Programare orientată pe obiecte

1. Clasele Object și Class
2. Interfețe Java

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA
Computer Science

Metode din clasa `Object`

- *Object* defineşte versiuni implicite ale următoarelor metode:
  - `toString()` – returnează un `String` (reprezentare "citibilă" a obiectului)
  - `equals(Object obj)` – returnează egalitatea referințelor; trebuie să fie suprascrisă pentru a realiza egalitatea de conţinut în subclase
  - `hashCode()` – returnează valoarea codului de dispersie pentru obiect; valorile sunt diferite pentru obiecte diferite
  - `getClass()` – returnează un obiect de tipul Class; există un obiect de tipul Class pentru fiecare clasă dintr-o aplicaţie
  - `notify()`, `notifyAll()`, `wait()`, `wait(long timeout)`, `wait(long timeout, int nanos)` – folosite la *multithreading*
  - `clone()` – creează şi întoarce o copie a acestui obiect; "copie" poate depinde de clasa obiectului
  - `finalize()` – destinat a efectua acţiuni de "curăţare" înainte ca obiectul să fie irevocabil abandonat

Egalitatea

- Există două feluri diferite de egalitate:
  - *Egalitatea de identitate* care înseamnă că două expresii au aceeaşi identitate (adică reprezintă acelaşi obiect)
    - Simbolul == testează egalitatea de identitate atunci când este aplicat datelor referinţă
  - *Egalitatea de conţinut* care înseamnă că două expresii reprezintă obiecte cu aceeaşi valoare/conţinut
    - Metoda *equals* din `Object` returnează true dacă și numai dacă este invocată cu două referințe identice; metoda poate fi suprascrisă în orice subclasă pentru a verifica egalitatea de conţinut

- Exemplu:

```
AOval ov1, ov2;
ov1 = new AOval(0, 0, 100, 100);
ov2 = new AOval(0, 0, 100, 100);
if (ov1 == ov2){ System.out.println("Egalitate de identitate");}
if (ov1.equals(ov2)){ System.out.println("Egalitate de continut");}
```

Clasa `Class`

- Clasa `Class` este definită astfel:

```
public final class Class extends Object
    implements Serializable, …
```

- Instanţele clasei `Class` reprezintă clase şi interfeţe dintr-o aplicaţie Java în curs de execuţie
- Un obiect de tipul `Class` conţine informaţii despre clasa a cărei instanţă este obiectul care apelează
- Nu are constructor propriu
- Obiectele `Class` sunt construite la execuţie de către JVM
- Există două moduri pentru a construi obiecte de acest tip:
  - `getClass()` din clasa `Object`
  - `forName()` din clasa `Class` (metodă statică)

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Clasa `Class`

- Metode:
  - `public String getName()`
    - returnează un `String` care reprezintă numele entităţii reprezentate de obiectul `Class` `this`
    - entitatea poate fi: clasă, interfaţă, tablou, tip primitiv, void
  - `public static Class forName(String className) throws ClassNotFoundException`
    - returnează un obiect de tipul `Class` care conţine informaţii despre clasa obiectului
  - `public Class[] getClasses()`
    - returnează un tablou de obiecte de tip `Class`;
    - toate clasele şi interfeţele, membri publici ai clasei sunt reprezentate de acest obiect `Class`

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Clasa `Class`

- Metode (continuare)
  - **`Field[] getFields`**
    - returnează un tablou care conţine obiecte **`Field`** care reflectă toate câmpurile accesibile public ale clasei sau interfeţei reprezentate de acest obiect **`Class`**
  - **`Method[] getMethods()`**
    - returnează un tablou care conţine obiecte **`Method`** care reflectă toate metodele publice *membre* ale clasei sau interfeţei reprezentate de acest obiect **`Class`**, inclusiv cele declarate de clasă sau interfaţă şi cele moştenite din superclase şi superinterfeţe
  - **`Constructor[] getConstructors()`**
    - returnează un tablou care conţine obiecte **`Constructor`** care reflectă toţi constructorii publici ai clasei sau interfeţei reprezentate de acest obiect **`Class`**

Operatorul `instanceof`

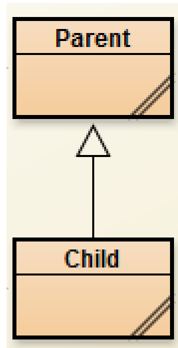
- Operatorul **`instanceof`** verifică dacă un obiect este de tipul dat ca al doilea argument al său

  `Obiect instanceof NumeClasa`

  - Va returna **`true`** dacă **`Obiect`** este de tipul **`NumeClasa`**; altfel va returna **`false`**
  - Observaţi că aceasta înseamnă că va returna **`true`** dacă **`Obiect`** are tipul *oricărei clase care este descendentă* a lui **`NumeClasa`**

  Exemplu:

```
Child c = new Child("Ana");
System.out.print(c instanceof Child); //true
System.out.print(c instanceof Parent); //true
```



Metoda getClass()

- Fiecare obiect moşteneşte aceeaşi metodă **getClass()** din clasa **Object**
  - Această metodă este marcată **final**, deci nu poate fi suprascrisă

- O invocare a lui **getClass()** pe un obiect returnează o reprezentare *numai* pentru clasa care a fost folosită cu operatorul **new** pentru a crea obiectul
  - Ex.
    ```
    Parent p = new Child("Bubu");
    System.out.print(p.getClass()); // Child
    ```
  - Rezultatele a oricare două asemenea invocări pot fi comparate cu **==** sau **!=** pentru a determina dacă ele reprezintă sau nu aceeaşi clasă

```
(obiect1.getClass() == obiect2.getClass())
```

`instanceof` şi `getClass()`

- Atât operatorul `instanceof` cât şi metoda `getClass()` se pot folosi pentru a verifica clasa unui obiect
- Totuşi, metoda `getClass()` este mai exactă
  - Operatorul `instanceof` doar testează clasa unui obiect
  - Metoda `getClass()` folosită într-un test cu `==` or `!=` testează dacă două obiecte *au fost instanţiate* din aceeaşi clasă

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Exemple

- Afişarea numelui unei clase folosind un obiect de tip **`Class`**

```
void printClassName(Object obj) {
   System.out.println(obj + " este de clasa " +
                      obj.getClass().getName());
}
```

- Alte exemple

```
Circle c = new Circle(5);
printClassName(c); //Cercul cu raza 5 este de clasa Circle
Class c1 = c.getClass();
System.out.println(c1.getName()); // "Circle"
Triangle t = new Triangle(7);
printClassName(t); //Triunghiul cu laturile 7 este de clasa Triangle
try {
   Class c2 = Class.forName("Triangle");
   System.out.println(c2.getName()); // "Triangle"
}
catch (ClassNotFoundException e) {
    System.err.println("Nu exista clasa \"Triangle\"" +
                       e.getMessage());
}
```

Specificaţiile şi Java

- Un program este asamblat dintr-o colecţie de clase care trebuie să "lucreze împreună" sau să "se potrivească una cu alta"
- Limbajul şi compilatorul Java ne pot ajuta să:
  - Scriem specificaţii de clase şi să
  - Verificăm că o clasă satisface corect (implementează) specificaţiile sale
- Există mai multe construcţii în Java:
  - Construcţia `interface` – ne permite să codificăm în Java informaţia pe care o specificăm, d.e. într-o diagramă de clasă
  - Construcţia `extends` – ne permite să codificăm o clasă prin adăugarea de metode la o clasa existentă
  - Construcţia `abstract class` – ne permite să codificăm o clasă incompletă care poate fi încheiată (terminată) printr-o altă clasă

Un exemplu

- Două persoane lucrează la acelaşi proiect, în acelaşi timp:
  - o persoană modelează un cont bancar
  - o alta scrie o clasă pentru plăţi lunare din cont
- Pentru a realiza acest lucru, cei doi trebuie să se înţeleagă cu privire la *interfaţă,* de exemplu:

```
/** SpecificatieContBancar specifica modul de comportare al contului bancar. */
public interface SpecificatieContBancar
{
    /** depune adauga bani in cont
    * @param suma - suma de bani de depus, un intreg nenegativ */
    public void depune(int suma);

    /** retrage scoate bani din cont daca se poate
    * @param suma - suma de retras, un intreg nenegativ
    * @return true, daca retragerea a avut succes;
    * return false, in caz contrar. */
    public boolean retrage(int suma);
}
```

Ce este o interfaţă?

- *Interfaţa* spune că, indiferent de clasa scrisă pentru a implementa o **`SpecificatieContBancar`**, clasa respectivă trebuie să conţină două metode, **`depune`** şi **`retrage`**, care să se comporte aşa cum s-a precizat
- În general, o *interfaţă* este un dispozitiv sau un sistem pe care entităţi ne-înrudite îl folosesc pentru a interacţiona
- Exemple:
  - O telecomandă reprezintă interfaţa dintre persoană şi televizor,
  - Limba română este o interfaţă între două persoane care o vorbesc
  - Protocolul de comportament din armată reprezintă interfaţa dintre indivizii de diferite grade

Ce este o interfaţă?

- În Java: o *interfaţă* este un tip, aşa cum şi o clasă este un tip
  - *Asemănător* unei clase, o interfaţă *defineşte metode*
  - *Spre deosebire* de o clasă, o interfaţă *nu implementează niciodată metode (valabil în Java 7 sau versiuni mai vechi); totuși începând cu Java 8 sunt permise și implementări implicite (default) de metode*
  - Clasele care implementează interfaţa implementează metodele definite de interfaţă
  - O clasă poate implementa mai multe interfeţe
- O interfaţă se foloseşte pentru a defini un *protocol de comportament* care poate fi implementat de către orice clasă de oriunde din ierarhia de clase

Definiţie. Utilitate

- Definiţie: *o interfaţă este o colecţie de constante şi definiţii de metode, colecţie care are un nume*
- O interfaţă nu este o clasă, ci un set de *cerinţe* pentru clasele care doresc să se conformeze interfeţei
- Interfeţele sunt folositoare pentru
  - *Reţinerea asemănărilor* între *clase ne-înrudite* fără a forţa o relaţie de clasă
  - *Declararea de metode* pe care una sau mai multe clase ar trebui să le implementeze
  - *Dezvăluirea interfeţei de programare* a unui obiect fără a-i dezvălui clasa
  - *Modelarea moştenirii multiple*, care permite ca o clasă să aibă mai mult de o superclasă

Definirea unei interfeţe

- Definirea unei interfeţe are două componente: declaraţia interfeţei şi corpul interfeţei
  - *Declaraţia* interfeţei defineşte diferitele atribute ale interfeţei, cum sunt numele şi dacă ea extinde alte interfeţe
  - *Corpul* interfeţei conţine declaraţiile de constante şi de metode pentru interfaţa respectivă

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Definirea unei interfeţe

- **Sintaxa:**

```
[modificator acces] interface NumeInterfata [extends AltaInterfata]{
        /* zona declarare constante */
        public static final int X = 10;

        /*zona declarare metode*/
        public void numeMetoda();
}
```

- **Implementarea unei interfețe:**

```
class NumeClasa implements NumeInterfata{
        public void numeMetoda(){
            // codul necesar pentru implementarea metodei
        }
}
```

Exemplu:

<<interface>>
SpecificatieContBancar
+depune(suma: Integer)
+retrage(suma: Integer)

<<class>>
CalculatorIpoteca
+faPlataIpoteca(suma: Integer)

<<class>>
ContBancar
+depune(suma: Integer)
+retrage(suma: Integer)
+obtineSold(): Integer

- Pentru a conecta cele două clase – scrieţi o metodă de lansare cam aşa:

```
ContBancar contulMeu = new contBancar();
CalculatorIpoteca calc = new CalculatorIpoteca(contulMeu);
. . .
calc.faPlataIpoteca(500);
```

O clasă care implementează o interfaţă

```java
/** ContBancar gestioneaza un singur cont bancar; cum este precizat
* in antetul sau, el implementeaza SpecificatieContBancar: */
public class ContBancar implements SpecificatieContBancar{
    private int sold; // soldul contului

    /** Constructor ContBancar initializeaza contul */
    public ContBancar() { sold = 0; }

    // implementarea metodelor din interfata SpecificatieContBancar:
    public void depune(int suma) { sold = sold + suma; }
    public boolean retrage(int suma) {
        boolean rezultat = false;
        if ( suma <= sold ) {
                sold = sold - suma;
                rezultat = true;
        }
        return rezultat;
    }
    /** cerSold raporteaza soldul current;  @return soldul */
    public int cerSold() { return sold; }
}
```

O clasă care referă o interfaţă

```
/** CalculatorPlatiIpoteca face plati de ipoteca */
public class CalculatorIpoteca{
    private SpecificatieContBancar _contBancar; // pastreaza adresa

    // unui obiect care implementeaza SpecificatieContBancar
    /** Constructor CalculatorPlatiIpoteca initializeaza calculatorul.
    * @param cont - adresa contului bancar in/din care se fac
    •   depuneri/retrageri */
    public CalculatorIpoteca(SpecificatieContBancar cont)
    {  _contBancar = cont; }

    /** faPlataIpoteca efectueaza o plata de ipoteca din contul bancar.
    •   @param suma - suma de platit */
    public void faPlataIpoteca(int suma){
        boolean ok = _contBancar.retrage(suma);
        if ( ok )
        {  System.out.println("Plata efectuata: " + suma); }
        else { ... error ... }
    }
    ...
}
```

Restricţii pentru interfeţe

- Toate metodele unei interfeţe trebuie să fie metode de instanţă **abstract;** începând cu Java 8 *se permit și metode de instanță cu comportament implicit și metode statice*
- Toate variabilele definite într-o interfaţă trebuie să fie **static final**, adică *constante*
  - Valorile se pot stabili la compilare sau se pot calcula la încărcarea clasei
  - Variabilele pot fi de orice tip
- Nu sunt permise blocuri de iniţializare statice
  - Fiecare iniţializare trebuie să fie o line pentru o variabilă
  - Începând cu Java 8, sunt permise metode statice pentru iniţializare în interfaţă

Instanţierea

- Pentru a putea folosi metode ale instanței dintr-o interfață trebuie să existe un obiect asociat care implementează interfaţa
- Nu se poate instanţia o interfaţă direct, dar se poate instanţia o clasă care *implementează* interfaţa
  - Exemplu

```
SpecificatieContBancar contulMeu = new ContBancar();
```

- Referinţele la un obiect se pot face via
  - numele clasei,
  - unul dintre numele superclaselor sale sau
  - unul dintre numele interfeţelor sale

Ce se pune într-o interfaţă

- Este considerat stil prost a scrie o interfaţă numai cu constante (static final)
  - De obicei aceşti calificatori sunt omişi:

```
interface MyConstants{
    double PI = 3.141592;
    double E = 1.7182818;
}
```

Pot fi accesate fie ca MyConstants.PI sau doar PI de orice clasă care implementează interfaţa

- O interfaţă ar trebui să aibă cel puţin o metodă abstractă
- Dacă tot ce doriţi este o colecţie de constante, folosiţi o clasă obişnuită cu `import static`

Un alt exemplu. O "bază de date"

- Proiectaţi o clasă numită `Database`, care să păstreze o colecţie de obiecte "înregistrare" (record), fiecare având o cheie (key) unică pentru identificare
- Comportamente esenţiale – o specificaţie neformală:
  - `Database` păstrează o colecţie de obiecte `Record`, unde fiecare `Record` are un obiect `Key`. Restul structurii oricărei `Record` nu are importanţă şi nu este cunoscut bazei de date
  - `Database` va avea metode `insert, find` şi `delete`
  - Înregistrările `Record` indiferent de structura lor internă, vor avea o metodă `getKey` care returnează obiectul cheie (`Key`) al înregistrării (`Record`)
  - Obiectele `Key` vor avea o metodă `equals` care să compare două chei dacă sunt egale şi să returneze true sau false

Diagrama de clase cu interfeţe

<<interface>>
Key
+equals(m: Key): boolean

<<interface>>
Record
+getKey(): Key

<<class>>
IntegerKey
+equals(m: Key): boolean
+getInt(): int

<<class>>
BankAccount
+deposit(amount: int)
+getBalance(): int
+getKey(): Key

<<class>>
Database
+insert(r: Record): boolean
+find(k: Key): Record
+delete(k: Key): boolean

Interfeţe pentru Record şi Key

```
/** Record esle un element de date care poate fi stocat intr-o baza de date */
public interface Record{
    /** getKey returneaza cheia care identifica in mod unic inregistrarea
    * @return obiectul de tip Key din inregistrare */
    public Key getKey();
}

/** Key reprezinta o valoare pentru identificare, o "cheie" */
public interface Key{
    /** equals compara pe sine cu o alta cheie, m, ca sa determine daca sunt egale
    * @param m - celalalta cheie
    * @return true, daca aceasta cheie si m au aceeasi valoare a cheii;
    * returneaza false, in caz contrar */
    public boolean equals(Key m);
}
```

Superinterfeţe

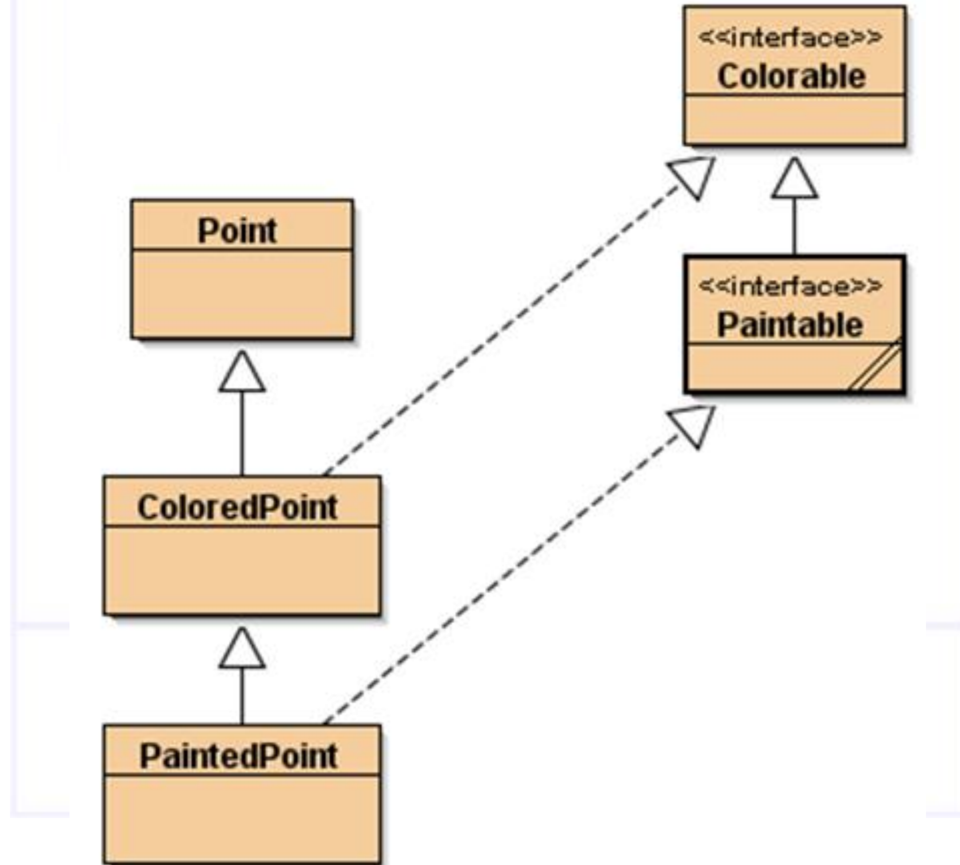
- Dacă este furnizată o clauză `extends,` atunci interfaţa în curs de declarare extinde fiecare dintre celelalte interfeţe numite şi din acest motiv moşteneşte metodele şi constantele fiecăreia dintre celelalte interfeţe numite

- Aceste alte interfeţe sunt numite *superinterfeţe directe* ale interfeţei în curs de declarare

- Orice clasă care implementează – `implements -` interfaţa declarată se consideră că *implementează toate interfeţele pe care această interfaţă le extinde şi care sunt accesibile clasei*

Super/sub interfeţe. Exemplu

- Diagrama de clase



Super/sub interfeţe. Exemplu

```
public interface Colorable {
   void setColor(int color);
   int getColor();
}
public interface Paintable extends Colorable {
   int MATTE = 0, GLOSSY = 1;
   void setFinish(int finish);
   int getFinish();
}
class Point {
   int x, y;
}
class ColoredPoint extends Point implements
                                    Colorable {
    int color;
    public void setColor(int color) {
       this.color = color;
    }
    public int getColor() {
      return color;
   }
}
```

```
class PaintedPoint extends ColoredPoint
                   implements Paintable {
    int finish;
    public void setFinish(int finish) {
         this.finish = finish;
    }
    public int getFinish() {
         return finish;
    }
}
```

- Interfaţa **Paintable** este o *subinterfaţă* a interfeţei **Colorable**
- Interfaţa **Colorable** este o *superinterfaţă* a lui **Paintable**

Interfețe predefinite. Exemplu

Interfața Comparable

- Interfața Comparable este definită în pachetul java.lang, fiind automat disponibilă fiecărui program
- Nu are decât următoarea metodă care trebuie implementată:

```
public int compareTo(Object other);
```

- Este responsabilitatea programatorului să urmeze semantica interfeței Comparable atunci când o implementează
- Metoda compareTo trebuie să returneze
  - Un număr negativ atunci când un obiect "este înainte de" parametrul other
  - Zero atunci când obiectul apelant "este egal cu" parametrul other
  - Un număr pozitiv dacă obiectul apelant "urmează după" parametrul other
- Semantica lui "equals" din metoda compareTo ar trebui să coincidă cu cea a metodei equals dacă se poate, dar aceasta nu este absolut necesar

Interfața Comparable. Exemplu

```
import java.util.*;
public class Person implements Comparable<Person>{
    // instance variables - replace the example below with your own
    private String nume;
    private String prenume;
    private int varsta;
    //contstructor
    public Person(String n, String pr, int v)    {
        // initialise instance variables
        nume = n;
        prenume = pr;
        varsta = v;
    }
    //@implements: compareTo(Object p) in Object
    public int compareTo(Person p)    {
        return nume.compareTo(p.nume);
    }
    //@Overrides: toString() in Object
    public String toString(){
        return (nume+ " " + prenume + ", " + varsta + " ani");
    }
}
```

Interfața Comparable. Exemplu

```java
public class Test {

    public static void afisare(Person[] v)    {
        for (int i = 0; i<v.length; i++)
            System.out.println(v[i].toString());
    }

    public static void main()    {
        Person[] v= new Person[3];
        v[0] = new Person("Pop", "Ioan", 20);
        v[1] = new Person("Ana", "Maria", 15);
        v[2] = new Person("Popescu", "Daria", 18);

        System.out.println("Vectorul initial de persoane:");
        afisare(v);
        Arrays.sort(v);
        System.out.println("\nVectorul sortat dupa nume:");
        afisare(v);

    }
}
```

Rezultatul afișat:

```
Vectorul initial de persoane:
Pop Ioan, 20 ani
Ana Maria, 15 ani
Popescu Daria, 18 ani

Vectorul sortat dupa nume:
Ana Maria, 15 ani
Pop Ioan, 20 ani
Popescu Daria, 18 ani
```

Sumar interfețe: Declararea constantelor în interfețe

- Definiții de constante care nu specifică explicit modificatorii recomandați; totusi, toate exemplele sunt identice:

```
public static final int x = 1; // ceea ce se obtine implicit
public int x = 1;
int x = 1;
static int x = 1;
final int x = 1;
public static int x = 1;
public final int x = 1;
static final int x = 1
```

- Oricare din aceste combinatii este legală, dar implicit toate variabilele declarate în interfețe vor avea modificatorii

  public static final

Sumar interfețe: Implementarea interfețelor

- Interfețele sunt niște contracte care obligă o clasă să implementeze anumite functionalități, fără ca să spună nimic despre cum trebuie implementate
- Interfețele pot fi implementate de orice clasă, de la orice nivel din arborele de moșteniri
- O interfață este ca o clasă abstractă 100%, și implicit este declarată abstractă
- Interfețele pot avea doar metode abstracte. Începând cu Java 8 sunt permise și metode concrete cu comportament implicit (*default*) și metode statice
- Metodele din interfețe sunt implicit publice și abstracte. Menționarea acestor modificatori este opțională
- Interfețele permit declararea constantelor, care implicit au modificatorii: public, static și final. Declararea explicită a acestor modificatori este opțională

Sumar interfețe: Implementarea interfețelor

- O clasă concretă care implementează o interfață are următoarele proprietăți:
  - Furnizează o implementare concretă a metodelor interfeței
  - Trebuie să respecte toate regulile de suprascriere a metodelor pe care le implementează
  - Permite tipul returnat să fie covariant
  - Nu poate declara noi excepții care nu au fost specificate în antetul metodei definite în interfață
- O clasă care implementează o interfață poate fi ea însăși abstractă; ea nu trebuie să implementeze neaparat metodele din interfață, însă lucrul acesta trebuie facut în prima subclasă concretă
- O clasă poate moșteni o singură clasă, dar poate implementa oricâte interfețe
- O interfață poate extinde una sau mai multe interfețe
- Interfețele nu pot extinde o clasă sau implementa vreo interfață

Interfețe în Java 7

- Java 7 sau versiuni mai vechi
  - Interfața poate avea 2 componente
    - Constante
    - Metode abstracte
  - Clasele concrete care implementează interfața **trebuie** să implementeze toate metodele definite de interfață
- Exemplu:

```java
public interface DemoInterface {
    public void div(int a, int b);
}

public class Demo implements DemoInterface{
    public void div(int a, int b)
    {
        System.out.print("Metoda div: ");
        System.out.println(a / b);
    }

    public static void main(String[] args)
    {
        DemoInterface x = new Demo();
        Demo y = new Demo();
        x.div(4,2);
        y.div(30,7);
    }
}
```

Interfețe în Java 8

- Începând cu Java 8
  - Interfața poate avea 4 componente
    - Constante
    - Metode abstracte
    - Metode *default*
    - Metode statice
  - Clasele concrete care implementează interfața pot să suprascrie comportamentul implicit al metodelor *default*
    - Nu este obligatoriu!

Interfețe în Java 8

- Exemplu:

```java
public interface DemoInterface {

  public void div(int a, int b);

  public default void suma(int a, int b) {
    System.out.print("Metoda default: ");
    System.out.println(a + b);
  }

  public static void mul(int a, int b) {
    System.out.print("Metoda statica: ");
    System.out.println(a * b);
  }

}
```

```java
public class Demo implements DemoInterface {

    public void div(int a, int b) {
        System.out.print("Metoda div: ");
        System.out.println(a / b);
    }

    public static void main(String[] args) {
        DemoInterface x = new Demo();
        Demo y = new Demo();
        y.div(4,2);
        y.div(30,7);

        DemoInterface z = new Demo();
        z.div(8, 2);
        z.suma(3, 2);
        DemoInterface.mul(4, 9);
    }
}
```