METODE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 5-6

- Principiul de desing al unei clase imutabile
- Interfețe
- Utilitate interfețe
- Interfețe Java 8 și 9

➢O **interfața** este un tip de date de referință utilizat pentru a specifica un comportament pe o clasă îl poate **implementa**.

- **Sintaxă:**

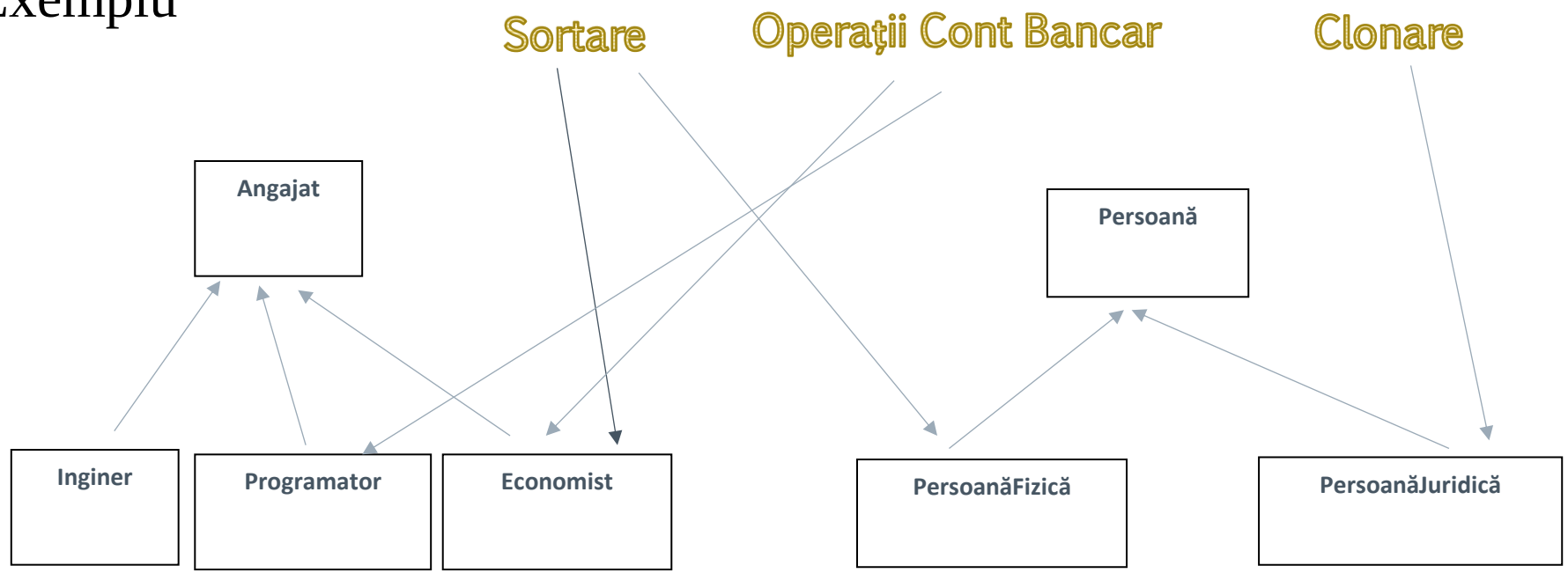
```
public interface numeInterfață{
    constante;
    metode fără implementare;
    metode default cu implementare;
    metode statice cu implementare;
}
```

- Datele membre sunt implicit **`public`, `static` și `final`**, deci sunt constante **care trebuie să fie inițializate**.

- Metodele membre sunt implicit `public`, iar cele fără implementare sunt implicit `abstract`.

- **La modul general, interfețele definesc un set de operații (capabilități) comune mai multor clase care nu sunt înrudite (în sensul unei ierarhii de clase).**

- Exemplu


Sortare
Operații Cont Bancar
Clonare
Angajat
Persoană
Inginer
Programator
Economist
PersoanăFizică
PersoanăJuridică

- Standardul Java oferă două interfețe pentru a compara obiectele în vederea sortării lor. Una dintre ele este interfața **`java.lang.Comparable`**, interfață care asigură o **sortare naturală** a obiectelor după un anumit criteriu.

```
public interface Comparable<Tip>{
      public int compareTo(Tip obiect);
}
```

  - Generalizând, într-o interfață se încapsulează un set de operații care nu sunt specifice unei anumite clase, ci, mai degrabă, **au un caracter transversal (trans-ierarhic)**.

  - Interfața în sine nu face parte dintr-o ierarhie de clase, ci este externă acesteia!!!

INTERFEȚE

- Implementarea unei anumite interfețe de către o clasă oferă o anumită certificare clasei respective (clasa este capabilă să efectueze un anumit set de operații). Astfel, o interfață poate fi privita ca o operație de tip **CAN_DO**.

- În concluzie, interfața poate fi văzută ca un serviciu (API) care poate fi implementat de orice clasă.

- Clasa își anunță intenția de a implementa serviciul respectiv, într-o maniera specifică, realizând-se **astfel un contract între clasă și interfață, cu o clauză clară: clasa trebuie să implementeze metodele abstracte din interfață.**

➢ **Implementarea unei interfețe** se realizează utilizând următoarea sintaxă:

```
[modificatori] class numeClasa implements numeInterfață_1,
                                › numeInterfață_2,..., numeInterfață_n
```

```
class Inginer implements Comparable<Inginer>{
  private String nume;
›   ..........................
  public int compareTo(Inginer ob){

        return this.nume.compareTo(ob.nume);
  }
}
```

➢ Observații

- Dacă o clasă **implementează două interfețe care conțin metode abstracte cu aceeași denumire**, atunci apare un conflict de nume care induce următoarele situații:
  - dacă metodele **au signaturi diferite**, clasa trebuie să implementeze ambele metode;
  - dacă metodele **au aceeași signatură și același tip pentru valoarea returnată**, clasa implementează o singură metodă;
  - dacă metodele **au aceeași signatură, dar tipurile valorilor returnate diferă**, atunci implementarea nu va fi posibilă și se va obține o eroare de compilare.

- În cazul câmpurilor cu același nume, conflictele se pot rezolva prefixând numele unui câmp cu numele interfeței (chiar dacă au tipuri diferite).

- **O interfață nu se poate instanția**, însă un obiect de tipul clasei care o implementează poate fi accesat printr-o referință de tipul interfeței.

- **În limbajul Java un obiect poate fi referit astfel:**

1. printr-o referință de tipul clasei sale => se pot accesa toate metodele publice încapsulate în clasă, alături de cele moștenite din clasa `Object`;

2. printr-o referință de tipul superclasei (polimorfism) => se pot accesa toate metodele moștenite din superclasă, cele redefinite în subclasă, alături de cele moștenite din clasa `Object`;

3. printr-o referință de tipul unei interfețe pe care o implementează => se pot accesa metodele implementate din interfață, alături de cele moștenite din clasa `Object`.

➢ Extindrea interfețelor

- Sintaxa pentru extinderea interfețelor

```
interface SubInterfata extends SuperInterfata1, SuperInterfata2, ....SuperIntervatan
```

- Utilitate: dacă o interfața implementată de către mai multe clase se modifică, atunci se modifică și codul claselor, astfel se preferă extinnderea interfeței într-o alta suninterfață!

➢ Utilitatea interfețelor

- **Definirea unor funcționalități ale unei clase**
- Definirea unor grupuri de constante
- Implementarea mecanismului de callback
- Interfețe marker
- Clase adaptor

➢ Definirea unor grupuri de constante

```
public interface ConstanteMatematice{
    double PI = 3.14159265358979323846;
    double SQRT_2 = 1.41421356237;
    double SQRT_3 = 1.73205080757;
    double LN_2 = 0.69314718056;
}
class TriunghiEchilateral {
   private double latura;

    public TriunghiEchilateral(double x){
        latura = x;
    }
    public double aria(){
        return latura*latura*ConstanteMatematice.SQRT_3/4;
    }}
```

➢ Mecanismul de callback

- O altă utilitate importantă a unei interfețe o constituie **posibilitatea de a transmite o metodă ca argument al unei alte metode (callback).**

- Implementarea mecanismului de callback în limbajul Java se realizează, de obicei, astfel:

1. se definește o **interfață care încapsulează metoda generică** sub forma unei metode abstracte;
2. se definește **o clasă care conține o metodă pentru realizarea prelucrării generice** dorite (metoda primește ca parametru o referință de tipul interfeței pentru a accesa metoda generică) – **metoda callback**;
3. se definesc **clase care implementează interfața**, respectiv clase care conțin implementările dorite pentru metoda generică din interfață;
4. se realizează prelucrările dorite apelând metoda din clasa definită la pasul 2 în care parametrul de tipul referinței la interfață se înlocuiește cu instanțe ale claselor definite la pasul 3.

➢ Exemplu mecanismul de callback

- Sortarea obiectelor folosind interfața **java.lang.Comparator<?>**

P1. Se definește o clasă care implementează interfața Comparator, oferind un criteriu de soratre

```
publicclass ComparatorVârste implements Comparator<Inginer>{
   public int compare (Inginer ing1, Inginer ing2){
        return ing2.getVârsta() - ing1.getVârsta();
    }
}
```

P2. Se apelează metoda statică `sort` a clasei utilitare `Arrays` care va primi ca parametru un obiect al clasei `ComparatorVârste` sub forma unei referințe de tipul interfeței `Comparator`:

```
Arrays.sort(t, new ComparatorVarste());
```

INTERFEȚE MARKER

- ✓ Interfețele marker sunt interfețe care nu conțin nicio constantă și nicio metodă, ci doar anunță mașina virtuală Java faptul că se dorește asigurarea unei anumite funcționalități la rularea programului.

- ✓ În standardul Java sunt definite mai multe interfețe marker, precum **java.io.Serializable** care este utilizată pentru a asigura salvarea obiectelor sub forma unui șir de octeți într-un fișiere binar sau **java.lang.Cloneable** care asigură clonarea unui obiect

- ✓ O clasă care implementează intefața `Clonable` permite apelul metodei `Object.clone()` pentru instanțele sale
- ✓ Prin convenție, o clasă care implementează interfața **`Cloneable`**, redefinește metoda **`Object.clone()`** (care are acces protejat) printr-o metodă cu acces public.
- ✓ Clonarea unui obiect presupune, în sine, copierea acestuia la o alta adresă HEAP alocată pentru obiectul destinație

CLASE ADAPTOR

✓O interfață poate să conțină multe metode abstracte. De exemplu, interfața `MouseListener` conține 8 metode asociate unor evenimente produse de mouse `mouseClicked()`, `mousePressed()`,`mouseReleased()` etc.)

```
interface Intref{
  public void m1();
  public void m2();
  public void m3();
: : : :

  public void m1000();
}
```

```
class Test implements Intref{
  public void m1(){ // some statements }
  public void m2(){ // some statements }
  public void m3(){ // some statements }

: : : :
public void m1000(){ // some statements } }
```

CLASE ADAPTOR

- O soluție o constituie definirea unei ***clase adaptor***, respectiv o clasă care să implementeze minimal (cod vid) toate metodele din interfață.

- Dacă pentru o clasă este necesară doar implementeze unui set restrâns de metode din interfață, atunci clasa poate să extindă clasei adaptor, **redefinind doar metodele necesare**.

```
abstract class Adapter
implements Intref
{
  public void m1(){};
  public void m2(){};
  public void m3(){};
  : : : :
  public void m1000(){};
}
```

```
class Test extends Adapter{
public void m1()
{
    System.out.println("This is m1() method.");
}
public void m80
{
  System.out.println("This is m80() method.");
  }
}
```

INTERFEȚE ÎN JAVA 8

- ✓ Un **dezavantaj major** al interfețelor specifice versiunilor anterioare Java 8 îl constituie faptul că modificarea unei interfețe necesită modificarea tuturor claselor care o implementează!!!

- ✓ Începând cu versiunea Java 8 o interfață poate să conțină și **metode cu implementări implicite (default)** sau **metode statice cu implementare**.

```
interface Interfață{
        ..........................
        default tipRezultat metodăImplicită(...){
                //implementare implicită
         }
        static tipRezultat metodăStatică(...){
                //implementare
         }
}
```

➢ Observații

- O clasă care implementează interfața **preia implicit implementările metodelor default**.

- Dacă este necesar, **o metodă default poate fi redefinită** într-o clasă care implementează interfața respectivă.

- O metodă dintr-o interfață poate fi și statică, dacă nu dorim ca metoda respectivă să fie preluată de către clasă. Practic, metoda va aparține strict interfeței, putând fi invocată doar prin numele interfeței.

➢ Extinderea interfețelor care conțin metode default

✓În momentul extinderii unei interfețe care conține o metodă default pot să apară următoarele situații:

- sub-interfața nu are nicio metodă cu același nume => clasa va moșteni metoda default din super-interfață;
- sub-interfața conține o metodă abstractă cu același nume => metoda redevine abstractă (nu mai este default);
- sub-interfața redefinește metoda default tot printr-o metodă default;

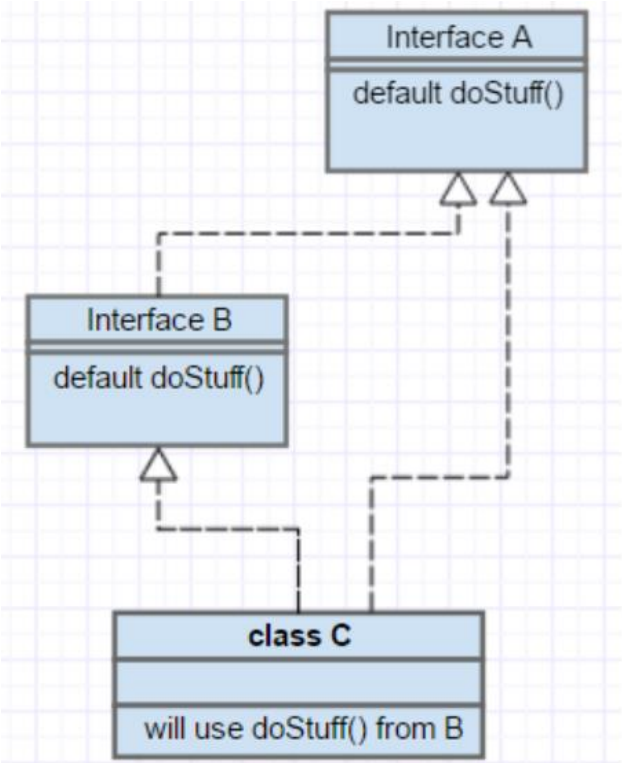
- sub-interfața extinde două super-interfețe care conțin două metode default cu aceeași signatură și același tip returnat => sub-interfața trebuie să redefinească metoda, eventual, poate să apeleze în implementarea sa metodele din super-interfețe folosind sintaxa `SuperInterfata.super.metoda();`

- sub-interfața extinde două super-interfețe care conțin două metode default cu aceeași signatură și tipuri returnate diferite => moștenirea nu este posibilă.

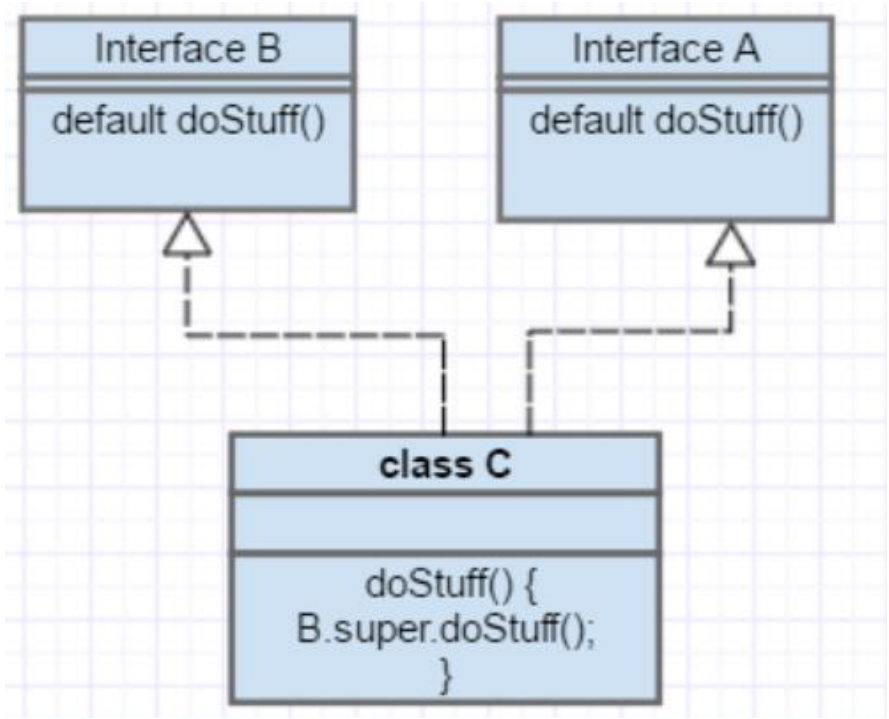
- **Reguli pentru extinderea interfețelor și implementarea lor (problema rombului)**

1. Clasele au prioritate mai mare decât interfețele (dacă o metodă default dintr-o interfață este redefinită într-o clasă, atunci se va apela metoda din clasa respectivă).

2. Interfețele "specializate" (sub-interfețele) au prioritate mai mare decât interfețele "generale" (super-interfețe).



3. **Nu există regula 3!** Dacă în urma aplicării regulilor 1 și 2 nu există o singură interfață câștigătoare, atunci clasele trebuie să rezolve conflictul de nume explicit, respectiv vor redefini metoda default, eventual apelând una dintre metodele default printr-o construcție sintactică de forma `Interfață.super.metoda()`.



CLASA StringBuilder

```
interface Poet
{
        default void write()
         {System.out.println("Poet's
default method");}
}
```

```
interface Writer
{
       default void write()
          {System.out.println("Writer's
default method");}
}
```

```
public class Multitalented implements Poet,Writer
{
      public static void main(String args[])
        {
                  Multitalented ob = new Multitalented();
                  author.write();
        }}
```

```
class Multitalented inherits
unrelated defaults for write()
from typesPoet and Writer
```


```
interface Poet
{
        default void write()
         {System.out.println("Poet's
default method");}
}
```

```
interface Writer
{
       default void write()
          {System.out.println("Writer's
default method");}
}
```

```
public class Multitalented implements Poet, Writer
{
      @Override
         public void write()
                {
                      System.out.println("Writing stories now days");
                }

}
```

INTERFEȚEIN JAVA 9

- **În Java 9** a fost adăugată posibilitatea ca o interfață să conțină metode private, statice sau nu.

- **Regulile de definire sunt următoarele:**
  - metodele private trebuie să fie definite complet (să nu fie abstracte);
  - metodele private pot fi statice, dar nu pot fi default.

```
public interface Calculator {
  default void calculComplex_1(…) {
        Cod comun
        Cod specific 1}
  default void calculComplex_2(…) {
    Cod comun
    Cod specific 2
}}
```


```
public interface Calculator{
    default void calculComplex_1(…) {
    codComun(…);
    Cod specific 1
}
    default void calculComplex_2(…) {
    codComun(…);
    Cod specific 2
}
    private void codComun(…) {
    Cod comun
}
}
```