

INTERFEȚE

- O **interfață** este un tip de date abstract utilizat pentru a specifica un comportament pe care trebuie să-l implementeze o clasă.

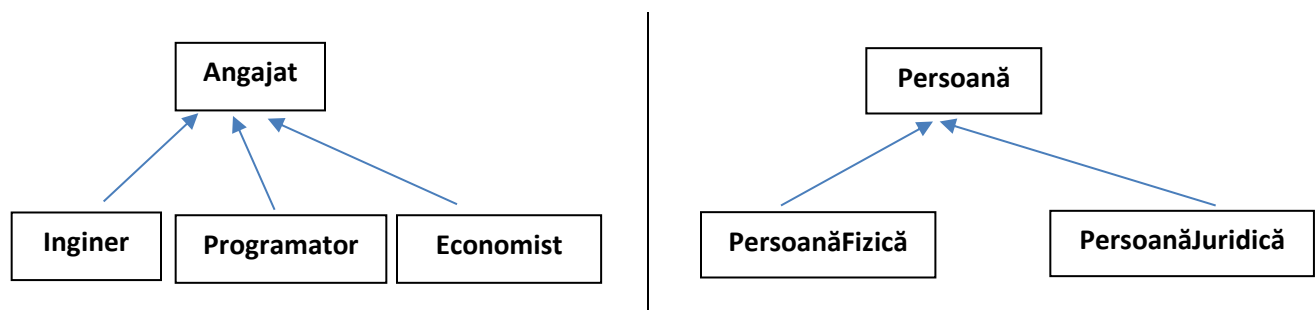
- **Sintaxă:**

```
public interface numeInterfață{
    constante    //public, static și final
    metode abstracte (fără implementare) //public și abstract
    metode implicite (default) cu implementare
    metode statice cu implementare
    metode private 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**.
- Interfețele definesc un set de operații (capabilități) comune mai multor clase care nu sunt înrudite (în sensul unei ierarhii de clase).

- **Exemple:**

Să presupunem că aveam următoarele ierarhii de clase:



După cum știți, una dintre operațiile des întâlnite în orice aplicație este cea de sortare (clasament, top etc.). În limbajul Java sunt definite metode generice (care nu țin cont de tipul elementelor) pentru a realiza sortarea unei structuri de obiecte, folosind un anumit criteriu de comparație (comparator). Astfel, într-o clasă se poate adăuga suplimentar un criteriu de comparație a obiectelor, sub forma unei metode (de exemplu, se poate realiza sortarea persoanelor juridice după cifra de afaceri, inginerii alfabetic după nume etc.). Cu alte cuvinte,

o interfață dedicată oferă o operație (capabilitate) de sortare, dar pentru a putea fi utilizată o clasă trebuie să specifice modalitatea de compararea a obiectelor.

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.

Un alt exemplu de operație pe care o poate realiza un obiect de tipul unei clase din ierarhiile de mai sus poate fi cel de plată online, folosind un cont bancar. Operația în sine poate fi realizată atât de către o categorie de angajați (de exemplu, programatori), cât și de persoane fizice sau juridice. Putem observa, din nou, cum o interfață care încapsulează operații specifice unei plăți online conține capabilități comune mai multor clase diferite conceptual (Angajat, PersoanăFizică etc.).

O interfață specifică unei plăți online poate să conțină următoarele operații:

- *autentificare* (pentru o persoană fizică se poate realiza folosind CNP-ul și o parolă, iar pentru o persoană juridică se poate folosi CUI-ul firmei și o parolă);
- *verificarea soldului curent*;
- *efectuarea unei plăți*.

```
public interface OperațiiContBancar{
    boolean autentificare();
    double soldCurent();
    void plată(double suma);
}
```

- 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 privită 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 manieră 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
```

- Se poate observa cum o clasă poate să implementeze mai multe interfețe în scopul de a dobândi mai multe capabilități. De exemplu, pentru o interfață grafică trebuie să tratăm atât evenimente generate de mouse, cât și evenimente generate de taste, deci vom implementa două interfețe: `MouseListener` și `KeyListener`.
- Revenind la exemplul anterior, clasa `Inginer` implementează interfața `Comparable`, oferind un criteriu de comparație (sortare alfabetică după nume):

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

- Pentru un tablou cu obiecte de tip `Inginer` se poate apela metoda statică `sort` din clasa utilitară `Arrays`:

```
Inginer tab[] = new Inginer[10];
.....
Arrays.sort(tab); //sortare naturală, metoda sort nu mai are nevoie de un alt
                  //argument pentru a specifica criteriul de sortare, ci se va utiliza
                  //implicit metoda compareTo implementată în clasa Inginer
```

- Clasa `Inginer` implementează interfața `OperatiiContBancar`, oferind implementări pentru toate cele trei metode abstracte:

```
class Inginer implements OperatiiContBancar{
    private String contBancar;
    .....
    public boolean autentificare(){
        //conectare la server-ul băncii pe baza CNP-ului și a unei parole
    }
    public double soldCurent(){
        //interogarea contului folosind API-ul server-ului băncii
    }
    void plată(double suma){
        //accesarea contului în scopul efectuării unei plăți folosind API-ul server-ului băncii
    }
}
```

- Clasa `PersoanăJuridică` implementează interfața `OperațiiContBancar`, oferind implementări pentru toate cele trei metode abstracte:

```
class PersoanăJuridică implements OperațiiContBancar{
    private String contBancar;
    .....
    public boolean autentificare() {
        // conectare la server-ul băncii utilizând CUI-ul firmei și o parolă
    }
    double soldCurent() {
        // interogarea contului folosind API-ul server-ului băncii
    }
    void plată(double suma) {
        //accesarea contului în scopul efectuării plății folosind API-ul server-ului băncii
    }
}
```

➤ **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 semnături diferite, clasa trebuie să implementeze ambele metode;
 - dacă metodele au aceeași semnătură și același tip pentru valoarea returnată, clasa implementează o singură metodă;
 - dacă metodele au aceeași semnătură, 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 acest caz, comportamentul obiectului este redus la cel oferit de interfață, alături de cel oferit de clasa `Object`:**

```
OperațiiContBancar p = new Inginer();
System.out.println("Sold curent: " + p.soldCurent());
```

➤ **În concluzie, 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`.

➤ Extinderea interfețelor

Să presupunem faptul că o interfață ce conține doar metode abstracte este implementată de mai multe clase $C1, C2, \dots, Cn$ etc. Fiecare clasă oferă o implementare pentru toate metodele abstracte din interfață implementată, astfel clasele $C1, C2, \dots, Cn$ pot fi instanțiate. Ulterior, dezvoltatorul dorește să mai introducă și alte funcționalități în interfață, respectiv alte metode abstracte. În acest caz, clasele $C1, C2, \dots, Cn$ devin abstracte și nu mai pot fi instanțiate!!! O soluție pentru a elimina acest neajuns, specifică până în versiunea Java 7, este aceea de a extinde interfața inițială și de a adăuga noile metode abstracte în subinterfața sa.

Sintaxa pentru extinderea interfețelor:

```
interface subInterfata extends superInterfata1, superInterfata2,
....superIntervatan
```

Exemplu: Să presupunem faptul că dorim să modificăm interfața `OperațiiContBancar` prin includerea unui nou serviciu, respectiv a metodei `void sendSMS(String message)` pentru a trimite un mesaj informativ de tip sms unui client ce folosește serviciul `OperațiiContBancar`. Pentru ca o serie de clase care implementează interfața `OperațiiContBancar` să nu fie afectate, se poate defini o subinterfața `OperatiiContBancarSMS` a superinterfeței `OperațiiContBancar`.

```
interface OperatiiContBancarSMS extends OperatiiContBancar {
    void sendSMS(String message);
}
```