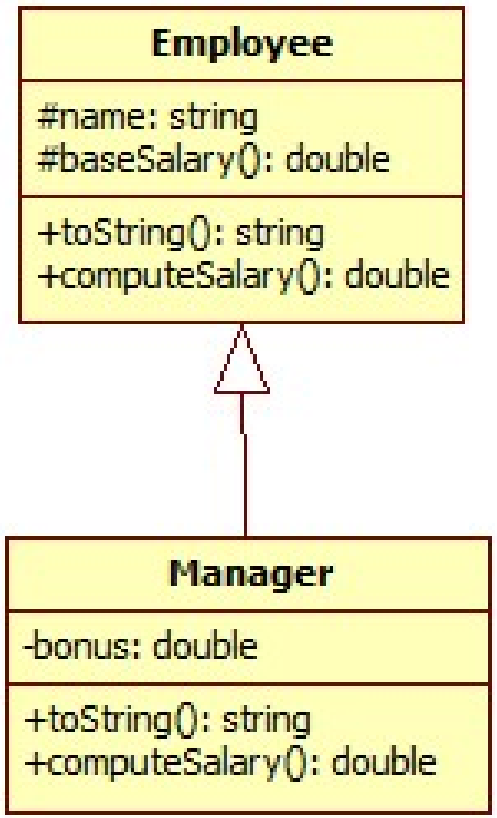
Exemplu

`lecture.examples.lecture2.AbstractClasses.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Exercițiu I

Scrieți codul Java corespunzător următoarei diagrame de clase UML (vezi slide următor).



Exercițiu II

- Compania are mai mulți angajați, unii dintre ei având rolul de manager.
- Metoda `toString` din clasa `Employee` returnează un `String` cu numele angajatului.
- Metoda `toString` din clasa `Manager` returnează un `String` compus din cuvântul "Manager" urmat de numele angajatului.
- Metoda `computeSalary` din clasa `Employee` returnează salariul de bază al angajatului.
- Metoda `computeSalary` din clasa `Manager` returnează salariul de bază, la care se adaugă bonusul managerului.

Exercițiu III

- Scrieți un program de test care creează mai mulți angajați (atât angajați cât și manageri) și îi adaugă pe toți într-o listă
- Scrieți o funcție care pentru afișează pe ecran numele și salariile tuturor angajaților primiți printr-un parametru de tip listă folosing valorile returnate de metodele `toString` și `computeSalary`.

Interfețe I

- Comparat cu o clasă abstractă, toate metodele unei interfețe sunt abstracte
- O interfață se declară folosind cuvântul rezervat `interface`.

```
public interface InterfaceName {
    [method declaration];
}
```

- O interfață poate conține doar declarații de metode (fără implementări – **vezi slide-ul următor** ☺)
- O interfață nu are constructori
- Toate metodele unei interfețe sunt în mod implicit publice
- O dată membru a unei interfețe este în mod implicit `public`, `static` și `final`.

Interfețe II

- Java 8 a introdus conceptul de metode implicite a unei interfețe (eng. *default method*)[1]
- Scopul lor este de a permite modificarea contractului unei interfețe (adăugarea de noi metode) în momentul în care se dorește actualizarea sa

[1] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

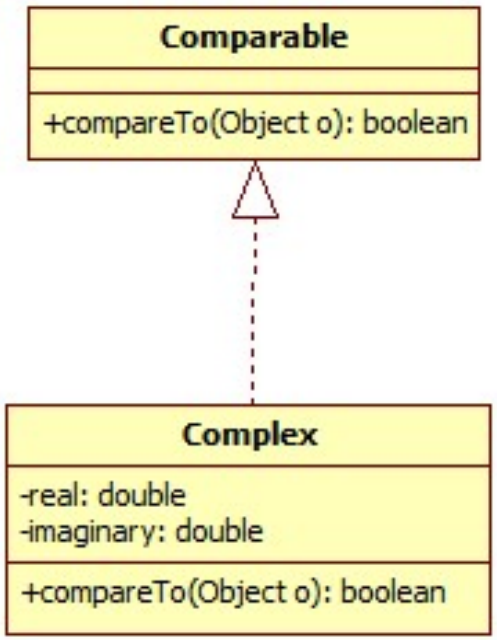
Implementarea interfețelor I

- O clasă poate implementa o interfață prin definirea metodelor declarate în acea interfață
- Relația este declarată prin utilizarea cuvântului cheie `implements`.

```
[public] class ClassName implements InterfaceName {
    [implementations of interface methods]
    // other definitions
}
```

- Dacă interfața are cel puțin o metodă care nu este implementată de clasă, atunci clasa trebuie să fie declarată `abstract`.
- Definirea metodelor implicite (eng. default methods) nu este obligatorie

Implementarea interfețelor II



Implementarea interfețelor III

- Este posibil ca o interfață să extindă una sau mai multe alte interfețe, precum și să adauge propriile sale metode (cuvântul cheie `extends`).
- O clasă poate implementa mai multe interfețe
- Clasa trebuie să definească toate metodele tuturor interfețelor implementate, în caz contrar trebuie declarată ca fiind `abstract`.
- În momentul implementării mai multor interfețe, trebuie să avem grijă la coliziuni (declarări de metode comune sau de metode implicite în Java 8+)
- E posibil să referim un obiect folosind o referința la una din interfețele pe care acesta o implementează – în acest caz, sunt accesibile doar metodele interfeței

Implementarea interfețelor IV

- Exemplu pentru metode statice și implicite în Java 8+

```
[public] default Return_type
                        method([parameters_list]) {
    // default implementation
}

[public] static Return_type
                        method([parameters_list]) {
    // implementation
}
```

- Metodele implicite pot fi suprascrise în clasele ce implementează interfața
- Metodele statice definesc metoda și nu pot fi suprascrise

Clase abstracte vs. interfețe

Clasă abstractă	**Interfață**
Poate include metode cu orice modificator de acces	Metodele sunt publice
Poate declara și defini orice tip de date membru	Toate datele membru sunt implicit `static` și `final`
Se pot defini constructori	Nu are constructori
Poate să nu aibă nici o metodă abstractă	Metodele non-implicite și non-statice sunt abstracte, dar se poate defini o interfață fără nici o metodă (interfață marker)
Nu poate fi instanțiată	Nu poate fi instanțiată

Operatorul `instanceof`

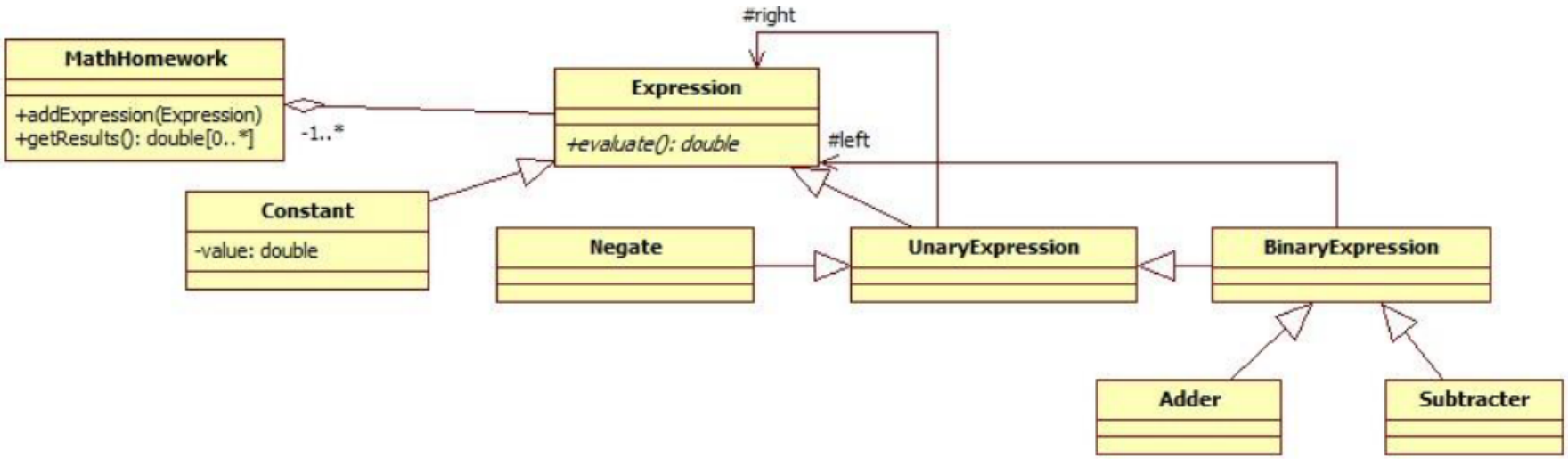
- Utilizat pentru a testa dacă o referință aparține unui anumit tip de dată
- Funcționare polimorfică (pe baza ierarhiei de moștenire)
- Returnează `false` când este apelat pe o instanță `null`

Exemplu

`lecture.examples.lecture2.Interfaces.java`[a]

[a] acesta este un link, toate exemplele sunt la adresa https://github.com/cs-ubbcluj-ro/MAP

Interfețe - exercițiu I



Interfețe - exercițiu II

Scrieți o aplicație pentru calcularea expresiilor matematice, după cum urmează:

- Interfața `Expression` conține metoda `evaluate()`
- Clasa `Constant` reprezintă o constantă și conține o valoare. Rezultatul evaluării unei expresii constante este valoarea sa
- Clasele `UnaryExpression` și `Negate` agreghează câte o expresie; prima returnează evaluarea expresiei conținute, iar a doua returnează negarea evaluării valorii expresiei conținute.
- Clasa `BinaryExpression` agreghează alte două expresii
- Clasele `Adder` și `Subtracter` sunt evaluate ca suma și respectiv diferența expresiilor agregate
- Clasa `MathHomework` permite crearea unei teme ce include mai multe expresii ce trebuie evaluate. Metoda `getResults()` returnează toate rezultatele expresiilor incluse

Interfețe - exercițiu III

- Scrieți o metodă care creează o temă care va evalua valorilor următoarelor două expresii: -5 + (9 - 3) și -(4 + 2) - (-10). Afișați rezultatele evaluării pe ecran

Rezumat

- Clase, clase abstracte și interfețe în limbajul Java
- Moștenire și polimorfism în Java
- **Săptămâna următoare**:
  - Pachete
  - Clase imbricate
  - Tipuri generice
  - Colecții Java (eng. *Java Collections Framework*)