INTERFEȚE

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

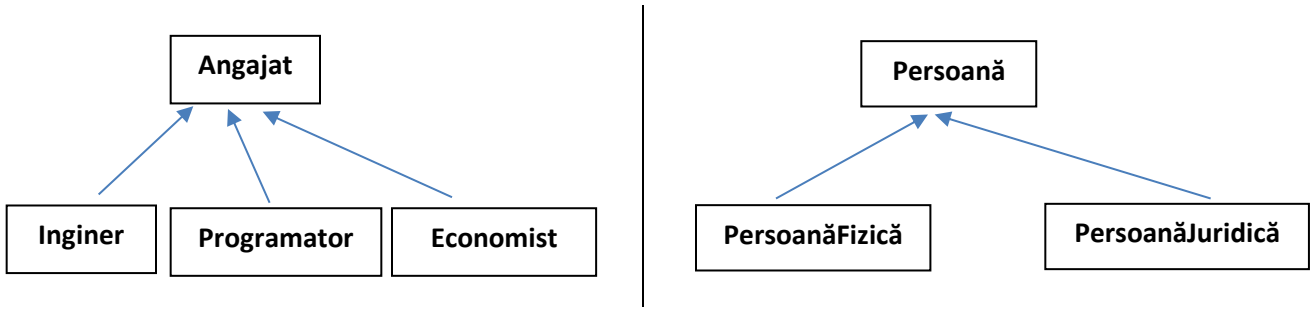
- **Sintaxă:**

```
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 plați 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 plați*.

```
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 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
```

- 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 OperațiiContBancar{
   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 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 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 interfața ce conține doar metode abstracte este implementată de mai multe clasa `C1, C2,..., Cn` etc. Fiecare clasă oferă o implementare pentru toate metodele abstracte din interfața implementată, astfel clasele `C1, C2,...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,...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 ca 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 implemntează interfața `OperațiiContBancar` să nu fie afectate, se poate defini o subinterfața `OperatiiContBancarSMS` a superinterfeței `OperațiiContBancar`.

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

Utilitatea interfețelor

1. **Definirea unor funcționalități ale unei clase**

   Așa cum am văzut mai sus, cu ajutorul interfețelor se pot defini funcționalități comune unor clase independente, care nu se află în aceeași ierarhie, fără a forța o legătura între ele (capabilități trans-ierarhice).

2. **Definirea unor grupuri de constante**

   O interfață poate fi utilizată și pentru definirea unor grupuri de constante. De exemplu, mai jos este definită o interfață care încapsulează o serie de constante matematice, utilizate în diferite expresii și formule de calcul. Clasa `TriunghiEchilateral` implementează interfața `ConstanteMatematice` în scopul de a folosi constanta `SQRT_3` (o aproximare a valorii $\sqrt{3}$) în formula de calcula a ariei unui triunghi echilateral:

```
public interface ConstanteMatematice{
    double PI = 3.141592653589793238 46;
    double SQRT_2 = 1.41421356237;
    double SQRT_3 = 1.73205080757;
    double LN_2 = 0.69314718056;
}

class TriunghiEchilateral implements ConstanteMatematice{
    double latura;

    public TriunghiEchilateral(double x){
        latura = x;
    }

    double Aria(){
        return latura*latura*ConstanteMatematice.SQRT_3/4;
    }
}
```

Totuși, metoda poate fi ineficientă, deoarece o clasă s-ar putea să folosească doar o constantă din interfața implementată sau un set redus de constante. Prin implementarea interfeței, clasa preia în semnătura sa toate constantele, ci nu doar pe acelea pe care le folosește. În acest sens, o metodă mai eficientă de încapsulare a unor constante este dată de utilizarea unei enumerări, concept pe care va fi prezenta într-un curs ulterior.

3. Implementarea mecanismului de callback

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

În limbajul Java nu putem transmite ca argument al unei funcții/metode un pointer către o altă metodă, așa cum este posibil în limbajele C/C++. Totuși, această facilitate, care este foarte utilă în diverse aplicații (de exemplu, în programarea generică), se poate realiza în limbajul Java folosind interfețele.

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);
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.

Exemplul 1: Să presupunem faptul că dorim să calculăm următoarele 3 sume:

$S_1 = 1 + 2 + \cdots + n$
$S_2 = 1^2 + 2^2 + \cdots + n^2$
$S_3 = [\text{tg}(1)] + [\text{tg}(2)] + \cdots + [\text{tg}(n)]$,

unde prin $[x]$ am notat partea întreagă a numărului real x.

Desigur, o soluție posibilă constă în implementarea a trei metode diferite care să returneze fiecare câte o sumă. Totuși, o soluție mai elegantă se poate implementa observând faptul că toate cele 3 sume sunt de forma următoare:

$$S_k = \sum_{i=1}^{n} f_k(i)$$

unde termenii generali sunt $f_1(i) = i, f_2(i) = i^2$ și $f_3(i) = [tg(i)]$.

Astfel, se poate implementa o metodă generică pentru calculul unei sume de această formă care să utilizeze mecanismul de callback pentru a primi ca parametru o referință spre termenul general al sumei.

Urmând pașii amintiți mai sus, se definește mai întâi o interfață care încapsulează funcția generică:

```
public interface FuncțieGenerică{
    int funcție(int x);
}
```

Într-o clasă utilitară, definim o metodă care să calculeze suma celor n termeni generici:

```
public class Suma{
    private Suma(){ //într-o clasă utilitară constructorul este privat!
    }

    public static int CalculeazăSuma(FuncțieGenerică fg , int n){
        int s = 0;

        for(int i = 1; i <= n; i++)
            s = s + fg.funcție(i);

        return s;
    }
```

```
}
```

Ulterior, definim clase care implementează interfața respectivă, oferind implementări concrete ale funcției generice:

```
public class TermenGeneral_1 implements FuncțieGenerică{
    @Override
    public int funcție(int x){
        return x;
    }
}

public class TermenGeneral_2 implements FuncțieGenerică{
    @Override
    public int funcție(int x){
        return x * x;
    }
}
```

La apel, metoda `CalculeazăSuma` va primi o referință de tipul interfeței, dar spre un obiect de tipul clasei care implementează interfața:

```
public class Test_callback {
  public static void main(String[] args) {
       FuncțieGenerică tgen_1 = new TermenGeneral_1();
       int S_1 = Suma.CalculeazăSuma(tgen_1, 10);
       System.out.println("Suma 1: " + S_1);

       //putem utiliza direct un obiect anonim
       int S_2 = Suma.CalculeazăSuma(new TermenGeneral_2(), 10);
       System.out.println("Suma 2: " + S_2);

       //putem utiliza o clasă anonimă
       int S_3 = Suma.CalculeazăSuma(new FuncțieGenerică() {
                   public int funcție(int x) {
                       return (int) Math.tan(x);
                   }
               }, 10);
       System.out.println("Suma 3: " + S_3);
  }
 }
```

Exemplul 2: Mai sus, am văzut cum sortarea unor obiecte se poate realiza implementând interfața `java.lang.Comparable` în cadrul clasei respective, obținând astfel un singur criteriu de comparație care asigură sortarea naturală a obiectelor. Dacă aplicația necesită mai multe sortări, bazate pe criterii de comparație diferite, atunci se poate utiliza interfața `java.lang.Comparator` și mecanismul de callback.

De exemplu, pentru a sorta descrescător după vârstă obiecte de tip `Inginer` memorate într-un tablou `t`, vom defini următorul comparator:

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

La apel, metoda statică `sort` a clasei utilitare `Arrays` va primi ca parametru un obiect al clasei `ComparatorVârste` sub forma unei referințe de tipul interfeței `Comparator`:

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