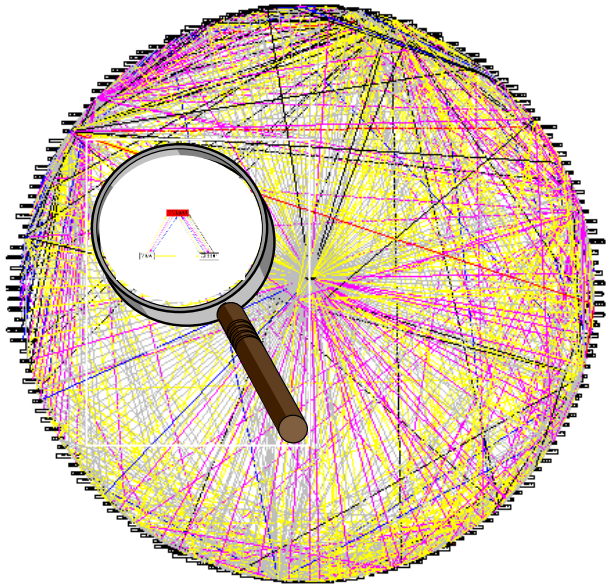
Metode avansate de programare

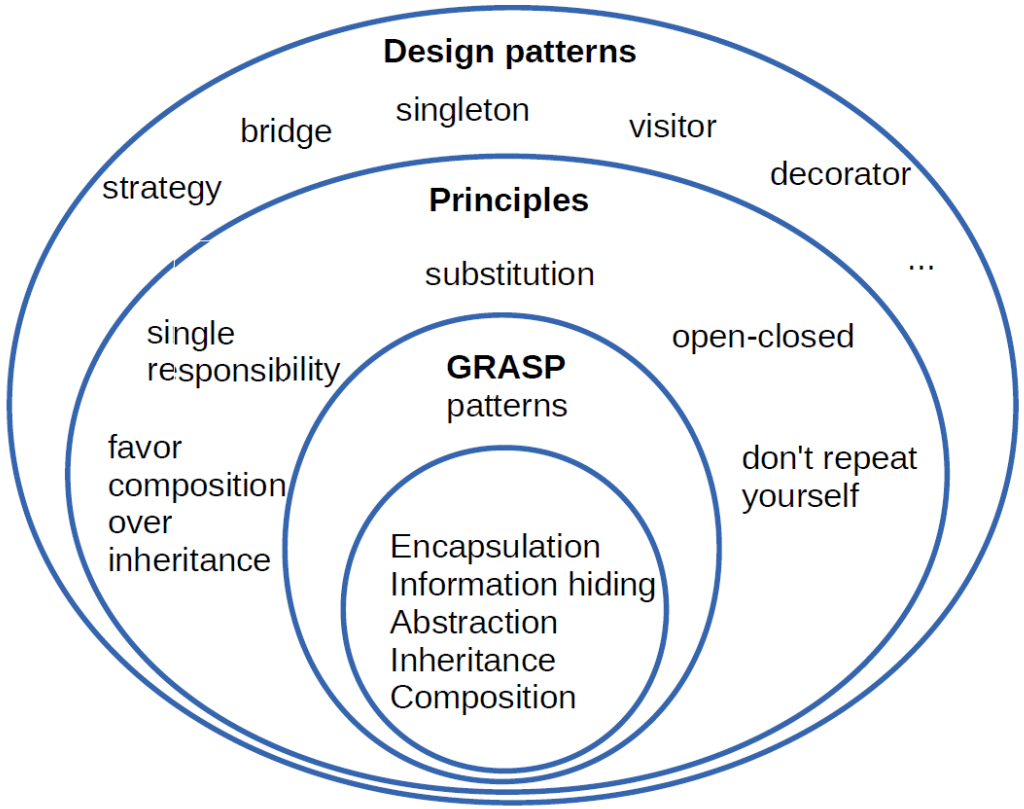
Informatică Română

Curs 1

Principii de proiectare



Imaginati-va ca: o modificare minora pe care o faceti sistemului soft, aceasta propaga modificari in 33% din clase …



- Cuplare scazuta
- Coeziune ridicata
- Complexitate gestionabila
- Abstractizare corespunzatoare

Principiile SOLID

- (SRP) Single Responsibility Principle
- (OCP) Open closed Principle
- (LSP) Liskov substitution Principle
- (ISP) Interface Segregation Principle
- (DIP) Dependency Inversion Principle

(SRP) Single Responsibility Principle

- Fiecare clasă ar trebui să aibă o singură responsabilitate
- Orice modul sau clasă trebuie să încapsuleze o singură funcționalitate.
- O clasă sau funcție, nu ar trebui să rezolve sau sa trateze mai mult decât un singur scop deoarece aceasta ar introduce cuplarea între cele două funcționalități.

(OCP) Open closed Principle

- *Entitățile software ar trebui să fie deschise pentru extindere, dar închise pentru modificare.*
- Când o componentă software este implementată și utilizată, orice modificare ulterioară a acelei componente poate duce la erori sau comportament nedefinit.
- Așadar, componentele software, odată implementate, nu ar trebui să fie modificate, ci să-și extindă funcționalitatea prin alte mijloace.

(LSP) Liskov substitution Principle

- (LSP) Liskov substitution Principle
- Orice clasă derivată poate înlocui clasa de bază și codul să funcționeze corect în continuare
- Acest principiu spune că, într-un program dacă S este subtip al lui T, atunci obiectele de tip T pot fi înlocuite cu obiecte de tip S, fără modificarea funcționalităților esențiale.
- Altfel spus noile clase extinse din părinte trebuie să fie capabile să înlocuiască clasa de bază în toate funcțiile sale fără a fi nevoie să aducem modificări nicăieri.

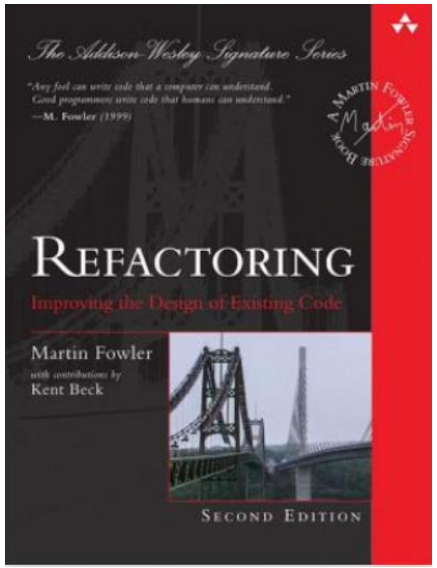
(ISP) Interface Segregation Principle

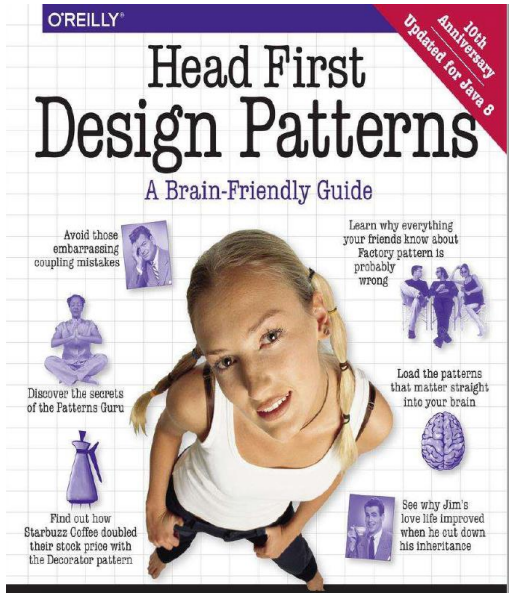
- In loc să avem o interfață cu foarte multe metode, din care doar câteva sunt folosite, mai bine avem mai multe interfețe mai mici și le folosim doar pe cele necesare
- Acest principiu expune ideea că niciun client nu are voie să fie forțat să depindă de metodele pe care nu le folosește
- Aceasta se realizează prin ”spargerea” interfețelor în module mai mici pe care clasele au opțiunea de a le implementa sau nu.
- Acest mod va ajuta și la ideea de decuplarea funcționalităților atunci când se dorește re-factorizarea produsului.

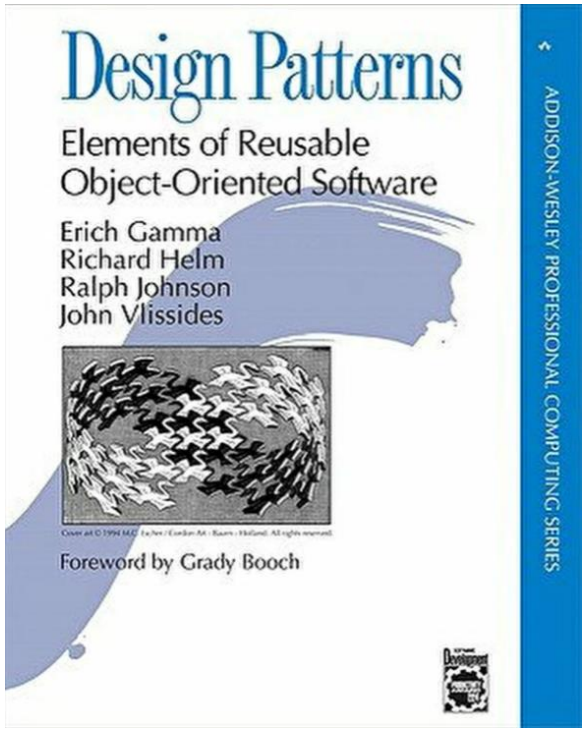
(DIP) Dependency Inversion Principle

- Dacă o clasă are o dependență, această dependență ar trebui să fie o interfață, nu o clasă concretă – acest principiu încurajează decuplarea claselor și e necesar pentru testarea unitară
- Acest principiu se referă la o anumită formă de a decupla module software. Principiul spune că:
  - Modulele de nivel înalt nu trebuie să depindă de modulele de nivel jos.
  - Ambele module trebuie să depindă de abstractizări.
  - Abstractizările nu trebuie să depindă de detalii ci detaliile trebuie să depindă de abstract

Referinte

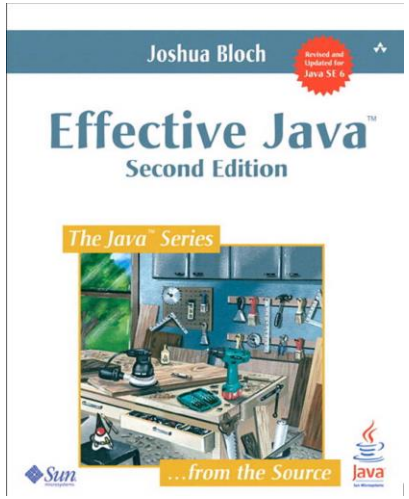


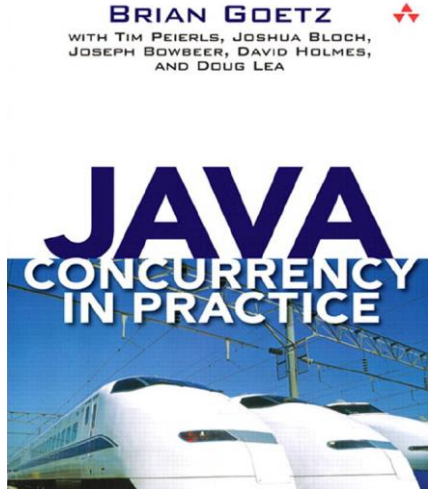




"If builders built houses the way programmers built programs, first woodpecker to come along would destroy civilization."

E.W. Dijkstrathe





Concepte POO de bază

- *Clasa*:
- *Obiect*:
- *Mesaj*:
- *Încapsularea:*
- *Moștenirea:*
- *Polimorfismul:*

Concepte POO de bază

- *Clasa*: reprezintă un tip de dată
    - Corespunde implementării un TAD
- *Obiect*: este o instanță a unei clase.
    - Obiectele interacționează între ele prin mesaje.
- *Mesaj*: folosit de obiecte pt a comunica.
    - Un mesaj este un apel de metodă.
- *Încapsularea* (separarea reprezentării de interfață)
    - datelor (starea)
    - operațiilor (comportamentul, interfața, protocolul)
- *Moștenirea:* reutilizarea codului, definirea unei ierarhii de obiecte
- *Polimorfismul* – capacitatea unei entități de a reacționa diferit în funcție de context

Conținut curs MAP

- Principii, euristici, reguli ale unei bune proiectări orientată obiect (aplicabilitate în limbajele Java si C#)

- Generics, Collections
- IO, XML, JSON, DB
- Lambda, streams
- GUI – JavaFX
- Reflection
- Concurrency

- Prima parte, Sapt [1..n]: **Java**
- A doua parte, Sapt [n+1..14]: **.NET, C#**

De ce Java?

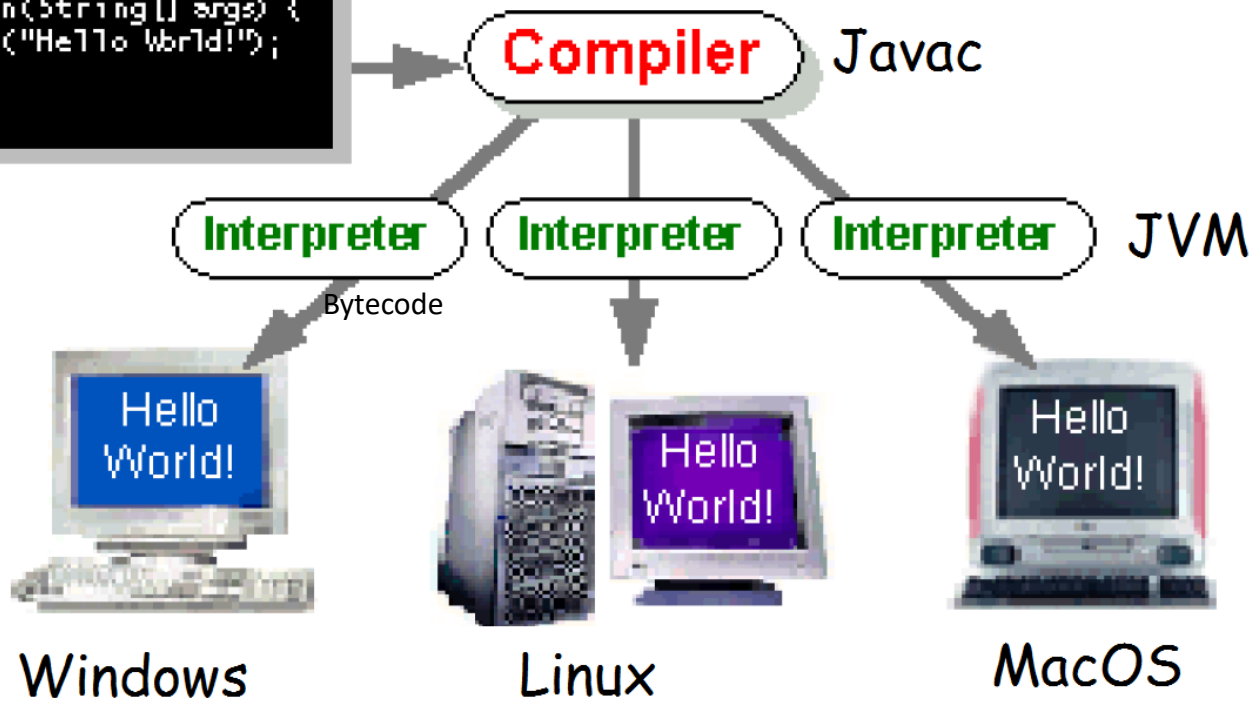
Probleme	Soluții
pointers	references (“safe” pointers)
memory leaks	garbage collection
error handling	exception handling
complexity	reusable code in APIs
platform dependent	portable code (Linux, Mac OS X, and Windows)

... plus
object oriented, security, networking, multithreading, web programming, mobile apps etc...

Java : Compilat si interpretat

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

HelloWorldApp.java




```
D:\__MAP>javac  HelloWordApp.java

D:\__MAP>java  HelloWordApp
Hello Word

D:\__MAP>_
```

Introducere în limbajul Java

- language syntax
- primitive data types,
- arrays
- classes
- interfaces
- packages,
- enums
- overriding, overloading,
- exceptions

Sintaxa

- Similară cu C++;
- Cuvinte cheie

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient
class	(goto)	protected	try
(const)	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof		

- Literali: “Hello World”, 'J', 'a', 'v', 'a', 10, 010, 0xA, 0b11, 12.3, 12.3d,12.3f, 12e3, 123L, true, false, null;
- Separatori: ( ) { } [ ] ; , .

- Operatori:

Access, method call: `.`, `[]`, `()`
Postfix: ***`expr++`***, ***`expr--`*** (R to L)
Other unary: ***`++expr`***, ***`--expr`***, `+`, `-`, `~`, `!`, `new`, ***`(aType)`***
Arithmetic: `*`, `/`, `%`
Additive: `+`, `-`
Shift: <<, >>, >>>
Relational: `<`, `>`, `<=`, `>=`, `instanceof`
Equality: `==`, `!=`
Logical (L to R): `&`, `^`, `|`, `&&`, `||`
Ternary: ***`condition ? (expr1):(expr2)`*** (R to L)
Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`
Precedence: from top to bottom
Tip: don’t rely too much on precedence rules: use parentheses

- http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html

Sintaxa cont

- Comentarii

Single line: `//Single line comment`

Multiple lines:
```
/* non-nested, multi-line
                    comment*/
```

Javadoc comment: `/** processed by javadoc */`

- Codificarea caracterelor – formatul UNICODE
  - Inlocuiește setul de caractere ASCII
  - Un caracter se reprezintă pe **2 octeți**
  - **65536** caractere, **\uxxxx (**0x0041 A)
  - Compatibil ASCII: primele 256 caractere sunt cele din ASCII

Tipuri de date

- Primitive: Boolean(variabila), char(16), byte(8), short(16), int(32), long(64), float(32), double(64)
- Referință: clasa, interfața, []
- Observații:
  - void nu este tip in Java, este doar un cuvânt cheie pentru cazurile în care dorim să indicăm că ceea ce se returnează este nimic (cast to void not allow!)
  - null

Variabile și constante

```java
byte a;  //declarare
int valoare = 100;  //declarare + initializare
final double PI = 3.14;  //constanta
long numarElemente = 12345678L;
String bauturaMeaPreferata = "apa";
```

- Constante - **final**

```java
final int MAX=100;
final int MAX2;
MAX2=100;
MAX2=150; //eroare
```

- var (Java 10)

```java
var str = "Java 10"; // infers String
var list = new ArrayList<String>(); // infers ArrayList<String>
var stream = list.stream(); // infers Stream<String>
```

Instrucțiuni

```
if (booleanExpr){
   // do something
}else { // else is optional
   // do something else
}
```

```
for (Tip variable : collection) {  //Java 5.0
   // loop body
}
for(Student s: studentsList){
   System.out.println(s);
}
```

```
while (booleanExpr) {
   // execute body
   // until booleanExpr becomes false
}

do {
   // execute body (at least once)
   // until booleanExpr becomes false
}while (booleanExpr);

for (int i=0; i < n; i++) {
   // execute loop body n times
}
```

```
switch (Expr){
   case Value1: instructions;
      break;
   case Value2: instructions;
      break;
   // ...
   default: instructions;
}
```

Expr poate fi:

- **byte**, **short**, **int**, **char** (sau clase învelitoare)
- **enum** types
- String (equals) (**new** in Java 7)

Tablouri unidimensionale

- **Declarare:**

```
tip[] nume_ tablou;      sau  tip nume_ tablou[];
```

- **Creare:**

```
nume_tablou=new tip[dim]; //se aloca spatiul in memorie
                          //indexarea 0 … dim-1
```

- **Referirea unui element:**

```
nume_tablou[index]
```

```
float[] vec;
vec=new float[10];
int[] sir=new int[3];
float tmp[];
tmp=vec //vec si tmp refera acelasi tablou
```

Tablouri unidimensionale

- Variabila ***length***: returneaza dimensiunea (capacitatea) tabloului

```
int[] sir=new int[5];

int lung_sir=sir.length; //lung=5;
```

- Creare si initializare

```
int[] dx={-1,0, 1};
```

Tablouri N-dimensionale

- **Rectangulare**

```
int[][] a;
a=new int[5][5];
a[0][0]=2;
int x=a[2][2]; //x=?
```

- **Nerectangulare**

```
int[][] a=new int[3][];
for(int i=0;i<3;i++)
      a[i]=new int[i+1];
int x=a.length; //x=?
int y=a[2].length; //y=?
```

Concepte de bază în programarea orientată obiect

- *Clasa*: reprezintă un tip de dată
    - Corespunde implementării un TAD
- *Obiect*: este o instanță a unei clase.
    - Obiectele interacționează între ele prin mesaje.
- *Mesaj*: folosit de obiecte pt a comunica.
    - Un mesaj este un apel de metodă.
- *Încapsularea* (separarea reprezentării de interfață)
    - datelor (starea)
    - Operațiilor (comportamentul, interfața, protocolul)
- *Moștenirea:* reutilizarea codului, definirea unei ierarhii de obiecte
- *Polimorfismul* – capacitatea unei entități de a reacționa diferit în funcție de context

Declararea/definirea unei clase în Java

```
//NumeFisier.java
[public] [abstract][final] class NumeClasa{
            [declaratii date membru (atribute, câmpuri)]
            [declaratii si implementare metode membru]
            [clase imbricate (interne)]
}
```

Observatii:

1. Daca clasa **NumeClasa** este declarata public, atunci clasa se salveaza intr-un fisier cu numele **NumeClasa.java.**

2. Intr-un fisier **.java** pot fi definite mai multe clase, insa cel mult una poate fi **publica**.

3. Diferente C++:
   - Nu sunt necesare 2 fisiere separate (.h, .cpp).
   - Metodele se implementeaza in locul declararii.
   - La sfarsitul clasei nu se pune **;**

Declararea/definirea unei clase în Java

```
//Persoana.java
public class Persoana{
//...
}
// Complex.java
class Rational{
//...
}
class Natural{
//...
}
public class Complex{
//...
}
```

Declararea datelor membru

```
[public] [abstract][final] class NumeClasa{
      [modificator_acces][abstract][static][final] Tip nume[=val_initiala];
}
```

```
Modificator_acces poate fi public, protected, private.
```

Observatii:

1. Modificatorul de acces trebuie precizat pentru fiecare atribut și are următoarea semnificație:

- **private:** membru accesibil doar clasei
- **protected:** membru accesibil doar clasei și subclaselor
- **public:** membru accesibil tuturor
- *implicit:* membru accesibil doar la nivel de pachet (Daca modificatorul de acces lipseste, atributul este vizibil in interiorul pachetului (directorului))

Declararea datelor membru. Exemple

```java
//Persoana.java
public class Persoana{
    private String nume;
    private int varsta;
//...
}
```

```java
//Punct.java
public class Punct{
    protected double x;
    protected double y;
//...
}
```

```java
//Cerc.java
public class Cerc{
    double raza;
    Punct centru;
    static final double PI=3.14;
}
```

Inițializarea atributelor

- *La locul declarării:* private double x=0;
- *În blocul special de initializare*

```
public class Rational{
    private int numarator;
    private int numitor;
    {
        numarator=0;
        numitor=1;
    }
}
```

- *În constructor.*
- *Observatie:* Daca un atribut nu este explicit initializat, atunci el va avea valoarea implicită corespunzatoare tipului său.

Declararea/definirea metodei constructor

Constructorul precizeaza o secventa de instructiuni care se vor executa imediat dupa crearea unui obiect.

```
[...] class NumeClasa{
          [modificator_acces] NumeClasa([lista_parametrilor_formali]){
           //secventa de instructiuni
        }
}
modificator_acces ∈ {public, protected, private}
lista_parametrilor_formali este de forma: Tip1 nume1[, Tip2 nume2[,...]]
```

```
//Complex.java
public class Complex{
    private double real, imag;
    public Complex(){
        real=0;
        imag=0;
    }
    …
```

```
…
public Complex(double real){
        this.real=real;
        imag=0;
    }
}
```

Supraîncărcarea constructorilor

```
//Complex.java
public class Complex{
    private double real, imag;
    public Complex(){
        real=0;
        imag=0;
    }
    public Complex(double real){
        this.real=real;
        imag=0;
    }
    public Complex(double real, double imag){ //...
    }
    public Complex(Complex c){ //...
    }
}
```

Observatii:

1. Constructorul trebuie sa aibă numele clasei (case sensitive).
2. Constructorul nu are tip returnat.
3. *Daca intr-o clasa nu se defineste nici un constructor*, compilatorul va genera automat un **constructor implicit – default constructor**, având modificatorul de acces public.

Referința la obiectul curent

- this: referință la obiectul curent (putem referi atributele sau metodele).

```
public class Complex{
   private double real, imag;
   Complex(double real){
      this.real=real;
      this.imag=0;
   }
   Complex(double real, double imag){
      this.real=real;
      this.imag=imag;
   }
   public Complex suma (Complex c){
      // compute the sum
      return this;
   }
}
```

Apelul unui alt constructor

- Un anumit constructor poate să apeleze un alt constructor al aceleiași clase. Apelul unui alt constructor trebuie să fie prima instructiune din constructorul apelant.
- Nu pot fi apelati doi constructori diferiti.
- Un constructor nu poate fi apelat din interiorul altor metode!

```
public class Complex{
    private double real, imag;
    Complex(double real){
        this(real,0);
    }
    Complex(double real, double imag){
        this.real=real;
        this.imag=imag;
    }
    public Complex suma (Complex c){
        this(real+c.real,imag+c.imag);
        return this;
    }
}

//erori?
```

Declararea si definirea metodelor

```
[...] class NumeClasa{
     [modificator_acces][atribute] TipR numeMetoda([lista_param_for]){
            //secventa de instructiuni
        }
    }
```

- modificator_acces ∈ {public, protected, private}
- lista_param_for este de forma Tip1 nume1[, Tip2 nume2[, ...]],
- TipR poate fi orice tip primitiv, clasa, tablou, sau void.

- Daca modificatorul de acces lipseste, metoda poate fi apelata din orice clasa definita in acelasi pachet (director)!

```
public class Circle {
    private float radius; //instance variable
    private Point center; //instance variable

    public float getRadius() { return radius; }

    /**
     * set the radius of the circle to a new value
     * @param newRadius
     */
    public void setRadius(float newRadius) {
        radius = newRadius;
    }

    /**
     *
     * @return compute and return the area of the circle
     */
    public float computeArea() { /* implementation */ return 0; }
}
```

Metode cu număr variabil de parametri

- Java 5.0 (Vararg)

```
public void setGrades(Student s, int ... grades){
   for (int g:grades )
   {
      //
   }
}
```

Vararg trebuie sa fie ultimul in lista de parametri!!!

Supraîncărcarea metodelor

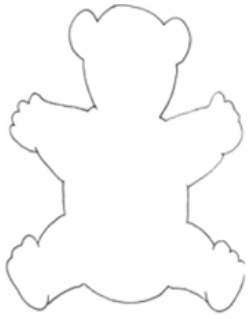
- Intr-o clasa pot fi definte mai multe metode cu acelasi nume, dar care au signaturi diferite.

- Prin *signatura* se intelege numarul parametrilor formali, tipul parametrilor formali si ordinea lor.

```
public class Complex{
    private double real, imag;
    // constructors ...
    public void aduna (double real){
        this.real+=real;
    }
    public void aduna(Complex c){
        this.real+=c.real;
        this.imag+=c.imag;
    }
    public Complex aduna(Complex cc){
        this.real+=cc.real;
        this.imag+=cc.imag;
        return this;
    }
}
//Erorrs?
```

Crearea obiectelor

- la execuția programului, prin operatorul new și sunt alocate în heap



```
public class Urs {
   private String specie;
   private String nume;
   private int varsta;

   public Urs(String specie, String nume, int varsta) {
      this.specie = specie;
      this.nume = nume;
      this.varsta = varsta;
   }
}
```

Referirea obiectelor

```
Urs ursPanda; //ursPanda referinta la obiect
ursPanda =new Urs("Panda","Domino",10);
Urs ursBrun =new Urs("Brun","Martin",15);
Urs polarel =new Urs("Polar","White",10);
```

Referirea obiectelor - null

`Urs ursOarecare=null;` //Varibila ursOarecare nu are asociat nici un obiect (ex. Telecomanda fara TV)

`ursOarecare=ursPanda;` Varibila ursOarecare si ursPanda refera acelasi obiect din memorie

Distrugerea obiectelor

- Destructor:
  - În Java nu exista destructor.
  - Memoria este dealocată de garbage collector.

Transmiterea parametrilor

- Prin *valoare*. Se copiaza valoarea lor pe stiva.
- Pentru parametri de tip obiect si pt tablouri, aceasta valoare este o referinta. Informatiile din obiect sau tablou pot fi modificate
- C++ ?

```
public void initStudent(Student s, String nume,int varsta, int anStudiu)
{
    //s=new Student(nume,varsta,anStudiu);   ??????????
    s.setNume(nume);
    s.setVarsta(varsta);
    s.setAnStudiu(anStudiu);
}


Student s=new Student();
initStudent(s);
System.out.println(s);
```

Vector de obiecte

```
Student [] studentsList=new Student [4];
studentsList[0]=new Student("Andrei",12,3);
studentsList[1]=new Student("Aprogramatoarei",18,1);
studentsList[2]=new Student("Dan",32,3);
studentsList[3]=new Student("Ion",22,3);

for(Student s: studentsList){
    System.out.println(s);
}
```

Clase învelitoare (wrapper classes)

Tip primitiv	Clasa învelitoare
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

```
Integer intObject1 = new Integer(34);
Integer intObject2 = new Integer("35");

Boolean boolValue1 = new Boolean("true");

Integer intObject3 = Integer.valueOf("36");
Integer intObject4 = Integer.valueOf("1001", 2); // 9 in baza
2

Integer intObj1 = 23;      //autoboxing
int intPrimitive = intObj1;    //unboxing
intPrimitive++;
Integer intObj2 = intPrimitive;    //autoboxing
```

Membri de tip clasă. Modificatorul static.

- **Variabile statice (de clasă)** – *valori memorate la nivelul clasei și nu la nivelul fiecărei instanțe.*

  Exemplu: declararea eficientă a constantelor

  ```
  static final double PI = 3.14;
  ```

  Pot fi referite si prin numele clasei: `System.out.println(Cerc.PI);`

- **Metode statice (de clasă)** – metode aplicabile la nivel de clasă și nu de instanță (pot opera doar pe variabile statice)
- *Metodele statice se pot apela **si doar** cu numele clasei (nu e necesar sa cream o instanta):*

  ```
  NumeClasa.numeMetodaStatica(listaParametri)
  ```

- A se vedea si contextul in care apare modificatorul **static** la clase interne (cursul urmator)

Utilizarea membrilor statici

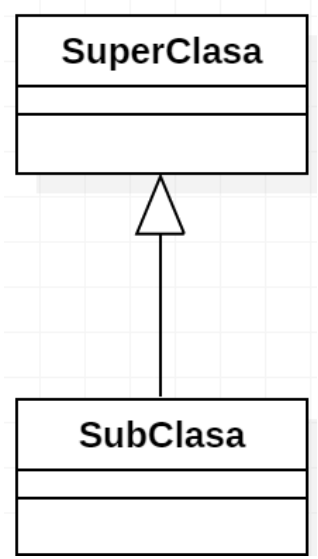
```
public class Exemplu {
    int x ; // Variabila de instanta
    static long n; // Variabila de clasa
    public void metodaDeInstanta() {
        n ++; // Corect
        x --; // Corect
    }
    public static void metodaStatica() {
        n ++; // Corect
        x --; // Eroare la compilare !
    }
}
```

```
Exemplu.metodaStatica(); // Corect
Exemplu obj = new Exemplu();
obj.metodaStatica(); // Corect
```

```
Exemplu.metodaDeInstanta(); // Eroare
Exemplu obj = new Exemplu();
obj.metodaDeInstanta(); // Corect
```

Mostenirea

```
class SuperClasa {
...
}
class SubClasa extends SuperClasa {
...
}
```



Mostenirea este o relație de tip **is - a**

- *... este o relație între clase prin care o clasă (superclasă / clasă de bază) pune la dispoziția altor clase (subclase / clase derivate)* ***structura și comportamentul*** *definite de ea.*

O clasă poate extinde direct cel mult o clasă. NU există moștenire multiplă de clasă în Java.

Programarea orientată pe obiecte este o metodă de implementare a programelor în care acestea sunt organizate ca și colecții de obiecte ce cooperează între ele [...], fiecare obiect fiind o instanță a unei clase, ***și fiecare clasă fiind membră a unei ierarhii de clase*** [clase unite prin relații de moștenire].

Constructori în contextul moștenirii

Q: În contextul moștenirii, cine ar fi cel mai în măsură să inițializeze câmpurile moștenite de la o clasă de bază ?

A: Constructorul acelei superclase!

- Dacă în superclasă există un constructor fara parametri (default sau nu), compilatorul introduce automat un apel la acel constructor ca primă instrucțiune.
- Dacă nu avem constructor fara parametri în clasa de bază, în constructorul subclaselor trebuie apelat explicit un constructor cu parametri din superclasă.
- Prima instrucțiune dintr-un constructor e fie apel la alt constructor al aceleiași clase, fie apel la un constructor din superclasa directă.

Exemplu grila

```
class A{
    int i;
    public A(int i){
        System.out.print("A()");
    }
}
class B extends A{
    float i = 3;
    public B(){
        System.out.print("B()");
    }
}
public class AB extends B{
    public static void main(String argv[]){
        AB[] x= new AB[]{new AB(), new AB()};
        for (AB obj: x) {
            obj.testMethodCA("AB");
        }
    }
    public void testMethodCA(String s){
        System.out.println(s);
    }
}
```

Constructori în contextul moștenirii. Exemplu

```
public class Persoana{
    protected String nume;
    protected byte varsta;
    public Persoana(){
        this("",(byte)0);
    }
    public Persoana(String nume, byte varsta){
        this.nume=nume;
        this.varsta=varsta;
    }
```

```
public class Student extends Persoana {
    private static final int COD_UNIVERSITATE = 15;
    private int anStudiu;

    public Student(String nume, byte varsta, int anStudiu)
    {
        super(nume,varsta); //apelconstr cu param din clasa de baza
        this.anStudiu=anStudiu;
    }
```

- Dacă în constructorul clasei Student prima instrucțiune nu ar fi fost `super(nume,varsta);`, s-ar fi apelat automat constructorul `Persoana()`. Dacă acesta nu exista, primeam eroare, deoarece nu avem constructor default in clasa Persoana.

Super în contextul moștenirii

```
public class Persoana{
    protected String nume;
    protected byte varsta;

    public Persoana(String nume, byte varsta){
        this.nume=nume;
        this.varsta=varsta;
    }
    public void socializeaza() {
        System.out.println(nume + " socializeaza  ... ");
    }
```

super

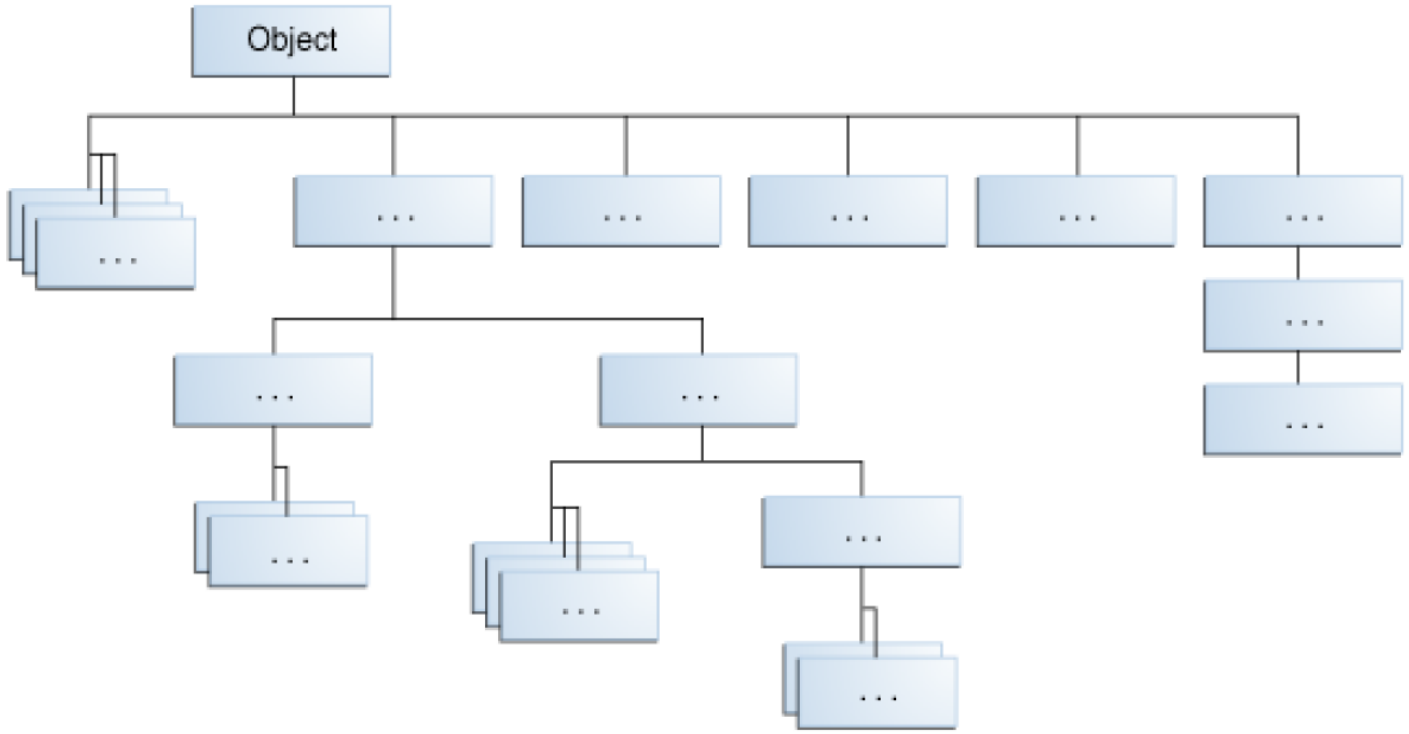
- Pentru a apela, din constructorul clasei derivate, un constructor al clasei de baza, apelul acestui constructor fiind prima instructiune din constructorul clasei derivate. super(nume, varsta)
- Pentru a referi un membru al clasei de baza in clasa derivata. super.membruClasaBaza

```
public class Student extends Persoana {
    …
    public Student(String nume, byte varsta, int anStudiu)
    {
            super(nume,varsta); //apel constr cu param din clasa de baza
            this.anStudiu=anStudiu;
    }

    public void seDistreaza(){
        super.socializeaza(); //apel metoda din clasa de baza
        System.out.println("Si canta ....");
    }
```

Object –Superclasă a tututor claselor

- *În Java orice clasă este derivată din Object!*



Metodele clasei Object

Toate obiectele, inclusiv tablourile, moștenesc metodele acestei clase:

- **toString :** returnează reprezentarea ca șir de caractere a unui obiect
- **equals :** testează egalitatea conținutului a două obiecte
- **hashCode :** returnează valoarea *hash* corespunzătoare unui obiect
- **getClass :** returnează clasa din care a fost instanțiat obiectul
- **clone :** creează o copie a obiectului (implicit: *shallow copy*)
- **finalize :** apelată de GC înainte de distrugerea obiectului
- **...**

Redefinirea (overriding) unei metode

- Uneori, implementarea moștenită a unei operații nu e adecvată/suficientă pentru o subclasă
- **Doar metodele de instanță se pot redefini!**

Redefinirea, in clasa Persoana, a metodei toString() din clasa Object

```
@Override
public String toString() {
    return "nume='" + nume + '\'' + ", varsta=" + varsta ;
}
```

Redefinirea, in clasa Student, a metodei toString() din clasa Persoana

```
@Override
public String toString() {
       return super.toString()+", "+ "an de studiu="+anStudiu;
}
```

Metoda redefinita in clasa derivate poate avea tipul returnat ca fiind un subtip al celui returnat de metoda in clasa de baza!

Legarea dinamică

```
Persoana stud=new Student("Andu",(byte)3,34);
stud.toString();
```

Ce metodă se apelează? toString() de la Persoană sau toString() de la Student?

În Java, orice metodă este implicit virtuală!

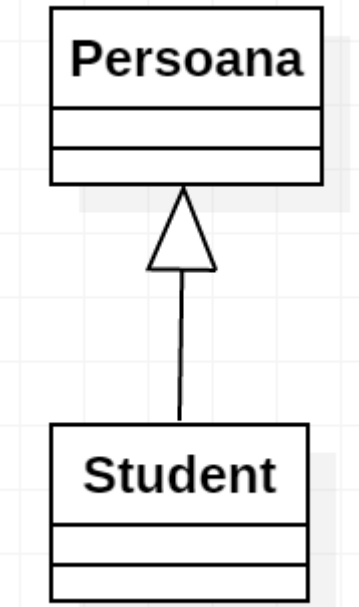
Overriding vs. overloading

- Suprascriere /redefinire / (overriding) - același nume, aceeași semnătură
- Supraîncărcare (overloading) - același nume, semnături diferite

Upcasting si downcasting

```
Persoana persoana;
Student student=new Student();
persoana=student;   //upcasting  (automat)


Student s2=(Student)persoana; //downcasting
explicit
```



Modificatori/Specificatori de acces în contextul moștenirii

- private - respectivul membru al clasei (atribut/metodă) poate fi accesat doar în interiorul clasei
- public - respectivul membru al clasei (atribut /metodă) poate fi accesat de oriunde
- protected - respectivul membru al clasei (atribut /metodă) poate fi accesat din interiorul clasei, din subclasele sale (pe this) sau din același pachet (pe orice obiect).

Modificatorul final

- *Metodele final* – nu mai pot fi redefinite (overriding) in clasele derivate
- *Clasele final* – nu mai pot fi extinse (e frunza in arborele ierarhiei de clase)
  - final class A{}
  - class B extends A {} – nu e posibil
- *Atributele, variabile locale, argumentele final* – constant
    - Constantele **`static final –`** trebuie initializate la declarare;
      - `static final double PI=2;`

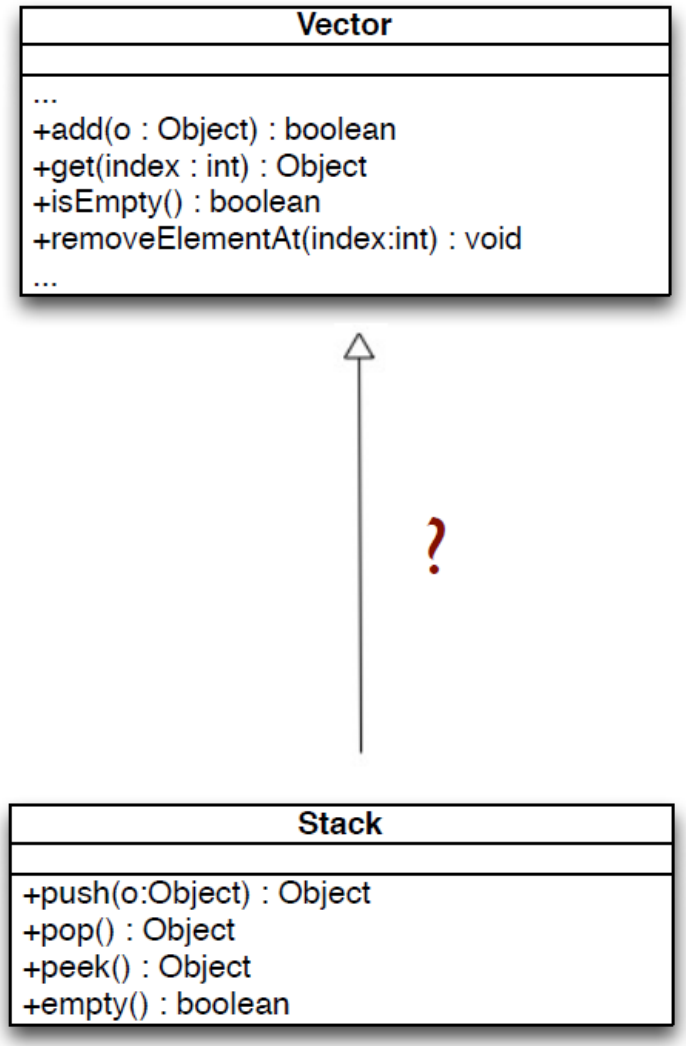
Moștenirea de clasă vs. compunerea obiectelor (object composition)

- Euristică importantă a Programării orientate pe obiecte:

  Nu folosiți moștenirea doar pentru a reutiliza codul unei superclase!

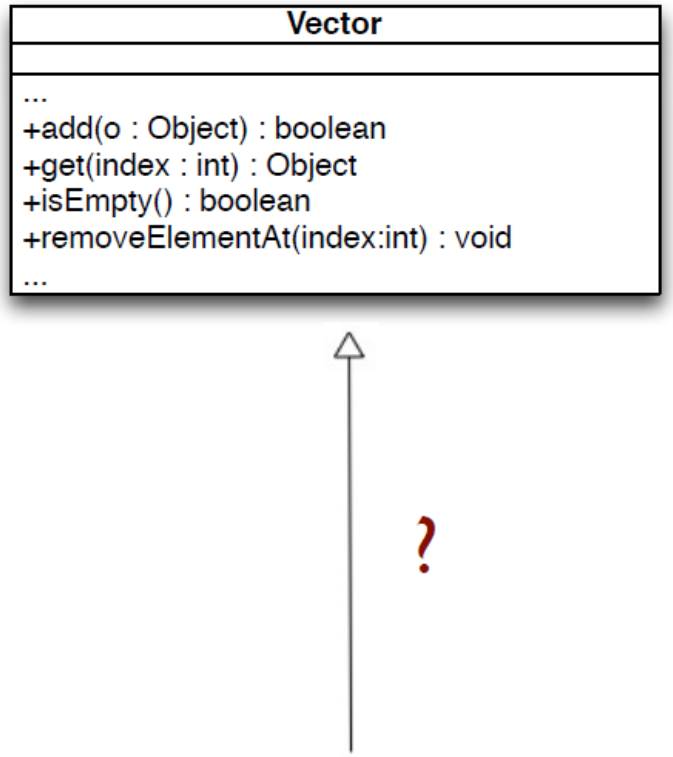
  Favorizează compunerea obiectelor în locul moștenirii de clasă! (Favor composition instead of inheritance!)

Studiu de caz




```
class Stack extends Vector {
    public Object push(Object o) {
            this.add(o);
            return o;
    }
    public Object pop() {
            Object r = this.get(this.size() - 1);
            this.removeElementAt(this.size() - 1);
            return r;
    }
        ...
}
```

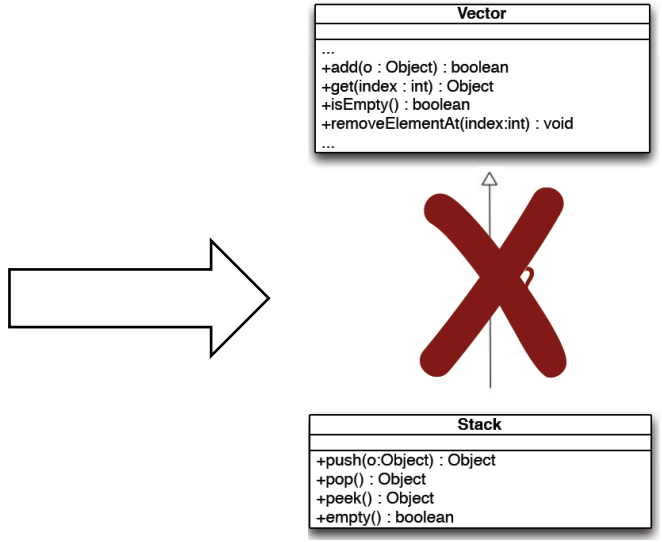
Studiu de caz




```
class Stack extends Vector {
    public Object push(Object o) {
            this.add(o);
            return o;
    }
    public Object pop() {
            Object r = this.get(this.size() - 1);
            this.removeElementAt(this.size() - 1);
            return r;
    }
        ...
}

public static void main(String[] args) {
    Stack stk = new Stack();
    stk.push(new Integer(5));
    stk.push(new Integer(10));
    stk.push(new Integer(15));
    Object p = stk.pop();
    System.out.println(p);
```

Așa NU!




```
    stk.removeElementAt(0);
     // adică cum (????)
     // operatiile ce nu caracterizeaza notiunea de stivă pot fi
folosite pe o stiva?
     ...
}
```

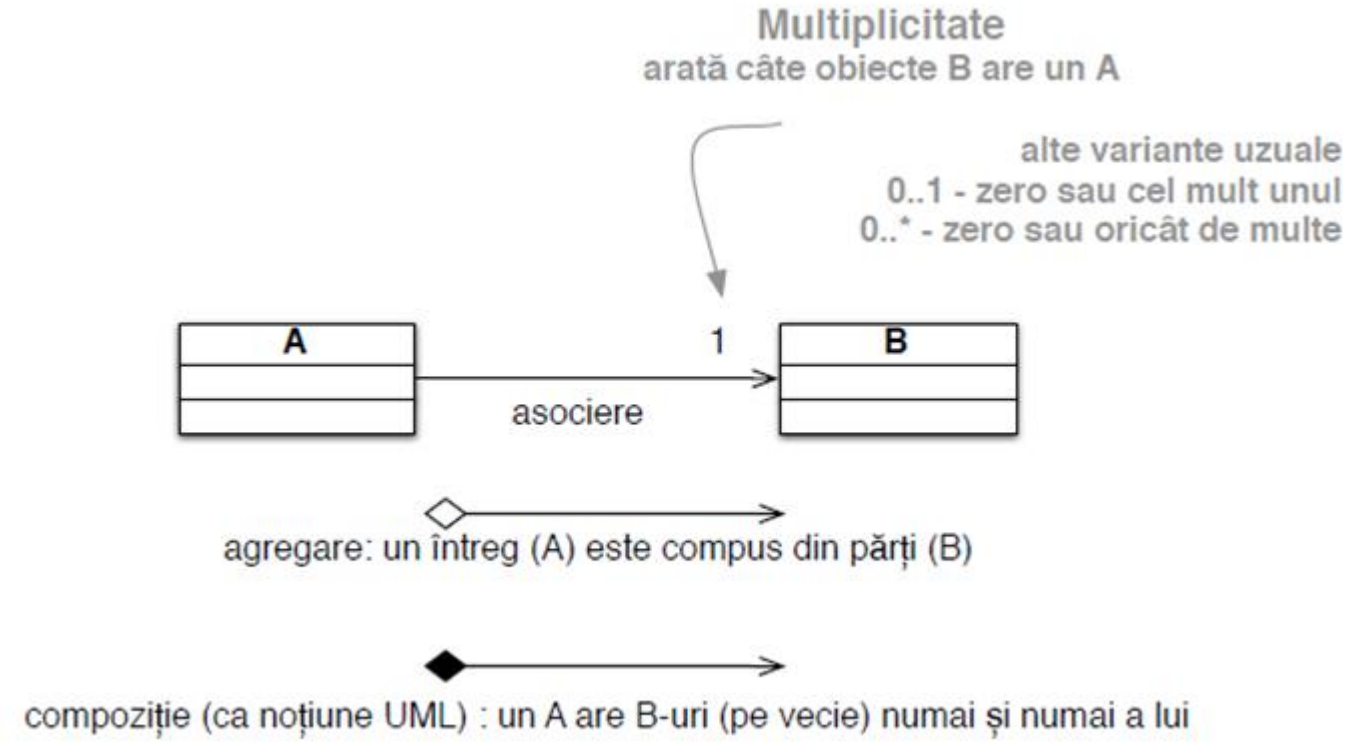
Soluția

- Compunerea de obiecte

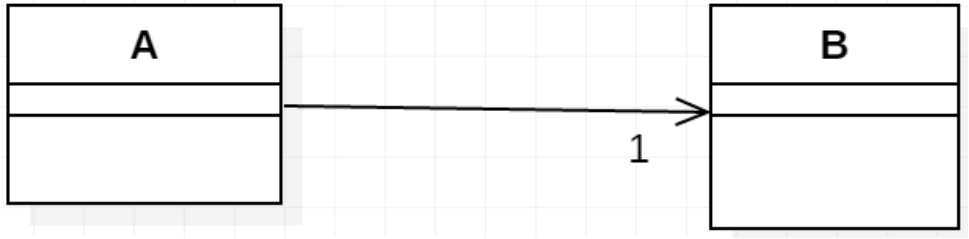
Compunerea de obiecte

- În esență, definirea de variabile de instanță într-o clasă ca referințe la obiecte (inclusiv când referințele sunt într-o variabilă de instanță de tip tablou).

```
class A {
        private B b;
}
```



Asociere

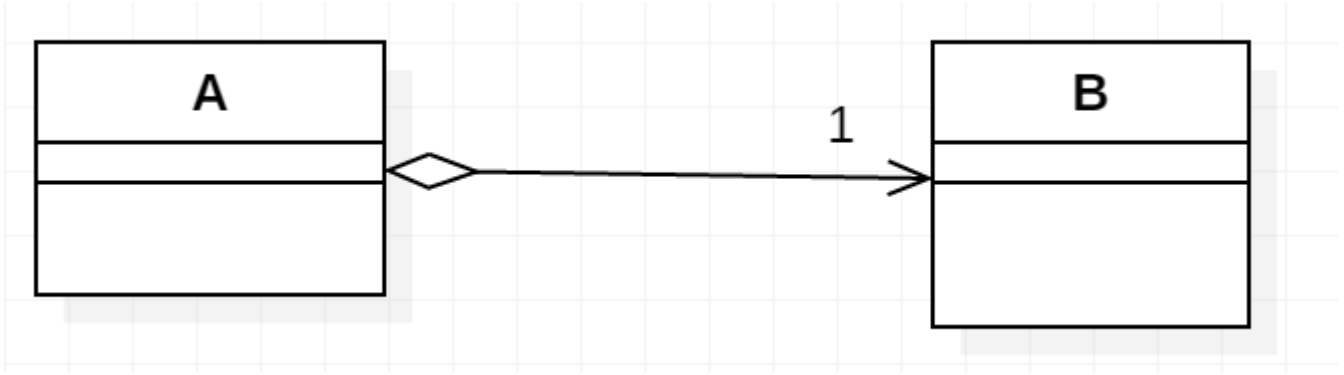



```
class A{
    private B b;
    public A() {}

    public void setB(B b){
        this.b=b;
    }
}
```

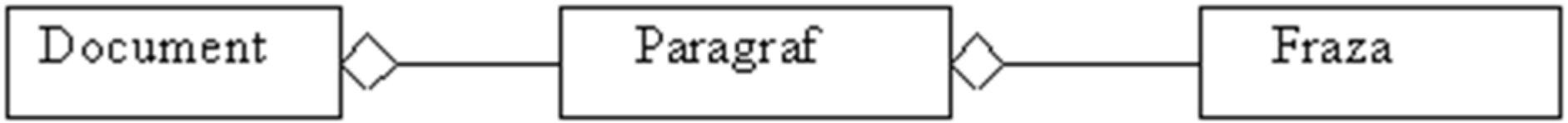
- A poate exista si fara B

Agregare



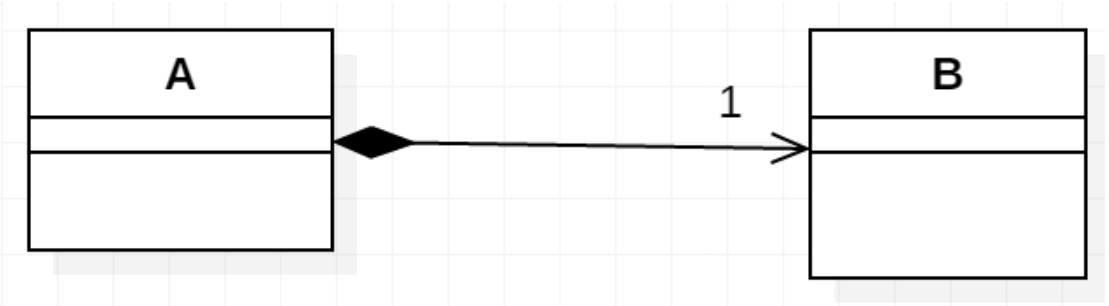

```
class A{
    private B b;
    public A(B b){
        this.b=b;
    }
}
```

- Agregatul (A) nu poate exista fara una dintre componente
- B exista fara A
- Distrugerea agregatului nu conduce la distrugerea componentelor



Clase agregat.

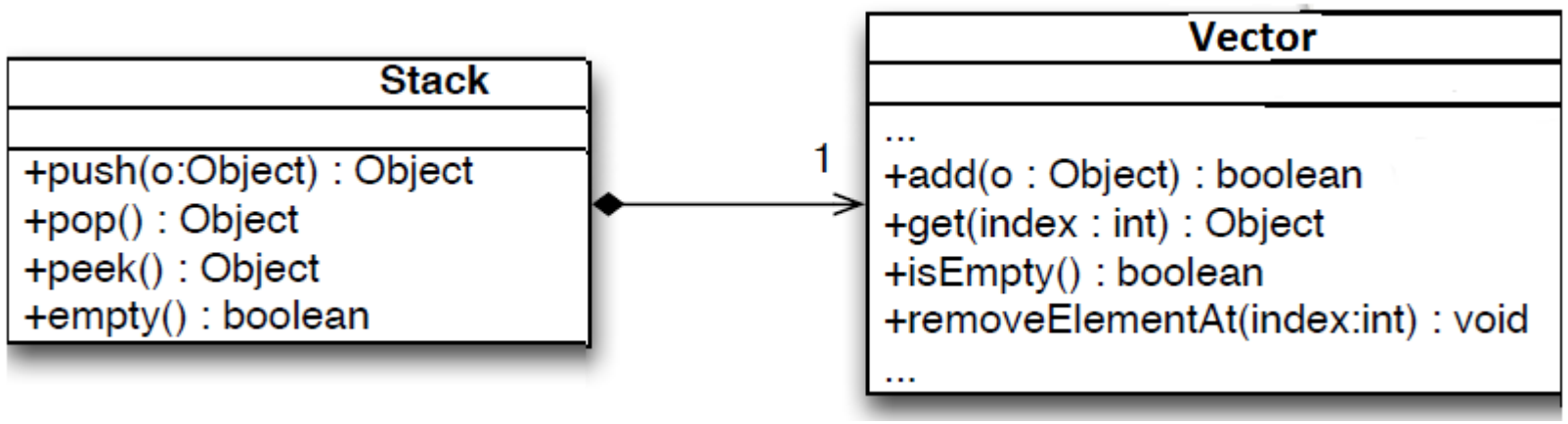
Compoziție




```
class A{
    private B b=new B();
    public A()
    {
        // sau aici b=new B()//
    }
}
```

- B este creat de catre A
- Distrugerea agregatului presupune si distrugerea componentei

Stiva corectată




```
class Stack {
    Vector v =new Vector();
    public Object push(Object o) {
        v.add(o);
        return o;
    }
    public Object pop() {
        Object r = v.get(this.size() - 1);
        v.removeElementAt(this.size() - 1);
        return r;
    }
    ...
}
```

- Stiva are un vector si e numai si numai al ei!!!
- Prin urmare avem un alt mod de reutilizare a codului!!!!

Clase abstracte

- Definesc concepte abstracte: formă, animal, figură, task ....
- *O metodă* se numește abstractă dacă se declară, dar nu se definește în clasa în care a fost declarată. O metoda abstractă se declară folosind cuvantul abstract.

```
[modificator_acces] abstract TipReturnat nume([lista_param_formali]);
```

- *O clasă abstractă* este o clasă ce poate sa contina (dar nu e obligatoriu) metode abstracte.
- *O clasa abstracta* se defineste folosind cuvantul abstract.
- *O clasa abstracta* nu poate fi instantiata. ~~**new**~~
- Daca o clasa are cel putin o metoda abstracta atunci ea trebuie declarata abstracta.
- *O clasa poate fi declarata abstracta fara a avea metode abstracte.*
- Daca o clasa mosteneste o clasa abstracta si nu defineste toate metodele abstracte ale clasei de baza, trebuie declarata si ea abstracta.

Clase abstracte

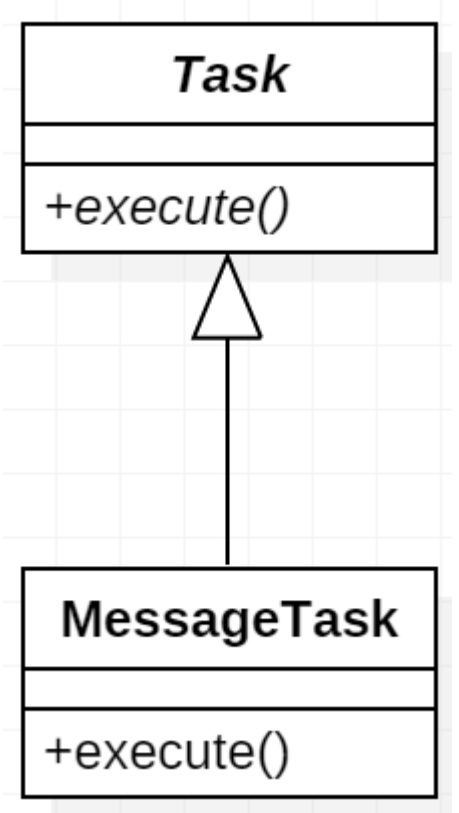
```
public abstract class Task {
    private String taskId;
    private String desc;

    public Task(String taskId, String desc) {
        this.taskId = taskId;
        this.desc = desc;
        System.out.println("Creating a task...");
    }

    public abstract void execute();
}
```

```
public class MessageTask extends Task {
    public String message;
    public MessageTask(String taskId, String desc, String message) {
        super(taskId, desc);
        this.message=message;
    }

    @Override
    public void execute() {
            System.out.println(message);
    }
}
```



Interfețe

- DEX: (inform.) frontieră convențională între două sisteme sau unități, care permite schimburi de informații după anumite reguli.
- *Interfata - contract, protocol de comunicare*
- Clasa - implementeaza (adera la) acel contract

Definirea unei interfețe

```
[public] interface NumeInterfata [extends SuperInterf1,
SuperInterf2...] {
        /* Corpul interfetei:
        Declaratii de constante publice
        Declaratii de metode abstracte publice
         Metode definite (default) Java 8
        */
}
```

- O interfata contine declaratii de metode abstracte (nedefinite), default (definite) si statice si date membru.
- *Orice data membru declarata in interfata este implicit* ***public, static, final***
- Toate metodele declarate intr-o interfata sunt implicit public
- O interfata poate sa nu contina nici o declaratie de metode sau date membru.
- O interfata poate mosteni alta interfata si poate adauga metode noi.
- Extinderea unei interfete: O interfata poate sa mosteneasca mai multe interfete (mostenire multipla de interfete) - pot aparea coliziuni; Cuvant rezervat extends

Definirea unei interfețe

```
public interface Formula {
    double PI=3.14; // este implicit public, static, final
    double calculate(double a, double b); //metoda abstracta

    default double sqrt(double a) {
        return Math.sqrt(a);
    }

    default double power(double a, double  b) {
        return Math.pow(a, b);
    }

    default double numarLaPatrat(double nr)
    {
        return power(nr,2);
    }

    default double numarLaCub(double nr)
    {
        return power(nr,3);
    }
}
```

Exemplu grila

- Care linii de cod din interfata *MyInterface* nu sunt corecte?

```
interface MyInterface {
    protected int X = 10;
    int y;
    int Z = 20;

    default int x() {
        return 0;
    }

    abstract void foo();

    final int f(int x);
}
```

Variante de raspuns:

a)

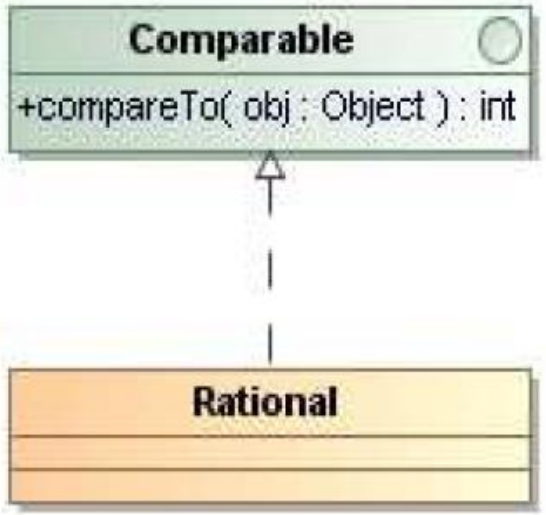
b)

c)

d)

Implementarea unei interfete

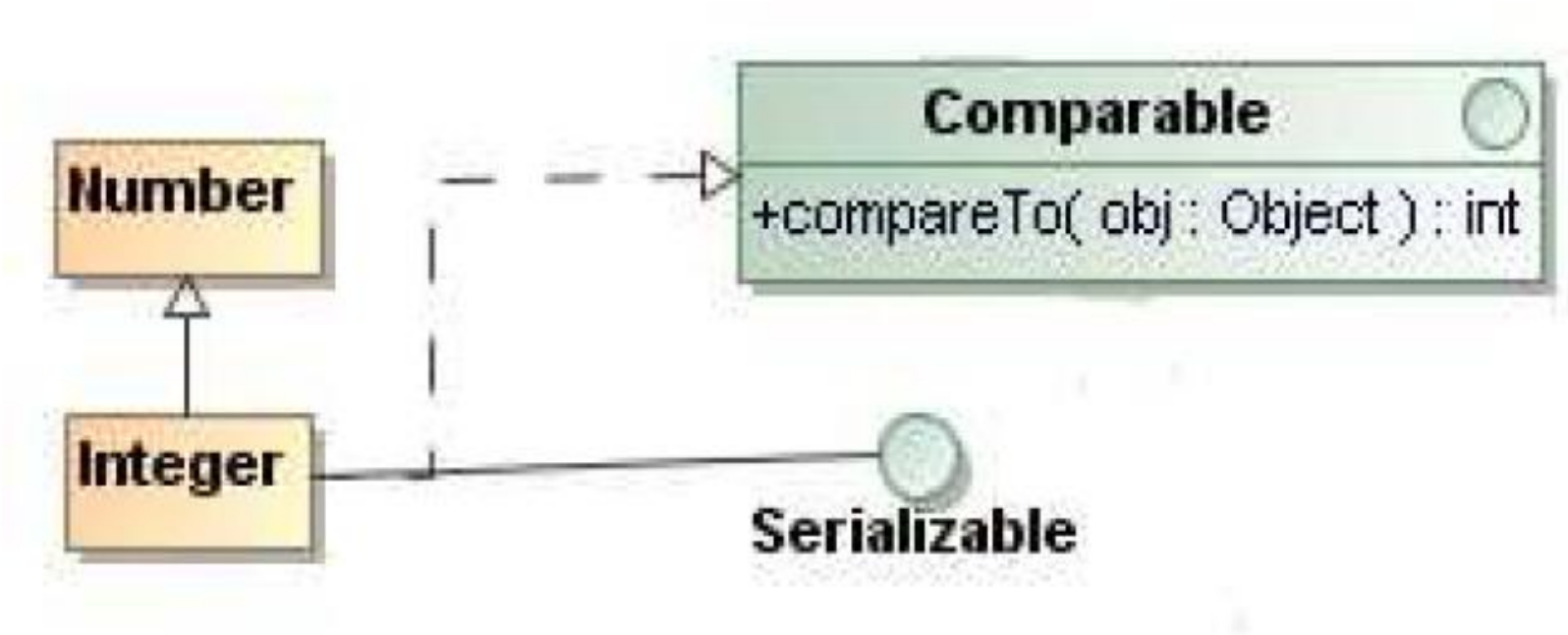
```
public interface Comparable{
    int compareTo(Object o);
}
public class Rational implements Comparable{
    private int numarator, numitor;
    //...
    public int compareTo(Object o){
        //...
    }
}
```



- Clasa se obliga sa defineasca toate metodele abstracte din interfata.
- Daca cel putin o metoda abstracta din interfata nu este definita in clasa, atunci clasa trebuie declarata abstracta!

Moștenire de clasă și implementarea de interfețe

- O clasa poate mosteni cel mult o clasa, dar poate implementa oricate interfete.



Good OO Design

- The following rules of thumb are essential to a good design:
  - "Program to an interface, not to an implementation."
  - “Hide and abstract as much as possible”.
  - "Favor object composition over inheritance."
  - “Minimize relationships among objects and organize related objects in packages”.