



Programare orientată pe obiecte

1. Pachete (packages)
2. Moștenire



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 instrucțiune în fișierul sursă care conține clasele

- Numele pachetului constă din unul sau mai mulți identificatori separați prin puncte



Organizarea claselor înrudite în pachete

- Spre exemplu, pentru a pune clasa **Database** prezentată anterior într-un pachet numit **oop.examples**, fișierul **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 și ieșire	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	interfața 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 greu 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 același pachet



Nume de pachete și determinarea locului unde se află clasele

- Folosiți pachete pentru a evita conflictele de nume

`java.util.Timer` VS. `javax.swing.Timer`

- Numele de pachete trebuie să fie neambigue

- Recomandare: Începeți cu numele invers al domeniului, e.g.

`ro.utcluj.cs.jim`: pentru clasele lui jim
(jim@cs.utcluj.ro)



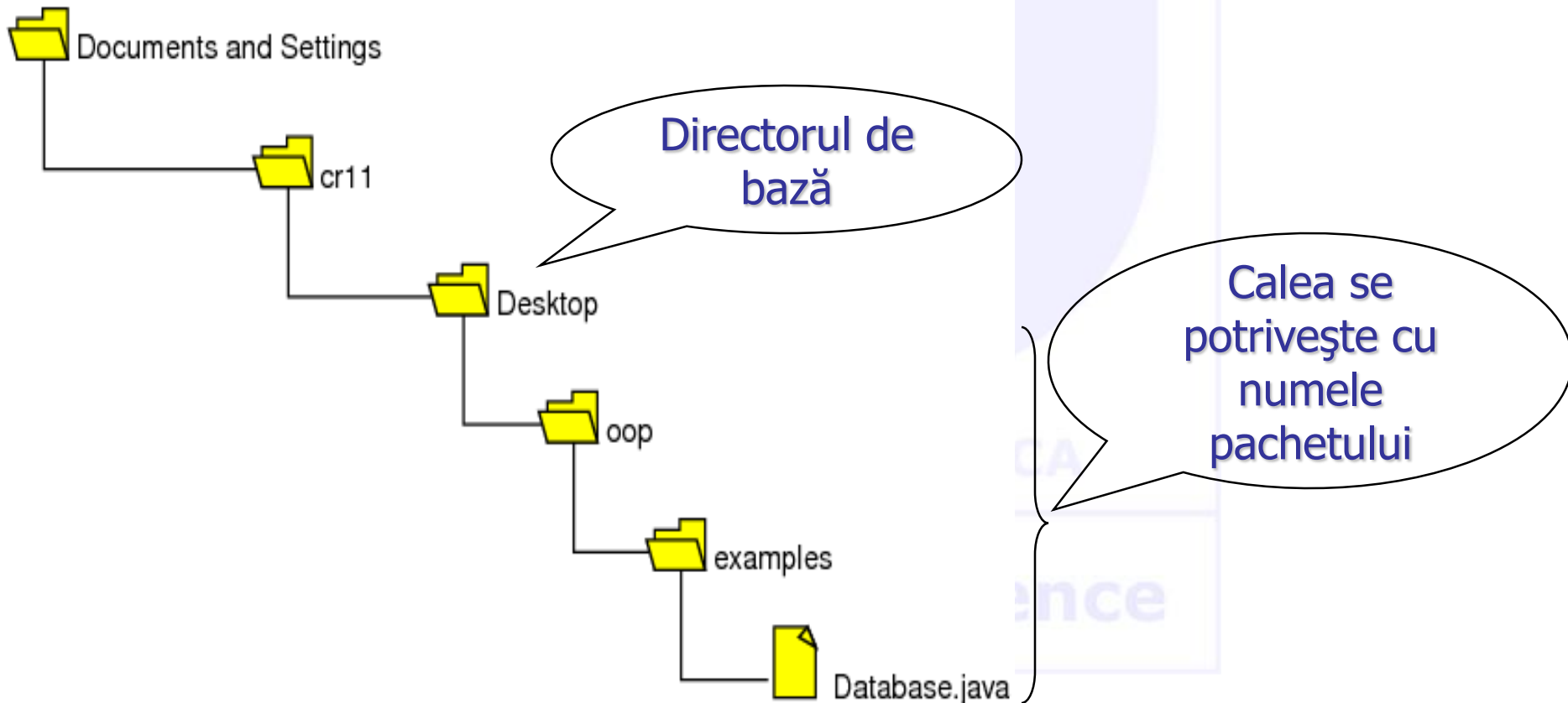
Nume de pachete și determinarea locului unde se află clasele

- 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 conține directoarele de bază care pot conține directoare de pachet



Directoare de bază și subdirectoare pentru pachete

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





Cum se construiește un pachet

1) Puneți 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) Stocați fișierele Java din pachet într-un director comun.

pachetDulciuri



Ciocolata.java

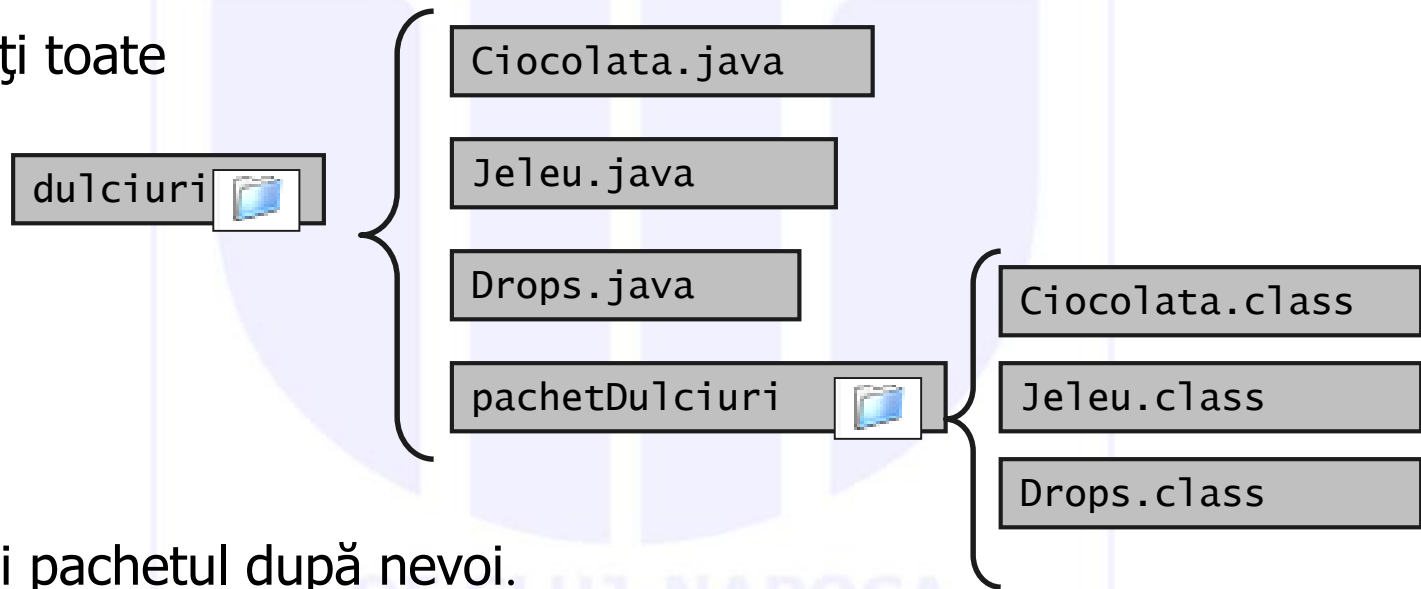
Jeleu.java

Drops.java



Cum se construiește un pachet

3) Compilați toate fișierele.



4) Importați 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 clasă existentă care se potrivește exact cerințelor problemei (o altă extremă).
 - Cel mai ușor lucru pentru programator
- Putem construi clase din clase existente bine testate și bine documentate.
 - Un fel de refolosire foarte tipic, numit refolosire prin compoziție!
- Putem refolosi o clasă existentă prin moștenire
 - Necesită mai multe cunoștințe decât refolosirea prin compoziție.



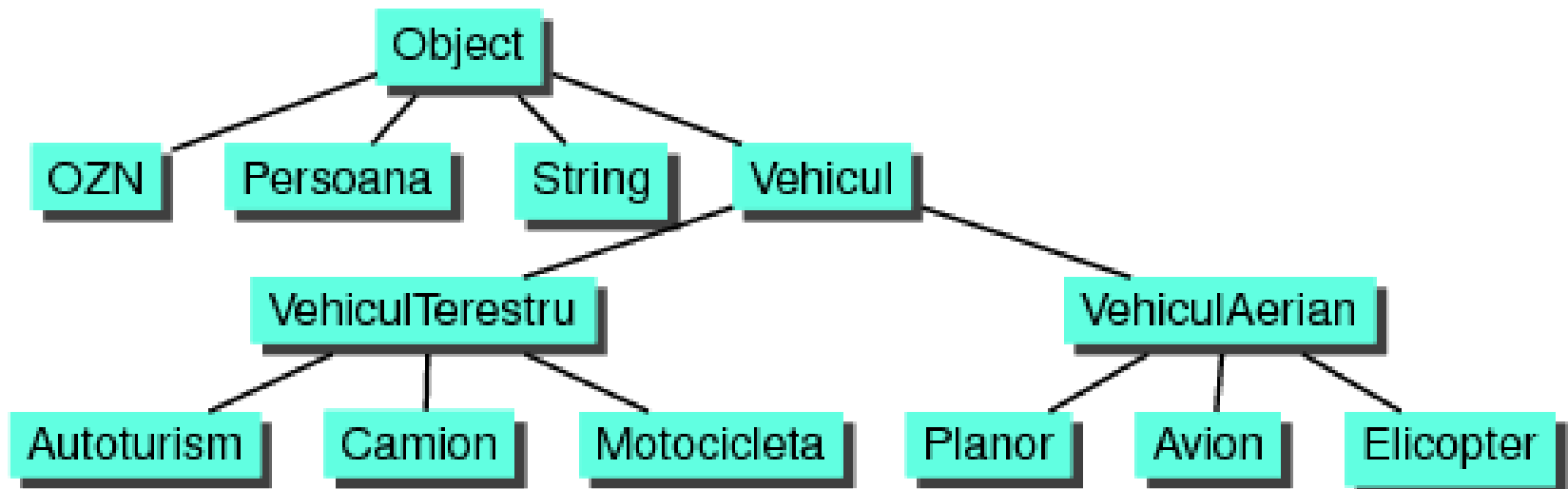
Introducere în moștenire

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



Introducere în moștenire

- Moștenirea modelează relații de tipul "*este o(un)*"
 - Un obiect "este un" alt obiect dacă se poate comporta în același fel
 - Moștenirea folosește *asemănările* și *deosebiri* pentru a modela grupuri de obiecte înrudite
- Unde există moștenire, există și o *ierarhie de moștenire* a claselor





Introducere în moștenire

- Moștenirea este un mod de:
 - organizare a informației
 - 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 același timp
 - D.e., **VehiculTerestru** este superclasă pentru **Camion** și subclasă pentru **Vehicul**

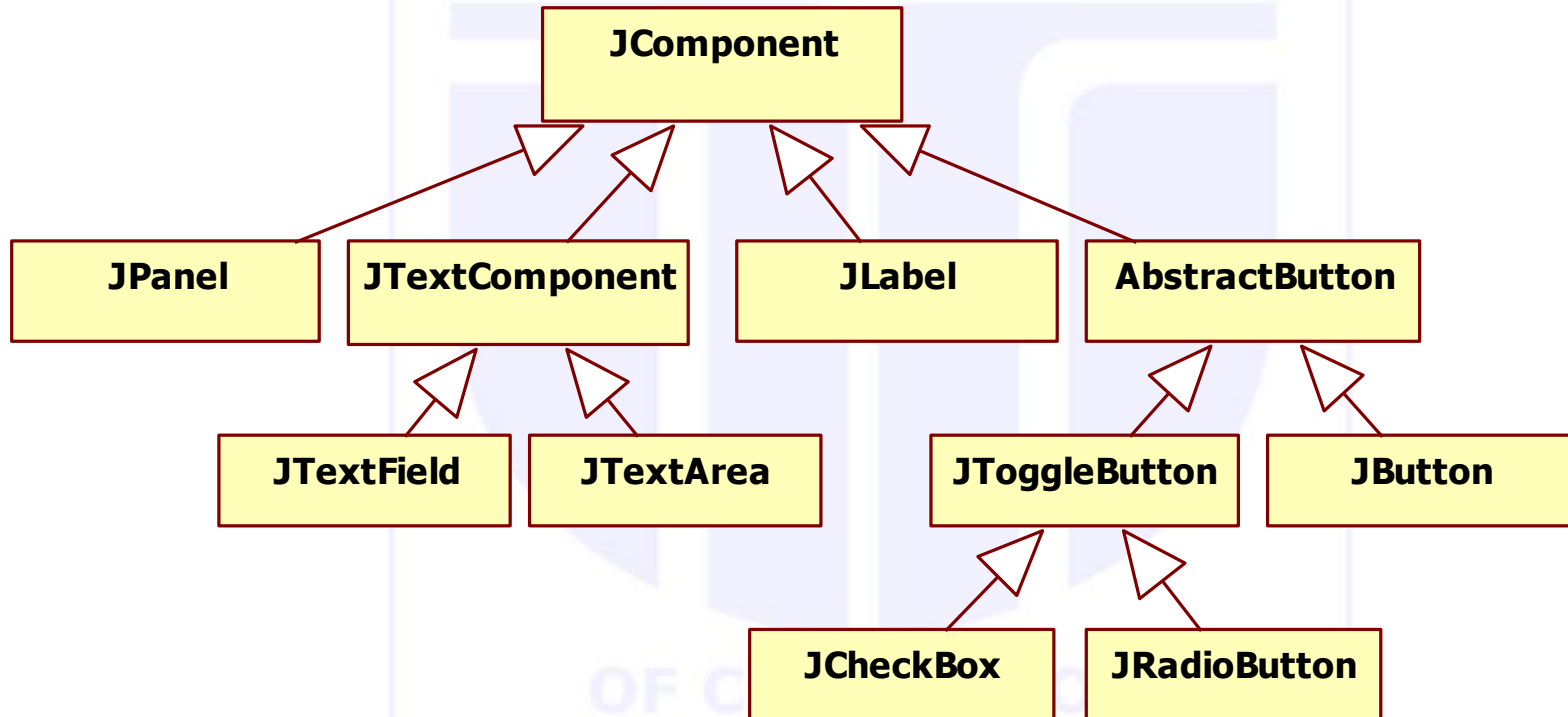


Introducere în moștenire

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



Exemplu. O parte a ierarhiei componentelor interfeței utilizator Swing

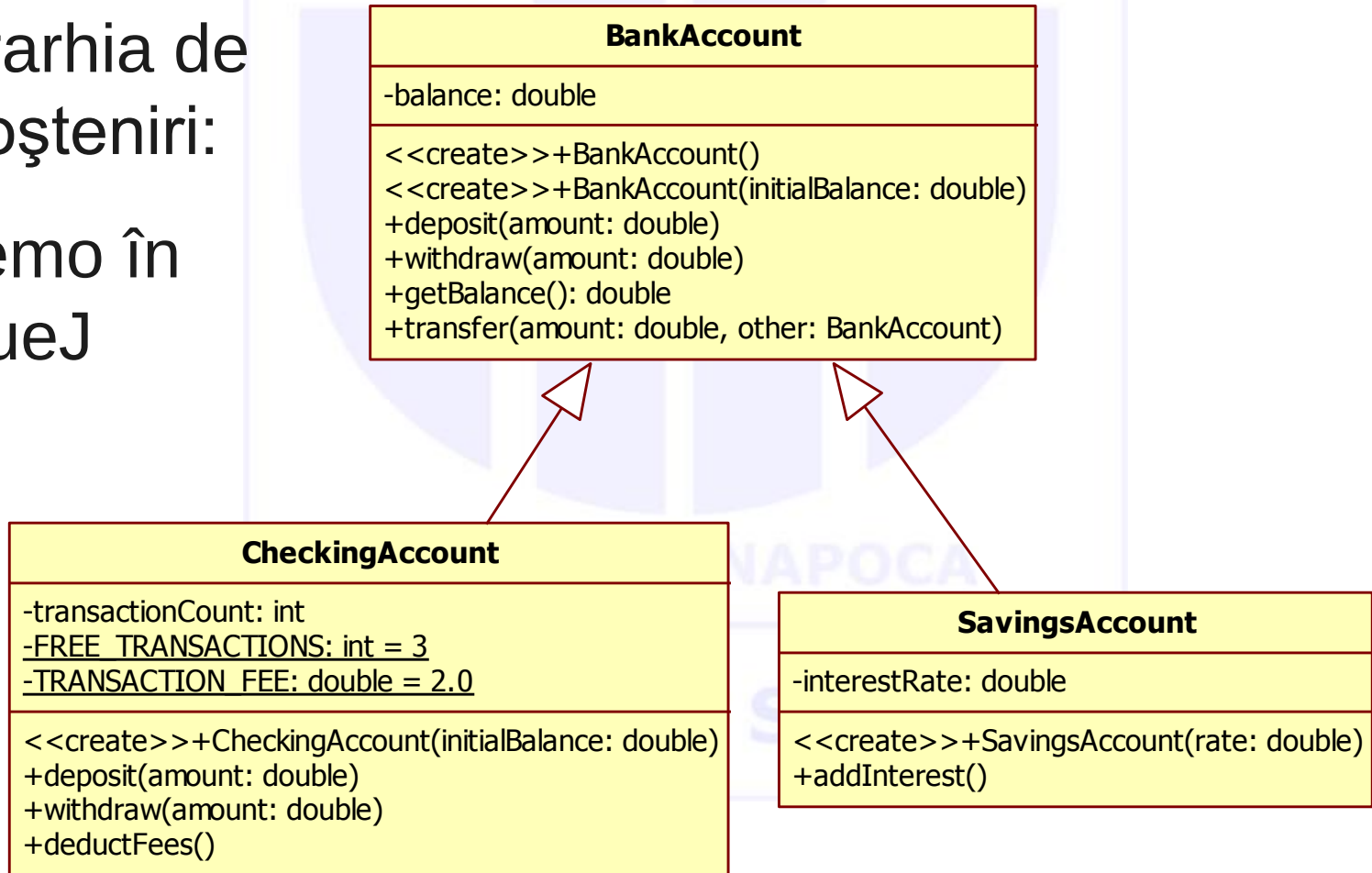


- Superclasa **JComponent** are metodele **getWidth**, **getHeight**
- Clasa **AbstractButton** are metode pentru a seta/obține textul și icoana unui buton



O ierarhie mai simplă: ierarhia unor conturi bancare

- Ierarhia de moșteniri:
- Demo în BlueJ





O ierarhie mai simplă: ierarhia unor conturi bancare

- Scurtă specificație
 - Toate conturile bancare suportă metoda **getBalance** – obține soldul contului
 - Toate conturile bancare suportă metodele **deposit** (depune) și **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**



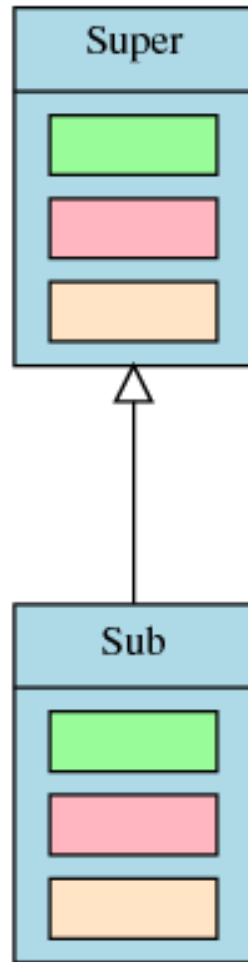
Clase derivate

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

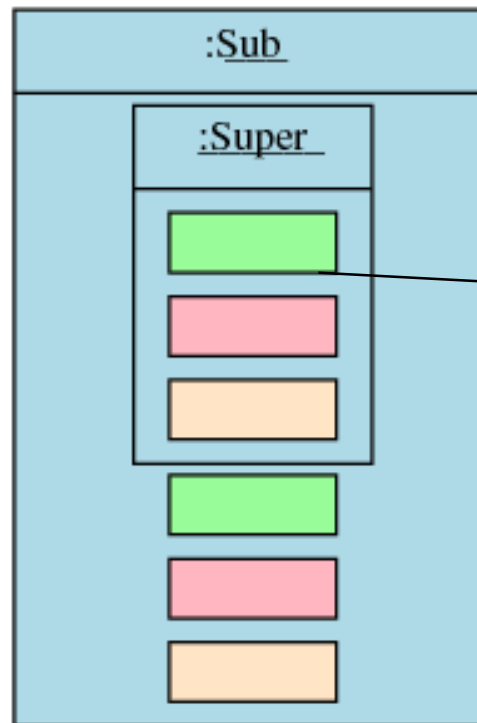
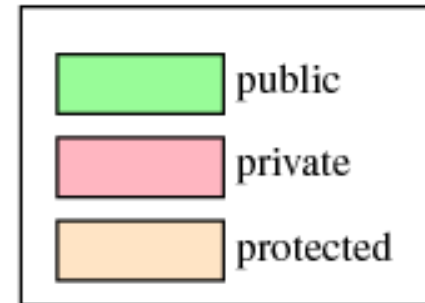
```
public class SavingsAccount extends BankAccount
```

- Sintaxa pentru moștenire:

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



Ierarhia de clase

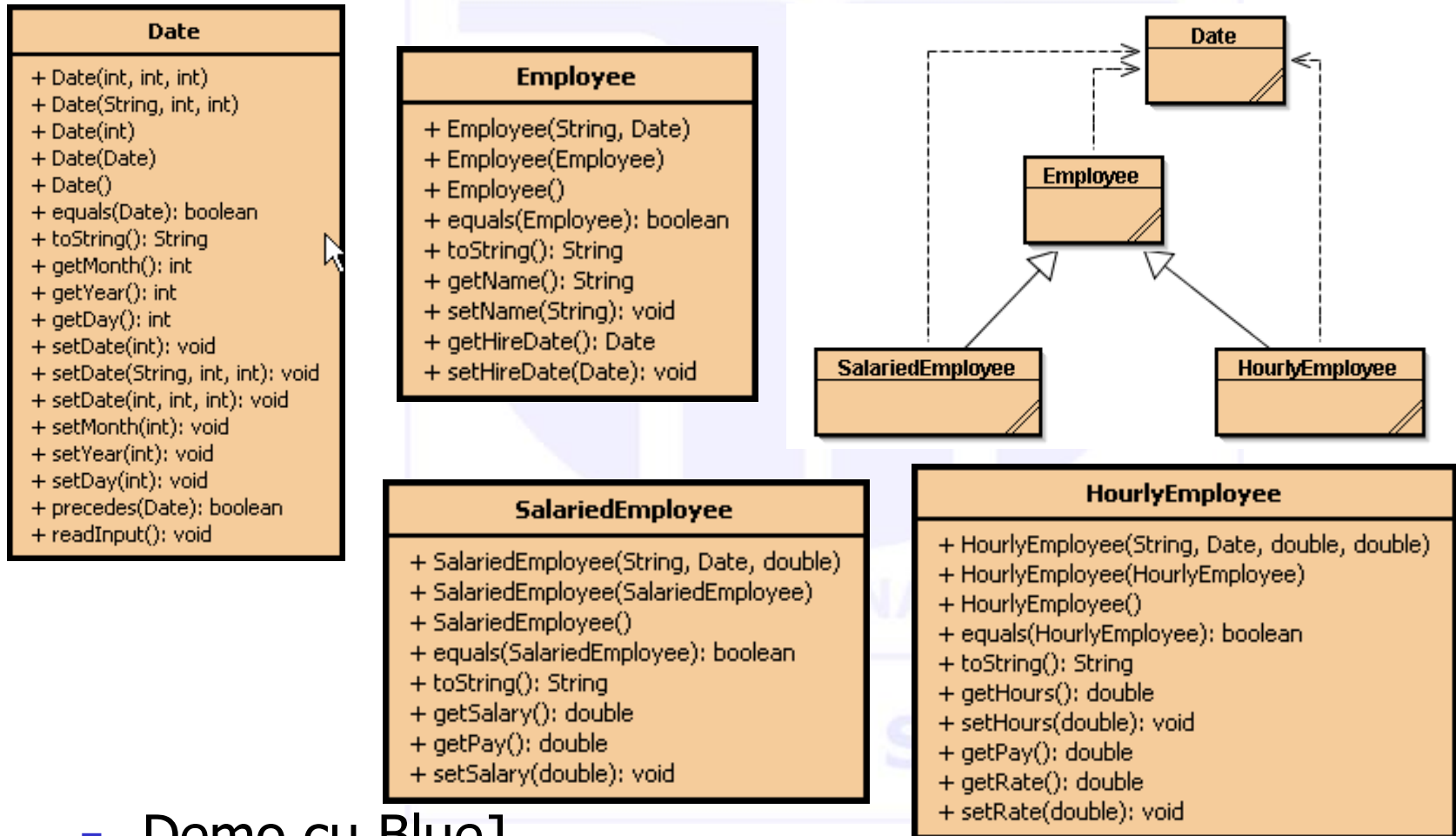


**Componentele
moștenite ale
superclasei sunt
parte a subclasei**

Instanțe



Un exemplu: Employees



■ Demo cu BlueJ



Clase derivate

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



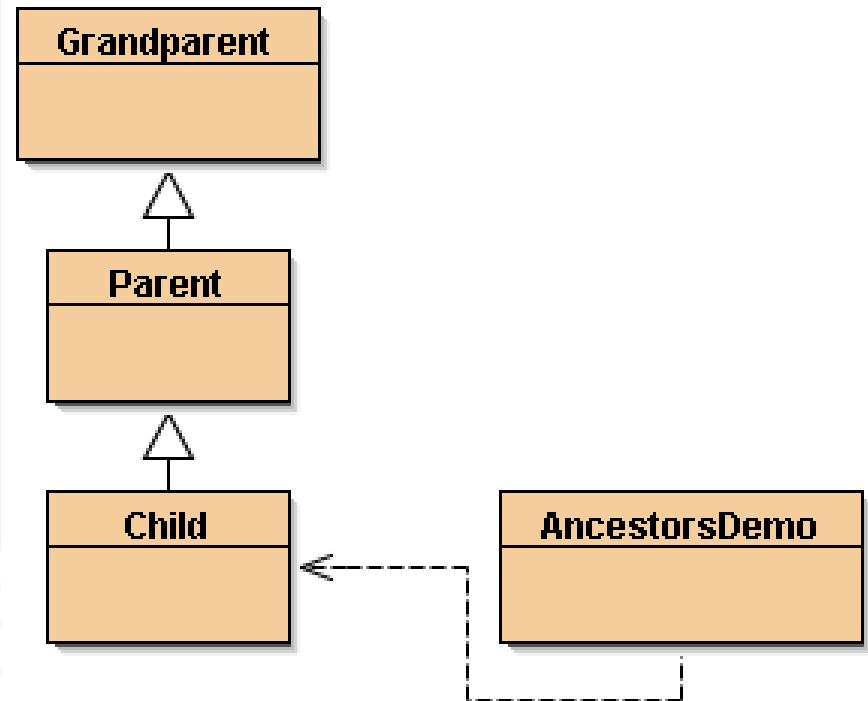
Clase derivate (subclase)

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



Clase părinți și clase copii

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





Clase abstracte

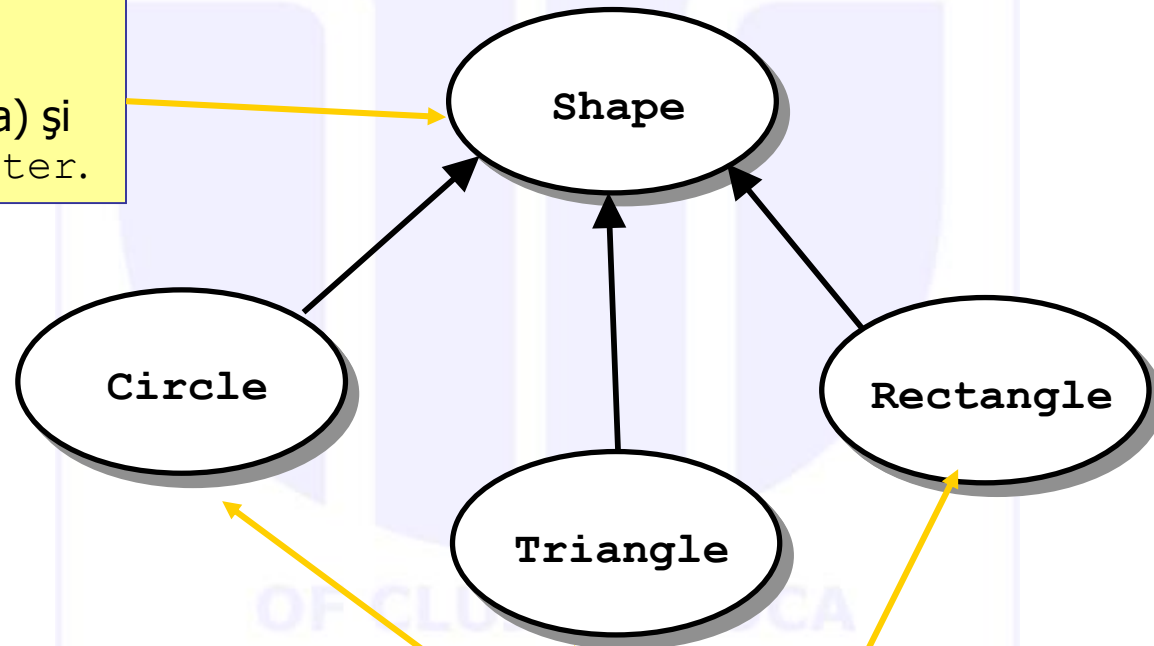
- O metodă sau o clasă abstractă se declară folosind cuvântul cheie **abstract**. De exemplu

```
public abstract double calcPay( double
hours );
```
- Dintr-o clasă abstractă nu se poate instanția nici un obiect
- Fiecare subclasă a unei clase abstracte care va fi folosită pentru a instanția obiecte trebuie să ofere implementări pentru toate metodele abstracte din superclasă.
- Clasele abstracte economisesc timp, deoarece nu trebuie să scriem cod "inutil" care n-ar fi executat niciodată.
- O clasă abstractă poate moșteni metode *abstracte*
 - dintr-o interfață sau
 - dintr-o clasă.



Exemplu: O clasă numită Shape (formă)

Superclasă: conține metodele abstracte `calculateArea` (calculează suprafața) și `calculatePerimeter`.



Subclasă: implementează metodele concrete `calculateArea` și `calculatePerimeter`.



Exemplu: O clasă numită Shape

```
/**
 * Abstract class Shape - base for inheritance for shapes
 */
public abstract class Shape {
    private static int counter;
    // Constructor
    public Shape() {
        counter++;
    }
    // calculate area
    public abstract double calculateArea();
    // calculate perimeter
    public abstract double calculatePerimeter();
    // get number of shapes
    public int getCount() {
        return counter;
    }
    protected void finalize() throws Throwable {
        counter--;
    }
}
```

Definiția superclasei.

Observați că această clasă este declarată `abstract`.

**Definiții de metode
abstracte.** Observați că este declarat doar antetul. Aceste metode **trebuie suprascrise (overridden)** în toate clasele concrete.



Exemplu: subclasa circle

```
/**  
 * Concrete class Circle - inherits from Shape  
 */  
public class Circle extends Shape {  
    private double r; // radius of circle  
    // Constructor  
    public Circle(double r) {  
        super();  
        this.r = r;  
    }  
    // calculate area  
    public double calculateArea() {  
        return Math.PI * r * r;  
    }  
    // calculate perimeter  
    public double calculatePerimeter() {  
        return 2.0 * Math.PI * r;  
    }  
    protected void finalize() throws Throwable {  
        super.finalize();  
    }  
}
```

Clasă concretă.

Clasa *nu trebuie* să conțină sau să moștenească metode abstracte. Metodele abstracte moștenite trebuie suprascrise.

Definiții de metode concrete.

Observați că aici este declarat corpul metodei.



Exemplu: subclasa Triangle

```
/**
 * Concrete class Triangle - inherits from Shape
 */
public class Triangle extends Shape {
    private double s; // side of Triangle
    // Constructor
    public Triangle(double s) {
        super();
        this.s = s;
    }
    // calculate area
    public double calculateArea() {
        return ( Math.sqrt(3.)/4 * s *s );
    }
    // calculate perimeter
    public double calculatePerimeter() {
        return 3.0 * s ;
    }
    protected void finalize() throws Throwable {
        super.finalize();
    }
}
```

Clasă concretă. Clasa *nu trebuie* să conțină sau să moștenească metode abstracte. Metodele abstracte moștenite trebuie suprascrise.

Definiții de metode concrete. Observați că corpurile metodelor sunt diferite de cele din `Circle`, dar semnăturile metodelor sunt *identice*.

Alte subclase ale lui `Shape` vor suprascrie și ele metodele abstracte `calculateArea` și `calculatePerimeter`



Exemplu: clasa TestShape

```
/**
 * Write a description of class TestShape here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class TestShape
{
    public static void main(String[] args)
    {
        // Create an array of Shapes
        Shape s[] = new Shape[2];
        // create objects
        s[0] = new Circle(2);
        s[1] = new Triangle(2);
        // Print out the number of Shapes
        System.out.println(s[0].getCount() + " shapes created");
        for (int i = 0; i < s.length; i++ ) {
            System.out.print(s[i].toString() + " ");
            System.out.print("Area = " + s[i].calculateArea());
            System.out.println(" Perimeter = " + s[i].calculatePerimeter());
        }
    }
}
```

**Creează obiecte
ale subclaselor
folosind referințe
la superclasă.**

**Apelează metodele
calculateArea și
calculatePerimeter.
Este apelată
automat versiunea
corespunzătoare a
fiecărei metode
pentru fiecare obiect.**

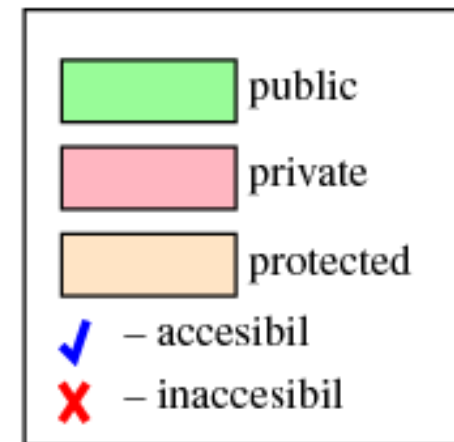
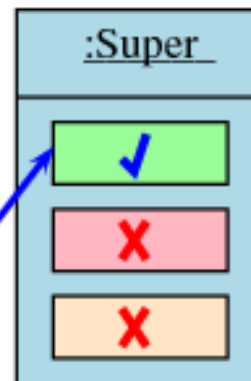


Variabile instanță

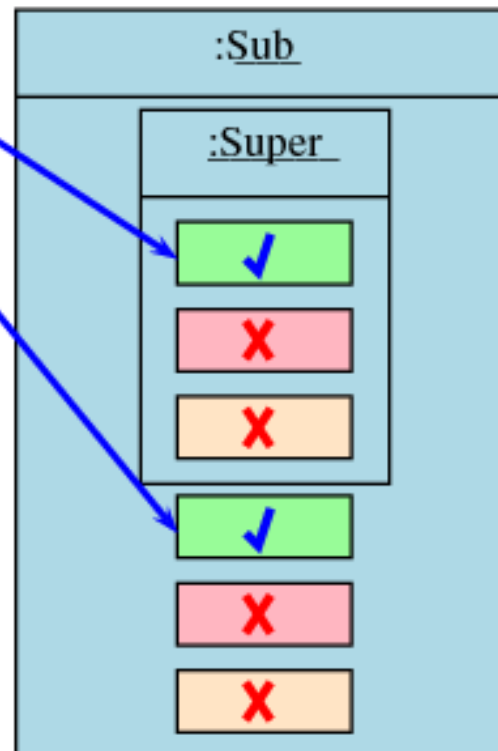
- Ca șablon general, subclasele:
 - Moștenesc capabilitățile **public** (metode)
 - Moștenesc proprietățile **private** (variabile instanță) dar nu au acces la ele
 - Moștenesc variabilele **protected** și le pot accesa
- O variabila declarată **protected** de o superclasă devine *parte a moștenirii*
 - variabila devine disponibilă pentru subclase, care o pot accesa *ca și cum ar fi proprie*
 - spre deosebire de aceasta, dacă o variabilă instanță este declarată **private** într-o superclasă, subclasele nu vor avea acces la ea
 - superclasa poate totuși oferi acces protejat la variabilele instanță private via metode *accesoare* și *mutatoare*



Accesibilitatea din
metodele Clientului



Doar membrii declarați **public** – definiți în cadrul clasei și cei moșteniți – sunt vizibili din exterior. Celelalte elemente sunt ascunse vederii din exterior





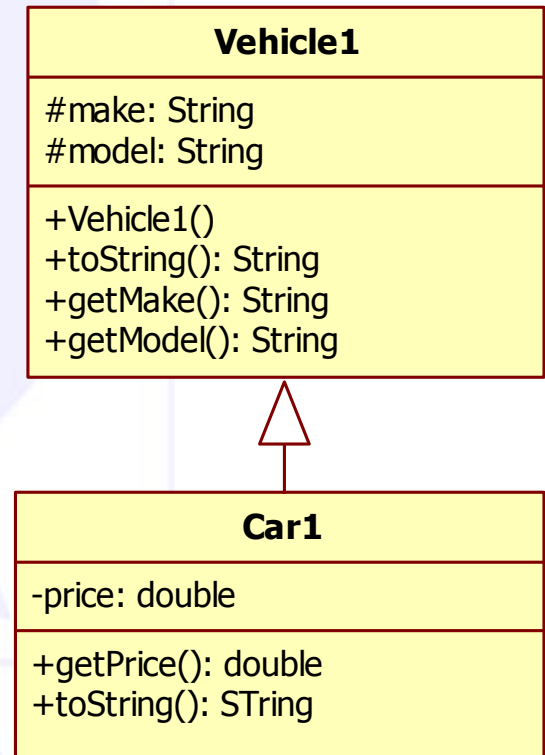
Variabile instanță **protected** fața de variabile instanță **private**

- Cum putem decide între **private** și **protected**?
 - folosiți **private** dacă doriți ca o variabilă instanță să fie *încapsulată* de către superclasă
 - d.e., uşile, ferestrele, bujiile unei maşini
 - folosiți **protected** dacă doriți ca variabila instanță să fie accesibilă subclaselor pentru a o modifica (şi nu doriți să faceți variabila mai general accesibilă prin metode accesoare/mutatoare)
 - d.e., motorul unei maşini



protected, Exemplu

```
public class Vehicle1 {  
    protected String make;  
    protected String model;  
    public Vehicle1() { make = ""; model = "";}  
    public String toString() {  
        return "Make: " + make + " Model: " + model;  
    }  
    public String getMake(){ return make;}  
    public String getModel() { return model;}  
}  
public class Car1 extends Vehicle1 {  
    private double price;  
    public Car1() { price = 0.0; }  
    public String toString() {  
        return "Make: " + make + " Model: " + model  
            + " Price: " + price;  
    }  
    public double getPrice(){ return price; }  
}
```





Suprascrierea unei definiții de metodă

- Deși o clasă derivată moștenește metode din clasa de bază, ea poate să le modifice – să le *suprascrive* dacă este necesar
 - Pentru a suprascrive o definiție de metodă, se pune pur și simplu o definiție nouă în definiția clasei, exact ca pentru orice altă metodă adăugată clasei derivate
- De obicei, tipul returnat nu poate fi schimbat la suprascrierea unei metode
- Totuși, dacă tipul este un *tip clasă*, atunci tipul returnat poate fi schimbat la acela al oricărei *clase descendente* al tipului returnat
- Acest lucru se cunoaște sub numele de *tip returnat covariant*
 - *Tipurile returnate covariant* sunt introduse în Java 5.0; ele nu sunt permise în versiuni anterioare de Java



Tipul returnat covariant

- Fiind dată următoarea clasă de bază:

```
public class BaseClass
{ . . .
    public Employee getSomeone(int someKey)
    . . .
```

- Este permisă următoarea modificare a tipului returnat în Java 5.0:

```
public class DerivedClass extends BaseClass
{ . . .
    public HourlyEmployee getSomeone(int someKey)
    . . .
```



Schimbarea permisiunii de acces a unei metode suprascrise

- Permisiunea de acces a unei metode suprascrise poate fi schimbată *de la private* în *clasa de bază* la *public* (sau alt *acces mai permisiv*) în *clasa derivată*
- Totuși, permisiunea de acces a unei metode suprascrise *nu poate fi modificată* de la public în clasa de bază *la o permisiune de acces mai restrictivă* în clasa derivată
 - Adică, putem relaxa permisiunile de acces într-o clasă derivată, nu o putem restrânge



Schimbarea permisiunii de acces a unei metode suprascrise

- Fiind dat următorul antet de metodă într-o clasă de bază:
`private void doSomething()`
- Următorul antet de metodă este valid într-o clasă derivată:
`public void doSomething()`
- Invers (din public în privat) nu se poate
- Fiind dat următorul antet de metodă într-o clasă de bază:
`public void doSomething()`
- Antetul de metodă următor *nu* este valid într-o clasă derivată:
`private void doSomething()`



Capcană: Suprascriere fața de supraîncărcare

- Nu confundați *suprascrierea (overriding)* unei metode într-o clasă derivată cu *supraîncărcarea (overloading)* numelui unei metode
 - Când o metodă este *suprascrisă*, noua definiție de metodă dată în clasă derivată are *exact același număr și tipuri de parametri ca în clasă de bază*
 - Când o metodă este într-o clasă derivată are o *semnătură diferită* în comparație cu metoda din clasă de bază, atunci avem de-a face cu *supraîncărcarea*
 - Observați că atunci când *clasă derivată suprascrie* metoda originală, *ea totuși moștenește și metoda originală* din clasă de bază



Modificatorul **final**

- Dacă se pune modificatorul **final** în fața definiției unei *metode*, atunci metoda respectivă *nu poate fi redefinită* într-o clasă derivată
- Dacă modificatorul **final** este pus în fața definiției unei *clase*, atunci clasa respectivă *nu mai poate fi folosită pe post de clasă de bază* pentru a deriva alte clase



Constructorul **super**

- O clasă derivată folosește un constructor al clasei de bază pentru a inițializa toate datele moștenite din clasa de bază

- Pentru a invoca un constructor al clasei de bază, se folosește o sintaxă specială:

```
public DerivedClass(int p1, int p2, double p3)
{
    super(p1, p2);
    instanceVariable = p3;
}
```

- În exemplul de mai sus, **super(p1, p2);** este un apel al constructorului clasei de bază



Constructorul **super**

- Un apel al unui constructor al clasei de bază nu poate folosi numele clasei respective, ci folosește în schimb cuvântul cheie **super**
- Apelul lui **super** trebuie să fie întotdeauna prima acțiune efectuată în definiția unui constructor
- Nu se pot folosi *variabile instanță* ca argumente ale lui **super**
 - Oare de ce acest lucru nu este permis?



Constructorul **super**

- Dacă într-o clasă derivată nu este prezentă o invocare a lui **super**, atunci constructorul fără argumente al clasei de bază va fi apelat automat
 - Aceasta poate cauza o eroare dacă clasa de bază nu are definit un constructor fără argumente
- Cum variabilele instanță moștenite ar trebui inițializate, iar menirea constructorului clasei de bază este să facă acest lucru, *întotdeauna ar trebui folosit un apel explicit al lui **super***



Constructori și subclase

- Cuvântul cheie **this** este folosit pentru a invoca un alt constructor al aceleiași clase. Exemplu:

```
public class Circle extends ClosedFigure
{
    private int radius;
    public Circle(int radius)
    {
        super();
        this.radius = radius;
    }
    public Circle()
    {
        this(1); // invoca constructorul cu argumentul 1
    }
    ...
}
```



Accesul la o metodă redefinită din clasa de bază

- În definiția unei metode dintr-o clasă derivată, versiunea suprascrisă a unei metode a clasei de bază poate totuși fi invocată
 - Pur și simplu prefixați numele metodei cu **super** și un punct
- ```
public String toString()
{
 return (super.toString() + "$" + wageRate);
}
```
- Cu toate acestea, la folosirea unui obiect al clasei derivate în afara definiției clasei, nu există nici o cale de invocare a versiunii unei metode suprascrise din clasa sa de bază



# Nu puteți folosi mai mulți **super**

- **super** poate fi folosit pentru a invoca o metodă doar dintr-un părinte (strămoș direct)
  - Repetarea lui **super** nu va invoca o metodă din vreo altă clasă strămoș
- Spre exemplu, dacă clasa **Employee** ar fi fost derivată din clasa **Person**, iar clasa **HourlyEmployee** ar fi fost derivată din clasa **Employee**, nu ar fi fost posibil să invocăm metoda **toString** a clasei **Person** dintr-o metodă a clasei **HourlyEmployee**

**super.super.toString() // ILEGAL!**



# Constructorul `this`

- În definiția unui constructor pentru o clasă, `this` poate fi folosit ca nume pentru invocarea unui alt constructor din aceeași clasă
  - Se aplică aceleași restricții de folosire ca pentru `super` și pentru `this`
- Dacă este necesar să fie apelat atât `super` cât și `this`, trebuie efectuat mai întâi apelul care folosește `this`, iar apoi constructorul invocat cu `this` trebuie să invoce `super` ca primă acțiune a sa



# Constructorul `this`

- Adesea, un constructor fără argumente folosește `this` pentru a invoca un constructor cu valori explicite
  - Constructor fără argumente (invocă un constructor cu valori explicite folosind `this` și argumente implicite):

```
public ClassName() {
 this(argument1, argument2);
}
```
  - Constructor cu valori explicite (primește valori implicite):

```
public ClassName(type1 param1, type2 param2) {
 . . .
}
```



# Constructorul `this`

```
public HourlyEmployee()
{
 this("No name", new Date(), 0d, 0d);
}
```

- Constructorul de mai sus va determina invocarea constructorului cu următorul antet:

```
public HourlyEmployee(String theName, Date
 theDate, double theWageRate, double
 theHours)
```



# Un obiect al unei clase derivate are mai mult de un tip

---

- Un obiect al unei clase derivate are tipul clasei derivate și are și tipul clasei de bază
- Mai general, un obiect al unei *clase derivate* are *tipul fiecăruia dintre clasele din ascendența sa*
  - De aceea, un obiect dintr-o clasă derivată poate fi asignat unei variabile de tipul oricărui părinte/strămoș al său



# Un obiect al unei clase derivate are mai mult de un tip

---

- Un obiect al unei clase derivate poate fi folosit ca parametru în locul oricăruia dintre clasele părinte/strămoș ale sale
- De fapt, un obiect dintr-o clasă derivată poate fi folosit în orice loc în care se poate folosi un obiect de tipurile părintelui/strămoșilor săi
- Observați, totuși, că relația nu merge și invers
  - Un tip strămoș/părinte nu poate fi niciodată folosit în locul unuia dintre tipurile derivate din el



# Capcană: termenii "subclasă" și "superclasă"

- Uneori termenii *subclasă* și *superclasă* sunt inversați din greșeală
  - O *superclasă* / clasă de bază este *mai generală* și mai cuprinzătoare, dar *mai puțin complexă*
  - O *subclasă* / clasă derivată este *mai specializată*, mai puțin cuprinzătoare și *mai complexă*
    - Pe măsură ce sunt adăugate mai multe variabile și metode, numărul obiectelor care pot satisface definiția clasei devine mai restrâns



# Folosirea claselor abstracte

- O clasă abstractă contribuie la implementarea subclaselor sale concrete.
- Este folosită pentru a exploata polimorfismul.
  - Pentru funcționalitatea specificată în clasele părinte se pot da implementări corespunzătoare fiecărei subclase concrete.
- Clasele abstracte trebuie să fie stabile.
  - Orice schimbare într-o clasă abstractă se propagă la subclase și la clienții lor.
- O clasă concretă poate extinde doar o singură clasă (abstractă)



# Interfețe, clase abstracte și clase concrete

- ○ ***interfață***
  - se folosește pentru a specifica funcționalitatea cerută de un client.
- ○ ***clasă abstractă***
  - oferă o bază pe care să se construiască clase server concrete.
- ○ ***clasă concretă***
  - completează implementarea server-ului care a fost specificată de o interfață;
  - furnizează obiecte la momentul execuției;
  - nu este, în general, potrivită ca bază pentru extindere.

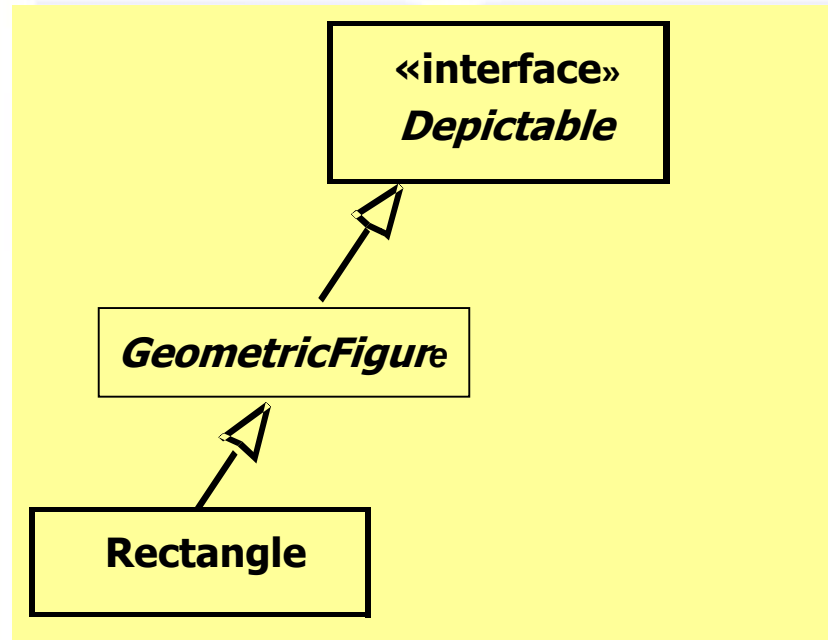


# Folosirea interfețelor

- Interfețele sunt abstracte prin definiție.
  - separă implementarea unui obiect de specificarea sa.
  - nu fixează nici un aspect al unei implementări.
- O clasă poate implementa mai mult de o interfață.
- Interfețele permit o folosire mai generalizată a polimorfismului; instanțe din clase relativ neînrudite pot fi tratate ca identice într-un scop anume.
- În programe, folosiți
  - *interfețe pentru a partaja comportament comun.*
  - *moștenirea pentru a partaja cod comun.*



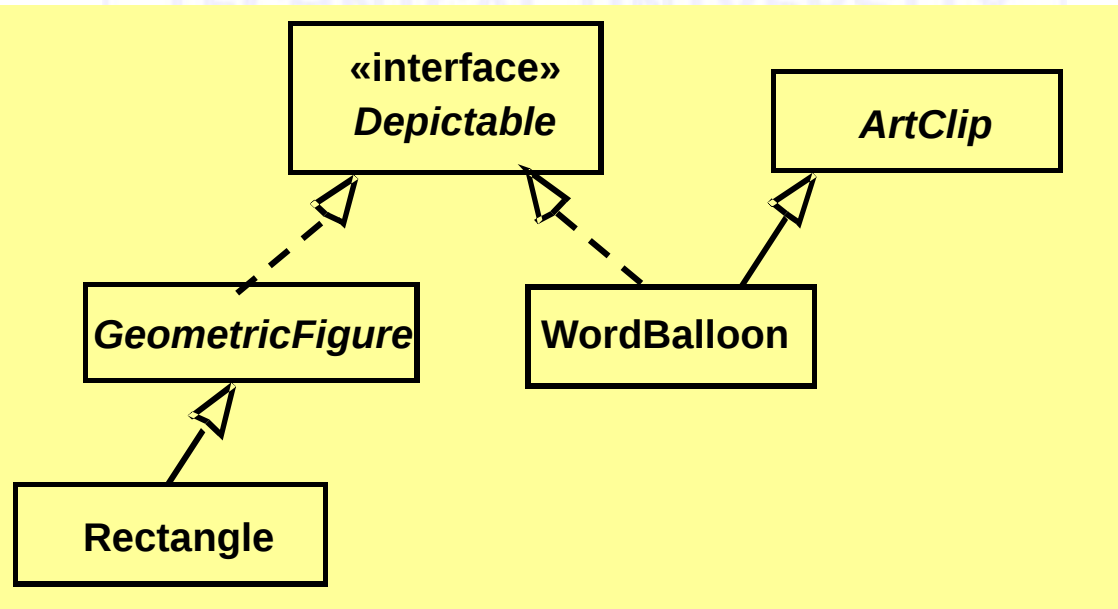
# Interfețe, clase abstracte și clase concrete



```
public boolean isIn (Location point, Depictable figure)
{
 Location l = figure.getLocation();
 Dimension d = figure.getDimension();
 ...
}
```



# Interfețe, clase abstracte și clase concrete



```
public boolean isIn (Location point, Depictable figure)
{
 Location l = figure.getLocation();
 Dimension d = figure.getDimension();
 ...
}
```

- Pot fi folosite instanțe ale lui **WordBalloon** ca parametri ai lui **isIn**.



# Exemplu de polimorfism

```
public class Base {
 protected int theInt = 100;

 ...
 public void printTheInt() {
 System.out.println(theInt);
 }
}
```

```
public class Doubler extends Base {
 ...
 public void printTheInt() {
 System.out.println(theInt*2);
 }
}
```

```
public class Tripler extends Base {
 ...
 public void printTheInt() {
 System.out.println(theInt*3);
 }
}
```

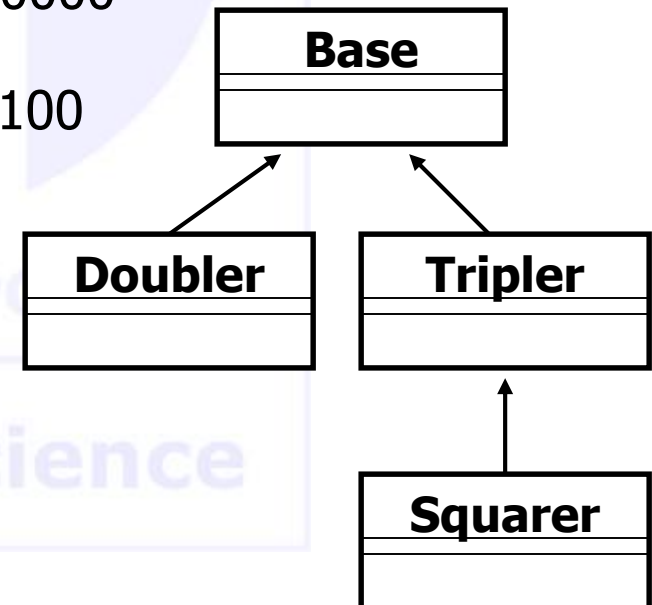
```
public class Squarer extends Tripler {
 ...
 public void printTheInt() {
 System.out.println(theInt*theInt);
 }
}
```



# Exemplu de polimorfism

```
Base theBase;
theBase = new Base();
theBase.printTheInt();
theBase = new Doubler();
theBase.printTheInt();
theBase = new Tripler();
theBase.printTheInt();
theBase = new Squarer();
theBase.printTheInt();
theBase = new Base();
theBase.printTheInt();
...
```

→ 100  
→ 200  
→ 300  
→ 10000  
→ 100



## Polimorfismul subtipurilor

Pe măsură ce apare legarea dinamică comportamentul (adică metodele) urmează obiectele



# Polimorfism

- O variabilă polimorfică poate părea a-și schimba tipul prin legare dinamică.
- Compilatorul înțelege întotdeauna tipul unei variabile potrivit declarației.
- Compilatorul permite o anumită flexibilitate prin modul de conformare la tip.
- La execuție, comportamentul unui apel de metodă depinde de tipul de object, nu de variabilă.
- Exemplu:

```
Base theBase;
theBase = new Doubler();
theBase = new Squarer();
theBase.printTheInt();
```



# De ce este util polimorfismul?

- Polimorfismul permite unei superclase sa rețină ceea ce este comun, lăsând specificitatea sa fie tratata de subclase.

Sa presupunem ca AView include o metoda **calcArea**, ca mai sus.

Atunci ARectangle trebuie scris ca ...

```
public class AView {
 ...
 public double calcArea() {
 return 0.0;
 }
}
```

```
public class ARectangle extends AView {
 ...
 public double calcArea() {
 return getWidth() * getHeight();
 }
}
```

iar AOval trebuie scris ca ...

```
public class AOval extends AView {
 ...
 public double calcArea() {
 return getWidth()/2. * getHeight()/2. * Math.PI;
 }
}
```

Considerați acum

```
public double coverageCost(AView v, double costPerSqUnit) {
 return v.calcArea() * costPerSqUnit;
}
```



# Moștenirea din clasa *Object*

- *Fiecare clasă din Java extinde o altă clasă.*
- Dacă nu se specifică, atunci extinde automat clasa numită ***Object***.
- Toate clasele Java sunt într-o ierarhie care la rădăcină clasa numită *Object* is the root of the hierarchy.
- Unele clase extind clasa *Object* direct, altele sunt subclase ale lui *Object* mai jos în ierarhie.
- Clasa *Object* este definită în **`java.lang`**



# Cum se decide care este metoda de executat

1. Dacă există o metodă concretă în clasa curentă, se execută aceea.
2. În caz contrar, se verifică în superclasa directă dacă există acolo o metodă; dacă da, se execută.
3. Se repetă pasul 2, verificând în sus pe ierarhie până când se găsește o metodă concretă și se execută.
4. Dacă nu s-a găsit nici o metodă, atunci este eroare
  - În Java și C++ programul nu se compilează



## Legarea dinamică

- Apare atunci când decizia privind metoda de executat nu se poate lua decât la execuția programului
  - Este nevoie de ea atunci când:
    - Variabila este declarată ca având tipul superclasei și
    - Există mai mult de o metoda polimorfică care se poate executa între tipul variabilei și subclasele sale



# De studiat

---

- Eckel: capitolele 8, 9
- Barnes: capitolele 8, 9
- Deitel: capitolele 9, 10



# Rezumat

- Pachete Java
  - organizarea claselor înrudite
  - API-uri Java
- Moștenire
  - clasă de bază, superclasă
  - clasă derivată, subclasă
  - ierarhii
  - constructorii **super** și **this**
- accesul la variabilele instanță
- Supraîncarcarea metodelor
- Interfețe, clase abstracte, clase concrete
- Polimorfism