Programare in C++, POO

4. DERIVARE SI MOSTENIRE

4.1 Definirea de clase derivate

O clasa D, derivata dintr-o alta clasa B (de baza), mosteneste de la clasa B datele si functiile publice (cu cateva exceptii), dar poate sa adauge date si/sau functii proprii si sa redefineasca oricare dintre metodele mostenite, in functie de cerintele clasei derivate D. O clasa derivata poate servi drept clasa de baza pentru derivarea altor clase.

Dintr-o clasa B se pot deriva (pot descinde) mai multe clase derivate D1, D2,.., fiecare cu caracteristicile ei proprii.

Derivarea constituie un mod total nou de reutilizare si adaptare a unor module existente: programatorul nu intervine pentru modificari intr-o clasa existenta C, dar isi defineste o clasa D, derivata din C, in care isi exprima noile cerinte. Derivarea permite "specializarea" unei clase intr-un anumit sens, dorit de utilizator.

Limbajul C++ permite doua moduri de derivare a claselor: derivarea publica si derivarea proprie ("private"). Cel mai frecvent se foloseste derivarea publica, care pastreaza pentru componentele mostenite acelasi nivel de accesibilitate din clasa de baza. Deci, metodele si datele publice din clasa de baza B raman publice si in clasa derivata public D, ceea ce lasa deschisa posibilitatea utilizarii ulterioare a clasei D drept clasa de baza pentru derivarea altor clase.

Datele si metodele declarate "private" intr-o clasa B nu sunt accesibile claselor derivate din B, dar sunt accesibile claselor prietene lui B. In schimb, datele declarate "protected" in clasa B sunt accesibile claselor derivate din B, dar nu si altor clase "straine". In concluzie, la definirea unei clase trebuie sa anticipam posibilitatea de a deriva din ea alte clase si care membri vor fi mosteniti. De multe ori se defineste simultan o familie (o ierarhie) de clase avand una sau cateva clase de baza, cu deschidere spre alte derivari posibile.

Pentru definirea unei clase D, derivata public dintr-o clasa de baza B se foloseste urmatoarea sintaxa:

```
 // clasa D este derivata din clasa B
class D : public B {
   ... // date proprii ale clasei D
public:
   ... // metode specifice clasei D
};
```

Daca nu se foloseste explicit cuvantul cheie "public", atunci se considera implicit ca este o derivare "private".

O clasa derivata D dintr-o clasa B are urmatoarele drepturi:

- Sa utilizeze membri publici si protejati ai clasei de baza B.
- Sa adauge noi date si/sau functii la cele mostenite.
- Sa redefineasca oricare dintre functiile mostenite, pastrand numele lor.

Clasa derivata nu mosteneste de la clasa de baza:

- Functiile constructor
- Operatorul de atribuire

Aceste functii vor fi definite de utilizator sau vor fi adaugate de catre compilator, in lipsa unor definitii explicite.

Exemplu: O clasa "lista ordonata" derivata dintr-o clasa "lista", la care orice adaugare se face la sfarsitul listei.

```
  // liste neordonate
class List {
private:
   // clasa "Nod" cu toti membrii accesibili ptr. List
  struct Nod {
    int val;        // valoare nod
    Nod * leg;      // legatura la nodul urmator
    Nod (int x) { val=x; leg=NULL;} // constructor
  };
protected:           // ptr a fi folosit de clasele derivate
   Nod * cap;        // cap de lista
public:
   List() {cap = new Nod(0);};
   ~List() ;
   void add (int);
   void print() ;
};
  // adaugare la sfirsit
void List:: add (int x ) {
 Nod* nou = new Nod(x);
 Nod* p=cap;
 while (p->leg !=NULL) {
  p=p->leg;
 }
 p->leg=nou;
}
  // destructor
List::~List () {
 Nod* p = cap, *q;
 while (p != NULL) {
   q=p;
   p=p->leg;
   delete q;
  }
}
  // afisare lista
void List:: print() {
 Nod * p= cap->leg;
 while (p !=NULL) {
   cout << p->val << " ";
  p=p->leg;
 }
 cout << endl;
}
  // liste ordonate
class ListOrd: public List {
 // aceleasi date
public:
   // metoda de adaugare redefinita
```

```
void add (int);
  // celelalte metode se mostenesc
};
void ListOrd:: add (int x) {
 Nod* nou = new Nod(x);
 Nod* p=cap; Nod * q;
 while (p!=NULL && x >p->val) {
  q=p; p=p->leg;
 }
 q->leg=nou; nou->leg=p;
}

void main() {
  ListOrd list;
  for (int k=6;k>=1;k--)
    list.add(k);
  list.print();
}
```

4.2 Constructori pentru clase derivate

O clasa derivata are propriul sau constructor, chiar daca efectul sau se reduce la efectul constructorului clasei de baza.

La construirea unui obiect dintr-o clasa derivata D ordinea de apelare implicita a functiilor constructor este:

- Constructorul clasei de baza B
- Constructorul clasei derivate D

La distrugerea unui obiect dintr-o clasa derivata D ordinea de apelare implicita a functiilor destructor este inversa:

- Destructorul clasei derivate D
- Destructorul clasei de baza B

Constructorul unei clase derivate are o sintaxa speciala care sa permita transmiterea de date la constructorul clasei de baza, executat inaintea sa:

```
D (tip x): B (x) { ... }
```

Exemplu: O clasa "istiva" derivata dintr-o clasa vector "ivector".

```
  // clasa vector de intregi
class ivector {
protected:
   int * vec;
   int n,nmax,inc;
   // extindere vector
  void extend () {
    int * aux=vec;
    nmax+=inc;
    vec = new int[nmax];
    for (int i=0;i<n;i++)
      vec[i]= aux[i];
    delete [] aux;
  }
public:
```

```
  ivector (int max=10, int incr=10)
   { nmax=max; inc=incr; n=0;
     vec= new int[nmax];
   }
  ~ivector () { delete [] vec;}   // destructor
  int & operator [] (int i)
   { assert (i >= 0 && i < nmax);
     return vec[i];
   }
  void add (int elem) {
    if ( n==nmax)
      extend();
    vec[n++]=elem;
  }
  int size() { return n; }
};
  // clasa stiva de intregi
class istiva: private ivector {
public:
 istiva (int ni=10, int inci=10): ivector(ni,inci) {};
 void push (int x) {
   add(x);
 }
 int pop () {
   assert ( n >0);
   return vec[--n];
 }
 int empty () { return n==0;}
} ;
  // utilizare stiva
void main () {
  istiva s(3);
  for (int i=1;i<=10;i++)
    s.push(i);
  while ( ! s.empty())
   cout << s.pop() << ' ';
  cout << endl;
}
```

Motivul derivarii "private" este acela de a interzice utilizarea unor functii ale clasei "ivector" pentru obiecte din clasa "stiva"; de exemplu, nu se poate folosi operatorul [] de acces direct la elementele stivei.

Justificarea definirii clasei "stiva" ca o clasa derivata este acela ca se poate refolosi functia "add" pentru adaugare la stiva (eventual si alte operatii din clasa de baza "ivector"). In plus, s-ar putea refolosi o functie "print" de afisare fara modificare a unui vector si eventual alte metode mostenite.

Chiar daca nu se transmit date de la un constructor la altul, se pastreaza aceeasi sintaxa:

```
D () : B () { ... }
```

Construirea unui obiect dintr-o clasa derivata D difera de construirea unui obiect din clasa de baza B atunci cind clasa D contine variabile suplimentare fata de B, care trebuie alocate si initializate.

Constructorul implicit, fara parametri, generat de compilator pentru o clasa derivata apeleaza automat constructorul implicit al clasei de baza. Un constructor cu parametrii pentru o clasa derivata trebuie definit explicit si trebuie sa fie urmat de ":" si de constructorul clasei de baza.

4.3 Comparatie intre derivare si agregare

Prin "agregare" sau "compozitie" se intelege utilizarea de obiecte ale unei clase B ca variabile membre ale unei clase compuse C.

Atat derivarea cat si compunerea permit reutilizarea functionalitatii clasei de baza, deci refolosirea unor functii din clasa de baza.

Exemplul urmator arata o clasa "stiva" care contine un obiect de tip "ivector".

```
  // clasa vector de intregi
class ivector {
protected:
   int * vec;
   int n,nmax,inc;
   void extend () {... }
public:
  ivector (int max=10, int incr=10)
   { nmax=max; inc=incr; n=0;
     vec= new int[nmax];
   }
  ~ivector () { delete [] vec;}   // destructor
  int & operator [] (int i)
   { assert (i >= 0 && i < nmax);
     return vec[i];
   }
  void add (int elem) {
    if ( n==nmax)
      extend();
    vec[n++]=elem;
  }
  int size() { return n; }
  void remove () {
    vec[--n]=0;
  }
};
  // clasa stiva de intregi
class istiva {
 ivector st;
public:
 istiva (int ni=10, int inci=10): st(ni,inci) {};
 void push (int x) {
   st.add(x);
 }
 int pop () {
   int sz=st.size();
   assert ( sz >0);
   int x= st[sz-1];
   st.remove();
   return x;
 }
 int empty () { return st.size()==0;}
} ;
```

Utilizarea acestei clase "stiva" este similara cu utilizarea clasei "stiva" derivate din "ivector".

Clasa compusa "stiva" nu are acces decit la membrii publici din clasa "ivector", ceea ce a necesitat adaugarea unei functii "remove" in clasa "ivector" si o exprimare mai greoaie in functia "pop".

In exemplul anterior trebuie remarcat constructorul clasei compuse "stiva", care seamana cu constructorul unei clase derivate, dar in loc de a folosi numele clasei de baza se folosesc numele obiectelor din compunerea clasei (care trebuie construite inaintea obiectului compus).

Constructorul clasei compuse "stiva" se mai poate scrie si astfel:

```
stiva::stiva (int di=10, int inci=10) {
   st= ivector(di,inci);
}
```

Un alt exemplu de legatura intre constructorii clasei compuse si constructorii claselor obiectelor componente este urmatorul:

```
  // clasa ptr. siruri de caractere
class String {
  char * str;   // adresa sir
public:
  String (char * s) {   // un constructor
    int sz= strlen(s); str = new char[sz];
    strcpy (str,s);
  }
  ...  // alte metode
};
 // clasa ptr o data calendaristica
class datcal {
  int z,l,a;     // zi,luna, an
public:
  datcal (int z1,int l1,int a1){
    z=z1; l=l1; a=a1;
  }
  ..  // alte metode
};
  // clasa pentru persoane
class Pers {
  String nume;     // nume persoana
  datcal nascut;   // data nasterii
public:
  Pers (char * numep, int zn, int ln, int an) :
    nume(numep), nascut(zn,ln,an) { }
  ... // alte metode
};
```

Alegerea intre derivare si agregare este determinata de interfetele publice ale celor doua clase: Daca noua clasa refoloseste o mare parte din interfata clasei vechi, atunci se recomanda derivarea; daca noua clasa are o interfata diferita de clasa veche dar refoloseste o parte din functiile acestei clase, atunci se prefera agregarea.

Cu alte cuvinte, derivarea permite reutilizarea interfetei unei clase iar agregarea permite reutilizarea functionalitatii unei clase.

Se mai spune ca relatia dintre o clasa derivata D si o clasa de baza B este o relatie de tipul "D este un fel de B" sau "D este un caz particular de B". Relatia dintre o clasa compusa C si clasa continuta B este o relatie de tipul "C contine un B".

Relatia dintre o stiva si un vector este fie "o stiva contine un vector", fie " o stiva este un fel de vector" (un caz particular de vector). Totusi, o stiva nu foloseste aproape nimic din interfata unui vector, deci ar fi preferabila compozitia in acest caz.

O problema specifica limbajului C++ legata de compozitie este aceea ca avem de ales intre o variabila de tip B si o variabila de tip pointer la B la definirea clasei compuse C. Exemplu:

```
  // varianta ptr. clasa stiva
class istiva {
 ivector * pst;   // pointer la un vector
public:
   // constructor ptr stiva
 istiva (int ni=10, int inci=10) {
   pst = new ivector (ni,inci) ;
 }
 void push (int x) {
   pst->add(x);
 }
 int pop () {
   int sz=pst->size();
   assert ( sz > 0);
   int x= (*pst)[sz-1];
   pst->remove();
   return x;
 }
 int empty () { return pst->size()==0;}
} ;
```

Din cauza notatiilor mai complicate s-ar parea ca nu exista motive ca sa folosim pointeri la obiecte ca membri ai altor clase, dar un pointer permite o flexibilitate mai mare la executie - el poate fi modificat cu adresele unor obiecte diferite si chiar de tipuri diferite.

4.4 Derivarea creaza subtipuri

Derivarea creaza o relatie intre tipuri, in sensul ca tipul derivat D este un subtip al tipului de baza B, iar o variabila de tip pointer la B poate fi inlocuita fara conversie printr-o variabila pointer la D. Aceasta posibilitate poate fi asemanata cu posibilitatea de a inlocui un pointer la "void" cu orice alt tip de pointer si poate fi folosita atat pentru a defini clase container cu date de orice tip, cat si functii ce pot primi date de orice tip.

Conversia de la tipul D* (sau D&) la tipul B* (sau B&) se face automat dar conversia inversa, de la B* la D* foloseste operatorul de conversie ("cast"). Explicatia este aceea ca un obiect de tip D contine toate datele unui obiect de tip B, deci folosind o adresa de obiect D putem accesa toate

datele obiectului B. Conversia inversa poate conduce la erori, iar in C++ compilatorul nu face nici o verificare la conversia de pointeri.

Posibilitatea de a inlocui o variabila de tip B* prin orice variabila de un tip D* (clasa D fiind derivata direct sau indirect din B) a condus la crearea unor familii de clase, de forma unui arbore care are ca radacina o clasa de baza cu foarte putine functii si fara date, din care sunt derivate direct sau indirect toate celelalte clase din familie. In felul acesta se creaza o familie de tipuri compatibile.

Exemplul urmator arata cum se poate defini si utiliza un vector de pointeri la obiecte "Object", in care se pot introduce adrese de orice obiecte derivate din "Object".

```
 // clasa de la baza ierarhiei
class Object {
public:
  void print() { cout << "? ";}
} ;
  // un vector de pointeri la Object
class Vector {
  Object* * vec;   // adresa vector cu componente Object*
  int n, nmax;     // dimensiune efectiva si maxima vector
public:
  Vector (int size) { vec=new Object* [nmax=size]; n=0; }
  ~Vector() { delete [] vec; }
  void print ();
  void add (Object* p) { vec[n++]=p; }
  Object* operator[] (int i) { assert (i <n); return vec[i];}
  int size() { return n;}
};
    // siruri de caractere
class String: public Object {
  char * str;
public:
  String (char * s) { str =new char[strlen(s)]; strcpy (str,s);}
  ~String() { delete str;}
  void print () { cout << str << ' ';}
 };
   // creare-afisare vector de nume
void main () {
 Vector a(10);  char sir[20];
 cout << "lista de nume: \n";
 while (cin) {
  cin >> sir;
  a.add (new String(sir));  // sirurile in vectorul a
 }
 for (int i=0;i< a.size();i++){
    String * pnume= (String*)a[i];
    pnume->print();
 }
 cout << endl;
}
```

La introducerea in vector se pierde tipul datelor adresate de pointeri, pentru ca se face o conversie automata de la tipul derivat (String*) la tipul de baza (Object*). Pentru afisarea corecta a continutului vectorului trebuie restabilit tipul pointerilor memorati, printr-o conversie explicita de la tipul Object* la tipul Nume*. Se putea scrie direct:

```
 ((String*)a[i])->print();  // toate parantezele sunt necesare
```

Varianta anterioara nu mai functioneaza daca in vector memoram date de tipuri diferite (prin adresele acestor date), pentru ca nu mai avem informatiile necesare restabilirii tipului initial al pointerilor.

In C++ standard nu exista posibilitatea determinarii tipului unei variabile la executie ("Run Time Type Identification" =RTTI), deoarece toate informatiile despre tip se pierd dupa compilare. Este posibil sa se introduca in fiecare clasa o metoda (sau o variabila) care sa aiba ca rezultat numele si/sau tipul clasei.

Solutia specifica POO foloseste functii virtuale si o legare "tarzie" a functiilor virtuale apelate.

4.5 Functii virtuale si polimorfism

O metoda declarata "virtual" intr-o clasa de baza poate fi redefinita in clasele derivate, fara a schimba tipul functiei si lista de argumente. Apelul unei functii virtuale, prin intermediul unui pointer sau printr-o referinta la clasa de baza, produce executia unei secvente diferite, in functie de tipul pointerului sau referintei. Exemplu:

```
  // clasa de baza
class Object {
public:
  virtual void print()  { cout << this << endl;}
};
  // o clasa derivata
class String : public Object {
 ...
public:
  void print() { ... }
};
  // efectul apelului unei functii virtuale
void main () {
   String s; String & sref; String * sptr;
   Object obj= s; Object& obref= sref; Object* obptr= sptr;
   obj.print();     // apeleaza functia Object::print()
   obref.print ();   // apeleaza functia String::print()
   obptr->print();   // apeleaza functia String::print()
}
```

Se recomanda ca o metoda sa fie declarata virtuala atunci cand ea este folosita de alte functii (ale aceleiasi clase sau de alte functii) si cand actiunea functiei este diferita in clasele derivate. Exemplu:

```
  // clasa de baza
class Object {
public:
  virtual int equals (Object& obj)  { return this== &obj; }
};
  // functie operator ne-virtuala
int operator == (Object ob1, Object ob2) {
  return ob1.equals(ob2);
}
```

Operatorul "==", definit ca mai sus poate fi folosit pentru orice clasa derivata direct sau indirect din clasa "Object", care redefineste metoda "equals" corespunzator cu datele clasei.

O functie declarata "virtual" in clasa de baza isi va pastra numele, tipul si argumentele in clasele derivate, dar va fi redefinita in fiecare clasa pentru a produce un alt efect. Exemplu:

```
  // clasa String ptr siruri de caractere
class String:public Object {
 char * str;
public:
 String (char* s="") {           // un constructor
  str= new char[strlen(s)+1];
  strcpy(str,s);
 }
 int equals (Object& obj) {
   String & strobj= *(String*) &obj;
   return strcmp (str, strobj.str)==0 ? 1:0;
 }
};ÿ
 ... // utilizare
 String s1 ("unu"), s2("unu");
 if ( s1==s2) cout << "egale \n";
 else cout << "diferite \n";
```

Definitia functiei String::equals din exemplul anterior este tipica pentru redefinirea unei metode virtuale din clasa "Object": parametrul trebuie sa fie de acelasi tip ca in declaratia din clasa "Object", dar pentru acces la datele din clasa "String" trebuie facuta conversia de la clasa "Object" la clasa "String".

Asocierea dintre apelul unei metode virtuale si corpul functiei se face la executie si nu la compilare, deci este o legare "tarzie".

O functie virtuala este numita si functie polimorfica, pentru ca are mai multe forme (definitii) dar un singur nume si mod de folosire.

Exemplul urmator arata necesitatea metodelor virtuale si legaturii intarziate intre apel si functia virtuala apelata. Metoda "print" va fi ulterior declarata ca metoda virtuala.

```
  // clasa de la baza ierarhiei
class Object {
public:
  void print() { cout << "? ";}   // afisare date din clasa
} ;
  // un vector de pointeri la Object
class Vector {
  Object* * vec;   // adresa vector cu componente Object*
  int n, nmax;     // dimensiune efectiva si maxima vector
public:
  Vector (int size) { vec=new Object* [nmax=size]; n=0; }
  ~Vector() { delete [] vec; }
  void print ();
  void add (Object* p) { vec[n++]=p; }
};
   // afisare vector
void Vector::print () {
  cout << "["
    for (int i=0; i<n; i++) {
      vec[i]->print();
      cout <<",";
    }
  cout << "\b]\n";
```

```
}
```

Pentru exemplificare vom folosi doua clase derivate din "Object" si un program care completeaza un vector de nume si un vector de numere, dupa care afiseaza cei doi vectori.

```
    // numere intregi
class Int : public Object {
  int x;
public:
  Int ( int val) { x=val;}
  void print () { cout << x << ' '; }
};
       // program de test
void main () {
 Vector a(10), b(20);
 char sir[20]; int x;
 do {
  cout << "sir: "; cin >> sir;
  a.add (new String(sir));  // sirurile in vectorul a
  if (cin==0) break;
  cout << "int: "; cin >> x;
  b.add (new Int(x));    // numerele in vectorul b
 } while (cin);
 a.print();
 b.print();
}
```

Programul nu afiseaza datele introduse in cei doi vectori, ci doar semne de intrebare, ceea ce arata ca nu se executa metodele "print" redefinite in clasele "String" si "Int" si se executa metoda "print" din clasa "Orice" in apelul din functia "Vector::print". Explicatia este aceea ca vectorul contine date de tip "Object *" si ca asocierea dintre o metoda si un tip clasa se face la compilare, deci apelul

```
vec[i]->print()
```

este echivalent cu apelul

```
vec[i]-> Object::print()
```

indiferent de tipul datelor aflate la adresa continuta in vec[i].

Ceea ce este necesar aici este ca selectarea functiei apelate sa se faca la executie in functie de tipul datelor adresate de pointer. Deci, daca vec[i] este de tip "String *" atunci trebuie selectat apelul

```
vec[i] -> String::print()
```

Solutia problemei este declararea functiei "print" ca metoda virtuala.

```
  // clasa cu o metoda virtuala
class Object {
public:
  virtual void print() { }
} ;
```

Un caz particular de metoda virtuala este destructorul clasei, care poate avea efecte diferite in clase diferite, dar derivate dintr-o clasa unica. In aceasta situatie se afla clasele container derivate dintr-o clasa colectie abstracta.

Pentru clasele care contin una sau mai multe metode virtuale compilatorul C++ creaza un tabel de pointeri la metodele virtuale ale clasei (VMT =Virtual Method Table). Fiecare obiect dintr-o clasa cu metode virtuale contine, pe linga datele clasei, si un pointer la tabelul VMT al clasei.

Variabila pointer vec[i] duce catre un obiect de tip "String" sau catre un obiect de tip "Int" si deci conduce la tabelul VMT al clasei "String" sau "Int"; din tabelul clasei se extrage adresa functiei "print" pentru clasa respectiva. In felul acesta, o singura variabila pointer vec[i] poate selecta o varianta sau alta a metodei virtuale "print".

4.6 Clase abstracte

Deoarece clasa "Object" nu contine date, nu se poate preciza efectul functiei de afisare; acesta urmeaza a fi definit in fiecare din clasele derivate din "Object". Clasa "Object" serveste doar ca baza pentru derivarea altor clase si nu ar avea sens sa declaram variabile de tipul "Object".

Pentru a cere compilatorului sa verifice acest caz de instantiere gresita a unei clase, se adauga declararii functiilor virtuale nule o egalare cu zero, rezultand functii virtuale "pure":

```
 // clasa cu metode virtuale pure
class Object {
public:
  virtual void print()=0 ;           // afisare
  virtual int egal ( Orice *) =0 ;   // comparatie la egalitate
} ;
```

O clasa cu metode virtuale pure este o clasa abstracta si nu poate fi folosita decat pentru derivarea altor clase. Functiile virtuale pure din clasa abstracta trebuie redeclarate in clasele derivate, chiar fara cuvantul "virtual". Daca functiile virtuale pure nu sunt redeclarate sau sunt declarate cu argumente de tipuri diferite, atunci se considera ca acea clasa derivata mosteneste functia virtuala pura, deci este si ea o clasa abstracta si nu poate fi instantiata.

Exemplul urmator reia mica familie de clase, de data aceasta cu functii virtuale pure.

```
class Object {
public:
  virtual void print()=0;
  virtual int equals (Orice * )=0;
} ;

class Vector {
  Object * * vec;   // vector de pointeri la Object
  int nmax, n;     // lungime maxima si efectiva
public:
  Vector (int size) { vec=new Object * [nmax=size]; n=0; }
  void print ();
  void add (Object * p) { vec[n++]=p; }
  int contains (Object * p) ;   // daca ob. de la adr. p este in vector
```

```
};

class String: public Object {
  char * str;
public:
  ...
  void print () { cout << np;}
  int equals (Object* ob) {
     String * p=(String*) ob;
     return strcmp(str, p->str)==0;
  }
 };

class Int : public Object {
  int x;
public:
  Int ( int val) { x=val;}
  void print () { cout << x << ' '; }
  int equals (Object * ob) { Int * p= (Int *)ob; return x == p->x ;}
};
       // metode clasa vector
int Vector:: contains (Object * ob) {
 for (int i=0; i<n;i++)
   if ( vec[i]->equals (ob))
     return 1;
 return 0;
}
   // afisare vector
void Vector::print () {
    for (int i=0; i<n; i++) {
      vec[i]->print(); cout <<",";
    }
  cout << "\n";
}
```

Metodele Vector::add si Vector::contains pot avea si argumente de tip Object (in loc de Object*), chiar daca vectorul contine pointeri.

De observat ca, in exemplul anterior, clasa "Vector" nu face parte din ierarhia de clase care are la baza clasa "Object", deoarece nu este derivata direct sau indirect din "Object". Daca declaram clasa "Vector" ca fiind derivata din "Object" atunci putem construi o matrice ca un vector de vectori, dar vom fi obligati sa implementam si metoda Vector::egal.

Pentru a obtine o clasa instantiabila derivata dintr-o clasa abstracta trebuie sa definim toate metodele virtuale pure din clasa de baza, chiar daca unele nu sunt necesare sau nu au sens pentru clasa derivata. Din acest motiv, in Java, clasa "Object" din care sunt derivate toate celelalte nu este o clasa abstracta (toate metodele sunt definite intr-un fel).

Mai trebuie observat ca, desi nu se pot declara variabile de tipul clasei abstracte "Object", se pot declara pointeri la tipul "Object". Acesti pointeri vor fi inlocuiti ulterior cu pointeri la o clasa concreta, derivata din "Object".