**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;**

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.
* 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:

|  |  |
| --- | --- |
| **Inginer**  **Programator**  **Angajat**  **Economist** | **PersoanăFizică**  **Persoană**  **PersoanăJuridică** |

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).

1. **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 ) în formula de calcula a ariei unui triunghi echilateral:

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;

}

}

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.

1. **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:

,

unde prin am notat partea întreagă a numărului real .

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:

unde termenii generali sunt și .

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 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

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:

publicclass 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 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, 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 intefața 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 anuntă mașina virtuală Java faptul că instanțele clasei ce 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 la apariția în mometul executarii a 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 referintă 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, insă 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 referinței de tip Departament! S-a realizat astfel ceea ce poartă denumirea de **shallow cloning.** Pentru a evita ca cele două câmpuri de tip Departament să aibă aceeași referintă, se crează o clonă care este independentă de obiectul original, astfel încăt orice modificarea a clonei să nu conducă și la modificarea obiectului original. Practic, se redefinește metoda clone() și in 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 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));

## }

## }

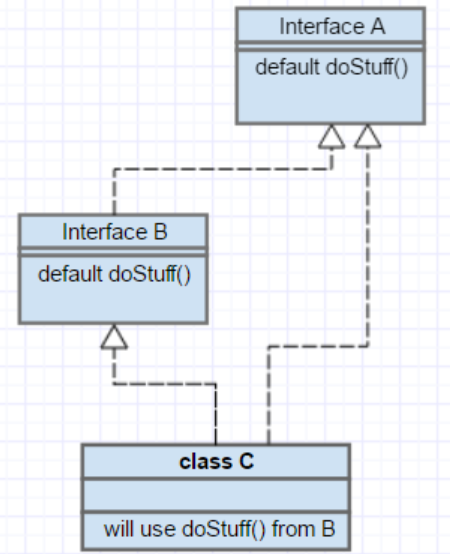
Dacă vom elimina comentariile din metoda main și vom rula programul, va apărea o eroare în momentul apelării metodei esteȘirVid din clasă. De ce?

* **Extinderea interfețelor care conțin metode default**

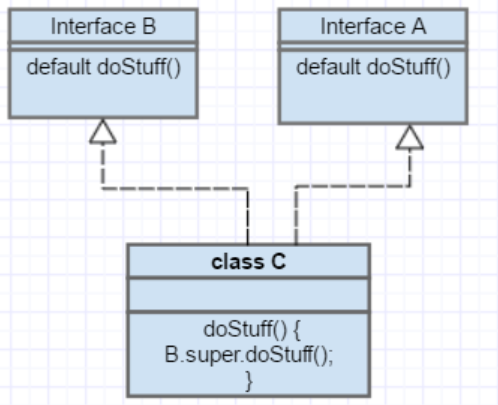
În momentul extinderii unei interfețe care conține o metodă default pot să apară următoarele situații:

* sub-interfața nu are nicio metodă cu același nume => 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 (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).



1. 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().



Exemplu: Considerăm două interfețe Poet și Writter

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();

}}

Apelul metodei write() pentru obiectul ob conduce la apariția unei erori la compliare, respectiv:

class Multitalented inherits unrelated defaults for write() from typesPoet and Writer  
  
Pentru a elimina ambiguitatea cauzată de implementarea celor doua interfețe ce conțin metode cu signatura identică, clasa trebuie să redefinească metoda respectivă.

public class Multitalented implements Poet, Writer

{

**@Override**

public void write()

{

System.out.println("Writing stories now days");

}  
  
}

* **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**

**}**

}