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, iar mașina virtuală va fi responsabilă de implementarea funcționalității respective. Practic, interfețele marker au rolul de a asocia metadate unei clase, pe care mașina virtuală să le folosească la rulare într-un anumit scop.

- 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.

Interfața java.lang.Cloneable

O clasă care implementează interfața `Clonable` permite apelul metodei `Object.clone()` pentru instanțele sale, în scopul de a realiza o copie a lor. Interfața în sine nu conține nicio metodă, fiind interfața marker, ci doar anunță mașina virtuală Java faptul că instanțele clasei care o implementează au funcționalitatea de clonare.

Prin convenție, o clasă care implementează interfața `Cloneable`, redefinește metoda `Object.clone()` (care are acces protejat) printr-o metodă cu acces public.

Exemplu: Considerăm clasa `Angajat`

```
public class Angajat {
     private String nume;
     private int varsta;
     private double salariu;

     public Angajat(String nume, int varsta, double salariu) {
          this.nume = nume;
          this.varsta = varsta;
          this.salariu = salariu;
     }

     //metode get, set, toString()

     //redefinirea metodei clone din clasa Object
     @Override
     public Object clone() throws CloneNotSupportedException
     {
          return super.clone();
     }
}
```

```
public class Clona {
  public static void main(String[] args) throws
                                CloneNotSupportedException {

      Angajat a1 = new Angajat("Matei", 23, 4675.67);

      Angajat a2 = (Angajat)a1.clone();
      }
}
```

Apelul metodei `clone()` pentru obiectul `a1` conduce, în momentul executării, la apariția excepției `java.lang.CloneNotSupportedException`, deoarece clasa `Angajat` nu implementează interfața marker `Cloneable!!!`

```
public class Angajat implements Cloneable{
......
}
```

Observație: Clonarea unui obiect presupune, în sine, copierea acestuia la o alta adresă HEAP alocată pentru obiectul destinație. Însă, în cazul în care obiectul conține o referință către alt obiect (agregare/compoziție), redefinirea metodei `clone()` nu alocă implicit o noua adresă HEAP pentru obiectul încapsulat. Să presupunem faptul că se specifică pentru fiecare `Angajat` și departamentul în care acesta activează. Considerăm astfel clasa `Departament`:

```
public class Departament {
       private int id;
       private String denumire;

       public Departament(int id, String denumire) {
             this.id = id;
             this.denumire = denumire;
       }
       //metode set, get și toString()
}
```

Modificăm clasa `Angajat`, adăugând câmpul `departament`:

```
public class Angajat implements Cloneable{
     private String nume;
     private int  varsta;
     private double salariu;
     private Departament departament;

     public Angajat(String nume, int varsta, double salariu,
                                        Departament departament) {
          this.nume = nume;
          this.varsta = varsta;
```

```
            this.salariu = salariu;
            this.departament = departament;
      }
      //metode get, set

      @Override
      public Object clone() throws CloneNotSupportedException {
            return super.clone();
      }
}

public class TestClona {
      public static void main(String[] args)
                                          throws CloneNotSupportedException {
            Departament departament = new Departament(1, "HR");
            Angajat a1 = new Angajat("Matei", 23, 4675.67, departament);

            Angajat a2 = (Angajat)a1.clone();

            a2.getDepartament().setDenumire("Finante");

            System.out.println(a1.getDepartament() + " " +
                                                a2.getDepartament());

      }
}
```

Clonarea obiectului `a1` s-a realizat cu succes, însă setarea câmpului `departament` pentru obiectul `a2` conduce și la modificarea câmpului `departament` pentru obiectul `a1`, deoarece metoda `clone()` din clasa `Angajat` realizează doar o clonă a referinței de tip `Departament`! S-a realizat astfel ceea ce poartă denumirea de *shallow cloning***.** Pentru a evita ca ambele câmpuri de tip `Departament` să aibă aceeași referință, se creează o clonă care este independentă de obiectul original, astfel încât orice modificarea a clonei să nu conducă și la modificarea obiectului original. Practic, se redefinește metoda `clone()` și în clasa `Departament`:

```
public class Departament implements Cloneable{
...........
@Override
    public Object clone() throws CloneNotSupportedException {
            return super.clone();
    }
}

public class Angajat implements Cloneable{

@Override
    public Object clone() throws CloneNotSupportedException {
     ..........................
```

```
    Angajat clona = (Angajat)super.clone();

    clona.setDepartament((Departament)clona.getDepartament().clone());
    return clona;
   }
}
```

CLASE ADAPTOR

- O interfață poate să conțină multe metode abstracte. De exemplu, interfața `MouseListener` conține 8 metode asociate unor evenimente produse de mouse (`mousePressed()`, `mouseReleased()` etc.). O clasă care implementează o astfel de interfață, evident, trebuie sa ofere implementare pentru toate metodele abstracte. Totuși, de cele mai mult ori, în practică o clasă va folosi un set restrâns de metode dintre cele specificate în interfață. De exemplu, din interfața `MouseListener` se folosește, de obicei, metoda asociată evenimentului `mouseClicked()`.
- O soluție pentru această problemă o constituie definirea unei *clase adaptor*, respectiv o clasă care să implementeze minimal (cod vid) toate metodele din interfață. Astfel, dacă o clasă dorește să implementeze doar câteva metode din interfață, poate să prefere extinderea clasei adaptor, redefinind doar metodele necesare.

ÎMBUNĂTĂȚIRI ADUSE INTERFEȚELOR ÎN JAVA 8 ȘI JAVA 9

- Un dezavantaj major al interfețelor specifice versiunilor anterioare Java 8 îl constituie faptul că modificarea unei interfețe necesită modificarea tuturor claselor care o implementează. O soluție posibilă ar fi aceea de a extinde interfața respectivă și de a încapsula în sub-interfață metodele suplimentare. Totuși, această soluție nu conduce la o utilizare imediată sau implicită a interfeței nou create. Astfel, pentru a elimina acest neajuns, începând cu versiunea Java 8 o interfață poate să conțină și metode cu implementări implicite (*default*) sau metode statice cu implementare.

```
interface numeInterfață{
   ........................
   default tipRezultat metodăImplicită(...){
        //implementare implicită
     }

   static tipRezultat metodăStatică(...){
        //implementare
   }
}
```

- În acest fel, 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.
- În plus, 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 în cadrul unei clase care implementează interfața sau, direct, prin numele interfeței. De regulă, o metodă statică este una de tip utilitar.

Exemplu: Considerăm interfața `InterfațăAfișareȘir` în care definim o metodă default `afișeazăȘir` pentru afișarea unui șir de caractere sau a unui mesaj corespunzător dacă șirul este vid. Verificarea faptului că un șir este vid se realizează folosind metoda statică (utilitară) `esteȘirVid`, deoarece nu considerăm necesar ca această metodă să fie preluată în clasele care vor implementa interfața.

```
public interface InterfațaAfișareȘir {
    default void afișeazăȘir(String str) {
        if (!esteȘirVid(str))
            System.out.println("Sirul: " + str);
        else
            System.out.println("Sirul este vid!");
    }
    static boolean esteȘirVid(String str) {
        System.out.println("Metoda esteȘirVid din interfață!");
        return str == null ? true : (str.equals("") ? true : false);
    }
}

public class ClasaAfișareȘir implements InterfațaAfișareȘir {
    //@Override -> nu se poate utiliza adnotarea deoarece metoda este statică și nu se preia din interfață
    public static boolean esteȘirVid(String str) {
        System.out.println("Metoda esteȘirVid din clasă!");
        return str.length() == 0;
    }
}

public class Test {
    public static void main(String args[]) {
        ClasaAfișareȘir c = new ClasaAfișareȘir();
        c.afișeazăȘir("exemplu");
        c.afișeazăȘir(null);

        //System.out.println(InterfațaAfișareȘir.esteȘirVid(null));
        //System.out.println(ClasaAfișareȘir.esteȘirVid(null));
    }
}
```

Dacă vom elimina comentariile din metoda `main` și vom rula programul, va apărea o eroare în momentul apelării metodei `esteȘirVid` din clasă. De ce?

- **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 => sub-interfața va moșteni metoda default din super-interfață;
  - sub-interfața conține o metodă abstractă cu același nume => metoda redevine abstractă în sub-interfață (i.e., metoda 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 (nu neapărat tot de tip default) și, 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 rescrisă 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 implementa metoda default, eventual apelând una dintre metodele default printr-o construcție sintactică de forma `Interfață.super.metoda()`.

  Exemplu: Considerăm două interfețe `Poet` și `Scriitor`:

```
interface Poet {
    default void scrie(){
        System.out.println("Metoda default din interfața Poet!");
    }
}

interface Scriitor {
    default void scrie(){
        System.out.println("Metoda default din interfața Scriitor!");
    }
}
```

```
class Multitalent implements Poet, Scriitor {
    public static void main(String args[]){
        Multitalent autor = new Multitalent();
        autor.scrie();
    }
}
```

Apelul metodei `scrie()` pentru obiectul `autor` conduce la apariția unei erori la compilare, respectiv:

```
class Multitalent inherits unrelated defaults for scrie() from types
Poet and Scriitor
```

Pentru a elimina ambiguitatea cauzată de implementarea celor doua interfețe care conțin metode cu signatura identică, clasa trebuie să redefinească metoda respectivă:

```
public class Multitalent implements Poet, Writer {
     @Override
     public void scrie()
     {
          System.out.println("Metoda din clasa Multitalent!");
     }
}
```

- **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.

- Principala utilitate a metodelor private este următoarea: dacă mai multe metode default conțin o porțiune de cod comun, atunci aceasta poate fi mutată într-o metodă privată și apoi apelată din metodele default. Astfel, o metodă privată nu este accesibilă din afara interfeței (chiar dacă este statică), nu este necesară implementarea sa în clasele care vor implementa interfața și nici nu va fi preluată implicit (deoarece nu este default) .

Exemplu: Considerăm următoarea implementare specifică versiunii Java 8:

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

Un dezavantaj evident este faptul că o secvență de cod este repetată în mai multe metode. O variantă de rezolvare ar putea fi încapsularea codului comun într-o metoda default:

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

    default void calculComplex_2(…) {
          codComun(…);
          Cod specific 2
    }

    default void codComun(…) {
          Cod comun
    }
}
```

Totuși, în acest caz metoda default care încapsulează codul comun va fi moștenită de către toate clasele care vor implementa interfața respectivă. Soluția oferită în Java 9 constă în posibilitatea de a încapsula codul comun într-o metoda privată (statică sau nu). Astfel, metoda privată nu va fi moștenită de către clasele care implementează interfața:

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

    default void calculComplex_2(…) {
          codComun(…);
          Cod specific 2
    }

    private void codComun(…) {
          Cod comun
    }
}
```

CONTROLUL MOȘTENIRII/IMPLEMENTĂRII ÎN JAVA

În versiunile anterioare versiunii Java 17, lansată în septembrie 2021, nu exista niciun mecanism prin care să poată fi controlată într-un mod detaliat extinderea unei clase. Astfel, extinderea unei clase putea fi controlată fie prin intermediul specificatorului `final` (restricționând-o astfel complet), fie declarând clasa accesibilă doar la nivel de pachet (i.e., specificatorul implicit de acces), ceea ce permite extinderea sa doar de către subclase aflate în același pachet (dar restricționează și accesarea sa din exteriorul pachetului!).

În versiunea Java 17 a fost introdus conceptul de *clasă*/*interfață sealed* care permite un control detaliat al moștenirii prin precizarea explicită a subclaselor/subinterfețelor care pot extinde o clasă/interfață (în cazul interfețelor și a claselor care o pot implementa).

O clasă *sealed* se declară astfel:

```
[specificatori] sealed class Clasă permits Subclase {
        ......................
}
```

Dacă o clasă extinde o altă clasă și/sau implementează anumite interfețe, atunci cuvântul `permits` și lista subclaselor care pot să implementeze clasa respectivă se vor scrie la sfârșitul antetului său, așa cum se poate observa din următorul exemplu:

```
public sealed class Angajat extends Persoana implements Comparable
        permits Economist, Paznic, Inginer {
        ......................
}
```

Declarările unei clase sealed și ale subclaselor permise trebuie să respecte următoarele reguli:

- clasa sealed și subclasele permise trebuie să facă parte din același modul sau, dacă sunt declarate într-un modul anonim, din același pachet;
- fiecare subclasă permisă trebuie să extindă direct clasa sealed;
- fiecare subclasă permisă trebuie să specifice în mod explicit modul în care va continua controlul moștenirii inițiat de superclasa sa, folosind exact unul dintre următorii modificatori:
  - `final`: subclasa respectivă nu mai poate fi extinsă;
  - `sealed`: subclasa respectivă poate fi extinsă doar în mod controlat (i.e., doar de subclasele pe care le permite explicit);

- `non-sealed`: subclasa respectivă poate fi extinsă fără nicio restricție (i.e., de orice altă clasă), deci o clasă sealed nu poate obliga subclasele sale să restricționeze ulterior moștenirea.

Pentru clasa `Angajat` din exemplul se mai sus, o variantă de declarare a subclaselor poate fi următoarea:

```
public final class Paznic extends Angajat {
	.....................
}

public non-sealed class Economist extends Angajat {
	.....................
}

public sealed class Inginer extends Angajat
	permits InginerElectronist, InginerMecanic {
	.....................
}
```

O clasă sealed nu poate conține în lista subclaselor permise clase de tip record, deoarece acestea nu pot extinde o altă clasă (i.e., orice clasă de tip record extinde în mod implicit clasa `java.lang.Record`), iar unei clase de tip record nu îi putem aplica modificatorul `sealed` deoarece clasele de acest tip nu pot fi extinse (i.e., sunt implicit de tip `final`).

În cazul unei interfețe, folosind modificatorul `sealed`, putem specifica subinterfețele care o pot extinde sau clasele care o pot implementa, astfel:

```
[public] sealed interface Interfață permits Subinterfețe, Clase {
	.....................
}
```

O subinterfață care extinde o interfață sealed trebuie să respecte reguli asemănătoare celor precizate în cazul claselor sealed, cu observația că unei interfață îi putem aplica doar modificatorii `sealed` și `non-sealed`. În lista claselor care pot implementa o interfață putem preciza și clase de tip record, cu observația că acestea vor fi implicit de tip `final`, deci nu putem să ele aplicăm modificatorii `sealed` și `non-sealed`.

În concluzie, folosind mecanismul de control al moștenirii/implementării, un programator poate crea ierarhii care modelează într-un mod complet și sigur un anumit concept.

ENUMERĂRI

O **enumerare** este un tip special de clasă care poate încapsula o serie de constante. Enumerările au fost adăugate în limbajul Java începând cu versiunea Java 5.

O enumerare se declară folosind următoarea sintaxă:

```
public enum DenumireaEnumerării {
      instanțele enumerării (constantele propriu-zise)
      [câmpuri private care reține valorile unei constante]
      [constructor privat care valorile asociate unei constante]
      [metode care furnizează valorile asociate unei constante]
}
```

De exemplu, putem să definim o enumerare care să conțină constante asociate celor 4 puncte cardinale astfel:

```
public enum PuncteCardinale {
    NORD, EST, SUD, VEST;
}
```

Observați faptul că numele constantelor sunt scrise cu litere mari, respectând astfel o regulă de bună practică în orice limbaj de programare!

Accesarea unei constante se realizează într-o manieră statică, prin numele său:

```
PuncteCardinale p = PuncteCardinale.NORD;
```

Observați faptul că un obiect de tip enumerare nu trebuie instanțiat folosind operatorul new!

Dacă dorim să asociem anumite valori constantelor dintr-o enumerare, atunci trebuie să declarăm în enumerarea respectivă date membre corespunzătoare (de obicei, de tip final și private), un constructor privat care să le inițializeze și, eventual, metode publice de tip get:

```
public enum PuncteCardinale{
    NORD(1, 'N'), EST(2, 'E'), SUD(3, 'S'), VEST(4, 'V');

    private final int valoare;
    private final char simbol;

    private PuncteCardinale(int valoare, char simbol) {
        this.valoare = valoare;
        this.simbol = simbol;
    }

    public int getValoare() {
        return valoare;
    }
```

```
    public char getSimbol() {
        return simbol;
    }
}
```

Orice enumerare este implicit o clasă de tip final care extinde clasa java.lang.Enum, deci o enumerare nu mai poate extinde nicio altă clasă și nici nu mai poate fi extinsă!

Clasa java.lang.Enum conține o serie de metode care vor fi moștenite implicit de orice enumerare:

- **public final** String name(): furnizează numele unei constante sub forma unui șir de caractere;
- **public final int** ordinal(): furnizează numărul de ordine al unei constante în cadrul enumerării.

Pentru orice enumerare, compilatorul va genera automat o metodă statică denumită values() care va furniza toate constantele din enumerarea respectivă sub forma unui tablou unidimensional:

```
for(PuncteCardinale pc: PuncteCardinale.values())
      System.out.println("Constanta "+pc.name()+" are indexul "+pc.ordinal());
```

De asemenea, clasa java.lang.Enum conține și metode redefinite din clasa Object:

- **public final boolean** equals(Object other)
- **public final int** hashCode()
- **public** String toString()

Metoda toString() poate fi redefinită pentru a furniza o descriere mai amănunțită a unei constante:

```
@Override
public String toString() {
      return "Constanta " + name() + " are valoarea " + valoare +
             " si simbolul " + simbol;
}
```

O enumerare poate să încapsuleze metode de instanță (vezi metodele de tip get din exemplul de mai sus) și metode statice. De exemplu, următoarea metodă statică va furniza constanta având asociată o valoare x sau null dacă nu există nicio constantă cu valoarea respectivă:

```
public static PuncteCardinale getPunctCardinalValoare(int x) {
        for (PuncteCardinale pc : PuncteCardinale.values())
            if (x == pc.getValoare())
                return pc;
        return null;
}
```

O enumerare poate să încapsuleze o metodă abstractă, caz în care fiecare instanță a enumerării (i.e., fiecare constantă) trebuie să implementeze metoda respectivă. De exemplu, metoda abstractă getSuccesor() va furniza succesorul fiecărui punct cardinal, în sensul acelor de ceasornic:

```
public enum PuncteCardinale {
    NORD(1, 'N') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.EST;
        }
    },
    EST(2, 'E') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.SUD;
        }
    },
    SUD(3, 'S') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.VEST;
        }
    },
    VEST(4, 'V') {
        @Override
        public PuncteCardinale getSuccesor() {
            return PuncteCardinale.NORD;
        }
    };

    private final int valoare;
    private final char simbol;

    private PuncteCardinale(int valoare, char simbol) {
        this.valoare = valoare;
        this.simbol = simbol;
    }

    public int getValoare() {
        return valoare;
    }

    public char getSimbol() {
        return simbol;
    }

    public abstract PuncteCardinale getSuccesor();
}
```

În încheiere, precizăm faptul că enumerările sunt foarte utile pentru a modela liste finite de constante (e.g., opțiunile dintr-un meniu, stările în care se poate afla o mașină sau un robot, denumirile monedelor acceptate pentru plată de către un magazin etc.). De asemenea, o enumerare cu o singură constantă reprezintă o modalitate foarte simplă de implementare a unei clase singleton (Java Singletons Using Enum - DZone Java).