Programare orientată pe obiecte

1. Clasele Object si 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)` – trebuie să fie suprascrisă pentru egalitatea de conţinut
  - `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. Semnificaţie lui "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.)
  - *Egalitatea de conţinut* care înseamnă că două expresii reprezintă obiecte cu aceeaşi valoare/conţinut.

Simbolul == testează egalitatea de identitate atunci când este aplicat datelor referinţă.

- Exemplu:

```
AOval ov1, ov2;
ov1 = new AOval(0, 0, 100, 100);
ov2 = new AOval(0, 0, 100, 100);
if (ov1 == ov2){ System.out.println("identity equality");}
 else {System.out.println("content equality");}
```

- Metoda *equals* din `Object` poate fi folosită pentru a verifica egalitatea de conţinut.

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 instanţe 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)

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 reprezentate de acest obiect `Class`

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`**

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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
  - 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 doua obiecte *au fost create* din aceeaşi clasă

Exemple

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

```
void printClassName(Object obj) {
   System.out.println(obj + " is of class " +
   obj.getClass().getName());
}
```

- Alte exemple

```
Circle c = new Circle(5);
printClassName(c);
Class c1 = c.getClass();
System.out.println(c1.getName()); // tipareste "Circle"
Triangle t = new Triangle(7);
printClassName(t);
try {
   Class c2 = Class.forName("Triangle");
   System.out.println(c2.getName()); // tipareste "Triangle"
}
catch (ClassNotFoundException e) {
    System.err.println("No class for \"Triangle\"" +
  e.getMessage());
}
```

Nevoia de specificaţii

- Un program este asamblat dintr-o colecţie de clase care trebuie să "lucreze împreună" sau să "se potrivească una cu alta"
  - Ce înseamnă că doua clase se potrivesc una cu alta?
- Exemplu:
  - Un radio portabil – are nevoie de baterii pentru a funcţiona. Ce fel de baterii?
  - "Două baterii AA (R6)" – o specificaţie
  - Scopurile acestei specificaţii:
    - pentru utilizator: îi spune ce componentă trebuie pusă în aparat pentru a funcţiona
    - pentru fabricantul aparatului: ce dimensiuni trebuie să aibă compartimentul bateriilor şi ce tensiune şi curent să folosească pentru alimentarea circuitelor
    - pentru producătorul bateriilor: mărimea, tensiunea şi curentul pentru baterii astfel ca alţii să le poată folosi

Specificaţiile şi Java

- Limbajul şi compilatorul Java ne pot ajuta să:
  - Scriem specificaţii de clase şi să
  - Verificăm că o clasă satisface corect (implementează) specificaţiile sale
- Vom studia:
  - 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

- Doua 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 scrisa pentru a implementa o `SpecificatieContBancar`, clasa respectivă trebuie să conţină doua 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 Dvs. 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
  - etc.

Ce este o interfaţă?

- 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*;
  - În loc de aceasta, 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 definiţii de metode, fără implementări, 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 următoarele:
  - *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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Definirea unei interfeţe

- Definiţia 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.

O clasa care referă o interfaţă Java

```
/** CalculatorPlatiIpoteca face plati de ipoteca */
public class CalculatorPlatiIpoteca
{
   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 CalculatorPlatiIpoteca(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 ... }
   }
   ...
}
```

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; }
   // observati ca metodele depune si retrage au acelasi nume cu metodele
   // 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 curent
   * @return soldul */
   public int cerSold() { return sold; }
}
```

<<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);
```

OF CLUJ-NAPOCA
Computer Science

Restricţii pentru interfeţe

- Restricţii referitoare la interfeţe:
  - Toate metodele unei interfeţe trebuie să fie metode de instanţă **abstract;** *nu se permit metode statice*.
  - Toate variabilele definite într-o interfaţa trebuie să fie **static final**, adică *constante*.
    - Valorile se pot stabili la compilare sau se pot calcula la încarcarea clasei.
    - Variabilele pot fi de orice tip.
  - Nu sunt permise blocuri de iniţializare statice.
    - Fiecare iniţializare trebuie să fie o line pentru o variabilă.
    - Nu sunt permise metode auxiliare, statice, pentru iniţializare în interfaţă. Dacă aveţi nevoie de ele, atunci acestea trebuie definite în afara interfeţei.

Instanţierea

- Metodele din interfaţă sunt *întotdeauna* metode ale instanţei.
  - Pentru a le putea folosi, trebuie să existe un obiect asociat care implementează interfaţa.
  - Nu puteţi instanţia o interfaţă direct, dar puteţi instanţia o clasă care *implementează* interfaţa.
  - Referinţele la un obiect se pot face via
    - numele clasei,
    - unul dintre numele superclaselor sale sau
    - unul dintre numele interfeţelor sale.

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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. Scriem doar, d.e.:

```
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 metoda abstractă.
- Daca 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

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);
}
```

Diagrama de clase cu interfeţe

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

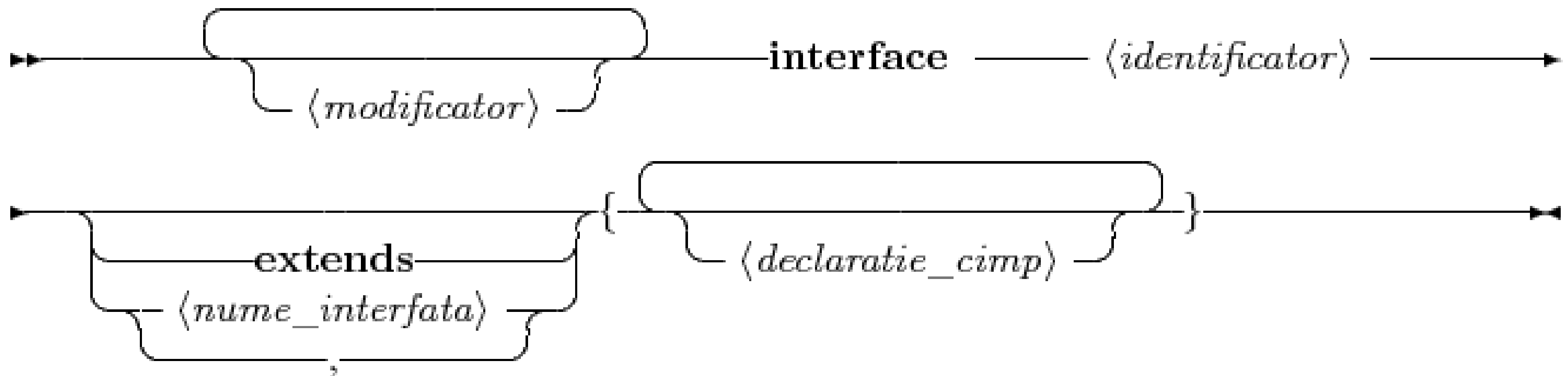
<<interface>>
Record
+getKey(): Key

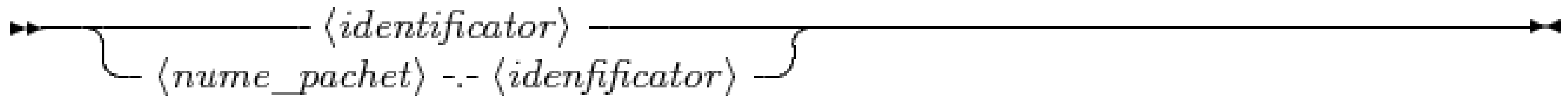
<<class>>
IntegerKey
+equals(m: Key): boolean
+getInt(): int

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

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

Diagrama de sintaxă pentru declararea unei interfeţe





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 numite sunt *superinterfeţe directe* ale interfeţei în curs de declarare.
- Orice clasă care implementează – **`implements -`** interfaţa declarată se consideră că *implementează toate interfeţele pe care aceasta interfaţă le extinde şi care sunt accesibile clasei*.
  - Vom reveni asupra interfeţelor când vom discuta moştenirea

Super/sub interfeţe. Exemplu

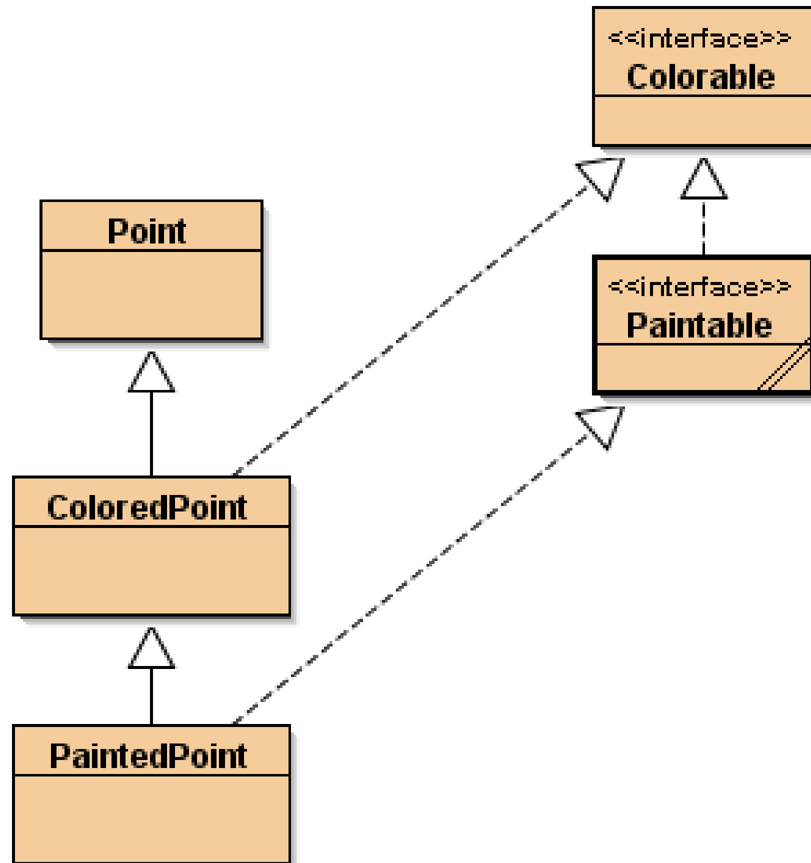
```
public interface Colorable {
   void setColor(int color);
   int getColor();
}
public interface Paintable extends
   Colorable {
   int MATTE = 0, GLOSSY = 1;
   void setFinish(int finish);
   int getFinish();
}
class Point {
   int x, y;
}
class ColoredPoint extends Point
   implements Colorable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

- Relaţiile sunt astfel:
  - Interfaţa **Paintable** este o *superinterfaţă* a clasei **PaintedPoint**.
  - Interfaţa **Colorable** este o *superinterfaţă* a clasei **ColoredPoint** şi a clasei **PaintedPoint**.
  - Interfaţa **Paintable** este o *subinterfaţă* a interfeţei **Colorable**, iar **Colorable** este o *superinterfaţă* a lui **Paintable**

Super/sub interfeţe. Exemplu
(continuare)

- Diagrama de clase pentru exemplul precedent



Cum se folosesc interfeţele

- Nu există un singur răspuns simplu
- Ca pentru orice caracteristică semantică dintr-un limbaj de programare, nu există reguli prestabilite şi rapide referitoare la situaţiile în care se potrivesc cel mai bine.
- Există totuşi reguli ghid şi strategii generale (altfel această caracteristică nu ar fi fost inclusă).
- Vom sugera câteva modalităţi de utilizare tipice.

Câteva moduri de folosire a interfeţelor

1. Pentru a clarifica funcţionalitatea asociată cu o anume abstractizare.
2. Pentru a abstractiza funcţionalitatea într-o metodă şi a o generaliza (cum este cazul pointerilor la funcţii în C – vedeţi exemplele de la sortare).
3. Pentru a implementa callbacks, cum se face în programarea GUI
4. Pentru a scrie cod mai general, dependent de implementare, uşor de extins şi întreţinut.
5. Pentru a simula constante globale.

Clarificarea funcţionalităţii

- Adeseori scriem cod sa facă ceva anume.
  - Nu va face niciodată altceva.
  - Nu suntem preocupaţi de extensibilitatea codului respectiv.
- Chiar şi în acest caz poate fi util să organizăm programul folosind interfeţe.
  - Face codul mai uşor de citit şi clarifică intenţia autorului.

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Callbacks

- Callback-urile sunt o tehnică generală de programare în care o metodă apelează o alta metodă care, la rândul său apelează metoda apelantă (de obicei pentru a informa apelantul că a avut loc o acţiune).
- Exemple relevante: **`Timer`** **`ActionListeners`** din Swing.

Pentru a scrie implementări mai generale

- O metodă care operează asupra unei variabile de tip interfaţă funcţionează automat pentru orice sub-tip al interfeţei respective.
- Acest lucru este mult mai general decât a scrie programul ca să opereze asupra unui anumit sub-tip.

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Abstractizarea funcţionalităţii

- O metodă poate fi adesea făcuta mai generală prin adaptarea a ceea ce face pe baza implementării altor metode pe care le apelează.
- sort(...) este un exemplu bun. O funcţie sort() poate sorta orice listă de articole dacă i se spune cum să compare oricare doua articole.
- Metodele numerice de rezolvare a ecuaţiilor diferenţiale depinde adeseori de derivarea discretă: putem face o asemenea rutină să fie generală prin specificarea independentă a tehnicii de derivate.

Rezumat

- Clasa `Object`
  - toate moştenesc din clasa Object
  - metode de suprascris
- Clasa `Class`
  - metode utile
- Operatorul `instanceof` vs metoda `getClass()`
- Interfeţe

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science