Programare orientată pe obiecte

1. Pachete (packages)
2. Moştenire

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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

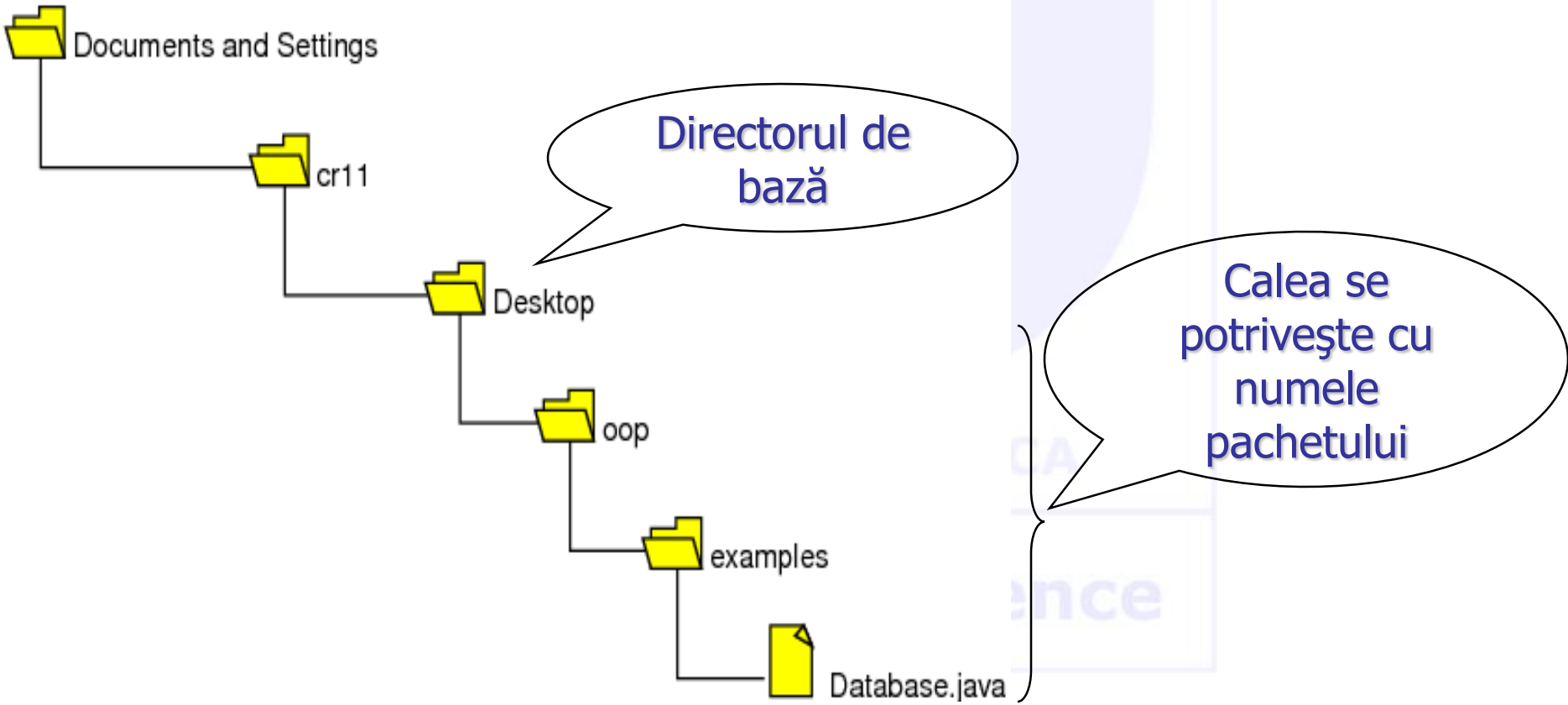
- Numele căii trebuie să se potrivească cu numele pachetului
  `oop/examples/Database.java`
- Numele căii începe cu calea spre clase

```
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

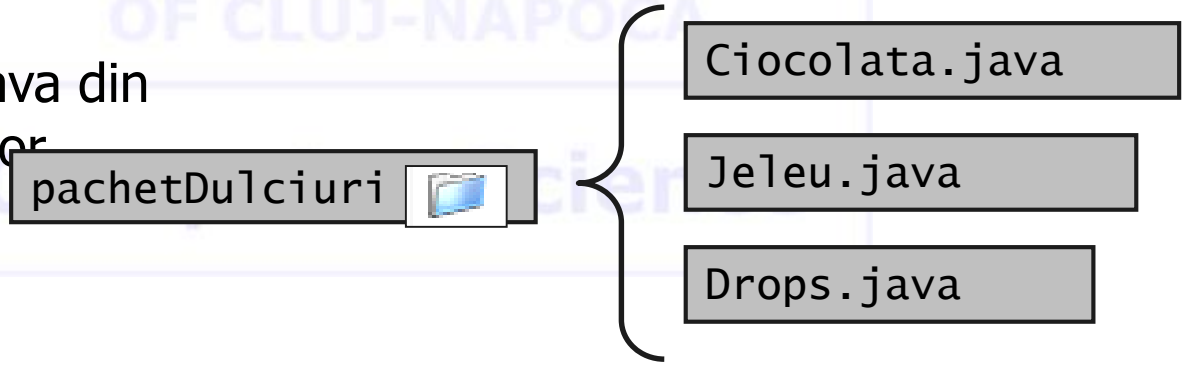
1) Puneţi o linie cu numele pachetului la începutul fiecărei clase.

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

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

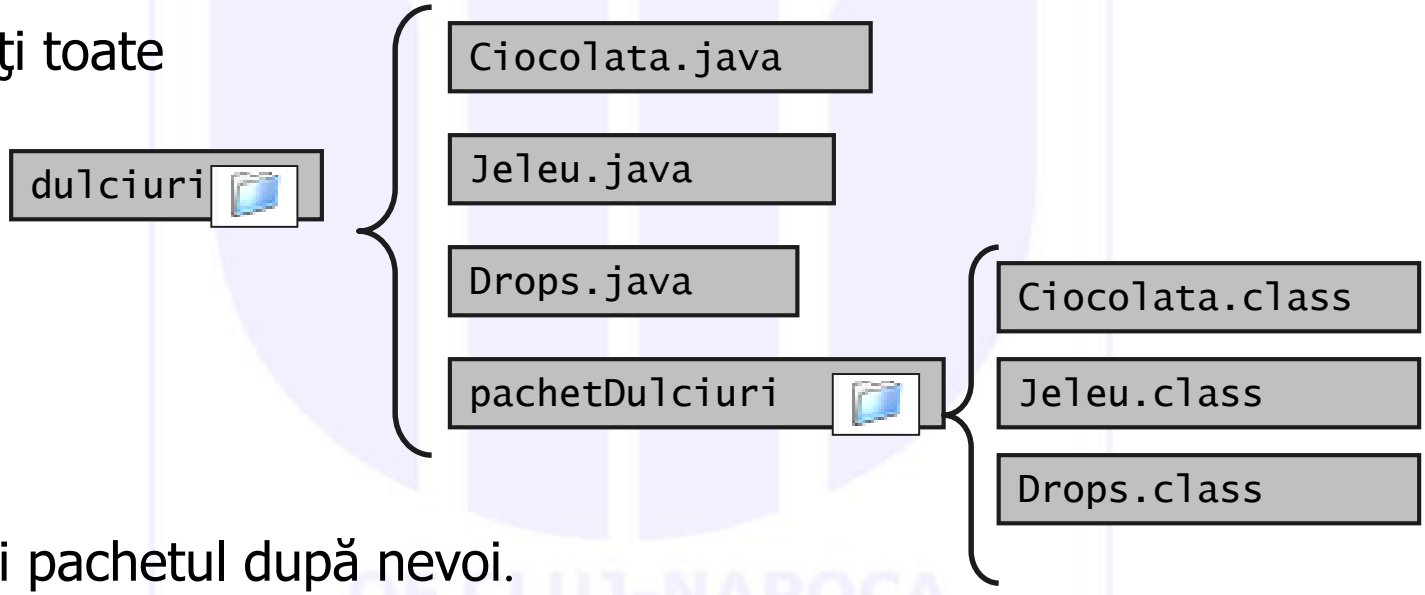
```
package pachetDulciuri;
public class Drops {
   . . .
}
```

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?

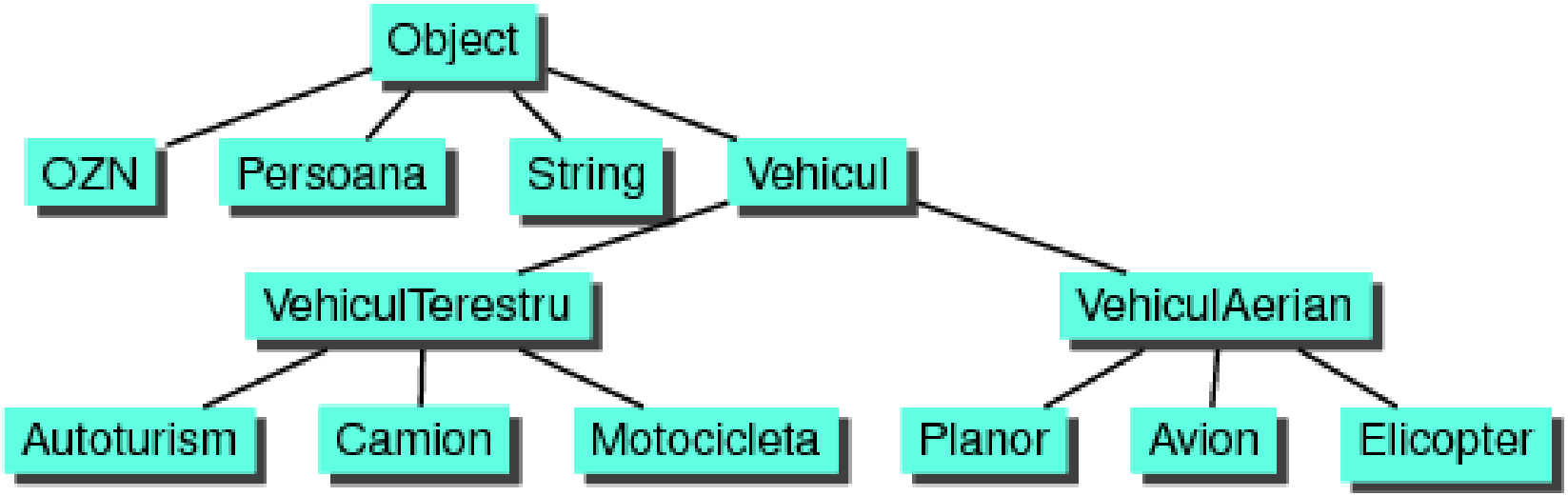
- Putem scrie clase de la început – fără a refolosi nimic (o extremă).
  - Ceea ce unii programatori doresc să facă întotdeauna
- Putem găsi o clasa existentă care se potriveşte exact cerinţelor problemei (o altă extremă).
  - Cel mai uşor lucru pentru programator
- Putem construi clase din clase existente 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



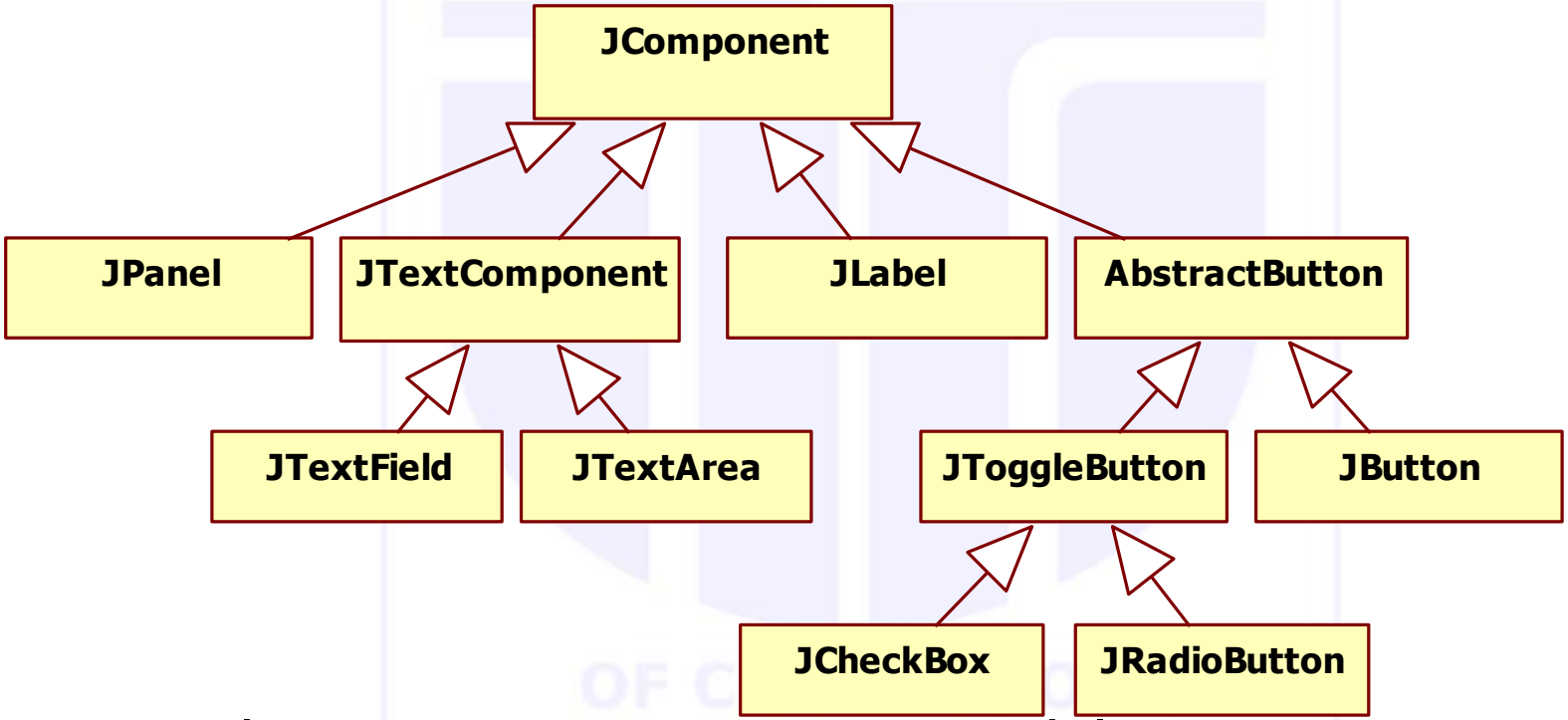
Introducere în moştenire

- 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

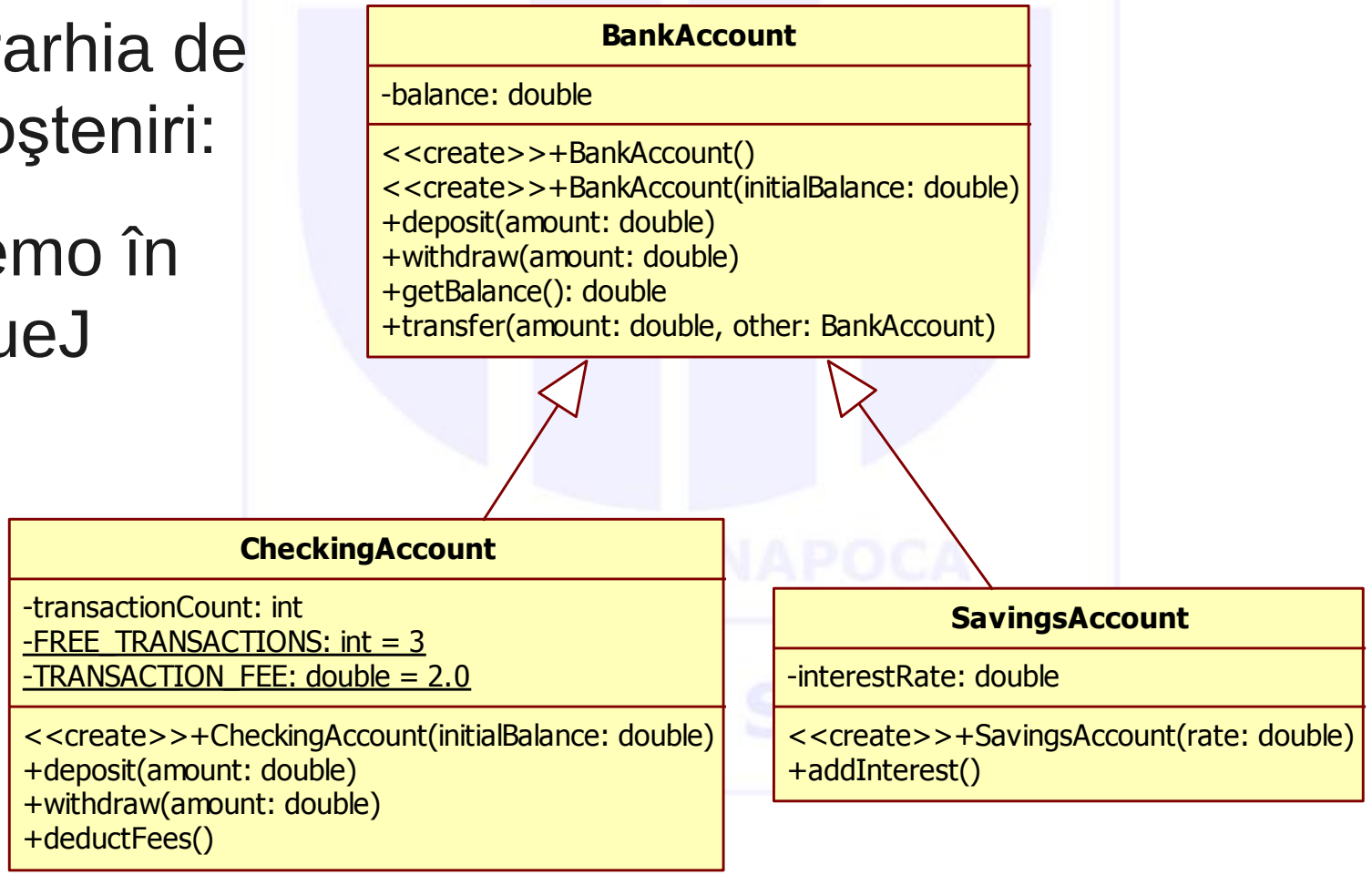
Exemplu. O parte a ierarhiei componentelor interfeţei utilizator Swing



- 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`**

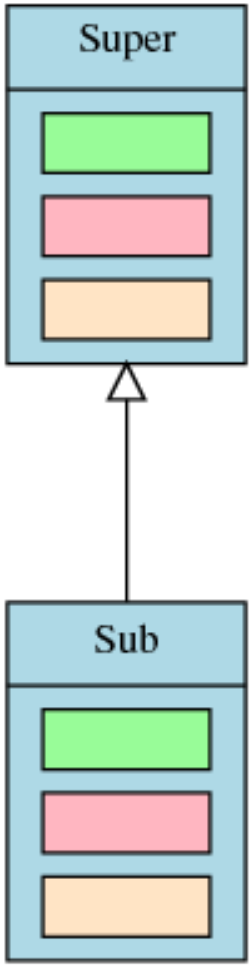
Clase derivate

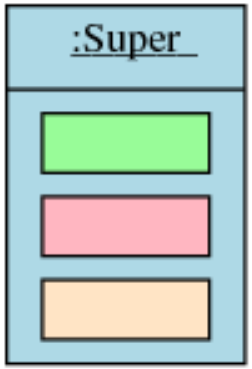
- Cum un cont de economii este un cont bancar, el este definit ca o clasă *derivată* a clasei **`BankAccount`**
  - O clasă *derivată* se defineşte prin adăugarea de variabile şi/sau metode la o clasă existentă
  - Fraza **`extends BaseClass`** trebuie adăugată în definiţia clasei derivate:

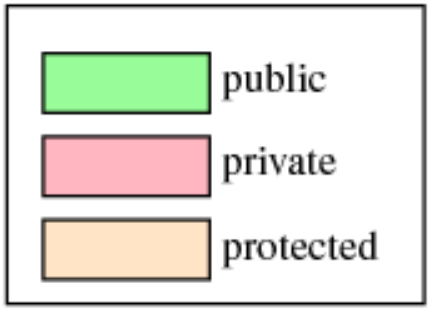
```
public class SavingsAccount extends BankAccount
```

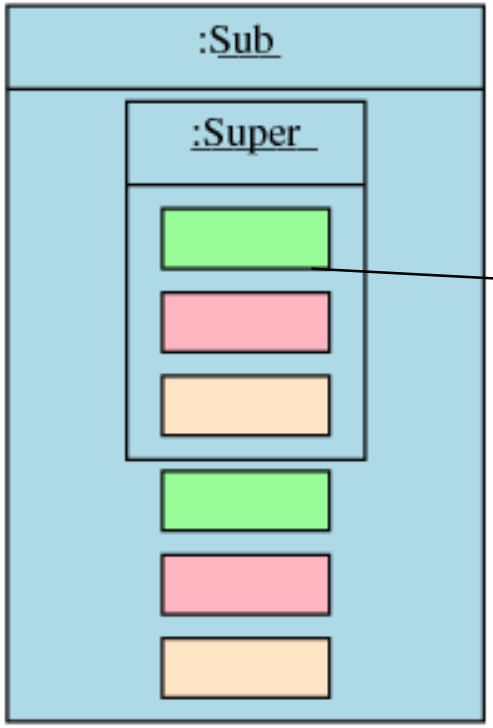
- Sintaxa pentru moştenire:

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









Componentele moştenite ale superclasei sunt parte a subclasei

Ierarhia de clase

Instanţe

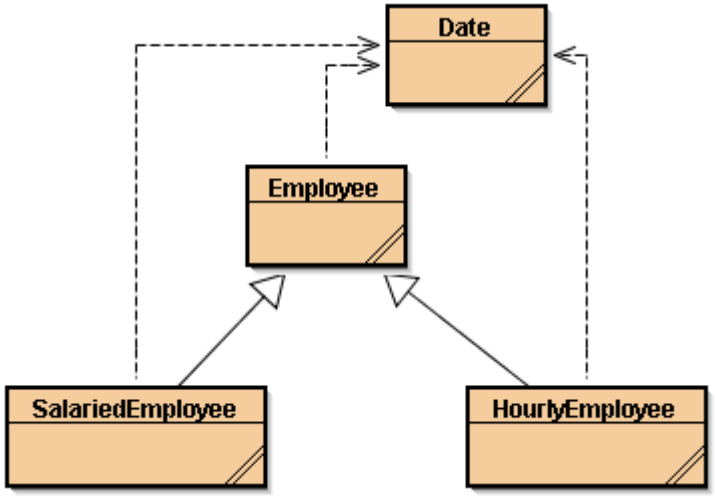
Un exemplu: Employees

Date

+ Date(int, int, int)
+ Date(String, int, int)
+ Date(int)
+ Date(Date)
+ Date()
+ equals(Date): boolean
+ toString(): String
+ getMonth(): int
+ getYear(): int
+ getDay(): int
+ setDate(int): void
+ setDate(String, int, int): void
+ setDate(int, int, int): void
+ setMonth(int): void
+ setYear(int): void
+ setDay(int): void
+ precedes(Date): boolean
+ readInput(): void

Employee

+ Employee(String, Date)
+ Employee(Employee)
+ Employee()
+ equals(Employee): boolean
+ toString(): String
+ getName(): String
+ setName(String): void
+ getHireDate(): Date
+ setHireDate(Date): void



SalariedEmployee

+ SalariedEmployee(String, Date, double)
+ SalariedEmployee(SalariedEmployee)
+ SalariedEmployee()
+ equals(SalariedEmployee): boolean
+ toString(): String
+ getSalary(): double
+ getPay(): double
+ setSalary(double): void

HourlyEmployee

+ HourlyEmployee(String, Date, double, double)
+ HourlyEmployee(HourlyEmployee)
+ HourlyEmployee()
+ equals(HourlyEmployee): boolean
+ toString(): String
+ getHours(): double
+ setHours(double): void
+ getPay(): double
+ getRate(): double
+ setRate(double): void

- 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

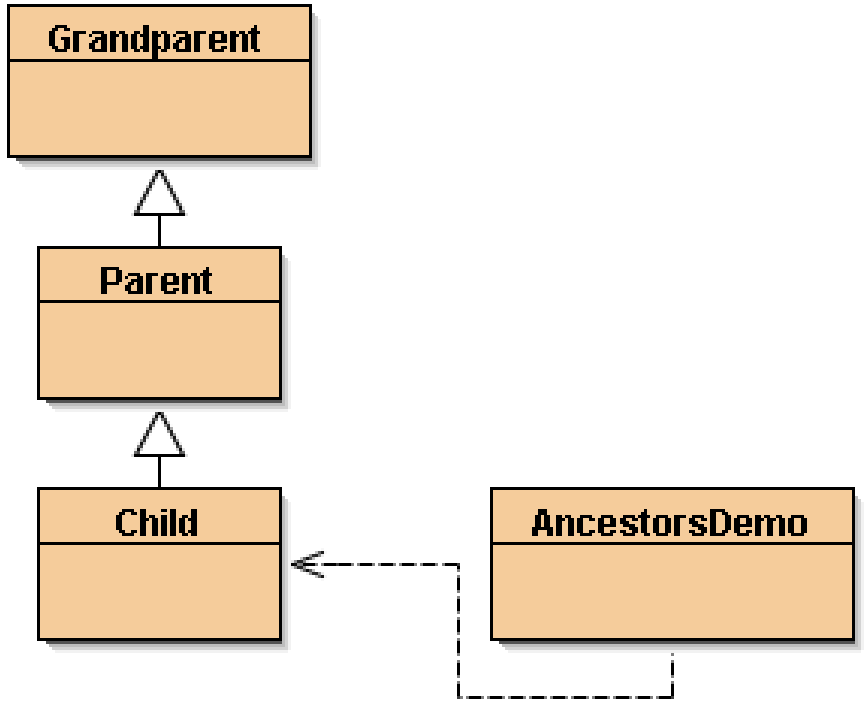
- Aşa cum moşteneşte variabilele clasei **`Employee`**, clasa **`HourlyEmployee`** moşteneşte şi toate metodele superclasei
  - Clasa **`HourlyEmployee`** moşteneşte metodele **`getName`**, **`getHireDate`**, **`setName`**, şi **`setHireDate`** (obţine numele, obţine data angajării, setează numele şi 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`**



Clase abstracte

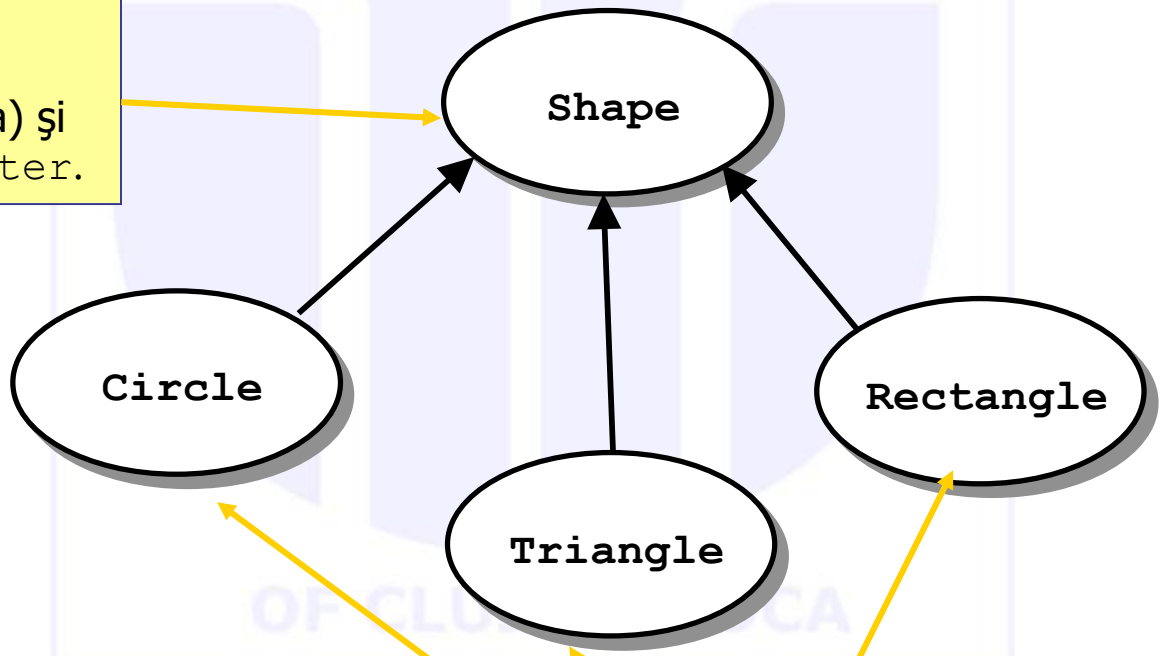
- O metodă sau o clasă abstractă se declară folosind cuvântul cheie **`abstract.`** De exemplu

```
public abstract double calcPay( double
hours );
```

- Dintr-o clasă abstractă nu se poate instanţia nici un obiect
- Fiecare subclasă a unei clase abstracte care va fi folosită pentru a instanţia 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 clasa abstractă poate moşteni metode *abstracte*
  - dintr-o interfaţă sau
  - dintr-o clasă.

Exemplu: O clasă numită `Shape` (formă)

Superclasă: conţine metodele abstracte `calculateArea` (calculează suprafaţa) şi `calculatePerimeter`.



Subclase: implementează metodele concrete `calculateArea` şi `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--;
    }
}
```

Definiţia superclasei. Observaţi că această clasă este declarată `abstract`.

Definiţii de metode abstracte. Observaţi 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 conţină sau să moştenească metode abstracte. Metodele abstracte moştenite trebuie suprascrise.

Definiţii de metode concrete. Observaţi 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 conţină sau să moştenească metode abstracte. Metodele abstracte moştenite trebuie suprascrise.

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

Alte subclase ale lui **`Shape`** vor suprascrie şi ele metodele abstracte **`calculateArea`** şi **`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());
        }
    }
}
```

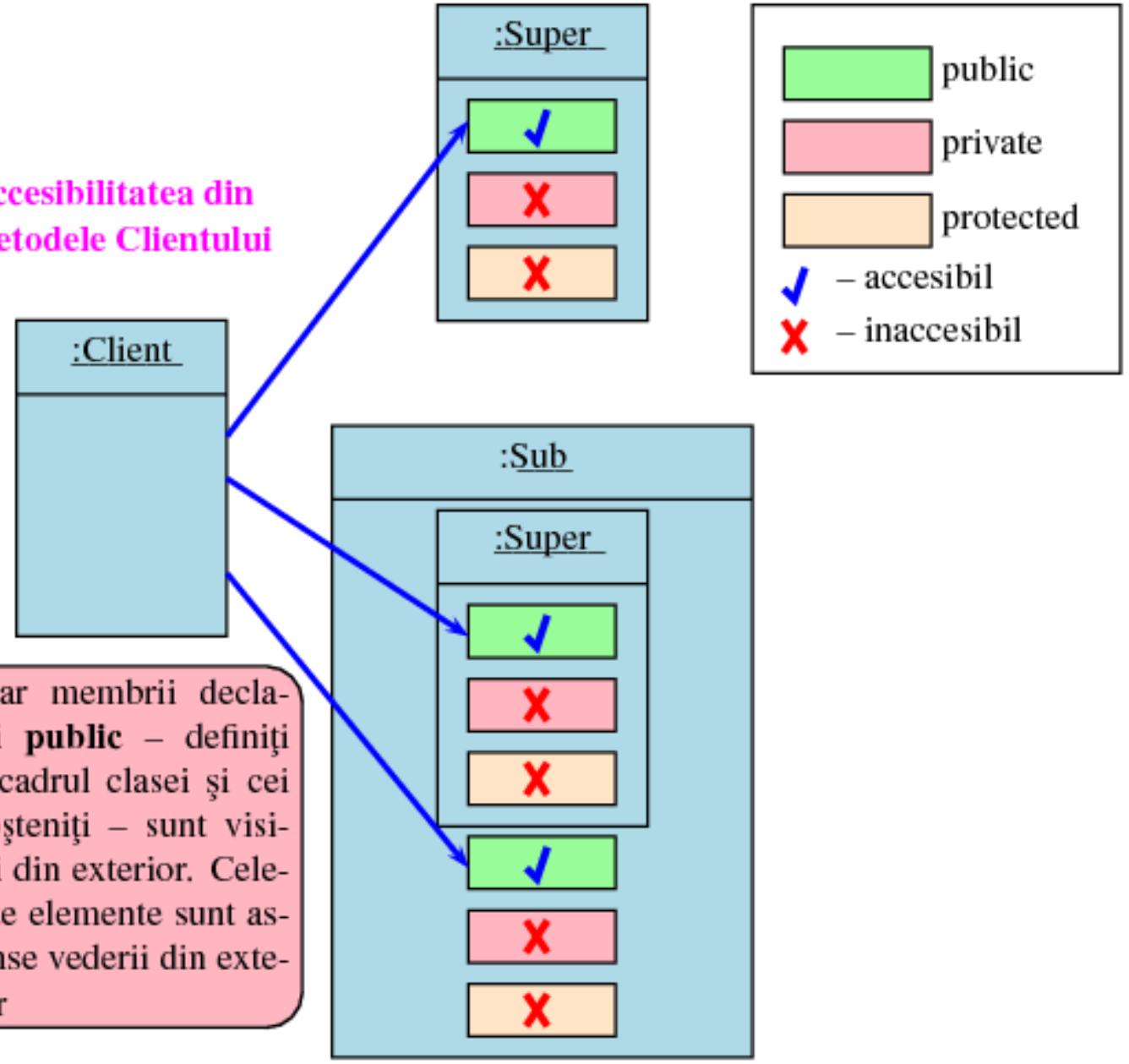
Creează obiecte ale subclaselor folosind referinţe la superclasă.

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

Variabile instanţă

- Ca şablon general, subclasele:
  - Moştenesc capabilităţile **`public`** (metode)
  - Moştenesc proprietăţile **`private`** (variabile instanţă) dar nu au acces la ele
  - Moştenesc variabilele **`protected`** şi le pot accesa
- O variabila declarată **`protected`** de o superclasă devine *parte a moştenirii*
  - variabila devine disponibilă pentru subclase, care o pot accesa *ca şi cum ar fi proprie*
  - spre deosebire de aceasta, dacă o variabilă instanţă este declarată **`private`** într-o superclasă, subclasele nu vor avea acces la ea
    - superclasa poate totuşi oferi acces protejat la variabilele instanţă private via metode *accesoare* şi *mutatoare*

Accesibilitatea din metodele Clientului

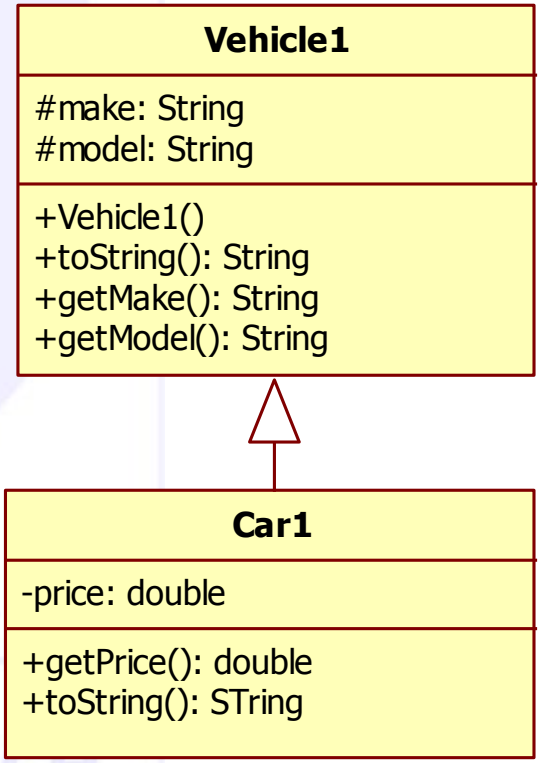


Variabile instanţă `protected` faţa de variabile instanţă `private`

- Cum putem decide între **`private`** şi **`protected`**?
  - folosiţi **`private`** dacă doriţi ca o variabilă instanţă să fie *încapsulată* de către superclasă
    - d.e., uşile, ferestrele, bujiile unei maşini
  - folosiţi **`protected`** dacă doriţi ca variabila instanţă să fie accesibilă subclaselor pentru a o modifica (şi nu doriţi să faceţi 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 definiţii de metodă

- Deşi o clasă derivată moşteneşte metode din clasa de bază, ea poate să le modifice – să le *suprascrie* dacă este necesar
  - Pentru a suprascrie o definiţie de metodă, se pune pur şi simplu o definiţie nouă în definiţia clasei, exact ca pentru orice altă metodă adăugată clasei derivate
- De obicei, tipul returnat nu poate fi schimbat la suprascrierea unei metode
- Totuşi, dacă tipul este un *tip clasă*, atunci tipul returnat poate fi schimbat la acela al oricărei *clase descendente* al tipului returnat
- Acest lucru se cunoaşte 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*
- Totuşi, 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 faţa de supraîncărcare

- Nu confundaţi *suprascrierea (overriding)* unei metode într-o clasă derivată cu *supraîncărcarea (overloading)* numelui unei metode
  - Când o metodă este *suprascrisă*, noua definiţie de metodă dată în clasa derivată are *exact acelaşi număr şi tipuri de parametri ca în clasa de bază*
  - Când o metodă este într-o clasă derivată are o *semnătură diferită* în comparaţie cu metoda din clasa de bază, atunci avem de-a face cu *supraîncărcarea*
  - Observaţi că atunci când *clasa derivată suprascrie* metoda originală, *ea totuşi moşteneşte şi metoda originală* din clasa de bază

Modificatorul `final`

- Dacă se pune modificatorul **`final`** în faţa definiţiei unei *metode*, atunci metoda respectivă *nu poate fi redefinită* într-o clasă derivată

- Dacă modificatorul **`final`** este pus în faţa definiţiei 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 foloseşte un constructor al clasei de bază pentru a iniţializa toate datele moştenite din clasa de bază
  - Pentru a invoca un constructor al clasei de bază, se foloseşte 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 foloseşte în schimb cuvântul cheie **`super`**
- Apelul lui **`super`** trebuie să fie întotdeauna prima acţiune efectuată în definiţia unui constructor
- Nu se pot folosi *variabile instanţă* 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 instanţă moştenite ar trebui iniţializate, iar menirea constructorului clasei de bază este să facă acest lucru, *întotdeauna ar trebui folosit un apel explicit al lui* **`super`**

Constructori şi subclase

- Cuvântul cheie **`this`** este folosit pentru a invoca un alt constructor al aceleiaşi 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 definiţia unei metode dintr-o clasă derivată, versiunea suprascrisă a unei metode a clasei de bază poate totuşi fi invocată
  - Pur şi simplu prefixaţi numele metodei cu **super** şi un punct

```
public String toString()
{
   return (super.toString() + "$" + wageRate);
}
```

- Cu toate acestea, la folosirea unui obiect al clasei derivate în afara definiţiei clasei, nu există nici o cale de invocare a versiunii unei metode suprascrise din clasa sa de bază

Nu puteţi folosi mai mulţi `super`

- **`super`** poate fi folosit pentru a invoca o metodă doar din-tr-un părinte (strămoş direct)
  - Repetarea lui **`super`** nu va invoca o metodă din vreo altă clasă strămoş
- 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 definiţia unui constructor pentru o clasă, **`this`** poate fi folosit ca nume pentru invocarea unui alt constructor din aceeaşi clasă
  - Se aplică aceleaşi restricţii de folosire ca pentru **`super`** şi pentru **`this`**

- Daca este necesar să fie apelat atât **`super`** cât şi **`this`**, trebuie efectuat mai întâi apelul care foloseşte **`this`**, iar apoi constructorul invocat cu **`this`** trebuie să invoce **`super`** ca primă acţiune a sa

Constructorul `this`

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

```
public ClassName(){
   this(argument1, argument2);
}
```

  - Constructor cu valori explicite (primeşte 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 şi are şi tipul clasei de bază
- Mai general, un obiect al unei *clase derivate* are *tipul fiecăruia dintre clasele din ascendenţa sa*
  - De aceea, un obiect dintr-o clasă derivată poate fi asignat unei variabile de tipul oricărui părinte/strămoş 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/strămoş 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/strămoşilor săi
- Observaţi, totuşi, că relaţia nu merge şi invers
  - Un tip strămoş/părinte nu poate fi niciodată folosi în locul unuia dintre tipurile derivate din el

Capcană: termenii "subclasă" şi "superclasă"

- Uneori termenii *subclasă* şi *superclasă* sunt inversaţi din greşeală
  - O *superclasă* / clasă de bază este *mai generală* şi mai cuprinzătoare, dar *mai puţin complexă*
  - O *subclasă* / clasă derivată este *mai specializată,* mai puţin cuprinzătoare şi *mai complexă*
    - Pe măsură ce sunt adăugate mai multe variabile şi metode, numărul obiectelor care pot satisface definiţia clasei devine mai restrâns

Folosirea claselor abstracte

- O clasă abstractă contribuie la implementarea subclaselor sale concrete.
- Este folosită pentru a exploata polimorfismul.
  - Pentru funcţionalitatea 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 şi la clienţii lor.
- O clasă concretă poate extinde doar o singură clasă (abstractă)

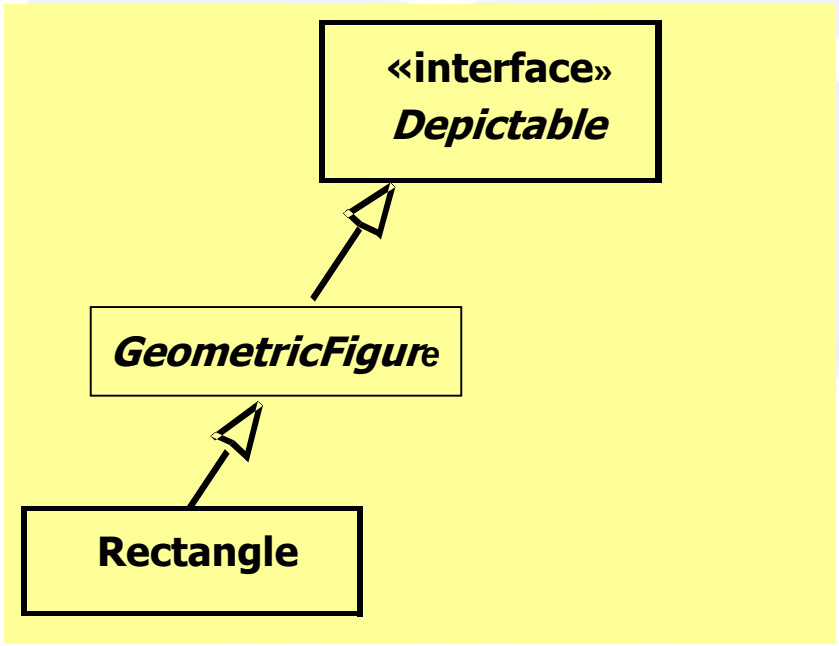
Interfeţe, clase abstracte şi clase concrete

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

Folosirea interfeţelor

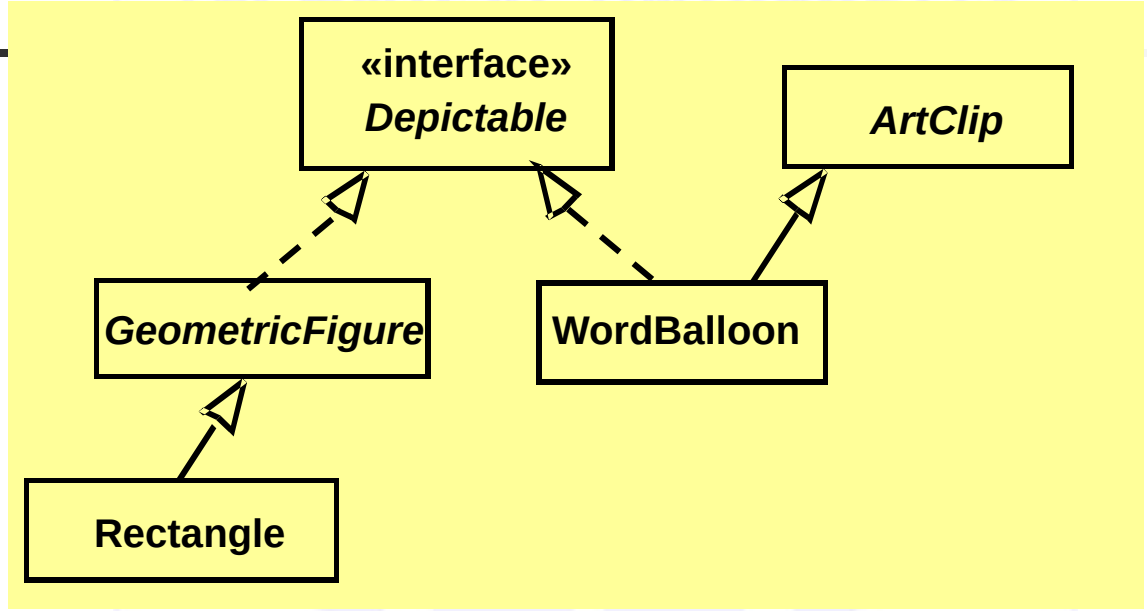
- Interfeţele sunt abstracte prin definiţie.
  - separă implementarea unui obiect de specificarea sa.
  - nu fixează nici un aspect al unei implementări.
- O clasă poate implementa mai mult de o interfaţă.
- Interfeţele permit o folosire mai generalizată a polimorfismului; instanţe din clase relativ neînrudite pot fi tratate ca identice într-un scop anume.
- În programe, folosiţi
  - *interfeţe pentru a partaja comportament comun.*
  - *moştenirea pentru a partaja cod comun*.

Interfeţe, clase abstracte şi clase concrete




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

Interfeţe, clase abstracte şi clase concrete




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

- Pot fi folosite instanţe 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();
…
```

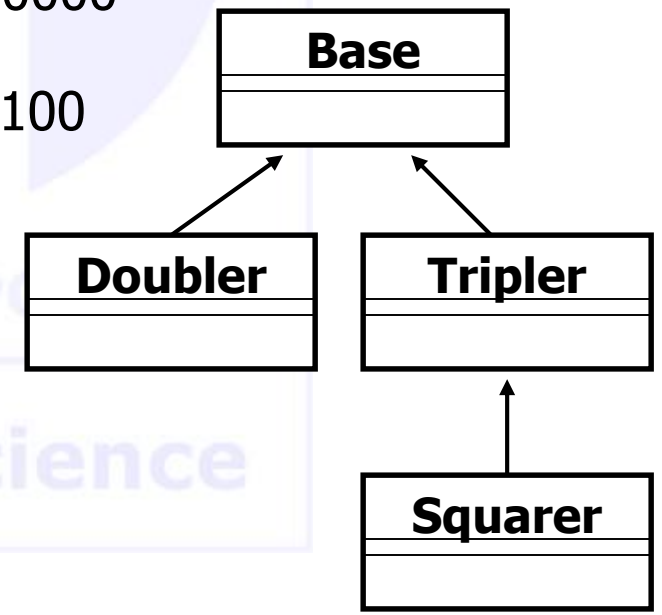
100

200

300

10000

100



Polimorfismul subtipurilor

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

Polimorfism

- O variabilă polimorfică poate părea a-şi schimba tipul prin legare dinamică.
- Compilatorul înţelege întotdeauna tipul unei variabile potrivit declaraţiei.
- Compilatorul permite o anume flexibilitate prin modul de conformare la tip.
- La execuţie, comportamentul unui apel de metodă depinde de tipul de *obiect* , nu de variabilă.
- Exemplu:

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

De ce este util polimorfismul?

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

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

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

Atunci ARectangle trebuie scris ca ...

```
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;
   }
}
```

Consideraţi acum

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

Moștenrea 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 definiță în `java.lang`

Cum se decide care este metoda de executat

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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Legarea dinamică

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

De studiat

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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Rezumat

- Pachete Java
  - organizarea claselor înrudite
  - API-uri Java
- Moştenire
  - clasă de bază, superclasă
  - clasă derivată, subclasă
  - ierarhii
  - constructorii `super` şi `this`
  - accesul la variabilele instanţă
  - Supraîncarcarea metodelor
- Interfeţe, clase abstracte, clase concrete
- Polimorfism