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 o clasă într-un pachet, trebuie scrisă o astfel de linie

```
package numePachet;
```

ca primă instrucţiune în fişierul sursă care conţine clasa

- Numele pachetului constă din unul sau mai mulţi identificatori separaţi prin puncte

Organizarea claselor înrudite în pachete

- Spre exemplu, pentru a pune clasa `Database` 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`

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 (două clase diferite având același nume – Timer – situate în două pachete diferite)

  `java.util.Timer` vs. `javax.swing.Timer`
- Numele de pachete trebuie să fie neambigue
- Numele căii trebuie să se potrivească cu numele pachetului

  `oop/examples/Database.java`
  - 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;.
```

Documents and Settings
cr11
Desktop
oop
examples
Database.java

Directorul de bază

Calea se potriveşte cu numele pachetului

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 {
    . . .
}
```

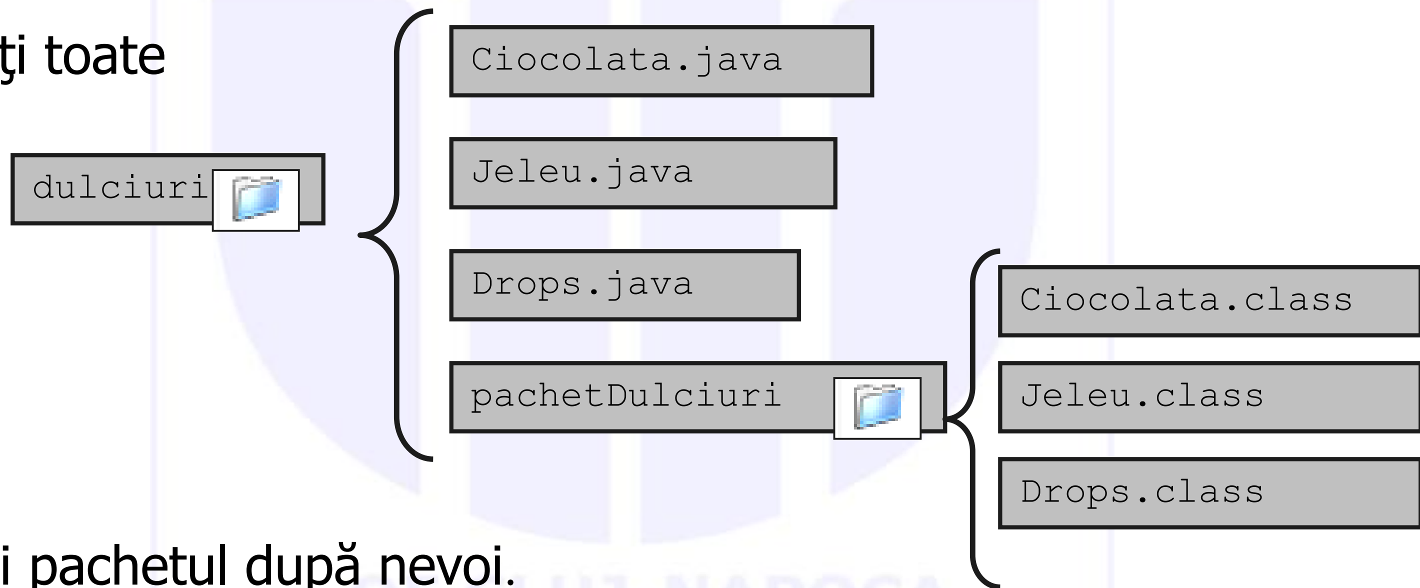
2) Stocaţi fişierele Java din pachet într-un director comun.

pachetDulciuri

- Ciocolata.java
- Jeleu.java
- Drops.java

Cum se construieşte un pachet

3) Compilaţi toate fişierele.



4) Importaţi pachetul după nevoi.

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

Cum să refolosim codul?

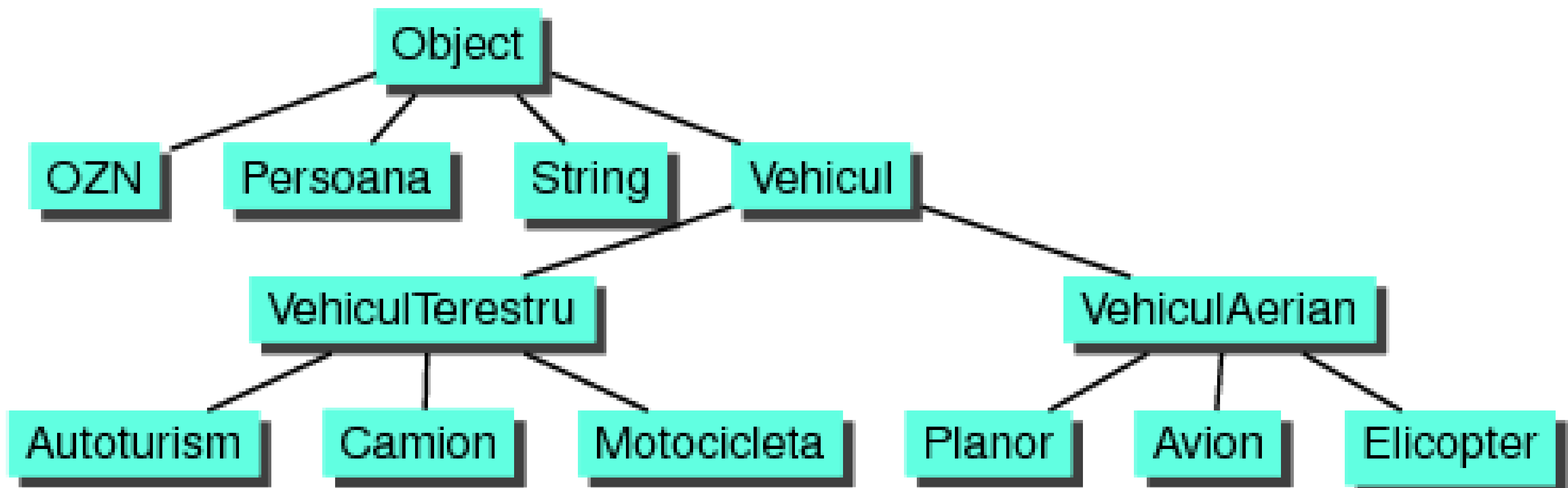
- Putem scrie clase de la început – fără a refolosi nimic (o extremă!)
  - Ceea ce unii programatori doresc să facă întotdeauna
- Putem găsi o clasa existentă care se potriveşte exact cerinţelor problemei (o altă extremă!)
  - Cel mai uşor lucru pentru programator
- Putem construi clase din clase existente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

Moştenirea

- ***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

Moştenirea

- 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



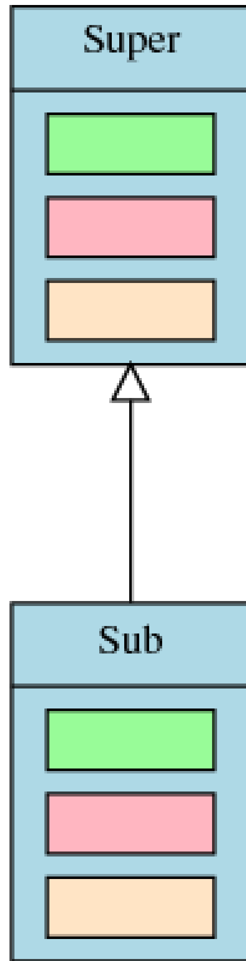
Moştenirea

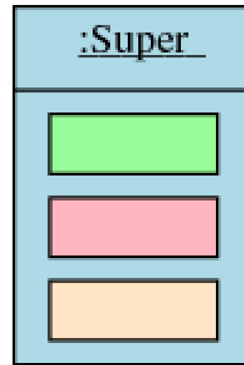
- Moştenirea este un mod de:
  - organizare a informaţiei
  - grupare a claselor similare
  - modelare a asemănărilor între clase
  - creare a unei taxonomii de obiecte
- **`Vehicul`** este numit *superclasă*
  - sau *clasă de bază* sau *clasă părinte*
- **`VehiculTerestru`** este numit *subclasă*
  - 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`**

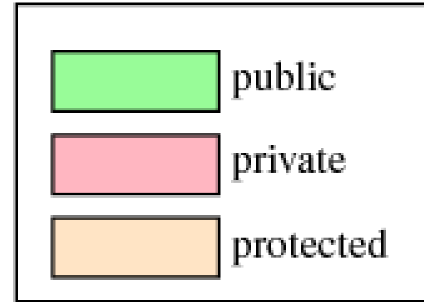
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

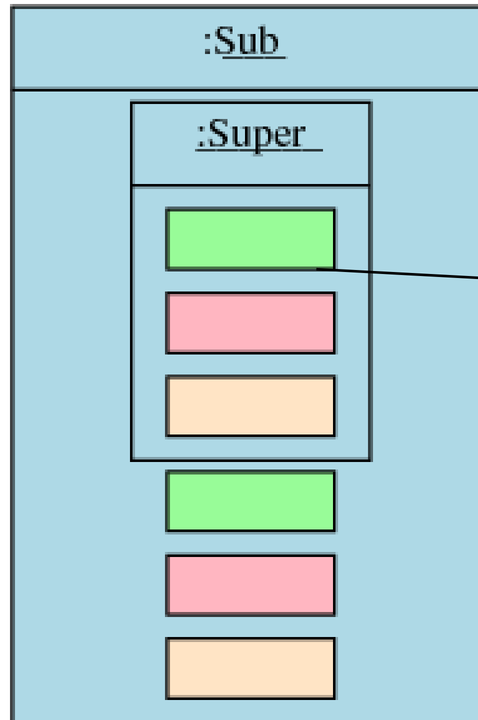
Moştenirea

- 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
- În Java se poate moşteni de la *o singură* superclasă
  - Nu există limite pentru adâncimea sau lăţimea ierarhiei de clase









Componentele moştenite ale superclasei sunt parte a subclasei

Ierarhia de clase

Instanţe

Exemplu: ierarhia unor conturi bancare

- Ierarhia de moşteniri:

BankAccount

-balance: double

<<create>>+BankAccount()
<<create>>+BankAccount(initialBalance: double)
+deposit(amount: double)
+withdraw(amount: double)
+getBalance(): double
+transfer(amount: double, other: BankAccount)

CheckingAccount

-transactionCount: int
-FREE_TRANSACTIONS: int = 3
-TRANSACTION_FEE: double = 2.0

<<create>>+CheckingAccount(initialBalance: double)
+deposit(amount: double)
+withdraw(amount: double)
+deductFees()

SavingsAccount

-interestRate: double

<<create>>+SavingsAccount(rate: double)
+addInterest()

Exemplu: 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`

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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:

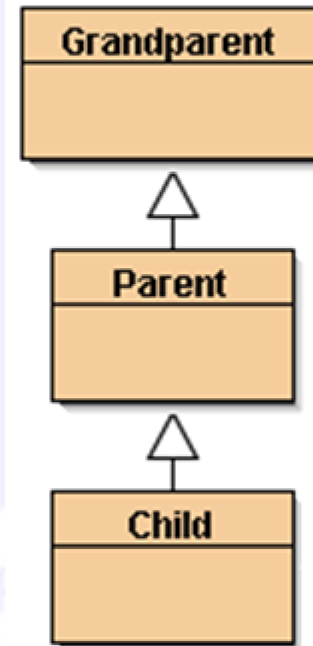
```
class NumeSubclasa extends NumeSuperclasa
{
    metode
    câmpuri de instanţă
}
```

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*, toate *variabilele instanţă*, precum şi toate *variabilele statice* 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 `Grandparent` este un strămoş al clasei `Child`, atunci clasa `Child` poate fi numită clasă *descendentă* a clasei `Grandparent`

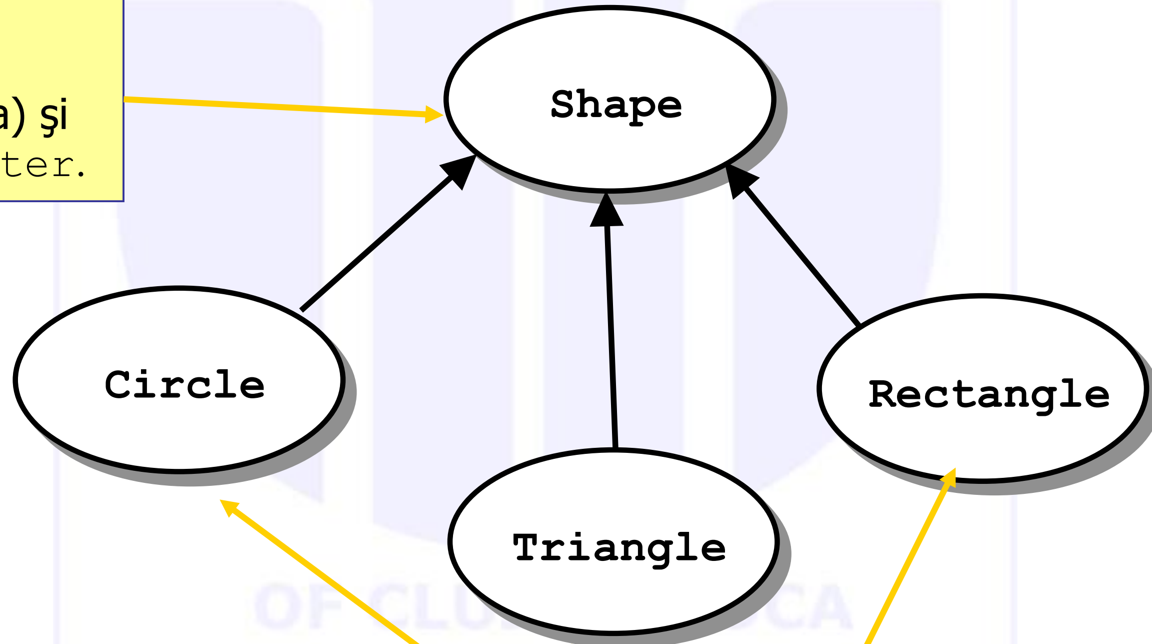


Clase abstracte

- O metodă sau o clasă abstractă se declară folosind cuvântul cheie **`abstract`**
- O clasă care conține cel puțin o metodă abstractă trebuie să fie abstractă
- 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 clasă 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`

```java
/**
 * 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.

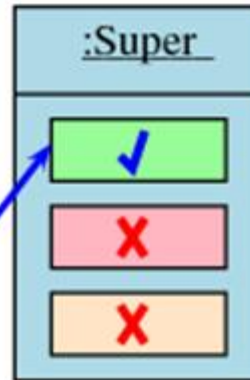
Apelează metodele `calculateArea` şi `calculatePerimeter`. Este apelată automat versiunea corespunzătoare a fiecărei metode pentru fiecare obiect.

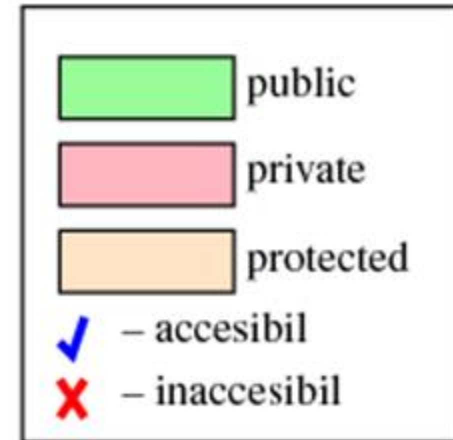
Variabile instanţă

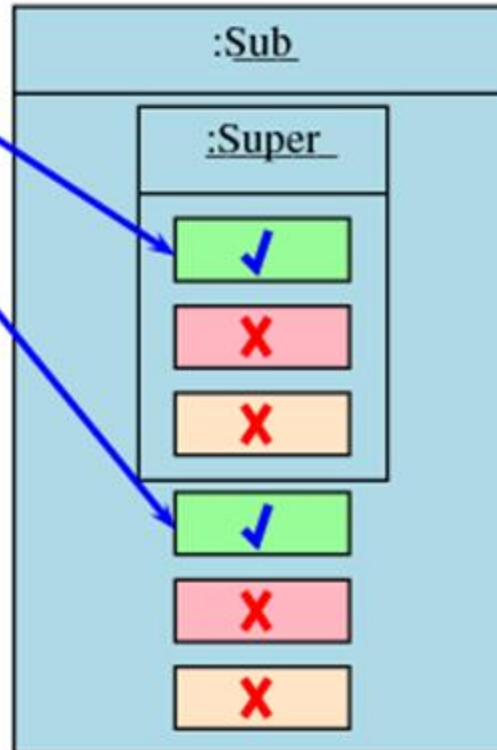
- Ca şablon general, subclasele
  - Moştenesc capabilităţile **`public`** (metode)
  - Moştenesc proprietăţile **`private`** (variabile instanţă) dar nu au acces la ele
  - Moştenesc variabilele **`protected`** şi le pot accesa

- O variabilă 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







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

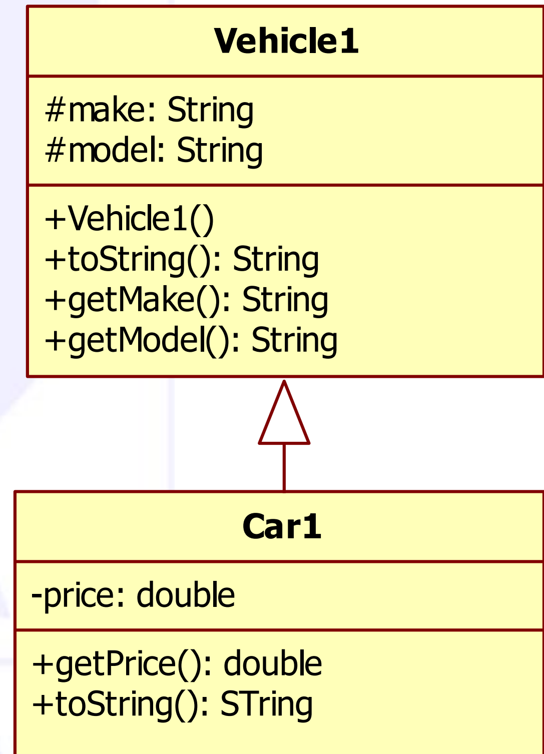
Variabile instanţă `protected` faţă de variabile instanţă `private`

- 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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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 BankAccount getAccount(int someKey)
  . . .
```

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

```
public class DerivedClass extends BaseClass
{ . . .
  public SavingsAccount getAccount(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 faţă 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 dintr-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 suprascrisă* 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

Accesul la o metodă redefinită din clasa de bază

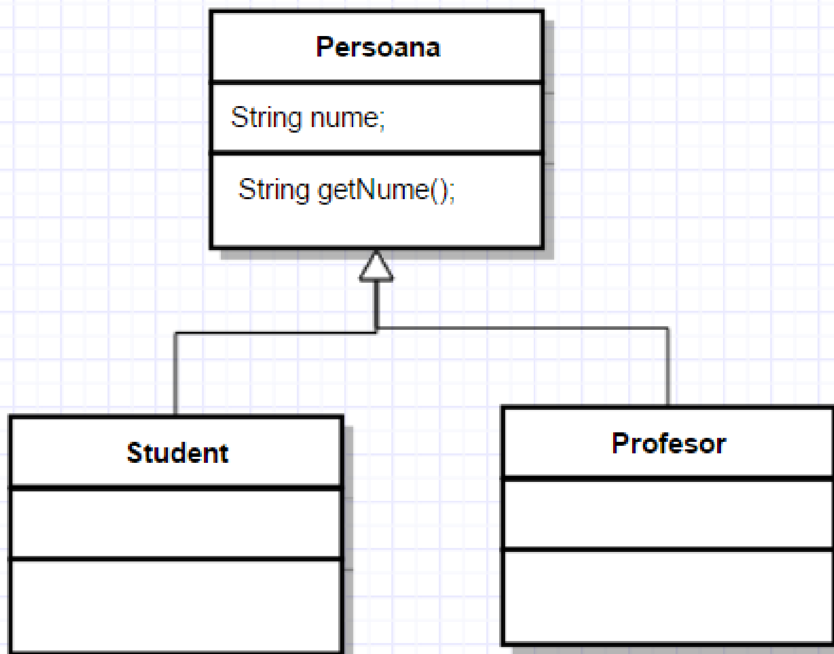
- În definiţia unei metode dintr-o clasă derivată, versiunea suprascrisă a unei metode a clasei de bază poate totuşi fi invocată
  - Pur şi simplu prefixaţi numele metodei cu `super` şi un punct

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

- 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

Construirea obiectelor în Java

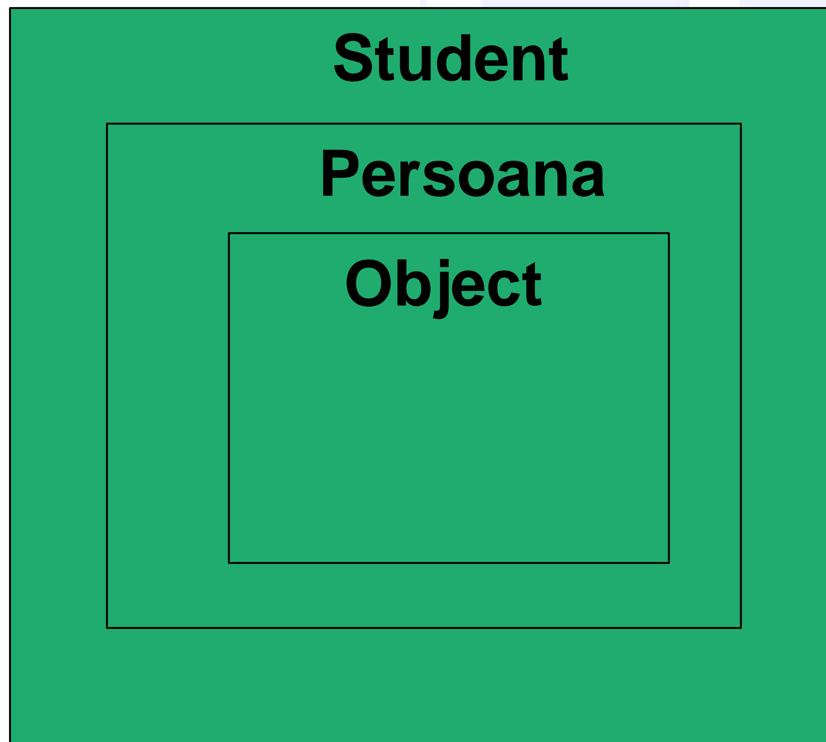
Exemplu de cod:




```
public class Persoana{
        private String nume;
        public String getNume(){
                return nume;
        }
}

class Student extends Persoana{

}

class Profesor extends Persoana{

}
```

Construirea obiectelor în Java

Student s = new Student();



Alocarea spațiului în memorie se face astfel:

- Se alocă spațiu pentru atributele din clasa **Object** (**atenție, clasa Object este moștenită implicit!**)
- Se alocă spațiu pentru atributele din clasa **Persoana**
- Se alocă spațiu pentru atributele din clasa **Student**

Construirea obiectelor în Java

- Cum interpretează compilatorul Java codul din acest exemplu?
  - Regula **1**: dacă o clasă nu extinde o altă clasă, atunci compilatorul inserează implicit: **extends Object**

```
public class Persoana{
   private String nume;
   //…

}
```

→

```
public class Persoana extends Object{
   private String nume;
   //…

}
```

Construirea obiectelor în Java

- Cum interpretează compilatorul Java codul din acest exemplu?
  - Regula **2**: dacă într-o clasă nu este definit nici un constructor, compilatorul creează implicit constructorul fără parametri

```
public class Persoana{
   private String nume;
   //…

}
```

→

```
public class Persoana extends Object{
   private String nume;
   Persoana(){
   }
   //…
}
```

Construirea obiectelor în Java

- Cum interpretează compilatorul Java codul din acest exemplu?
  - Regula **3**: prima linie din interiorul constructorului trebuie să fie
    - fie apelul unui alt constructor: **this(<params>)**
    - fie apelul unui constructor din superclasă: **super(<params>)**

    Altfel, compilatorul apelează implicit constructorul superclasei fără parametri super()

```
public class Persoana{
   private String nume;
   //…

}
```

→

```
public class Persoana extends Object{
   private String nume;
   Persoana(){
      super();
   }
}
```

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
  - Observaţi, totuşi, că relaţia nu merge şi invers!

Polimorfism

- Poli = mai multe
- Morphos = forme

- Polimorfismul se referă la această proprietate a obiectelor de a avea mai multe forme

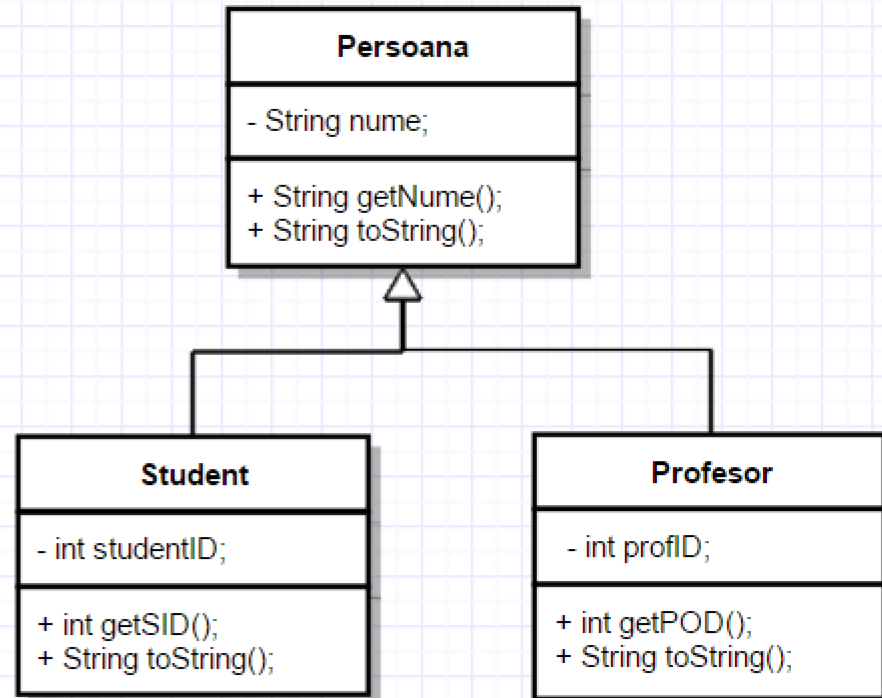
- Spre exemplu, un obiect de tip Persoana poate referi spre un obiect de tip Student:

  Persoana p= new **Student**("Ana", 2854);

Polimorfism

Dându-se diagrama de clase alăturată, ce va afișa următorul cod?

```
Persoana  p[] = new Persoana[3];
p[0] = new Persoana("Ion");
p[1] = new Student("Ana", 1234);
p[2] = new Profesor("Mara", 8);
 for(int i = 0; i < p.length; i++) {
        System.out.println( p[i] );
}
```



Rezultate afișate:
Ion
1234: Ana
8: Mara

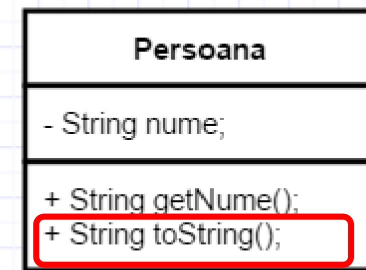
Polimorfism

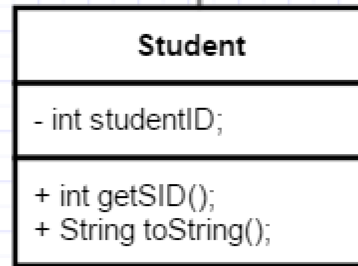
Decizii luate la compilare vs. în timpul execuției

- Reguli pentru compilare
  - Compilatorul cunoaște doar tipul referință al obiectului
  - Caută în clasa tipului referință dacă există metoda care se dorește a fi apelată
  - Și returnează antetul metodei (semnătura)

Semnătura metodei:
String toString();

```
Persoana p= new Student("Ana", 2854);
p.toString();
```





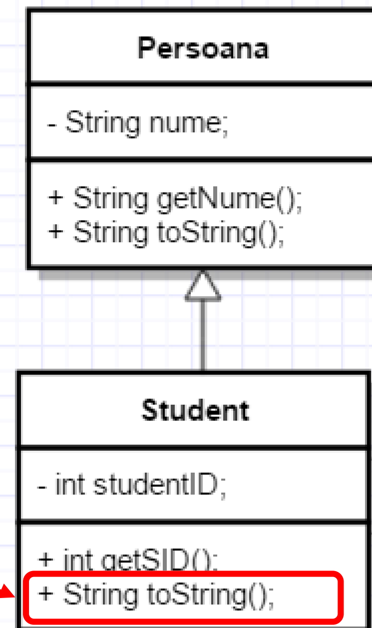
Polimorfism

Decizii luate la compilare vs. în timpul execuției

- Reguli pentru execuție
  - Se va urma tipul obiectului creat efectiv în momentul execuției
  - Semnătura returnată în momentul compilării trebuie să se potrivească cu metoda din clasa actuală a obiectului
    - În cazul în care metoda nu este gasită în clasa actuală, se caută mai sus în ierarhia de clase

```
Persoana p = new Student("Ana", 2854);
p.toString();
```

Metoda care se execută:



Polimorfism

Decizii luate la compilare vs. în timpul execuției

- Ce se întâmplă la execuția următoarelor variante de cod?
- Persoana p = new Student("Ana",1234); p.getSID();

R: Eroare de compilare

Soluție: ((Student) p).getSID();

- pentru a evita erorile la execuție, folosiți:

```
if( p instanceof Student ) {
        // se execută doar dacă p "este-un" Student la execuție
        ( (Student)s ).getSID();
}
```

- Student s = new Persoana("Ion");

R: Eroare de compilare

Solutie: - nu există

Polimorfism. Exemplu 1

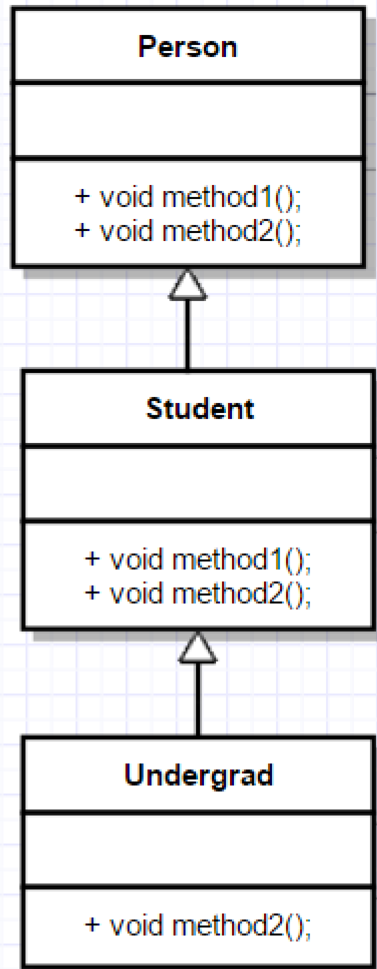
```
public class Persoana {
private String name;
public Person(String name) {
        this.name = name;
}
public boolean isAsleep(int hr) {
        return 22 < hr || 7 > hr;
}
public String toString() {
        return name;
}
public void status(int hr) {
  if (this.isAsleep(hr))
        System.out.println("Now
        offline: " + this);
  else
        System.out.println("Now
        online: " + this);
}
}
```

```
public class Student extends
Persoana {
public Student(String name) {
 super(name);
}
public boolean isAsleep(int hr) {
//suprascriere
return 2 < hr && 8 > hr;
}
public static void main(String[]
                         args){
 Persoana p;
 p = new Student("Ana");
 p.status(1);
}
}
```

Rezultate afișate:

Now online: Ana

Polimorfism. Exemplu 2




```
public class Person {
    public void method1() {
     System.out.print("Person 1 ");
    }
    public void method2() {
     System.out.print("Person 2 ");
}}
```

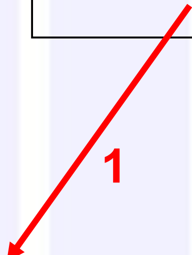
```
class Student extends Person {
    public void method1() {
     System.out.print("Student 1 ");
     super.method1();
     method2();
    }
    public void method2() {
     System.out.print("Student 2 ");
}}
```

```
class Undergrad extends Student {
    public void method2() {
     System.out.print("Undergrad 2 ");
}}
```

- Ce se va afișa la executarea următoarelor linii de cod?

  Person p = new Undergrad();
  p.method1();

```
public class Person {
   public void method1() {
    System.out.println("Person 1 ");
   }
   public void method2() {
    System.out.println("Person 2 ");
}}

class Student extends Person {
   public void method1() {
    System.out.println("Student 1 ");
    super.method1();
    method2();   //this.method2();
   }
   public void method2() {
    System.out.println("Student 2 ");
}}

class Undergrad extends Student {
   public void method2() {
    System.out.println("Undergrad 2 ");
}}
```



1

2

3

Rezultate afișate:
Student 1
Person 1
Undergrad 2

Polimorfism. Exemplu 2

- Discuții:

1. Se execută întâi method1() din clasa Student, deoarece în clasa Undergrad nu există o metodă cu acestă semnătură, astfel se execută prima metodă gasită mergând în sus în ierarhia de clase. Se afișează "Student 1"
2. Apoi se apelează method1() din clasa Person (indicată de apelativul *super*, care în momentul compilării stabilește că apelul trebuie făcut catre method1() din clasa Person). Se afișează "Person 1"
3. Se apelează method2() din clasa Undergrad, deoarece compilatorul interpretează apelul "method2();" ca "**this**.method2()", unde this se referă la obiectul din care se face apelul, și anume obiectul concret creat în momentul execuției, care este de tip Undergrad

Polimorfism

- Reguli în ceea ce privește apelul metodelor folosind operatorii **this** și **super**:
  - Când apelăm o metodă cu **super** (ex: super.method1()), legarea se face la compilare
    - Atunci se verifică care e clasa părinte
  - Când apelăm o metodă cu **this** (ex. this.method2(), sau pur și simplu method2()), legarea se face în momentul execuției, în funcție de tipul concret al obiectului creat
    - Aceasta mai poartă numele de **legare dinamică**

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Polimorfism. 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 metodă polimorfică care se poate executa între tipul variabilei şi subclasele sale

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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 găseşte o metodă concretă şi se execută
4. Dacă nu s-a găsit nici o metodă, atunci Java semnalează o eroare de compilare

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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:

```
Person p;
p = new Student();
p = new Undergrad();
p.method1();
```

De ce este util polimorfismul?

- Polimorfismul permite unei superclase să reţină ceea ce este comun, lăsând specificitatea să fie tratată de subclase

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

Să presupunem că AView include o metodă `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;
}
```

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 concrete
- O ***clasă concretă***
  - completează implementarea efectivă a metodelor abstracte care au fost specificate de o interfaţă sau printr-o clasă abstractă
  - furnizează obiecte la momentul execuţiei
  - nu este, în general, potrivită ca bază pentru extindere

Folosirea claselor abstracte

- O clasă abstractă contribuie la implementarea subclaselor sale concrete
- Este folosită pentru a exploata *polimorfismul*
  - Pentru funcţionalitatea specificată în clasa 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 sau concretă)

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*