Programare orientată pe obiecte

1. Interfeţe Java
2. Pachete (packages)
3. Moştenire (I)

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 ai dezvălui clasa
  - Modelarea moştenirii multiple, care permite ca o clasă să aibă mai mult de o superclasă

Definirea unei interfeţe

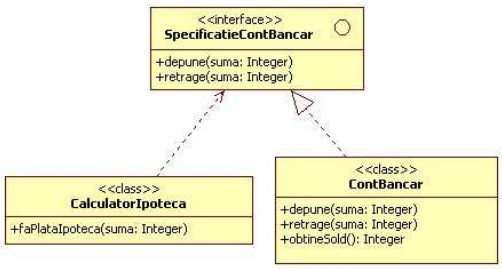
- Definiţia unei interfeţe are două componente: declaraţia interfeţei şi corpul interfeţei
  - *Declaraţia* interfeţei defineşte diferitele atribute ale interfeţei, cum sunt numele şi dacă ea extinde alte interfeţe.
  - *Corpul* interfeţei conţine declaraţiile de constante şi de metode pentru interfaţa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

- 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);
```


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.

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);
}
```


Exemplul în BlueJ

- BlueJ demo şi discuţie

Diagrama de clase cu interfeţe

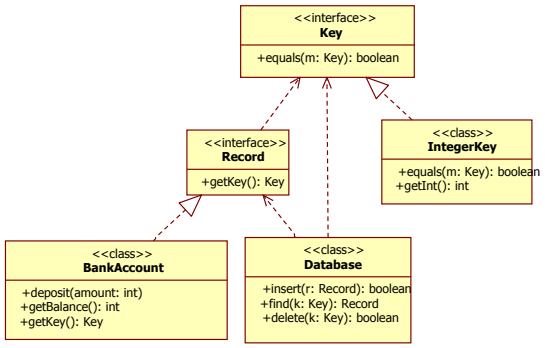
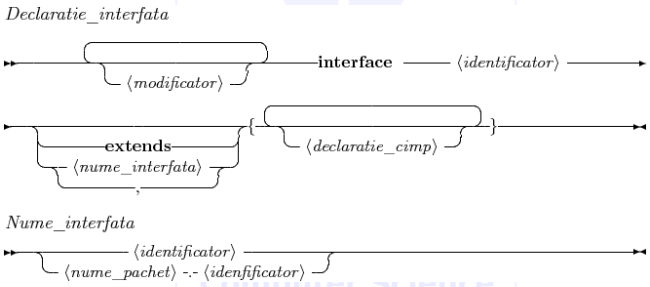


Diagrama de sintaxă pentru declararea unei interfeţe



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

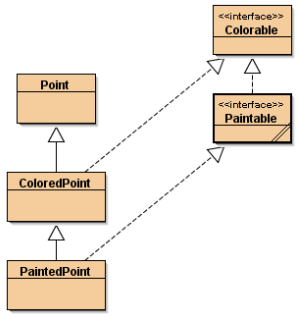
```
public interface Colorable {
   void setColor(int color);
   int getColor();
}
public interface Paintable extends
   Colorable {
   int MATTE = 0, GLOSSY = 1;
   void setFinish(int finish);
   int getFinish();
}
class Point {
   int x, y;
}
class ColoredPoint extends Point
   implements Colorable {
       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. Aceasta face codul mai uşor de citit şi clarifică intenţia autorului.

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.

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.

Organizarea claselor înrudite în pachete

- Pachet (package): set de clase înrudite
- Pentru a pune clase într-un pachet, trebuie scrisă o linie de genul

```
package numePachet;
```

  ca primă instrucţiune în fişierul sursă care conţine clasele
- Numele pachetului constă din unul sau mai mulţi identificatori separaţi prin puncte

Organizarea claselor înrudite în pachete

- Spre exemplu, pentru a pune clasa **`Database`** prezentată anterior într-un pachet numit **`oop.examples`**, fişierul **`Database.java`** trebuie să înceapă astfel:

```
package oop.examples;
public class Database
{
   . . .
}
```

- Pachetul implicit nu are nume, deci nu are o specificare **`package`**
- BlueJ demo

Organizarea claselor înrudite în pachete

Pachet	Scop	Exemplu de clasă
java.lang	suport pentru limbaj	Math
java.util	utilitare	Random
java.io	intrare şi ieşire	PrintScreen
java.awt	Abstract Windowing Toolkit	Color
java.applet	Applets	Applet
java.net	Networking	Socket
java.sql	accesul la baze de date	ResultSet
java.swing	interfaţa utilizator swing	JButton
org.omg.CORBA	Common Object Request Broker Architecture	IntHolder

Importul pachetelor

- Se poate folosi întotdeauna o clasă fără import

```
java.util.Scanner s = new java.util.Scanner(System.in);
```

  - Dar e greoi să folosim nume calificate complet
- Import ne permite să folosim nume mai scurte pentru clase

```
import java.util.Scanner;
. . .
Scanner in = new Scanner(System.in)
```

- Putem importa toate clasele dintr-un pachet `import java.util.*;`
- Nu este nevoie să importăm `java.lang`
- Nu este nevoie să importăm alte clase din acelaşi pachet

Nume de pachete şi determinarea locului unde se află clasele

- Folosiţi pachete pentru a evita conflictele de nume
  `java.util.Timer` vs. `javax.swing.Timer`
- Numele de pachete trebuie să fie neambigue
- Recomandare: începeţi cu numele invers al domeniului, e.g.
  `ro.utcluj.cs.jim`: pentru clasele lui jim (jim@cs.utcluj.ro)

Nume de pachete şi determinarea locului unde se află clasele

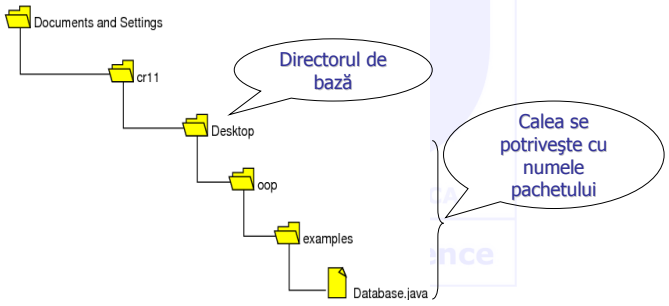
- Numele căii trebuie să se potrivească cu numele pachetului
  `oop/examples/Database.java`
- Numele căii începe cu calea spre clase

```
export CLASSPATH=/home/jim/lib:.
set CLASSPATH=D:\Documents and
Settings\cr11\Desktop;.
```

- Calea spre clase conţine directoarele de bază care pot conţine directoare de pachet

Directoare de bază şi subdirectoare pentru pachete

```
set CLASSPATH=C:\Documents and Settings\cr11\Desktop;.
```



Cum se construieşte un pachet

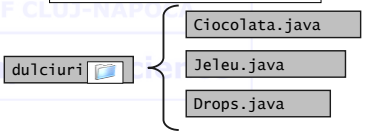
1) Puneţi o linie cu numele pachetului la începutul fiecărei clase.

```
package pachetDulciuri;
public class Ciocolata {
   . . .
}
```

```
package pachetDulciuri;
public class Jeleu {
   . . .
}
```

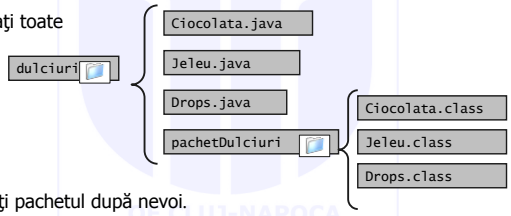
```
package pachetDulciuri;
public class Drops {
   . . .
}
```

2) Stocaţi fişierele Java din pachet într-un director comun.



Cum se construieşte un pachet

3) Compilaţi toate fişierele.



4) Importaţi pachetul după nevoi.

```
import dulciuri.pachetDulciuri.*;
public class ConsumatorDulciuri { . . .
```


Cum să refolosim codul?

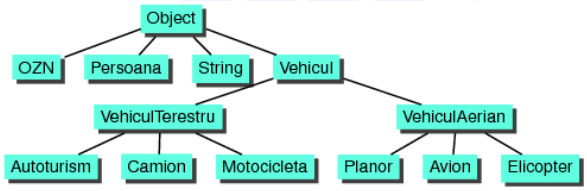
- Putem scrie clase de la început – fără a refolosi nimic (o extremă).
  - Ceea ce unii programatori doresc să facă întotdeauna
- Putem găsi o clasa existentă care se potriveşte exact cerinţelor problemei (o altă extremă).
  - Cel mai uşor lucru pentru programator
- Putem construi clase din clase existente bine testate şi bine documentate.
  - Un fel de refolosire foarte tipic, numit refolosire prin compoziţie!
- Putem refolosi o clasă existentă prin moştenire
  - Necesită mai multe cunoştinţe decât refolosirea prin compoziţie.

Introducere în moştenire

- *Moştenirea* este una din tehnicile principale ale programării orientate pe obiecte
- Folosind această tehnică:
  - se defineşte mai întâi o formă foarte generală de clasă şi se compilează, apoi
  - se definesc versiuni mai specializate ale clasei prin adăugarea de variabile instanţă şi de metode
- Despre clasele specializate se spune că *moştenesc* metodele şi variabilele instanţă ale clasei generale

Introducere în moştenire

- Moştenirea modelează relaţii de tipul "*este o(un)*"
  - Un obiect "este un" alt obiect dacă se poate comporta în acelaşi fel
  - Moştenirea foloseşte *asemănările şi deosebirile* pentru a modela grupuri de obiecte înrudite
- Unde există moştenire, există şi o *ierarhie de moştenire* a claselor



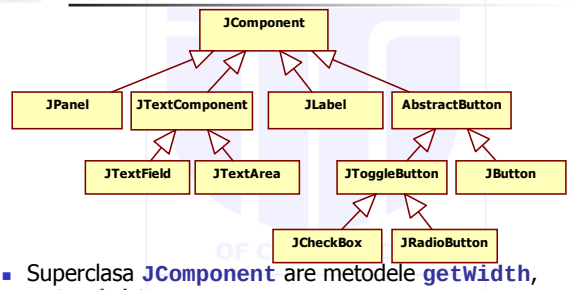
Introducere în moştenire

- Moştenirea este un mod de:
  - organizare a informaţiei
  - grupare a claselor similare
  - modelare a asemănărilor între clase
  - creare a unei taxonomii de obiecte
- `Vehicul` este numit *superclasă*
  - sau *clasă de bază* sau *clasă părinte*
- `VehiculTerestru` este numit *subclasă*
  - sau *clasă derivată* sau *clasă fiică*
- Oricare clasă poate fi de ambele feluri în acelaşi timp
  - D.e., `VehiculTerestru` este superclasă pentru `Camion` şi subclasă pentru `Vehicul`

Introducere în moştenire

- În Java se poate moşteni de la *o singură* superclasă
  - altminteri, nu există limite pentru adâncimea sau lăţimea ierarhiei de clase
  - C++ permite unei subclase să moştenească de la mai multe superclase (lucru care are tendinţa de a produce erori)
- În Java fiecare clasă extinde clasa `Object` fie direct, fie indirect
- O clasă are în mod automat toate variabilele instanţă şi metodele clasei de bază şi poate avea şi metode suplimentare şi/sau variabile instanţă
- Moştenirea este avantajoasă deoarece permite să se *refolosească* codul, fără a fi nevoie să fie copiat în definiţiile claselor derivate

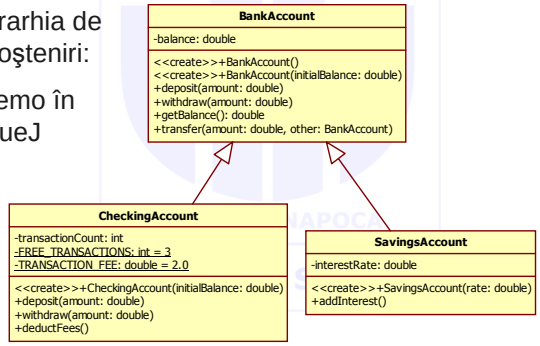
Exemplu. O parte a ierarhiei componentelor interfeţei utilizator Swing



- Superclasa **`JComponent`** are metodele **`getWidth`**, **`getHeight`**
- Clasa **`AbstractButton`** are metode pentru a seta/obţine textul şi icoana unui buton

O ierarhie mai simplă: ierarhia unor conturi bancare

- Ierarhia de moşteniri:
- Demo în BlueJ



O ierarhie mai simplă: ierarhia unor conturi bancare

- Scurtă specificaţie
  - Toate conturile bancare suportă metoda **`getBalance`** – obţine soldul contului
  - Toate conturile bancare suportă metodele **`deposit`** (depune) şi **`withdraw`** (retrage), dar implementările diferă
  - Contul de cecuri (**`CheckingAccount`**) are nevoie de o metodă pentru deducerea taxelor de prelucrare – **`deductFees`**; contul de economii (**`SavingsAccount`**) are nevoie de o metodă pentru adăugarea dobânzii – **`addInterest`**

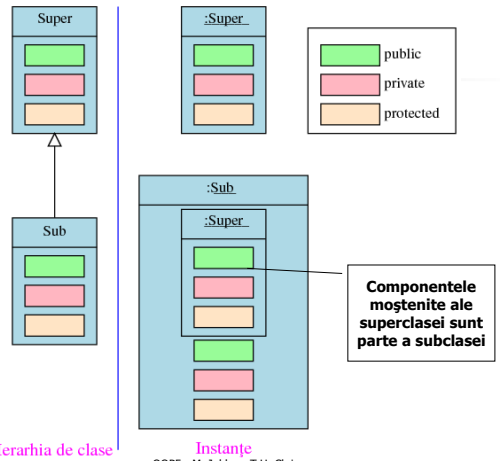
Clase derivate

- Cum un cont de economii este un cont bancar, el este definit ca o clasă *derivată* a clasei **`BankAccount`**
  - O clasă *derivată* se defineşte prin adăugarea de variabile şi/sau metode la o clasă existentă
  - Fraza **`extends BaseClass`** trebuie adăugată în definiţia clasei derivate:

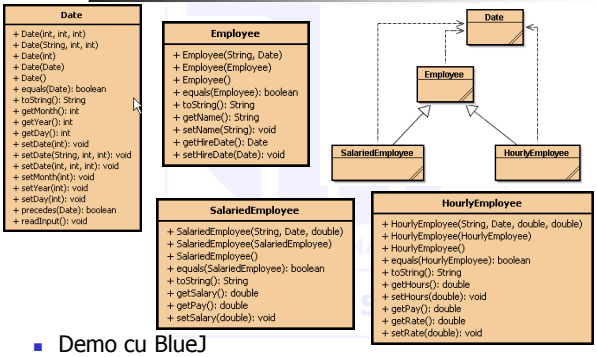
```
public class SavingsAccount extends BankAccount
```

- Sintaxa pentru moştenire:

```
class NumeSubclasa extends NumeSuperclasa
{
   metode
   câmpuri de instanţă
}
```

Un exemplu: Employees



- Demo cu BlueJ

Clase derivate

- La definirea unei clase derivate, ea moşteneşte variabilele instanţă şi metodele clasei de bază pe care o extinde
  - Clasa **`Employee`** (angajat) defineşte variabilele instanţă **`name`** (nume) şi **`hireDate`** (data angajării) în definiţia sa
  - Clasa **`HourlyEmployee`** (ziler) are şi aceste variabile instanţă, dar ele nu sunt specificate în definiţia sa
  - Clasa **`HourlyEmployee`** are variabile instanţă suplimentare **`wageRate`** (salariu pe oră) şi **`hours`** (ore lucrate) în definiţia sa

Clase derivate

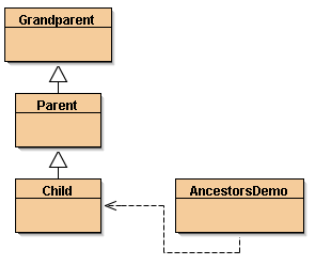
- Aşa cum moşteneşte variabilele clasei **`Employee`**, clasa **`HourlyEmployee`** moşteneşte şi toate metodele superclasei
  - Clasa **`HourlyEmployee`** moşteneşte metodele **`getName`**, **`getHireDate`**, **`setName`**, şi **`setHireDate`** (obţine numele, obţine data angajării, setează numele şi setează data angajării) din clasa **`Employee`**
  - Orice obiect de clasa **`HourlyEmployee`** poate invoca una dintre aceste metode, exact ca pe oricare altă metodă

Clase derivate (subclase)

- O clasă derivată, numită şi *subclasă*, este definită pornind de la o altă clasă definită deja, numită *clasă de bază* sau *superclasă*, prin adăugarea (şi/sau modificarea) de metode, variabile instanţă şi de variabile statice
  - Clasa derivată *moşteneşte* toate *metodele publice*, toate *variabilele instanţă publice şi private*, precum şi toate *variabilele statice publice şi private* din clasa de bază
  - Clasa derivată *poate adăuga* variabile instanţă, variabile statice şi/sau metode
- *Definiţiile* variabilelor şi metodelor moştenite *nu apar* în clasa derivată
  - Codul este reutilizat fără a fi nevoie să fie copiat explicit, cu excepţia cazului în care creatorul clasei derivate nu *redefineşte* una sau mai multe dintre *metodele* clasei

Clase părinţi şi clase copii

- O clasă de bază este numită adesea *clasă părinte*
  - Clasa derivată se mai numeşte şi *clasă fiică (copil)*
- Aceste relaţii sunt adesea extinse astfel că o clasă este părintele unui părinte . . . al unei alte clase şi se numeşte *clasă strămoş*
  - Dacă clasa **A** este un strămoş al clasei **B**, atunci clasa **B** poate fi numită clasă *descendentă* a clasei **A**



Rezumat

- Interfeţe Java
  - definiţii
  - declarare
  - utilitate
  - restricţii
  - exemple
- Pachete Java
  - organizarea claselor înrudite
  - API-uri Java
- Moştenire
  - Superclase
  - Subclase (clase derivate)