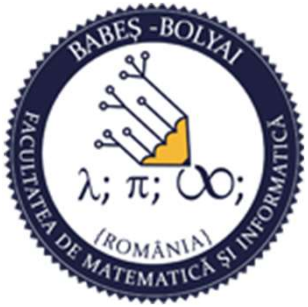

UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică şi Informatică




# Programare orientată obiect

Curs 10

Laura Dioşan

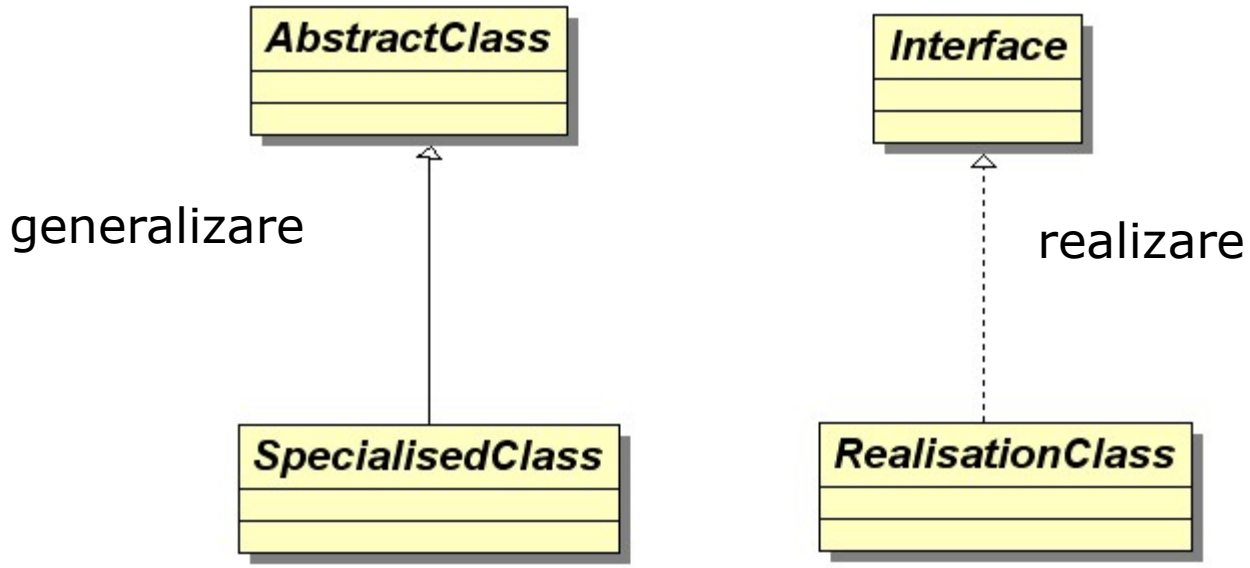
# POO

- Clase
  - Polimorfism
  - Interfeţe
  - Colecţii cu elemente generice
  - Excepţii
  - Spaţii de nume

# Interfața

- descrie comportamentul sau capacităţile unei clase fără a recurge la o implementare particulară
- Reprezintă un contract între furnizor şi clienţi, definind ceea ce este necesar pentru fiecare implementare, dar doar în termeni de servicii care trebuie furnizate, nu şi în modul în care aceste servicii trebuie realizate
- în C++, interfaţa = o clasă abstractă care conţine doar metode virtuale pure
  - poate fi derivată din 0 sau mai multe interfeţe de bază
  - nu poate fi derivată dintr-o clasă de bază (ordinară)
  - poate conţine doar metode publice virtuale pure
  - **nu** poate conţine constructori
  - **nu** poate conţine metode statice
  - **nu** poate conţine date membre

# Diagrama UML



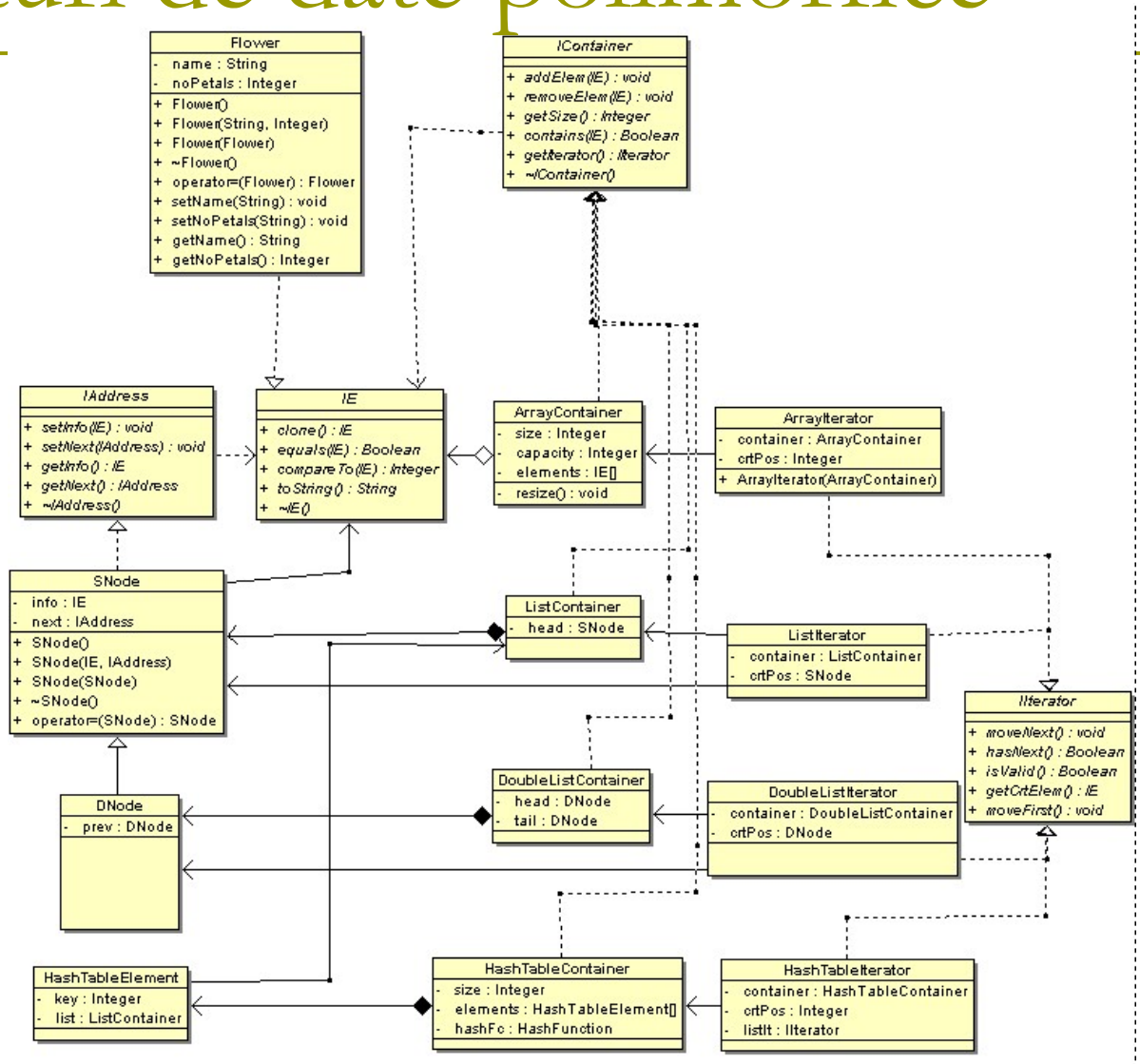
# Containere

- Un container este un obiect care stochează o colecţie de alte obiecte (elementele sale).
- Containerul:
  - gestionează spaţiul pentru elemente
  - oferă funcţii de acces la elemente, direct sau prin intermediul iteratorilor
- Unele containere au funcţii comune şi împart aceleaşi funcţionalităţi
- Alegerea unui anumit tip de container depinde de:
  - funcţionalităţile oferite de container
  - eficienţa (complexitatea) acestor funcţionalităţi

# Clasificarea containerelor

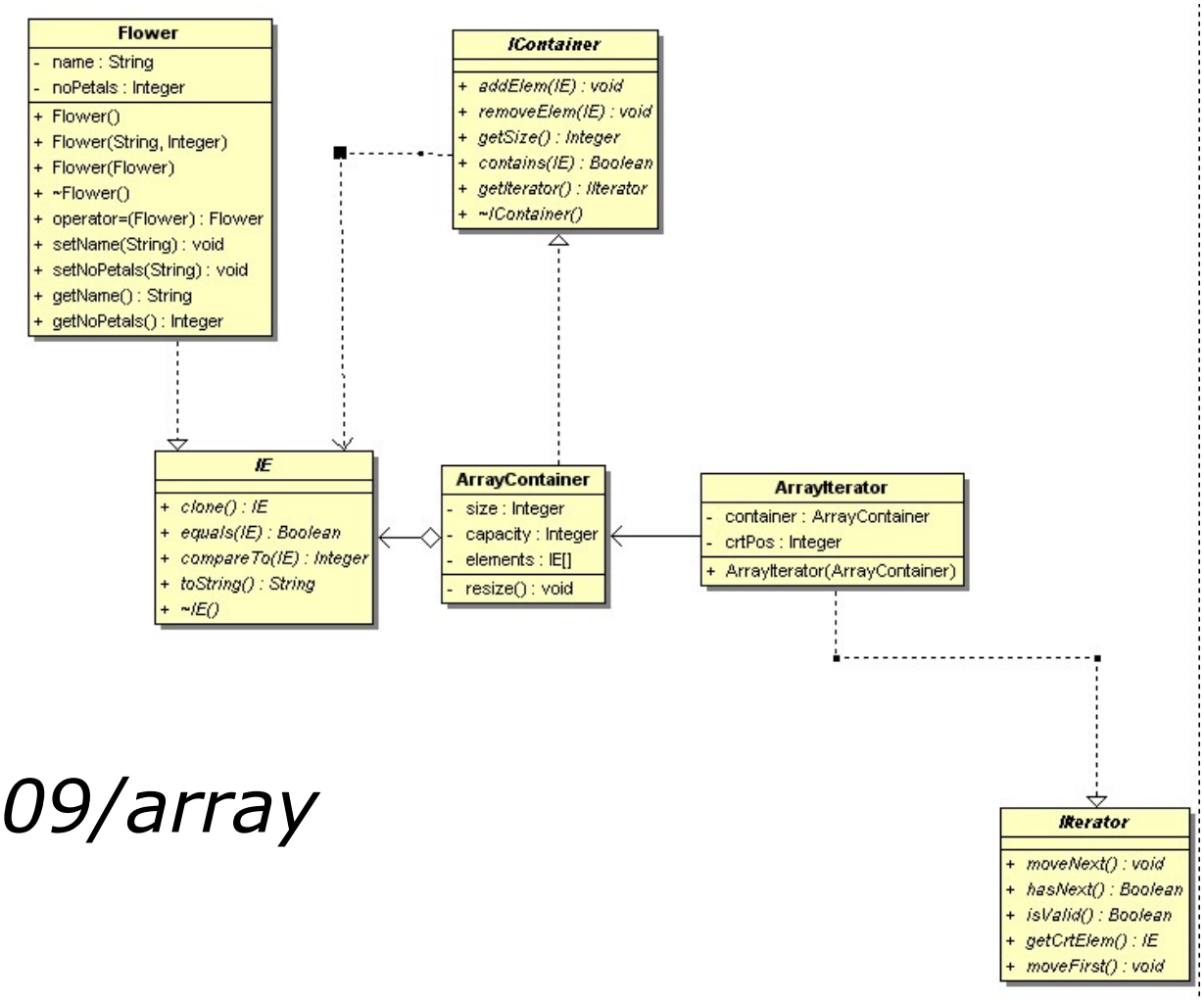
- Containere secvenţiale
  - Vector
  - Listă
  - *Deque* (coadă cu 2 capete)
- Containerere adaptate
  - Stivă
  - Coadă
  - Coadă cu priorităţi
- Containere asociative
  - Multime
  - Mulţime multiplă
  - Dicţionar
  - Dicţionar multiplu
  - Tabelă de dispersie

# Structuri de date polimorfice



# Structuri de date polimorfice

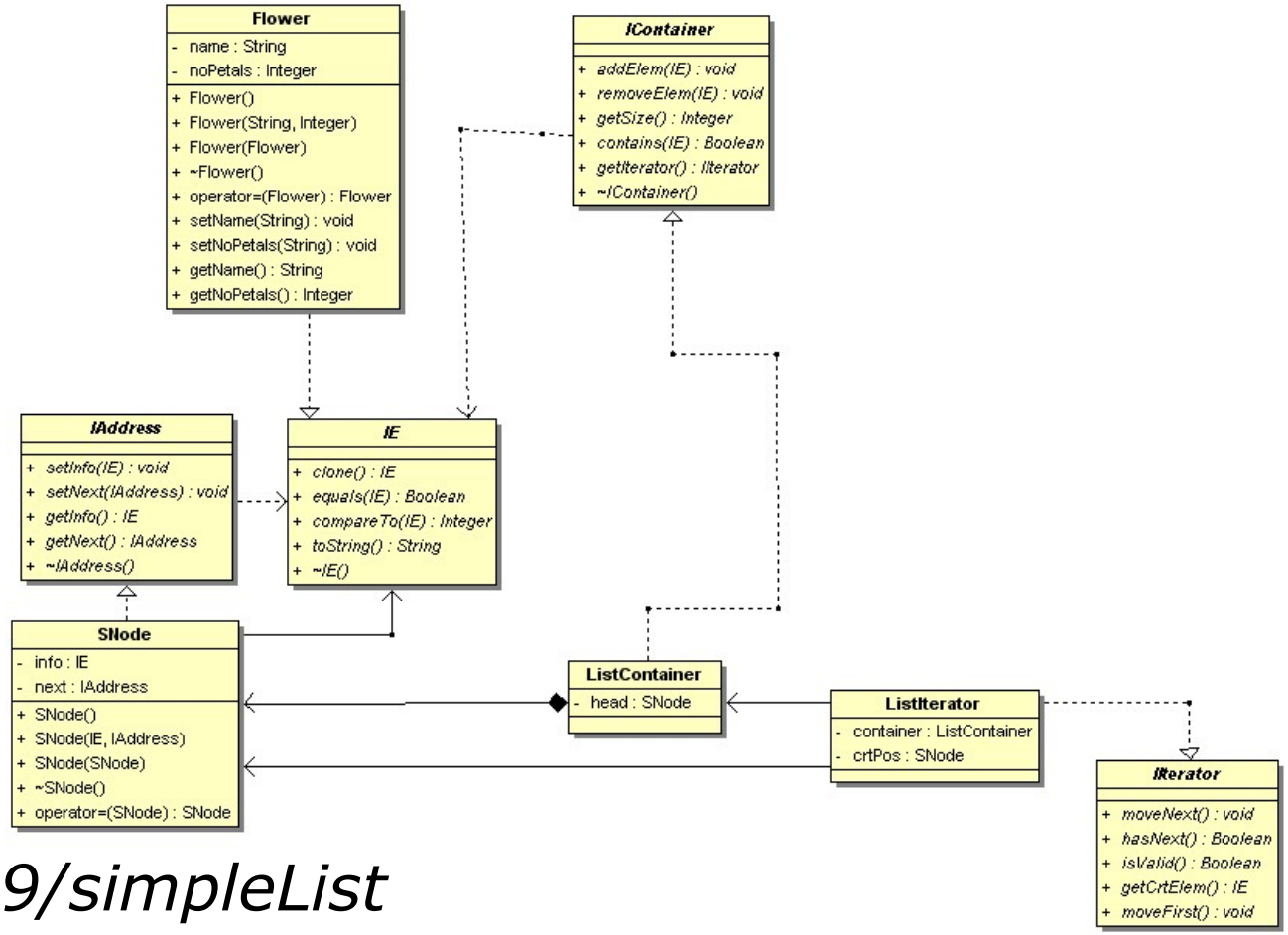
- Şiruri polimorfice




- exemplu în *09/array*

# Structuri de date polimorfice

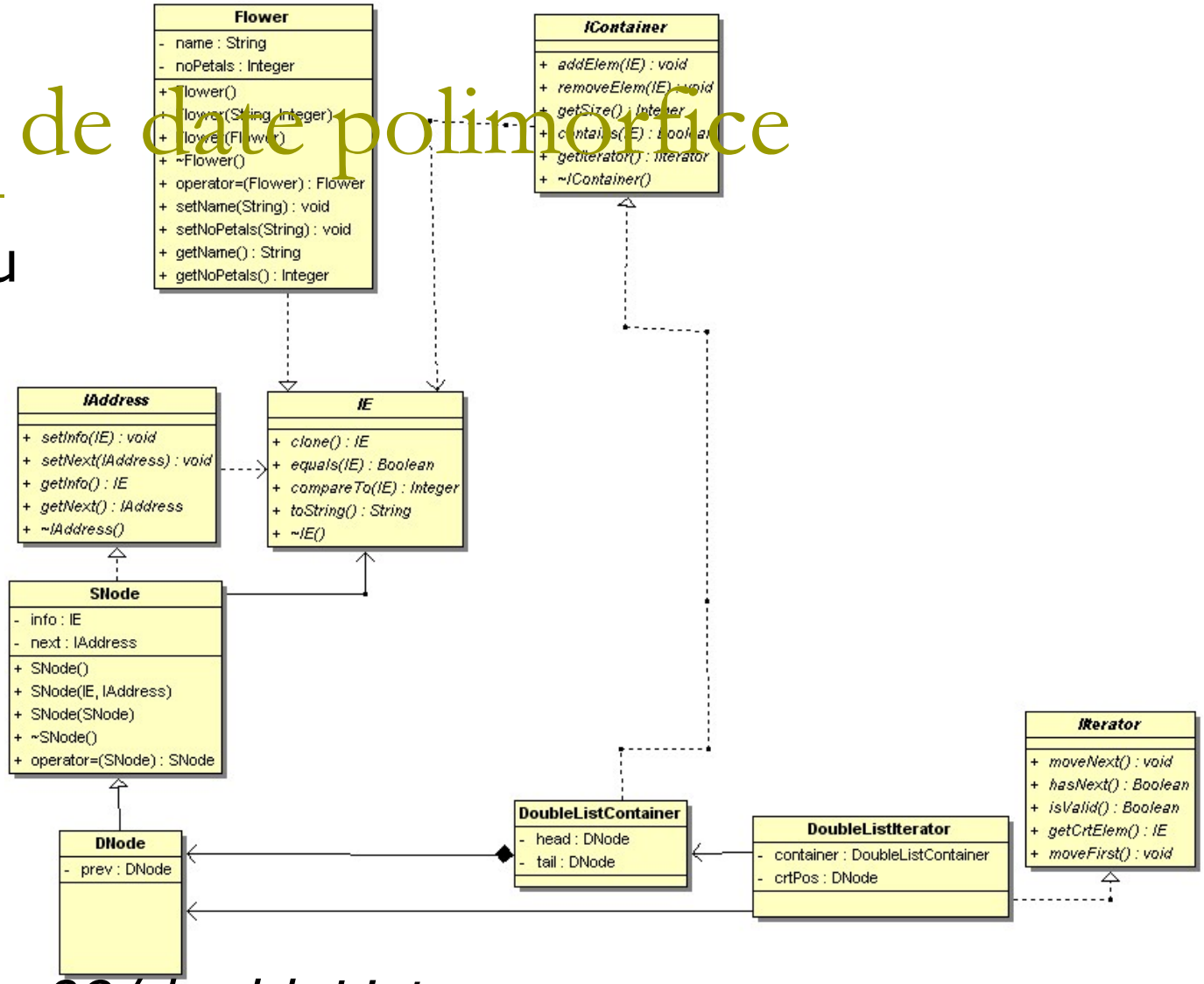
- Liste simplu înlănţuite polimorfice




- exemplu în *09/simpleList*

# Structuri de date polimorfice

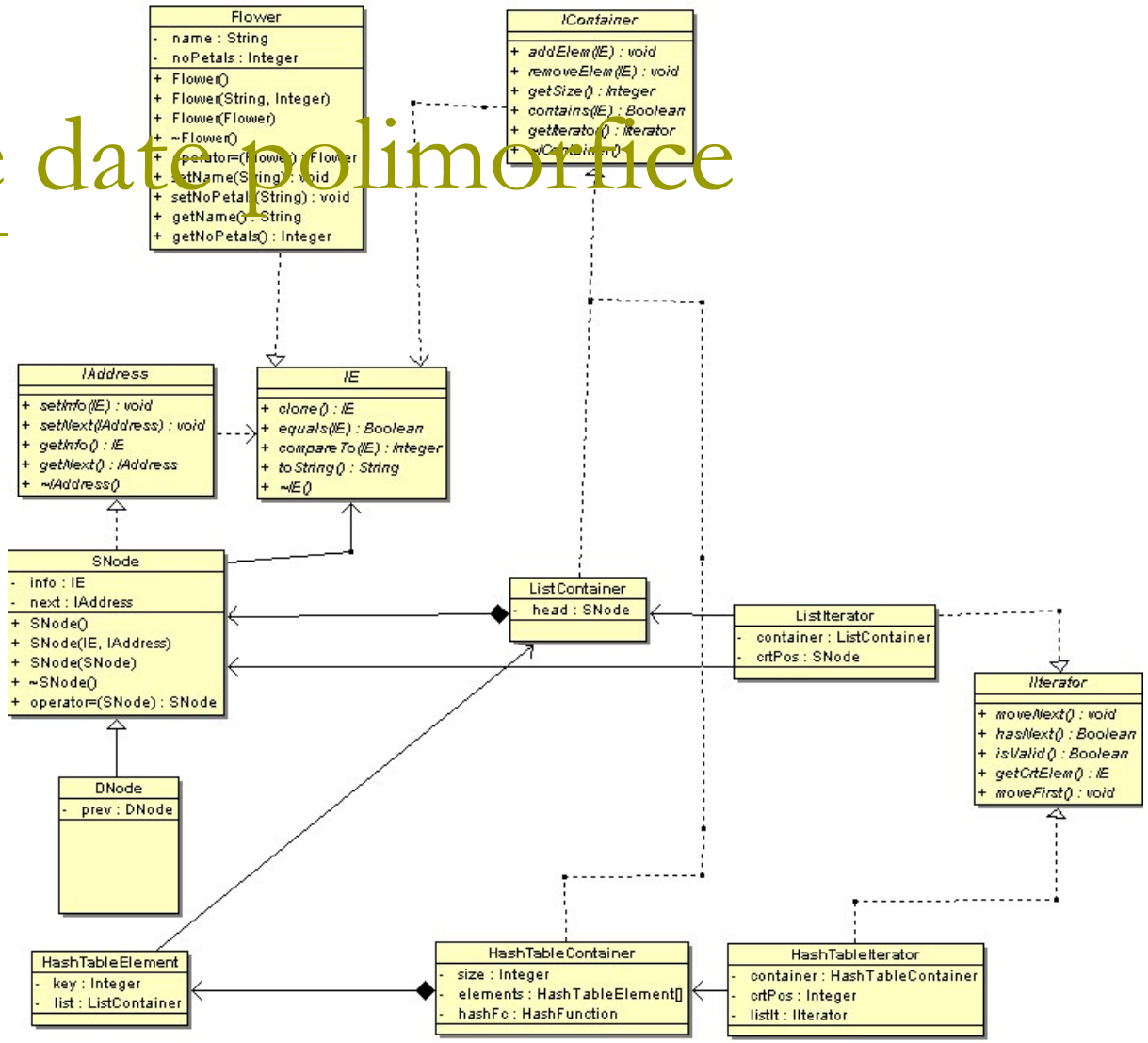
- Liste dublu înlănţuite polimorfice




- exemplu în *09/doubleList*

# Structuri de date polimorfice

- Tabele de dispersie polimorfice




- exemplu în *09/hashTable*

# Structuri de date polimorfice

- Dicţionare polimorfice
  - exemplu în *09/map*
- Dicţionare multiple polimorfice
  - exemplu în *09/multipleMap*

# Temă

- Să se implementeze mulţimi polimorfice
- Să se implementeze mulţimi multiple polimorfice
  - urmând exemplele deja prezentate

# POO

- Excepţii
- Spaţii de nume

# Prinderea erorilor

- Abordarea tradiţională
  - Cum?
    - funcţii speciale (pentru programe mici)
      - *assert*()
      - *assure*()
      - *require*()
    - Steguleţe de eroare (ca variabile globale)
    - Coduri de retur
  - Dezavantaje
    - codul nu este clar
    - anumite erori pot fi ignorate – a se vedea funcţia *printf*
    - codurile de erorare şi cele logice pot coincide
    - Dificultăți de înlănțuire

# Prinderea erorilor

- Abordarea modernă
  - Prinderea excepţiilor
  - Excepţia
    - O situaţie anormală care apare în timpul execuţiei
      - accesarea unui indice în afara limitelor
      - folosirea unui pointer nul
      - împărţire la 0
    - Oferă posibilitatea tratării situaţiilor excepţionale prin transferul controlului unor funcţii speciale numite *handlers*

# Aruncarea unei excepții

- Când apare o situaţie specială
  - Se aruncă o excepţie = se pot trimite informaţii despre respectiva eroare într-un context mai larg prin crearea unui obiect care să conţină acele informaţii şi aruncarea lui în contextul curent
  - A se consulta exemplul din directorul *09/simpleException*
- Ce se întâmplă când o funcţie apelată aruncă o excepţie?
  - Se creează o copie a excepţiei şi se întoarce această copie (chiar dacă funcţia curentă întoarce un alt tip de dată)

# Elementele componente ale unei excepții

- ***try***
  - marchează codul susceptibil de a arunca excepţii
- ***catch***
  - *Handler*-ul excepţiei (locul în care excepţia ajunge şi este tratată)
  - are un parametru → tipul excepţiei
  - Câte un *handler* pentru fiecare tip de excepţie
    - mai multe clauze ***catch***
    - mecanismul de prindere a excepţiilor caută primul *handler* al cărui parametru se potriveşte cu exceţia aruncată
    - se execută doar clauza ***catch*** care s-a potrivit
  - Dacă nu există nici o clauză ***catch*** sau nici una dintre clauzele ***catch*** nu se potrivesc cu excepţia aruncată, atunci programul se termină (anormal: este apelată funcţia *terminate()* sau *abort()*)
- ***throw***
  - aruncă o excepţie (de un anumit tip)

# Elementele componente ale unei excepții

```
try{
    //cod care poate arunca o exceptie
}
catch (ExcepClass &exc){
    //cod care trateaza exceptia
    //exceptia poate fi de tipul ExcepClass sau
    //orice alt tip derivat din acesta
}
catch (...){
    //cod care trateaza orice alte exceptii
}
```

# Excepții - derulare

- Pași:
  - Plasarea codului susceptibil de a arunca excepții în cadrul blocului ***try***
  - Construirea unuia sau mai multor blocuri ***catch*** care să trateze excepțiile
  - Dacă codul din blocul ***try*** (sau orice alt cod apelat din blocul ***try***) aruncă o excepție, se transferă controlul (din blocul try) în blocul ***catch***

- Observații:
  - Blocul care aruncă excepția și blocul o care prinde pot aparține unor funcții diferite
  - Excepțiile se propagă în lanț între funcții
  - Excepțiile pot fi re-aruncate (folosind ***throw***) – în acest caz ele trebuie tratate de alt bloc ***try-catch***

# Excepții

- ***throw*** acceptă un parametru – obiectul aruncat ca excepție și transmis ca argument *handle*r-ului excepției
  - În cazul mai multor *handler*-e, ele pot avea parametri de tipuri diferite

- Tipuri de excepţii
  - Tipuri predefinite (int, char, double, char*)
    - A se consulta exemplul din directorul 09*/simpleException*
  - tipuri definite de utilizator (obiecte)
    - obiectul excepţie este transmis ca:
      - referinţă
      - pointer
    - pentru a evita apelul constructorului de copiere al clasei excepţie

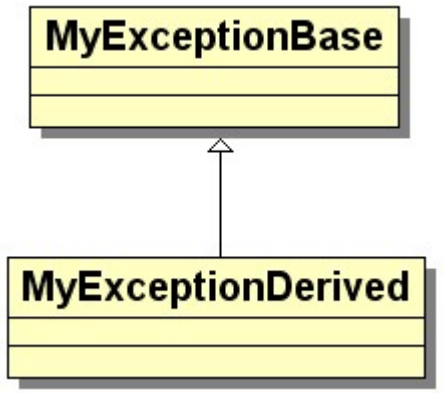
# Excepții ca obiecte

- Obiectul excepție
  - Transmiterea informațiilor despre situația apărută
- Obiectul excepție este distrus abia apoi după ce excepția a fost tratată
  - ultimul bloc catch s-a terminat
- Transmiterea excepțiilor
  - Aruncarea unei excepții prin valoare
  - Prinderea unei excepții prin referință
    - Se evită copierea obiectelor
    - Se păstrează polimorfismul
  - Tipul excepției așteptate în blocul ***catch*** este confruntat cu tipul excepției aruncate
    - Dacă coincide, *handler*-ul prinde și tratează excepția
  - În cazul mai multor blocuri ***catch*** potrivirea se realizează în ordinea apariției
    - Dacă obiectele aruncate aparțin unei ierarhii de tipuri, tipul cel mai particular trebuie prins primul

# Exceptions - example

- A se consulta exemplul din directorul *09/MyExc*
- Aruncarea unei excepţii, prinderea unei singure excepţii
  - funcţia *oneException()*
- Aruncarea mai multor excepţii, prinderea primei excepţii aruncate
  - funcţia *moreThrownExceptions*
- Aruncarea unei excepţii, mai mulţi handleri catch, prinderea primului tip potrivit
  - funcţia *morePossibleExceptions*
- Prinderea oricărei excepţii
  - clauza *catch(…)*
    - trebuie să fie plasată la sfârşit (altfe, va prinde orice fel de excepţie)
  - funcţia *anyException*
- Nici una dintre clauzele ***catch*** nu se potriveşte cu excepţia aruncată
  - funcţia *exceptionThatDoesntMatchAnyCatchClause*

# Polimorfismul excepțiilor




- Presupune că avem relaţia:

- Dacă un cod este susceptibil de a arunca o excepţie de baza, atunci:
  - Dacă se prind doar excepţii de tip părinte (bază), compilatorul va face potrivirea cu clauza ***catch*** repsectivă
  - Dacă se prind ambele tipuri de excepţii (bază şi derivată), compilatorul va face potrivirea cu clauza catch de tip bază

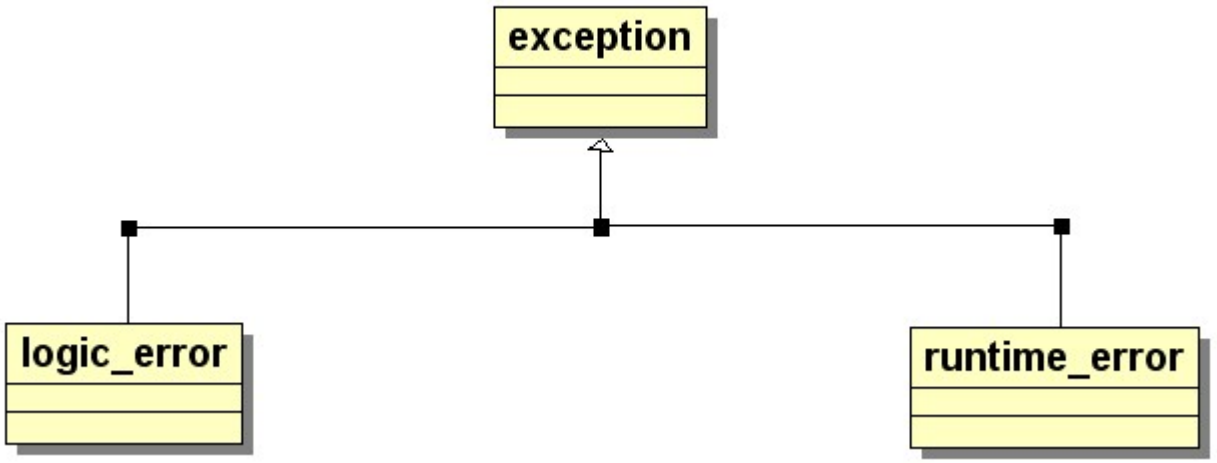
- Dacă un cod este susceptibil de a arunca o excepţie derivată, atunci:
  - Dacă se prind doar excepţii de tip derivat, compilatorul va face potrivirea cu clauza ***catch*** repsectivă
  - Dacă se prind ambele tipuri de excepţii (bază şi derivată), compilatorul va face potrivirea cu prima clauza ***catch*** care se potriveşte (bază sau derivată)
  - Dacă se prind doar excepţii de bază, compilatorul va face potrivirea cu clauza catch de tip bază  (excepţia derivată este convertită la excepţie de bază)

- a se consulta exemplul din directorul *09/ExcPolymorphism*

- În biblioteca standard C++ există class *std::exception*
  - Trebuie suprascrisă metoda *char* what()*

# Excepții standard

- Toate clasele de excepţii standard sunt derivate din clasa **exception** (definită în headerul **<exception>**)
  - **logic_error** – erori logice de programare
    - *e.g.* folosirea unui argument invalid
    - pot fi detectate în prealabil prin inspecţie
  - **runtime_error** – apar ca rezultat a unor cauze nebănuite
    - *e.g.* probleme hardware sau de memorie
    - pot fi detectate doar în momentul execuţiei



# Excepții standard

- a se consulta exemplul din directorul *09/vectorExcep*

# Polimorfismul excepțiilor

- Numărul blocurilor ***try*** nu trebuie să crească exponențial cu dimensiunea programului

- Clasele de tip excepție trebuie organizate în ierarhii
  - Minimizarea numărului de *handlere*
  - Prinderea orientată-obiect a excepțiilor
    - Același *handler* este folosit pentru prinderea mai multor tipuri de excepții – *dynamic binding*

# Specificarea excepțiilor

- Anunţarea utilizatorului despre excepţiile pe care le poate arunca o funcţie
  - ➔ utilizatorul poate trata aceste excepţii
  - a se consulta exemplul din directorul *09/excepSpecif*

- Reutilizarea cuvântului rezervat **throw** urmat, în paranteză, de lista tuturor tipurilor de potenţiale excepţii pe care le aruncă funcţia respectivă
  - *void fc() noexcept;* - funcția nu aruncă nici o excepție
  - *void fc() noexcept(false);* - funcția s-ar putea să arunce o excepție
  - *void fc();* - funcţia poate arunca orice fel de excepţie
    - funcţiile *division1()* şi *run1()*
  - *void fc() throw(Exc1);*- funcţia aruncă excepţii de tipul *Exc1*
  - *void fc() throw(Exc1, Exc2);* - funcţia aruncă excepţii de tipul *Exc1* şi *Exc2*
    - funcţiile *division2()* şi *run2()*
  - *void fc() throw();* - funcţia nu aruncă nici o excepţie
    - funcţiile *division3()* and *run3()*
  - Dacă o funcţie aruncă o excepţie nespecificată ➔ se apelează automat metoda *unexpected()* (*terminate()* → *abort()*)
    - funcţiile *division4()* and *run4()*
- Dacă nu se cunosc excepţiile care pot apărea, nu se specifică excepţiile
- Specificarea excepţiilor se foloseşte mai ales pentru clasele ne-şablonate (*non-template classes*)

# Rearuncarea unei excepții

- utilizarea ***throw*** fără argument în cadrul unui *handler*
- Se re-aruncă o excepţie atunci când anumite resurse trebuie eliberate
- Dacă apare o excepţie
  - se prinde *orice* excepţie
  - Se eliberează resursele
  - Se permite contextului superior să trateze excepţia
- a se consulta exemplul din directorul *09/rethrowExc*

# Crearea şi distrugerea obiectelor

- Realizarea "*curăţeniei în cămară*"

- Managementul resurselor --> *exception-safe code*
  - Dacă un cod aruncă excepţii, atunci
    - obiectele complet create (constructorul a fost complet executat) sunt automat distruse
    - a se consulta exemplul din directorul *09/ExcConstrDestr*

  - Când apare o excepție
    - se recomandă să nu rămână resurse nealocate
    - operațiile pot fi executate complet sau parțial (tranzacții)

  - De aceea
    - fiecare resursă (memorie, fișier) --> crearea unei clase
    - orice pointer să fie încapsulat într-un obiect (fiind gestionat automat de către compilator); aceste obiecte trebuie să fie locale → se apelează automat destructorul când eecuția se mută în alt scop
    - utilizați RAII (Resource Acquisition Is Initialisation)

# *Exception-safe code*

- RAII (Resource Acquisition Is Initialization)
  - Resurse: memorie, fișiere, sockets, conexiuni la baze de date, etc.
  - Resursele sunt obținute înainte de utilizare și apoi eliberate după ce este terminat lucrul cu ele (cât mai curând posibil)
  - Ne-eliberarea resurselor poate cauza probleme (eg. *memory leaks*)

# Exception-safe code

- Soluții
  - Eliberarea memoriei în blocul ***catch***

```
try{
    int* v = new int{ 3 };
    throw std::exception("o exceptie");
    delete v;
}
catch (std::exception &exc){
    cout << exc.what();
}
```

```
class SmartPointer
{
private:
    int* data;

public:
    SmartPointer(int* p) : data(p){
        cout << "s-a alocat...";
    };
    ~SmartPointer(){
        if (this->data != NULL){
            delete this->data;
            this->data = NULL;
            cout << "s-a dealocat...";
        }
    };
    int& operator*(){
        return *this->data;
    }
};
```

  - *Smart pointers*

```
try
{
    SmartPointer a{ new int{ 3 } };
    throw std::exception{ "An exc has occured, but all the res were properly managed \n" };
    // no need to delete + no more leaks
}
catch (std::exception& e)
{
    cout << e.what();
}
```

# Excepții

- Se evită folosirea excepţiilor atunci când se lucrează:
  - cu evenimente asincron
  - cu condiţii pentru erori ne-progresive
  - cu scheme de control

- Avantajele excepţiilor
  - se pot evita situaţiile de eroare
  - codurile excepţiilor sunt independente de logica aplicaţiei
  - Se pot prinde mai multe tipuri de erori într-un singur loc (moștenire)
  - Funcțiile necesită mai puțini parametri (in&out) – mai ușor de înțeles

# Spaţii de nume

- Introduc un domeniu de vizibilitate care nu poate conţine duplicate
- Previn coliziunile de nume ale datelor

File *Library01.h*

```
namespace Library01{
      class Flower{
      };
}
```

File *Library02.h*

```
namespace Library02{
      class Flower{
      };
}
```

File *test.cpp*

```
#include "Library01.h"
#include "Library02.h"

int main(){
   //Flower f;   //error
            //'Flower' : undeclared identifier
   Library01::Flower f1;
   Library01::Flower f2;
   return 0;
}
```

# Spaţii de nume

- Declarare (definire, creare)

```
namespace [name]{
  //declarations
};
```

- Declararea este similară declarării unei clase, cu următoarele excepţii:
  - Nu există protecţie (toate elementele sunt publice)
  - Nu se pune “;” la sfârşit
  - Spaţiile de nume nu se pot instaţia (nu se pot crea variabile de tip spaţiu)
  - Definirea unui spaţiu de nume se poate face doar în scop global sau imbricat în alt spaţiu de nume
  - Se poate împărţi un spaţiu de nume în mai multe fişiere
    - a se consulta exemplul din directorul *09/nameSpace*

# Spaţii de nume

- Se pot declara spaţii de nume anonime
  - Compilatorul va genera automat un nume unic pentru un astfel de spaţiu

File *Library.h*

```
namespace {
      class Flower{
      };
     const char* NAME = “rose”;
}
```

# Spaţii de nume

- Accesul la elementele unui spaţiu de nume
  - Prin folosirea operatorului de rezoluţie

File *Library.h*

```
namespace Library{
      class Flower{
         private:
                  char* name;
         public:
            //…
            char* getName(){
                  return name;
            }
      };
     const char* NAME = “rose”;
}
```

File *test.cpp*

```
#include “Library.h”

int main(){
      Library::Flower f;
      char* s1 =Library::NAME;
      std::cout << “message” ;
      char* s2 = Library::Flower::getName();
      return 0;
}
```

# Spații de nume

- Este posibilă injectarea elementelor **friend** în declararea unui spaţiu de nume (în cadrul declarării unor clase)

File *Library.h*

```
namespace Library{
      class Flower{
        friend void fcFriend();
      }
}
```

- funcţia ***fcFriend()*** este membră a spaţiului de nume ***Library***.

# Spaţii de nume

- Directiva ***using***
  - Importă un întreg spaţiu de nume o dată
  - Permite accesul la elementele spaţiului
  - Expune toate elementele spaţiului
  - Se poate suprascrie un nume prin folosirea directivei **using**

File *Library01.h*

```
namespace Library01{
      class Flower{
      };
      const int MIN = 0;
}
```

File *Library02.h*

```
namespace Library02{
      class Flower{
      };
      const int MAX = 10;
}
```

File *test.cpp*

```
#include "Library01.h"

int main(){
  using namespace Library01;
  int m = MIN;
  Flower f1;
  using namespace Library02;
  int n = MAX;
  //Flower f2;
      //error: 'Flower' : ambiguous symbol
  return 0;
}
```

# Spații de nume

- Directiva ***using***
  - Injecteză un nume la un moment dat în cadrul scopului curent
  - Introduce un element al unui spaţiu de nume

File *Library01.h*

```
namespace Library01{
      class Flower{
      };
      const int MIN = 0;
}
```

File *Library02.h*

```
namespace Library02{
      class Flower{
      };
      const int MAX = 10;
}
```

File *test.cpp*

```
#include "Library01.h"

int main(){
  using Library01::Flower;
  Flower f1;  // Library01::Flower;
  using namespace Library02;
  Library02::Flower f2; //Library02::Flower
  return 0;
}
```

- O florarie (locatie, orar, nume) care gestioneaza mai multe flori (specie, cantitate, costUnitar) pe care le poate trimite clientilor (nume, adresa, telefon, iban) pe baza comenzilor (florile, adresa de livrare, data livrarii, telefon, cost) efectuate de acestia
  - Management de comenzi -> CRUD pt czi
  - Management de clienti -> CRUD -> clienti
  - Management de flori -> CRUD -> flori
  - Flow principal = plasare comanda

1. Implementare clasa Floare
2. Teste pt clasa Floare
3. Implementare Florarie
4. Teste pt Florarie
5. Implementare Comanda
6. Testare comanda
7. Impleemntare Client
8. Testare Client

# Cursul următor

- GUI