Reutilizarea codului. Moştenirea Claselor

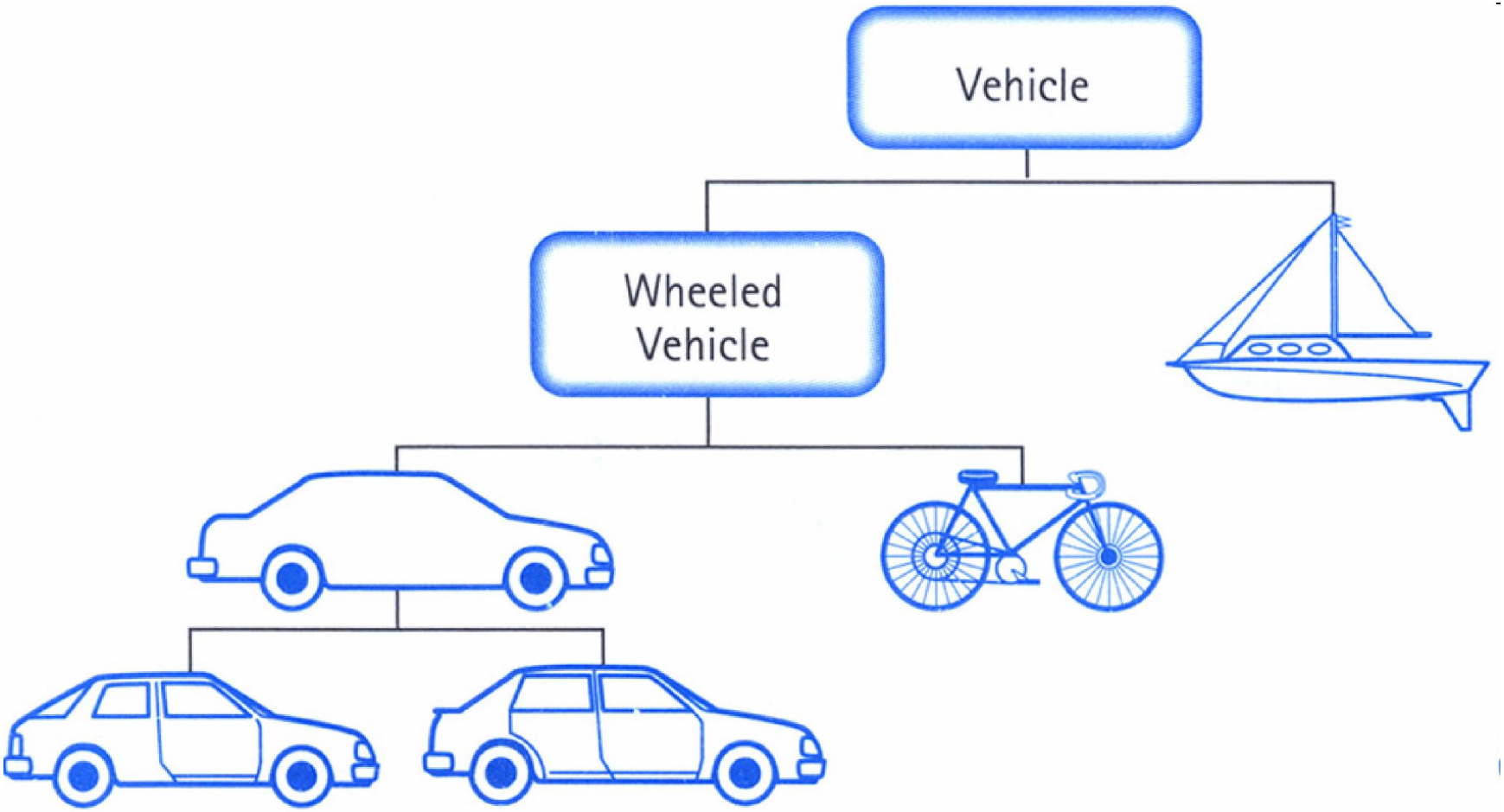
Ce este **moştenirea**?

- Mecanismul prin care o clasă preia *structura* (datele membru) şi *comportamentul* (metodele) unei alte clase la care se adaugă elemente specifice.
- În limbajul C++ acest mecanism este implementat prin conceptul de *moştenire*.
- Stabileşte o relație de genul "*este un / este o*" (**is-a / an**).
- **Clasă de bază** = clasa *de la care se preia* structura şi comportamentul.
- **Clasa derivată** = clasa *care preia* structura şi comportamentul.

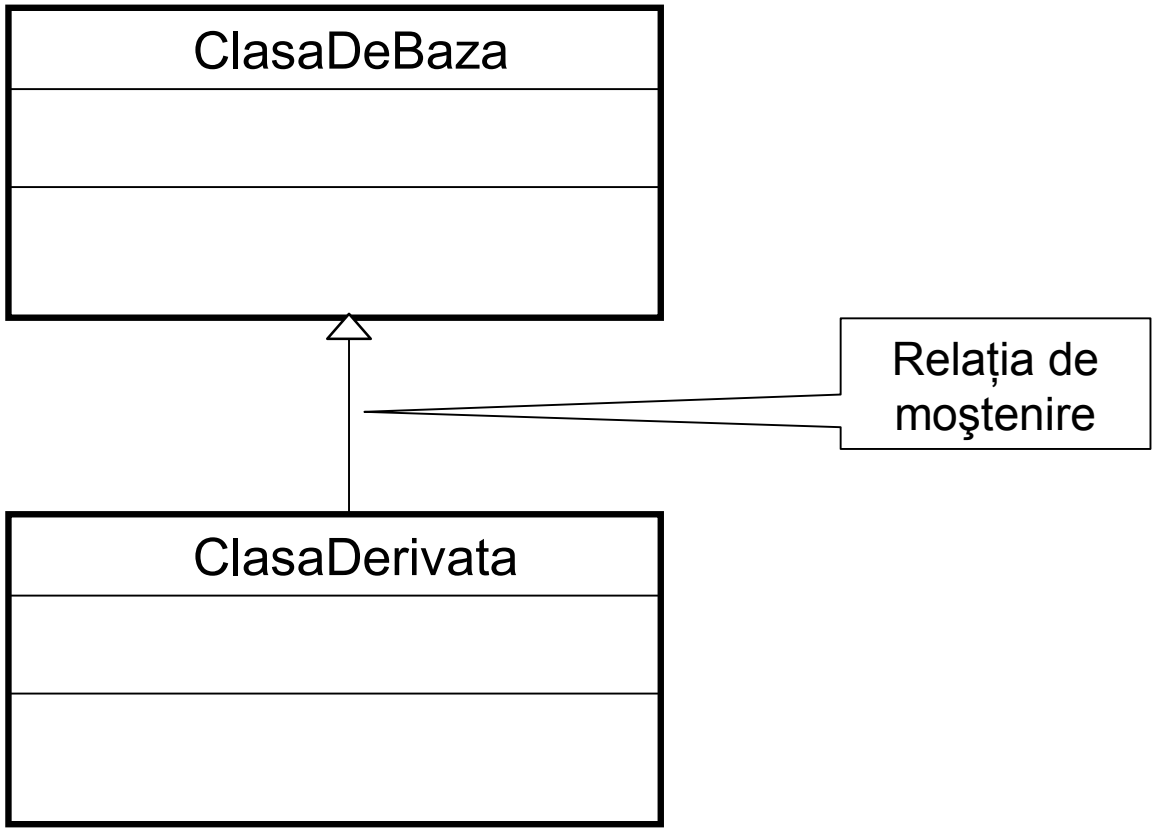
Avantaje

- posibilitatea reutilizării codului.
- obținerea extensiei unei clase fară a fi necesară o recompilare a clasei inițiale.
- utilizarea **polimorfismului** în timpul execuției programului prin folosirea funcțiilor **virtuale** (discuție în cursurile următoare).

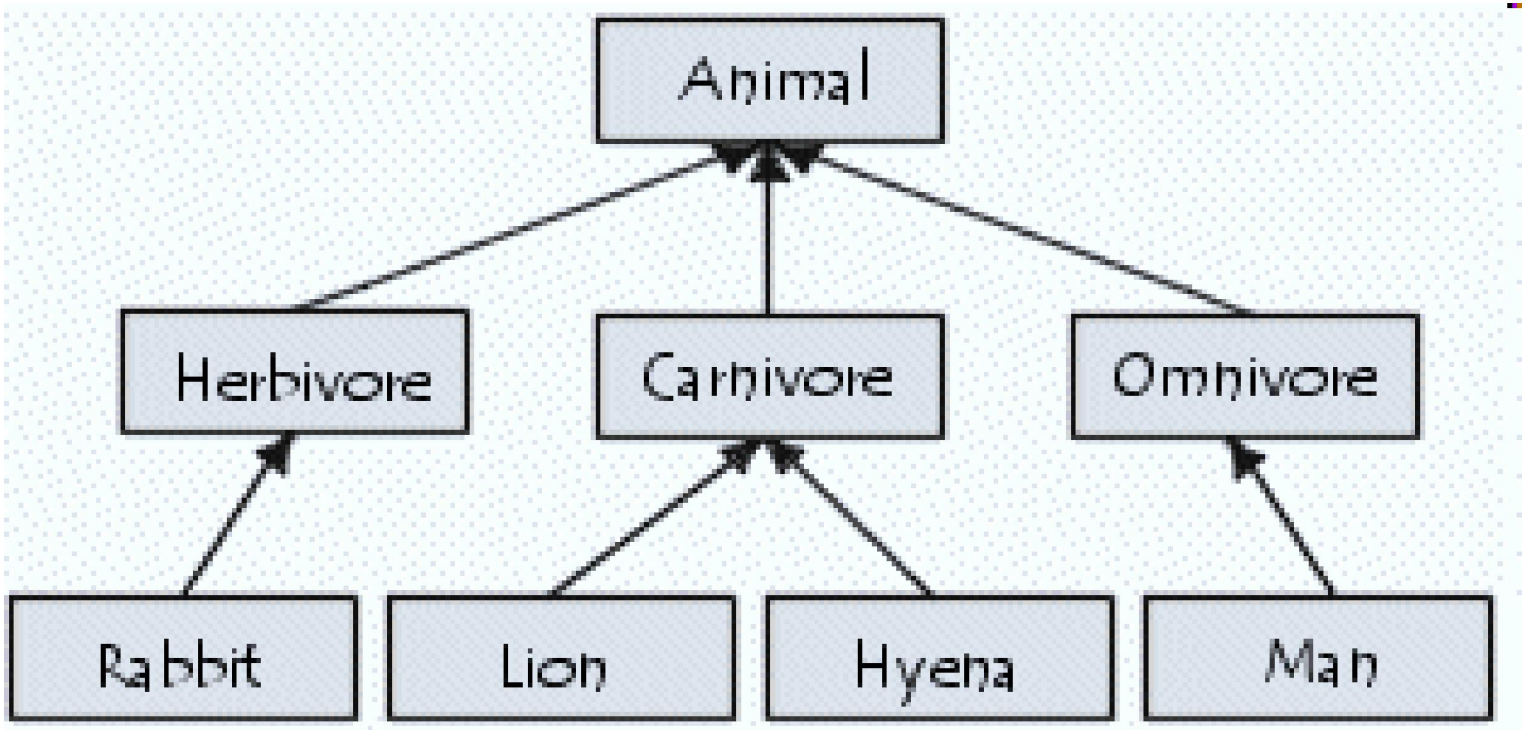
Ierarhii de clase



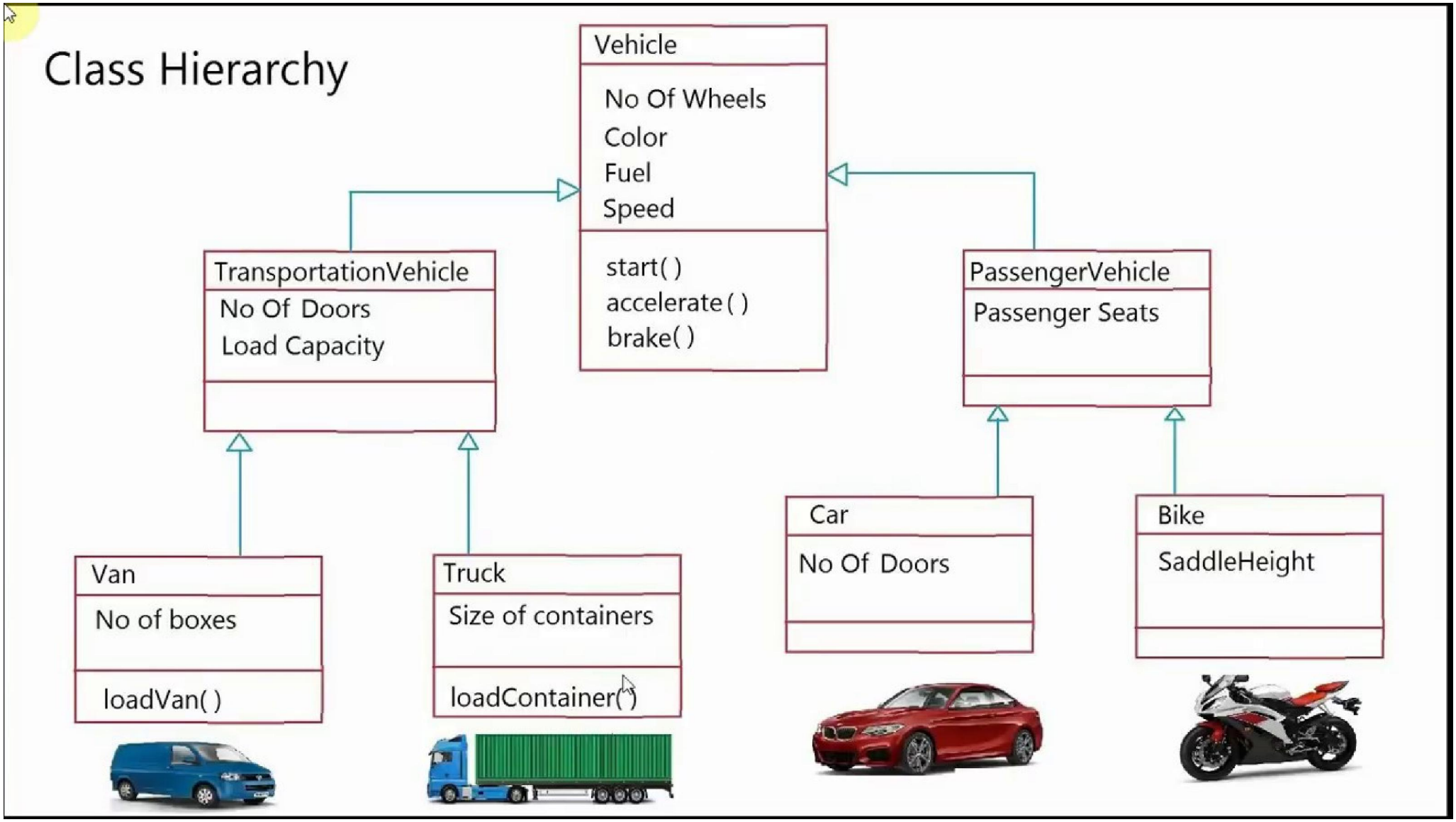
Reprezentare UML



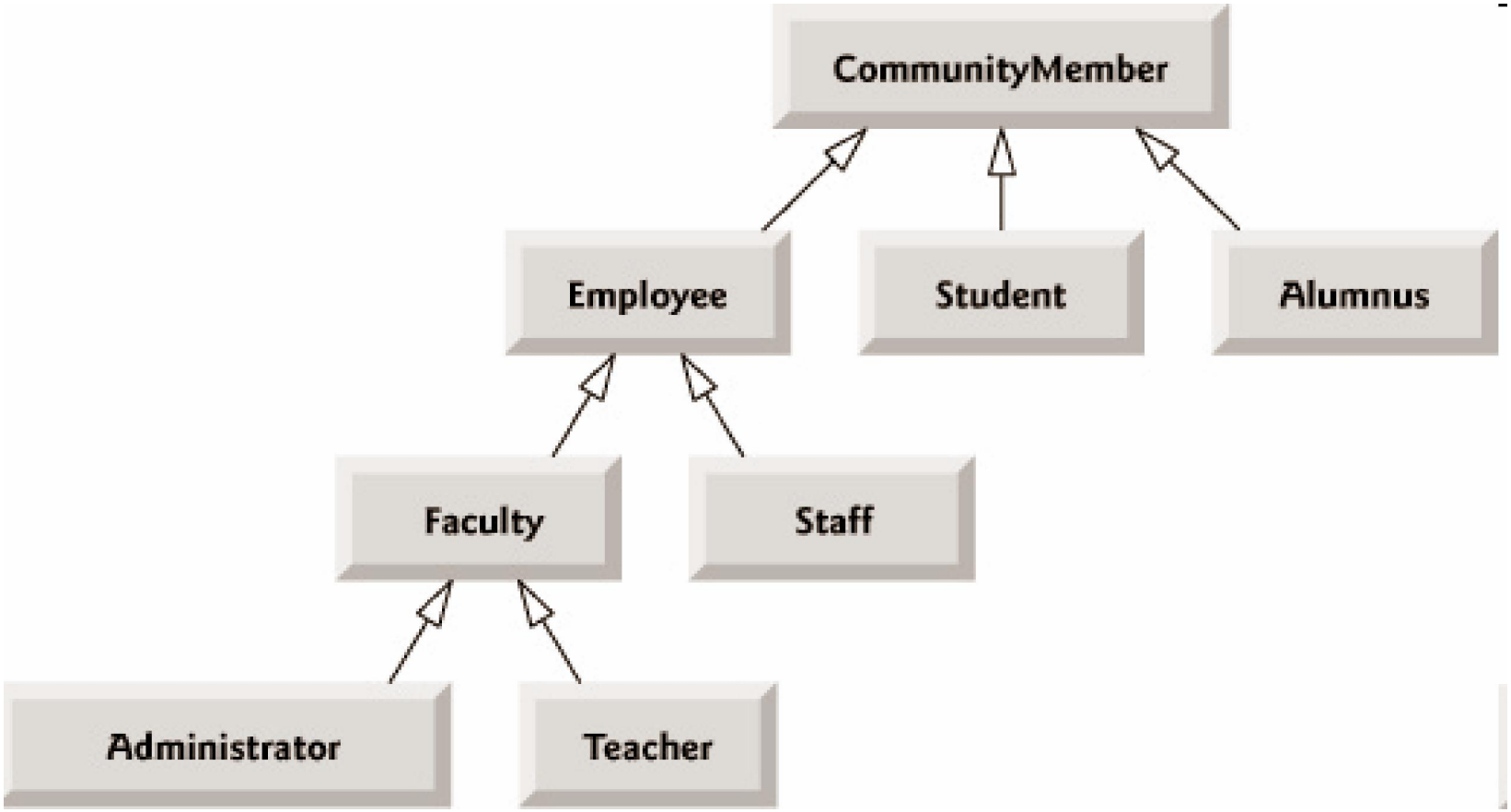
Ierarhii de clase



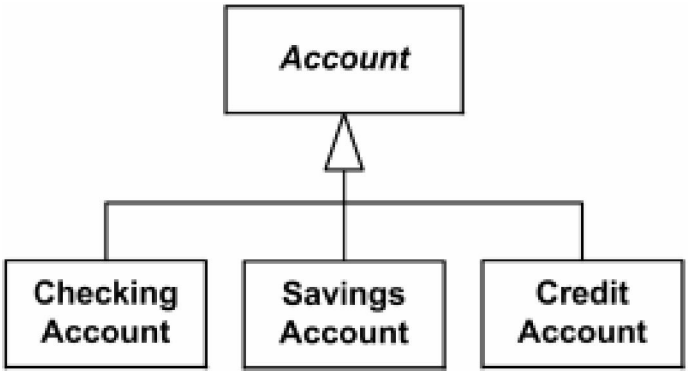
Ierarhii de clase

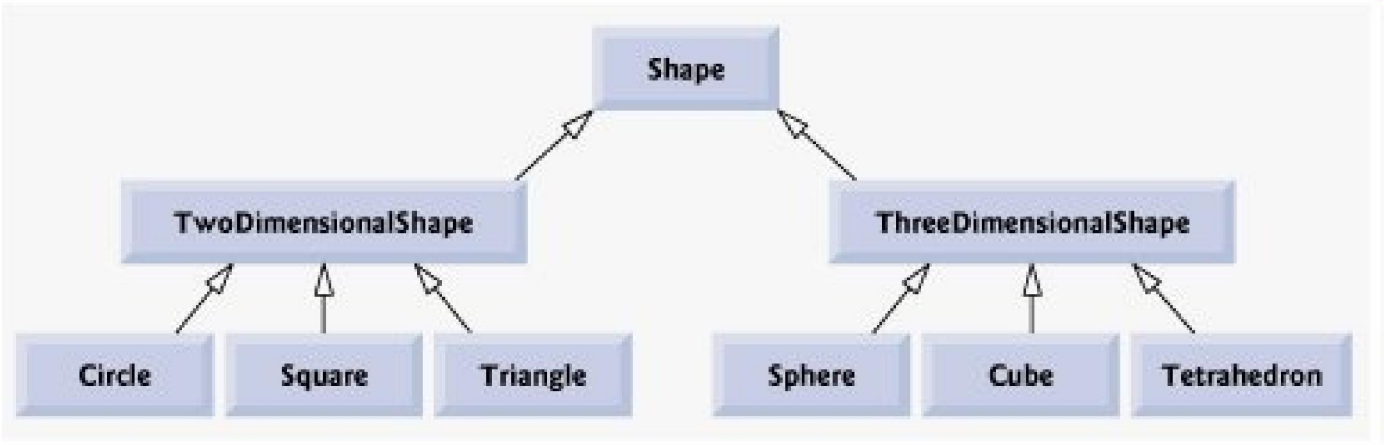


Ierarhii de clase

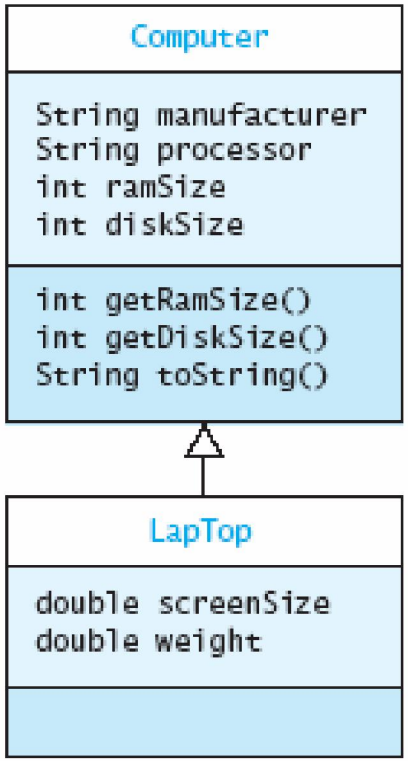


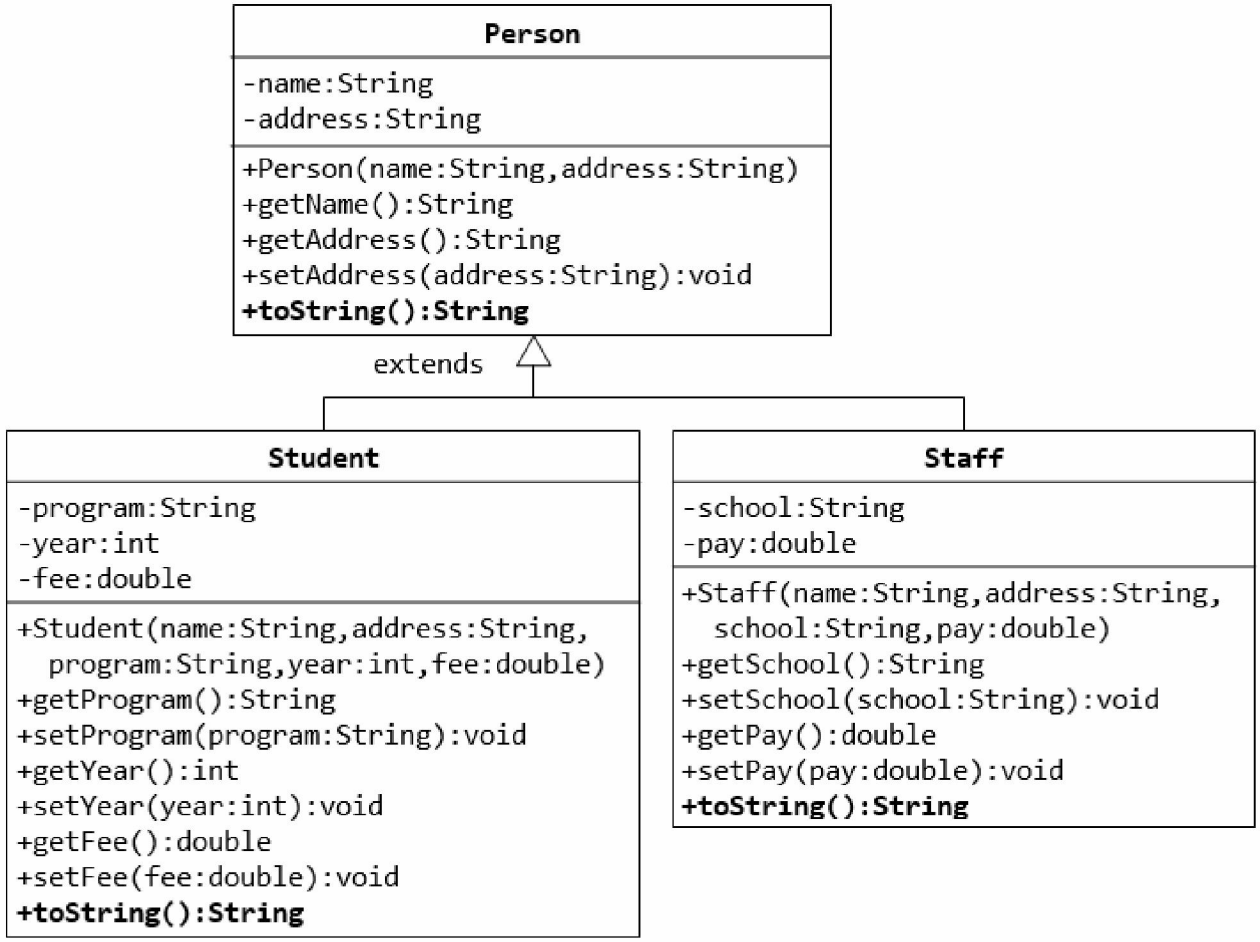
Ierarhii de clase





Exemplu

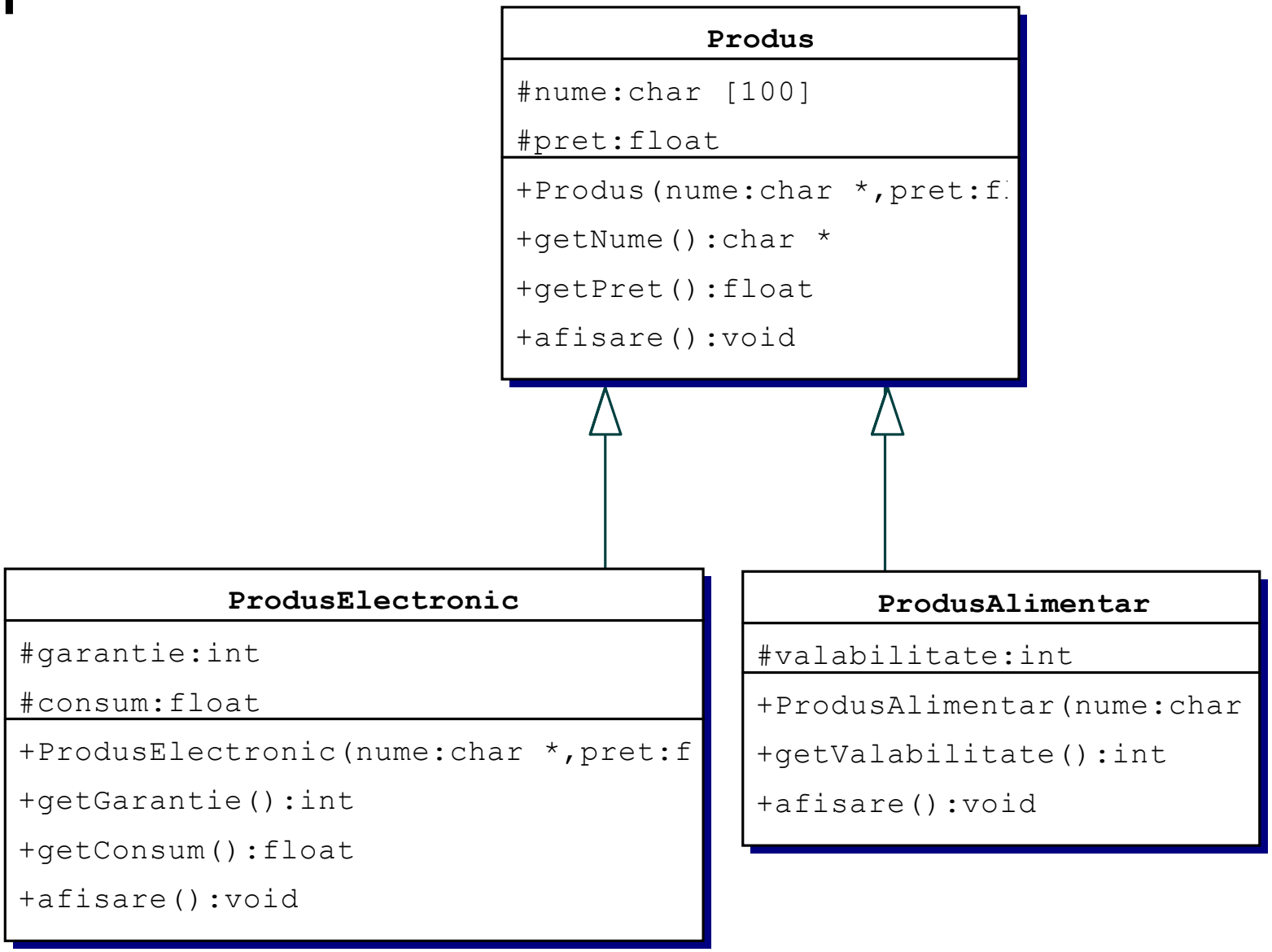

Computer
String manufacturer
String processor
int ramSize
int diskSize
int getRamSize()
int getDiskSize()
String toString()
LapTop
double screenSize
double weight


Person
-name:String
-address:String
+Person(name:String,address:String)
+getName():String
+getAddress():String
+setAddress(address:String):void
+toString():String
extends
Student
-program:String
-year:int
-fee:double
+Student(name:String,address:String,
program:String,year:int,fee:double)
+getProgram():String
+setProgram(program:String):void
+getYear():int
+setYear(year:int):void
+getFee():double
+setFee(fee:double):void
+toString():String
Staff
-school:String
-pay:double
+Staff(name:String,address:String,
school:String,pay:double)
+getSchool():String
+setSchool(school:String):void
+getPay():double
+setPay(pay:double):void
+toString():String

Terminologie

- clasa `Computer`, clasa `Persoana` – *clasă de bază*, *clasă părinte*, *superclasă*.
- clasa `Laptop`, clasa `Student`, clasa `Staff` – *clasă derivată, clasă fiu, subclasă*.
- *membru moştenit* (atribut, metodă) – un membru definit în clasa de bază, şi utilizat nemodificat în clasa derivată (`manufacturer`, `address`, `getRamSize()`).
- *membru redefinit* (atribut, metodă) – un membru definit în clasa de bază, şi definit din nou în clasa derivată (`toString()`).
- *membru nou* – membru declarat numai în clasa derivată (`screenSize`, `school`, `pay`).

Exemplu


Produs
#nume:char [100]
#pret:float
+Produs(nume:char *,pret:f
+getNume():char *
+getPret():float
+afisare():void
ProdusElectronic
#garantie:int
#consum:float
+ProdusElectronic(nume:char *,pret:f
+getGarantie():int
+getConsum():float
+afisare():void
ProdusAlimentar
#valabilitate:int
+ProdusAlimentar(nume:char
+getValabilitate():int
+afisare():void

Sintaxa definirii unei clase derivate

- Sintaxa definirii unei clase derivate este următoarea:

```
class IdClasaDerivata: modif_acces1 IdClasaB1, …, modif_accesN
   IdClasaBn {
    //date si metode specifice clasei derivate
};
```

unde:

- IdClasaDerivata este numele clasei derivate;
- IdClasaB1, …, IdClasaBn sunt clasele de bază de la care se moştenesc datele şi metodele;
- modif_acces1, …, modif_accesN sunt modificatori de acces: *public, protected, private.*

Accesul asupra membrilor moşteniți

Protectia în clasa de bază	Modificator de acces utilizat în lista claselor de bază	Drept de acces în clasa derivată
public private protected	public public public	public inaccesibil protected
public private protected	protected protected protected	protected inaccesibil protected
public private protected	private private private	private inaccesibil private

Accesul asupra membrilor moşteniți

- Moştenirea de tip `public` păstrează în clasa derivată nivelul de acces al membrilor *publici sau protected* din clasa de bază.
- Moştenirea de tip `protected` face ca membrii *publici* din clasa de bază să devină *protected* în clasa derivată. Membrii *protected* din clasa de bază rămân la fel şi în clasa derivată.
- Moştenirea de tip `privat` face ca toți membrii *publici* sau *protected* să devină *privați* în clasa derivată.

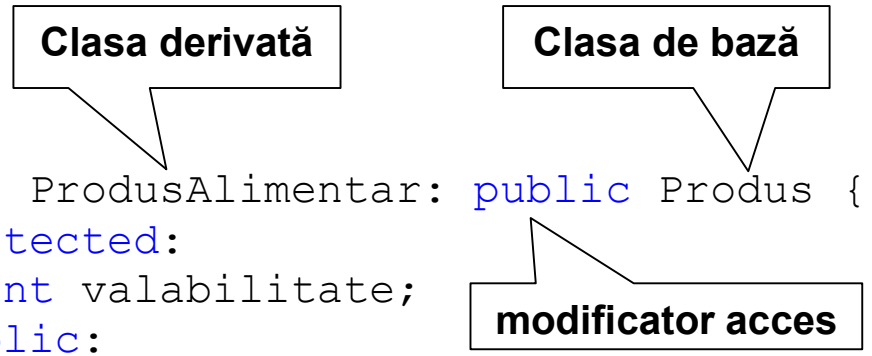
Moştenire `private`. Moştenire `protected`.

- Moştenirea privată sau protejată se întâlneşte destul de rar.
- Orice dată sau metodă membră a unei clase de bază se moşteneşte în clasa derivată, diferența fiind făcută de permisiunea de acces.
- Nu se moştenesc constructorii şi destructorii precum şi operatorul de atribuire ‘=‘.

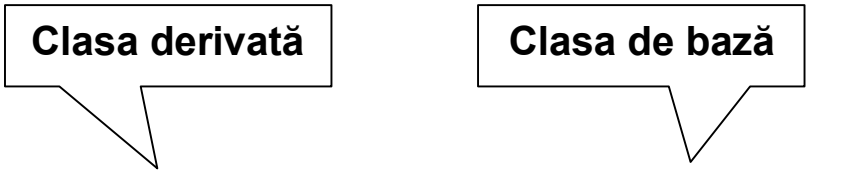
Exemplu

```
class Produs {
  protected:
    char nume[100];
    float pret;
  public:
    Produs(const char*, float);

    char* getNume();
    float getPret();
    void afisare();
};
```

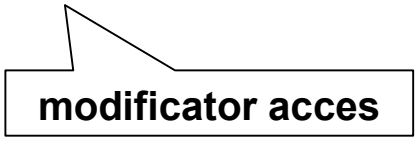
```
class ProdusAlimentar: public Produs {
  protected:
    int valabilitate;
  public:
    ProdusAlimentar(const char*, float,
                    int);

    int getValabilitate();
    void afisare();
};
```



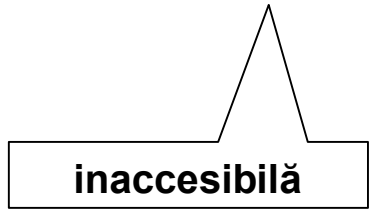
Exemplu




```
class ProdusElectronic: private Produs {
  protected:
    int garantie;
    float consum;
  public:
    ProdusElectronic(const char*, float,
                     int, float);

    int getGarantie();
    float getConsum();
    void afisare();
};
```




```
int main() {
  ProdusAlimentar pa("Iaurt", 15, 30);
  pa.afisare();

  cout << "Pret p.a." << pa.getPret();
  ProdusElectronic pe("LCD", 1000,24,12);
  pe.afisare();
  cout << "Pretul p.e." << pe.getPret();

  return 0;
}
```



Utilizarea datelor şi a metodelor moştenite

```
void Produs::afisare() {
  cout << "Nume: " << nume << endl;
  cout << "Pret: " << pret << endl;
}
```

```
void ProdusAlimentar::afisare() {
  Produs::afisare();
  cout<<"Valabilitate: "<<valabilitate;
  cout << endl;
}
```

```
int main() {
  ProdusAlimentar pa("Iaurt", 15, 30);
  pa.afisare();
  pa.Produs::afisare();
  return 0;
}
```

Output

```
Nume: Iaurt
Pret: 15
Valabilitate: 30
```

```
Nume: Iaurt
Pret: 15
```

Protected vs private

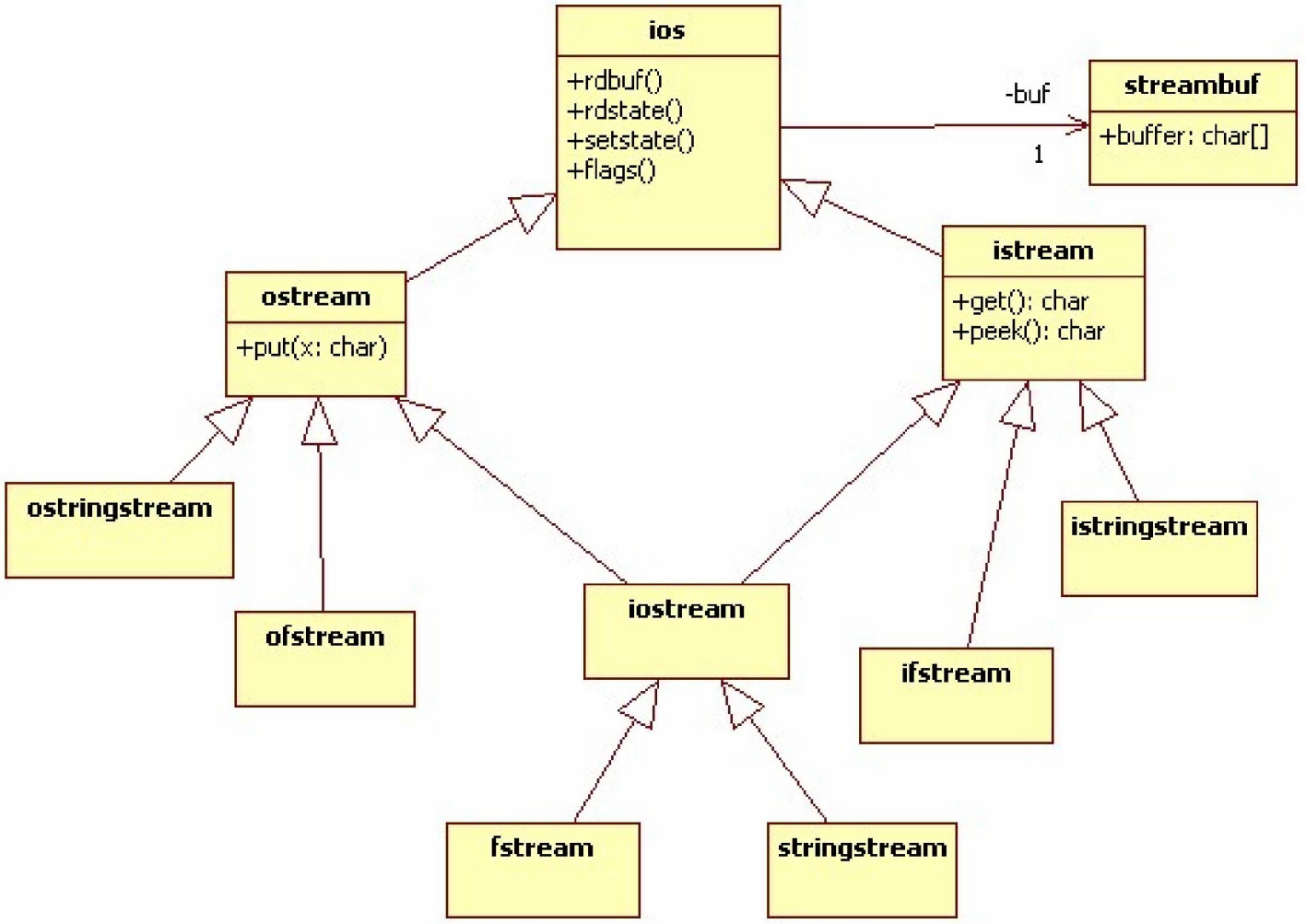
- Specificatorul de acces `protected` indică un nivel mai redus al încapsulării.
- Clasele derivate pot accesa datele membre *protejate* din clasa de bază.
- Să presupunem că în clasa de bază avem o dată membră *protejată* ce păstrează o dată calendaristică sub forma unui şir de caractere.
- La un moment dat, dorim să reprezentăm data calendaristică prin intermediul clasei `Date`.
  - dacă data membră ce păstrează o dată calendaristică este *privată*, putem să facem modificarea destul de uşor.
  - dacă data membră este `protected`, atunci trebuie să mergem prin tot lanțul de derivare şi să efectuăm modificările necesare.
  - va fi nevoie să realizăm şi modificări asupra documentației claselor implicate.

Tipuri de moştenire

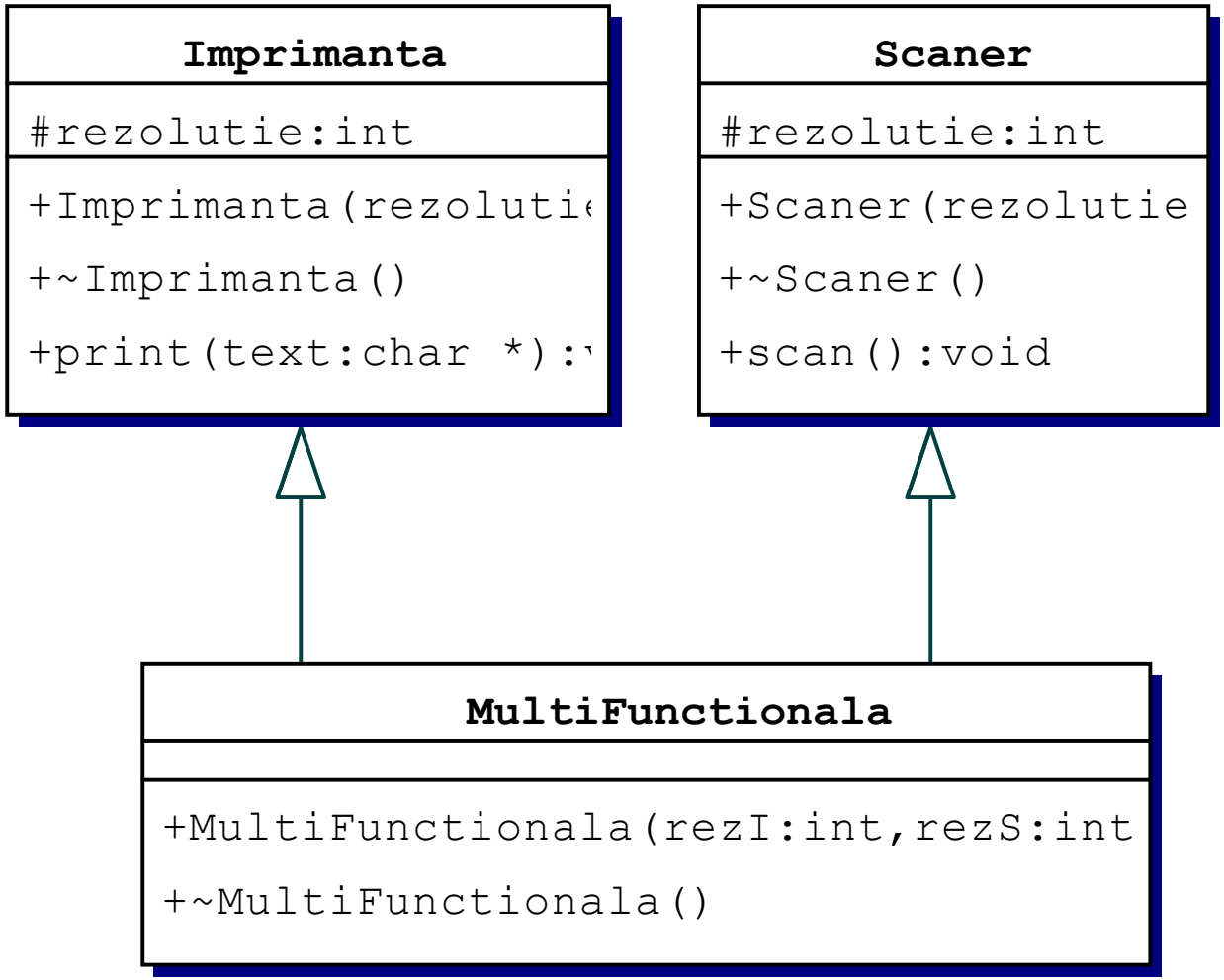
- *Moştenire simplă* – clasa derivată preia caracteristicile şi metodele unei singure clase de bază.
- *Moştenire multiplă* - clasa derivată preia caracteristicile şi metodele de la mai multe clase de bază.

Moştenire multiplă. Exemplu

- ierarhia de clase de IO



Moştenire multiplă. Exemplu



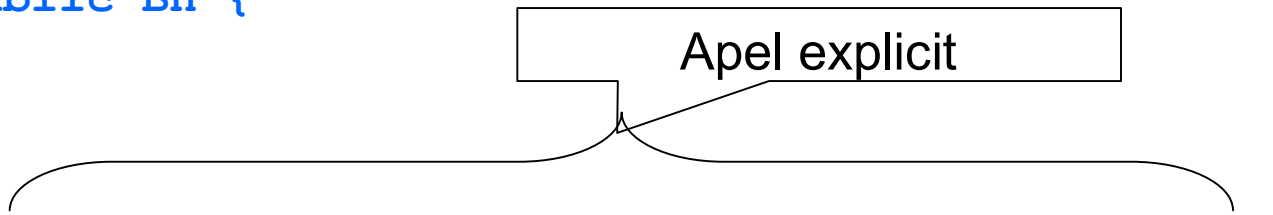
Constructori în procesul de moştenire

- Dacă o clasă D este derivată din mai multe clase (`B1, …, Bn`), atunci constructorul clasei D trebuie să aibă suficienți parametri pentru a inițializa datele membru ale claselor `B1, …, Bn`.
- La crearea unui obiect al clasei D se vor apela mai întâi constructorii claselor `B1, …,Bn`, în ordinea specificată în lista de moştenire, pentru a se iniția liza datele membre ale claselor de bază şi apoi se vor executa instrucțiunile constructorului clasei D.

Apelul constructorilor claselor de bază

- se poate face **explicit**

```
class D: public B1, …, public Bn {
   D(lista de parametri);
};

D::D(lista de parametri):B1(sub_lista_param1),...,Bn(sub_lista_paramn){
   instructiuni
}
```



- sau se poate face **implicit** – dacă constructorul clasei derivate nu apelează în mod explicit câte unul din constructorii din clasele de bază, atunci compilatorul va apela automat constructorul implicit al fiecărei clase de bază.

Destructori în procesul de moştenire

- La distrugerea unui obiect al clasei derivate, compilatorul va apela mai întâi destructorul clasei derivate şi apoi destructorii claselor de bază, în ordinea inversă în care au fost apelați constructorii.

Exemplu

```
class Imprimanta {
  protected:
    int rezolutie;
  public:
    Imprimanta(int rezolutie = 600) {
      this->rezolutie = rezolutie;
      cout << "Apel Constr. Imprimanta\n";
    }

    ~Imprimanta() {
      cout << "Apel Destr. Imprimanta\n";
    }

    void print(char *text) {
      cout << "Print " << text << "\n";
    }
};
```

```
class Scaner {
  protected:
    int rezolutie;
  public:
    Scaner(int rezolutie = 1200) {
      this->rezolutie = rezolutie;
      cout << "Apel Constr. Scaner\n";
    }

    ~Scaner() {
      cout << "Apel Destr. Scaner\n";
    }

    void scan() {
      cout << "Scanez\n";
    }
};
```

Exemplu

```
class MultiFunctionala: public Imprimanta,
                        public Scaner {
  public:
    MultiFunctionala(int rezI, int rezS):
                Imprimanta(rezI), Scaner(rezS) {
      cout << "Apel Constr. MultiFunctionala\n";
    }

    ~MultiFunctionala(){
      cout << "Apel Destr. MultiFunctionala\n";
    }
};

int main(){
  MultiFunctionala mf(300, 600);
  mf.print("hello");
  mf.scan();
  return 0;
}
```

OUTPUT

```
Apel Constr. Imprimanta
Apel Constr. Scaner
Apel Constr. MultiFunctionala
Print hello
Scanez
Apel Destr. MultiFunctionala
Apel Destr. Scaner
Apel Destr. Imprimanta
```

Se afişează la distrugerea obiectului mf

Constructorul de copiere în procesul de moştenire

- Constructorii sunt **funcții membre ce nu se moştenesc** (deci implicit nici constructorii de copiere).
- Să considerăm următoarea clasă derivată:

```
class D: public B1, public B2, …, public Bn {
  …
};
```

- Dacă **clasa derivată D** nu are definit **un constructor de copiere,** atunci compilatorul va genera în mod automat unul ce va realiza copierea datelor membre moştenite, apelând la constructorii de copiere ai claselor de bază **`B1, …, Bn`**, (definiți de utilizator sau generați de compilator), iar datele specifice clasei derivate D se vor copia *bit cu bit*.

Constructorul de copiere în procesul de moştenire

- Dacă clasa derivată **D** are definit constructorul de copiere atunci acesta se va ocupa atât de copierea datelor membre moştenite de la clasele de bază **B1, …, Bn** (poate apela explicit constructorii acestora), cât şi de copierea datelor specifice clasei **D**, apelând eventual la constructorii claselor de bază:

```
class D: public B1, public B2, …, public Bn {
  …
  D (const D &ob): B1(ob), B2(ob),…, Bn(ob) {
    // copierea datelor specifice lui D din ob
  }
};
```

Constructorul de copiere în procesul de moştenire

- Să considerăm următoarea ierarhie de clase:

```
class B;
class D: public B { };
D d1, d2 = d1;
```

- Dacă constructorul de copiere **nu există în nici o clasă** (D şi B), atunci el este generat de compilator (copiere bit cu bit).
- Dacă constructorul de copiere **există doar în clasa de bază B**: se apelează acesta pentru membrii din *clasa de bază* şi respectiv constructorul de copiere implicit pentru ceilalți membri din *clasa derivată*.
- Dacă **există doar în clasa derivată D**, se apelează
  - implicit constructorul fără parametri al clasei de bază B (dacă există);
  - explicit alți constructori din clasa de bază B;
  - precum şi constructorul de copiere al clasei D.
- Dacă **există şi în clasa de bază B şi în clasa derivată D**, se apelează
  - implicit constructorul fără parametri al clasei de bază B (dacă există);
  - explicit alți constructori din clasa de bază B sau constructorul de copiere definit în clasa de bază B;
  - şi constructorul de copiere pentru membrii din clasa derivată.

Supraîncărcarea operatorului ‘=‘

- Supraîncărcarea operatorului ‘=‘ **nu se moşteneşte**.
- Dacă **clasa derivată D** nu are supraîncărcat **operatorul de atribuire** atunci compilatorul va realiza copierea datelor membre moştenite, apelând la supraîncărcările operatorului de atribuire `operator=` definite în clasele de bază **`B1, …, Bn`** (dacă există), iar datele membre specifice clasei D vor fi copiate *bit cu bit*.
- Dacă **clasa derivată** are supraîncărcat **operatorul de atribuire,** atunci acesta va realiza copierea tuturor datelor membre.

Supraîncărcarea operatorului ‘=‘

- Dacă clasa derivată supraîncarcă operatorul ‘=‘, atunci el trebuie să realizeze copierea atât a datelor membre moştenite de la clasele de bază **B1, …, Bn** (poate apela explicit metoda **operator=** a claselor de bază), cât şi pentru datele specifice clasei **D**:

```
class D: public B1, public B2, …, public Bn {
  . . .
  D& operator= (const D &ob) {
    if (this != &ob){
    //copierea datelor specifice lui D din ob
      B1::operator=(ob);
      …
      Bn::operator=(ob);
   }

    return *this;
  }
};
```

Exemplul 1

```
class Persoana {
  protected:
    char *nume;
    int varsta;

  public:
    Persoana(const char*, int = 0);
    Persoana(const Persoana&);

    ~Persoana();

    void afisare();
    Persoana& operator = (Persoana&);
};

Persoana::Persoana(const char* nume, int v) {
  this->nume = new char[strlen(nume) + 1];
  strcpy(this->nume, nume);
  this->varsta = v;
  cout << "Apel constr. Persoana\n";
}

Persoana::Persoana(const Persoana &p) {
  nume = new char[strlen(p.nume) + 1];
  strcpy(nume, p.nume);
  varsta = p.varsta;
  cout << "Apel Constr. de copiere Persoana\n";
}

Persoana::~Persoana() {
  delete[] nume;
  cout << "Apel destructor Persoana\n";
}

void Persoana::afisare() {
  cout << "Nume: " << nume << endl;
  cout << "Varsta: " << varsta << endl;
}

Persoana& Persoana::operator = (Persoana &p) {
  cout << "Apel atribuire Persoana\n";
  if (this != &p) {
    delete[] nume;
    nume = new char[strlen(p.nume) + 1];
    strcpy(nume, p.nume);
    varsta = p.varsta;
  }
  return *this;
}
```

Exemplul 1 (cont.)

```
class Student: public Persoana {
  protected:
    char* facultate;
  public:
    Student(const char*, int, const char*);
    Student(const Student&);
    ~Student();

    void afisare();
    Student& operator = (Student&);
};

Student::Student(const char* nume, int varsta,
 const char* facultate): Persoana(nume, varsta) {
  this->facultate = new char[strlen(facultate)+1];
  strcpy(this->facultate, facultate);
  cout << "Apel constr. Student\n";
}
```

```
Student::Student(const Student &s):
                  Persoana(s.nume, s.varsta) {
  facultate = new char[strlen(s.facultate) + 1];
  strcpy(facultate, s.facultate);
  cout << "Apel Constr. de copiere Student\n";
}
```

```
Student::~Student() {
  delete[] facultate;
  cout << "Apel destructor Student\n";
}

void Student::afisare() {
  Persoana::afisare();
  cout << "Fac: " << facultate << endl;
}
```

```
Student& Student::operator = (Student &s) {
  cout << "Apel atribuire Student\n";
  if (this != &s) {
    Persoana::operator=(s);
    delete[] facultate;
    facultate = new char[strlen(s.facultate)+1];
    strcpy(facultate, s.facultate);
  }

  return *this;
}
```

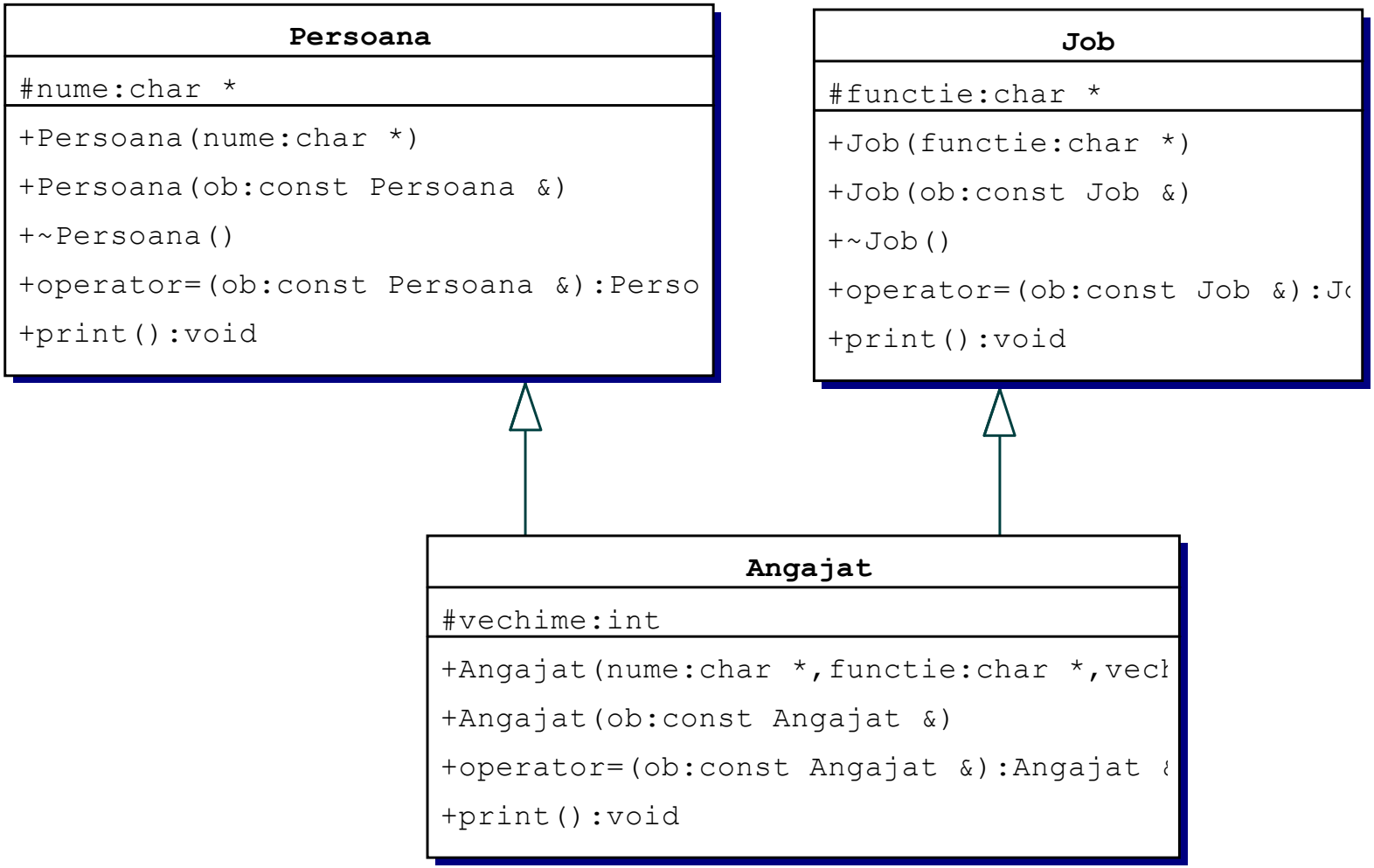
Apelul metodei mostenite din clasa Persoana

Exemplul 1 (cont.)

```
int main() {
  Student s1("Andrei", 21, "Informatica");
  s1.afisare();

  Student s2 = s1;
  s2.afisare();

  Student s3("Mircea", 23, "Medicina");
  s1 = s3;
  s1.afisare();

  return 0;
}
```

```
Apel constr. Persoana
Apel constr. Student
Nume: Andrei
Varsta: 21
Fac: Informatica
Apel constr. Persoana
Apel Constr. de copiere Student
Nume: Andrei
Varsta: 21
Fac: Informatica
Apel constr. Persoana
Apel constr. Student
Apel atribuire Student
Apel atribuire Persoana
Nume: Mircea
Varsta: 23
Fac: Medicina
Apel destructor Student
Apel destructor Persoana
Apel destructor Student
Apel destructor Persoana
Apel destructor Student
Apel destructor Persoana
```

Exemplul 2


Persoana
#nume:char *
+Persoana(nume:char *)
+Persoana(ob:const Persoana &)
+~Persoana()
+operator=(ob:const Persoana &):Perso
+print():void
Job
#functie:char *
+Job(functie:char *)
+Job(ob:const Job &)
+~Job()
+operator=(ob:const Job &):Jo
+print():void
Angajat
#vechime:int
+Angajat(nume:char *,functie:char *,vech
+Angajat(ob:const Angajat &)
+operator=(ob:const Angajat &):Angajat &
+print():void

Exemplul 2 (cont.)

```
class Persoana {
 protected:
  char* nume;

 public:
  Persoana(const char* str) {
    nume = new char[strlen(str) + 1];
    strcpy(nume, str);
    cout << "Apel constructor Persoana\n";
  }

  Persoana(const Persoana& ob) {
    nume = new char[strlen(ob.nume) + 1];
    strcpy(nume, ob.nume);
    cout << "Apel constructor copiere Persoana\n";
  }

  ~Persoana() {
    delete[] nume;
    cout << "Apel destructor Persoana\n";
  }

  Persoana& operator = (const Persoana& ob) {
    if (this != &ob) {
      delete[] nume;
      nume = new char[strlen(ob.nume) + 1];
      strcpy(nume, ob.nume);
    }
    cout << "Apel operator= Persoana\n";
    return *this;
  }

  void print() {
    cout << "Nume:" << nume << endl;
  }
};
```

Exemplul 2 (cont.)

```
class Job {
 protected:
  char* functie;

 public:
  Job(const char* fname) {
    functie = new char[strlen(fname) + 1];
    strcpy(functie, fname);
    cout << "Apel constructor Job\n";
  }

  Job(const Job& ob) {
   functie = new char[strlen(ob.functie) + 1];
   strcpy(functie, ob.functie);
   cout << "Apel constructor copiere Job\n";
  }

  ~Job() {
    delete[] functie;
    cout << "Apel destructor Job\n";
  }

  Job& operator= (const Job& ob) {
   if (this != &ob) {
    delete[] functie;
    functie = new char[strlen(ob.functie) + 1];
    strcpy(functie, ob.functie);
   }
   cout << "Apel operator= Job\n";
   return *this;
  }

  void print() {
    cout << "functie: " << functie << endl;
  }
};
```

Exemplul 2 (cont.)

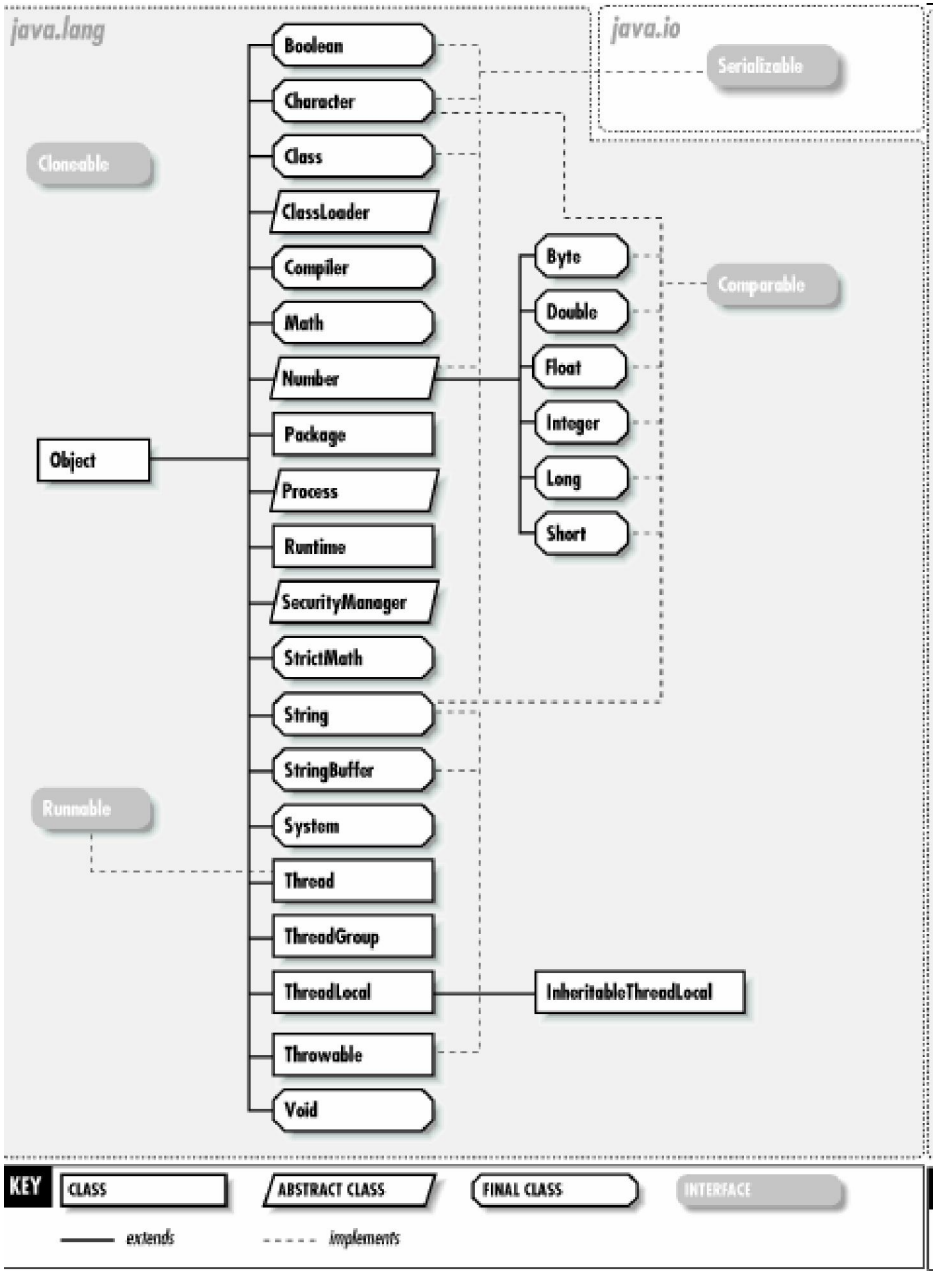
```
class Angajat: public Persoana, public Job {
 protected:
  int vechime;

 public:
  Angajat(const char* nume, const char* functie,
          int v = 0): Persoana(nume), Job(functie) {
    vechime = v;
    cout << "Apel constructor Angajat\n";
  }

  Angajat(const Angajat& ob): Job(ob),
                              Persoana(ob) {
    vechime = ob.vechime;
    cout << "Apel constructor copiere Angajat\n";
  }

  Angajat& operator= (const Angajat& ob) {
    if (this != &ob) {
      Persoana::operator= (ob);
      Job::operator= (ob);
      vechime = ob.vechime;
    }
    cout << "Apel operator= Angajat\n";
    return *this;
  }

  void print() {
    Persoana::print();
    Job::print();
    cout << "Vechime: " << vechime << endl;
  }
};
```

Exemplul 2 (cont.)

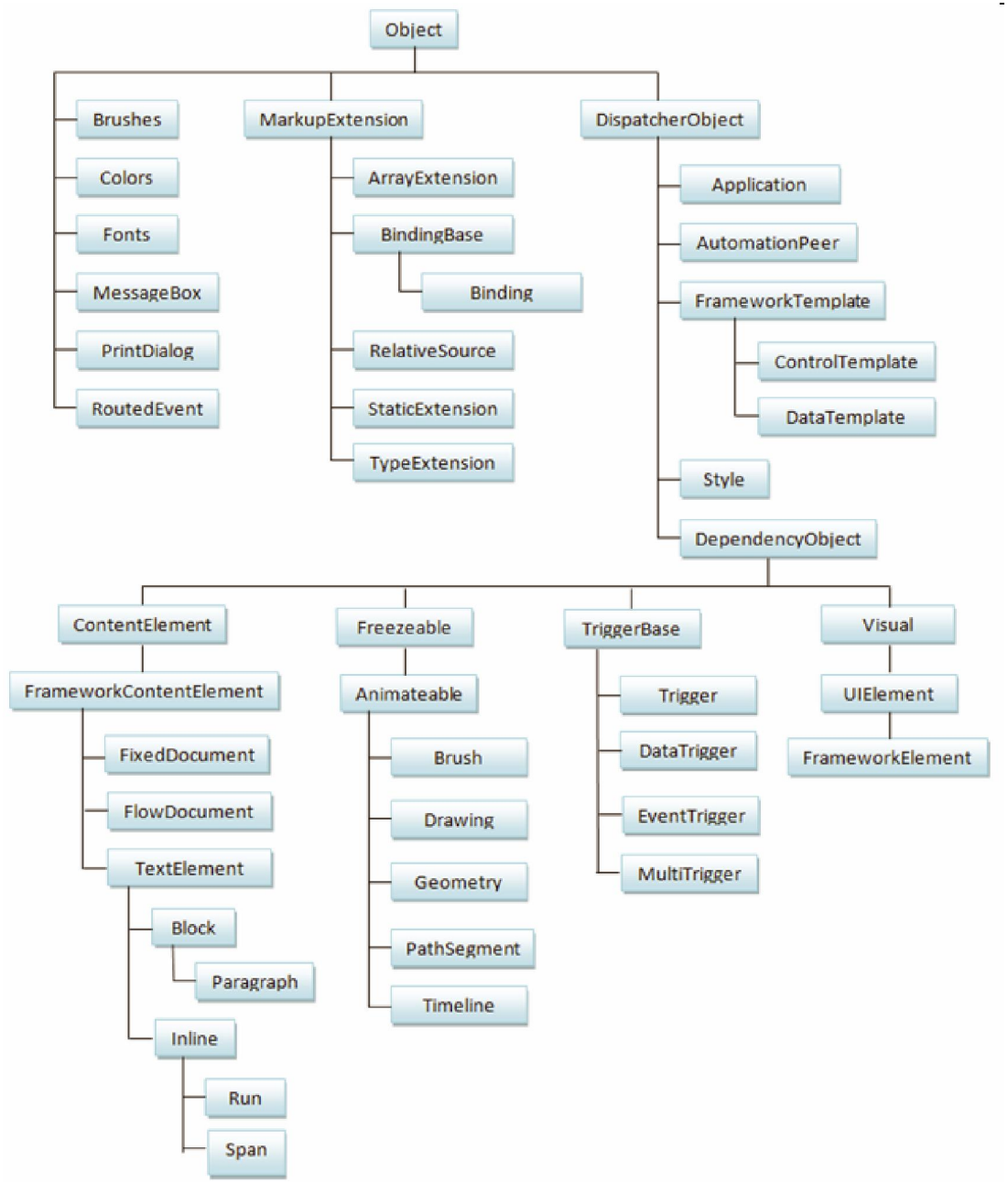
```
int main() {
  Angajat a1("Tedy", "programator", 1);
  a1.print();

  Angajat a2 = a1;
  a2.print();

  Angajat a3("Gheorghe", "tester", 4);
  a3 = a2;
  a3.print();

  return 0;
}
```

```
Apel constructor Persoana
Apel constructor Job
Apel constructor Angajat
Nume:Tedy
functie:programator
Vechime:1
Apel constructor copiere Persoana
Apel constructor copiere Job
Apel constructor copiere Angajat
Nume:Tedy
functie:programator
Vechime:1
Apel operator= Persoana
Apel operator= Job
Apel operator= Angajat
Nume:Tedy
functie:programator
Vechime:1
Apel destructor Job
Apel destructor Persoana
Apel destructor Job
Apel destructor Persoana
Apel destructor Job
Apel destructor Persoana
```

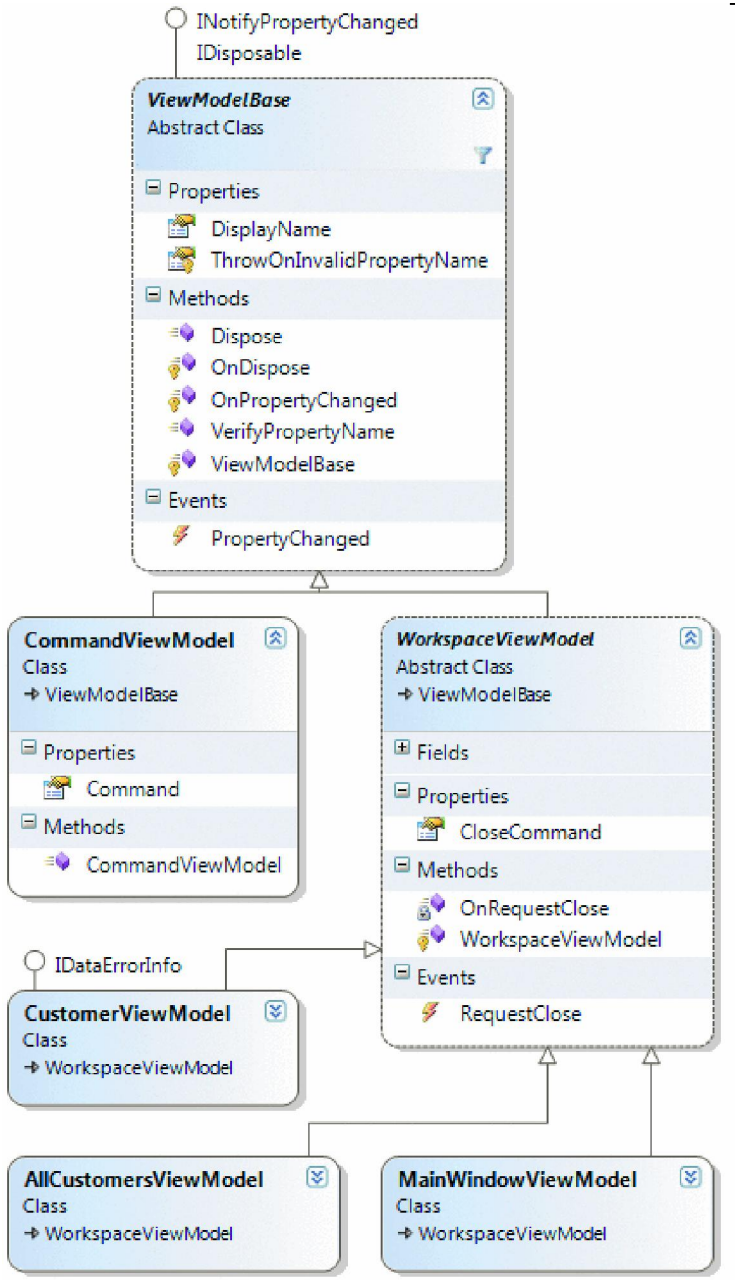
Ierarhia de clase din pachetul `java.lang`



Ierarhie de clase WPF (C#)



Şablonul de proiectare MVVM



Relația de moştenire - Concluzii

- Relația de moştenire este o relație între două sau mai multe clase, în cadrul căreia clasa derivată moşteneşte comportamentul şi atributele unei clase / unor clase existente.
- Moştenirea are drept scop reutilizarea codului existent fără modificări sau cu mici modificări.
- Obiectele clasei derivate întotdeauna pot fi tratate drept obiecte ale clasei de bază.
- De exemplu, dacă avem două clase, `Persoana` şi `Student`, unde clasa `Student` este derivată din clasa `Persoana`: un obiect de tip `Student` poate fi utilizat ca un obiect de tip `Persoana`.

Relația de moştenire - Concluzii

- O clasă derivată poate, la rândul ei, să fie clasă de bază pentru o altă clasă, ce poate moşteni de la ea.
- Orice modificare în clasa de bază, afectează toate clasele derivate.
- Orice modificare în clasa derivată, nu afectează clasa de bază.
- Toate proprietățile clasei de bază sunt disponibile şi în clasa derivată.
- Proprietățile caracteristice clasei derivate nu sunt disponibile şi în clasa de bază.

Relația de moştenire - Concluzii

- Nivele de acces pentru date şi funcții:
  - ***public*** – datele şi funcțiile membre aflate sub influența modificatorului public, pot fi accesate *atât din interiorul clasei cât şi din afara clasei*.
  - ***protected*** - datele şi funcțiile membre aflate sub influența acestui modificator pot fi accesate atât *din interiorul clasei (şi din funcțiile prietene) cât şi din clasele derivate*.
  - ***private*** – datele şi funcțiile membre aflate sub influența acestui modificator pot fi accesate numai *din interiorul clasei (şi din funcțiile prietene).* NU pot fi accesate din afara clasei. *În mod implicit datele şi funcțiile sunt private*.

Temă

- Implementați clasele `Student, Profesor` ce derivă din clasa `Persoana`.
- Implementați o ierarhie de clase ce reprezintă diferite figuri geometrice din plan (`Shape`, `Circle`, `Rectangle`, `Triangle`).
- Implementați ierarhia de clase `Telefon`, `TelefonFix`, `TelefonMobil`.
- Implementați ierarhia de clase `Instrument`, `InstrumentDeSuflat`, `InstrumentCuCorzi, InstrumentDePercutie, Saxofon, Trompeta`.
- Implementați ierarhia de clase `Caruta`, `Bicicleta`, `Autoturism` (se vor introduce şi alte clase, dacă sunt necesare).
- Implementați ierarhia de clase `Animal, Vertebrat, Mamifer, Pasare`, `Caine`, `Pisica`, `Vultur`.

Referințe

- http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/oop/streams/streams.htm
- https://docstore.mik.ua/orelly/java-ent/jnut/ch12_01.htm
- https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern
- https://soumya.wordpress.com/2010/01/10/wpf-simplified-part-10-wpf-framework-class-hierarchy/