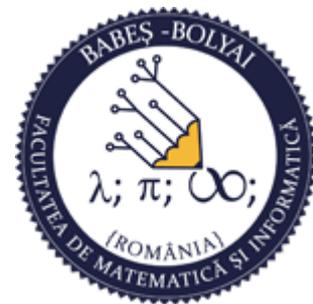




UNIVERSITATEA BABEŞ-BOLYAI  
Facultatea de Matematică și Informatică



# Programare orientată obiect

---

Curs 09

Laura Dioşan

# POO

---

## □ Clase

- Polimorfism
- Interfețe
- Colecții cu elemente generice
  
- Excepții
  
- Spații de nume

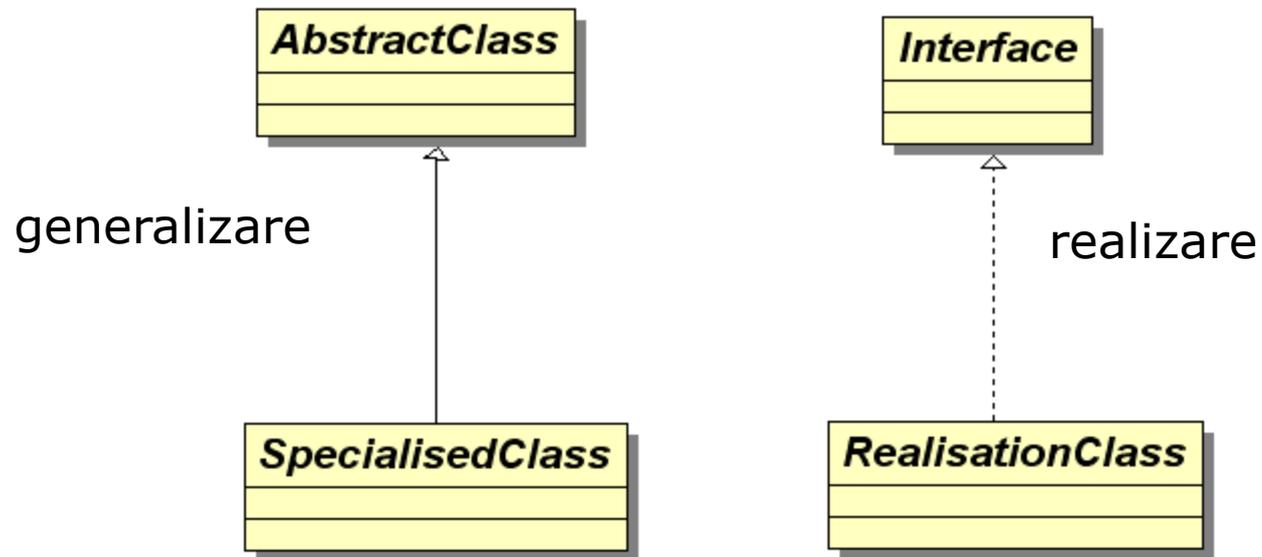
# Interfața

---

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

# Diagrama UML

---



# Containere

---

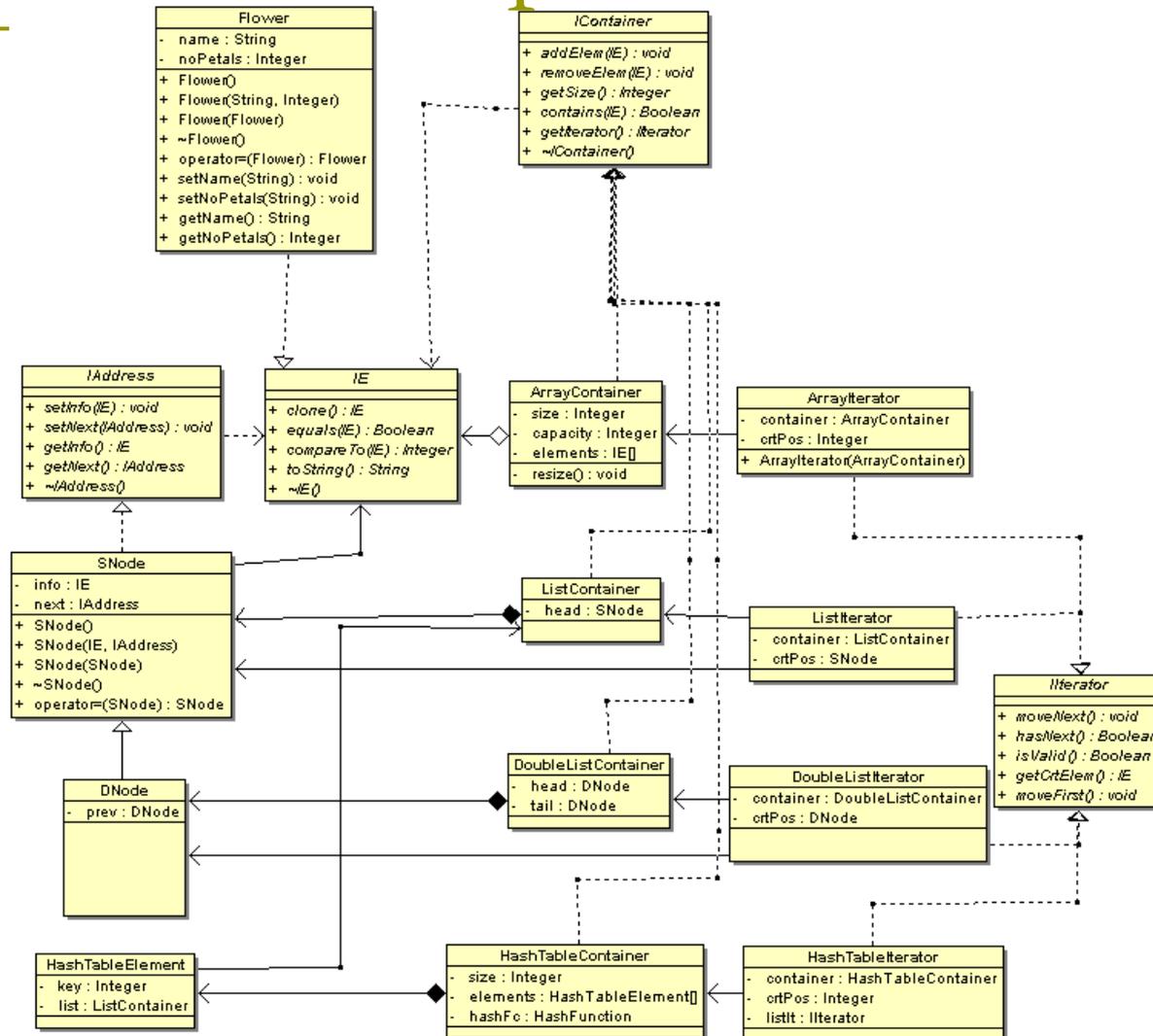
- Un container este un obiect care stochează o colecție de alte obiecte (elementele sale).
- Containerul:
  - gestionează spațiul pentru elemente
  - oferă funcții de acces la elemente, direct sau prin intermediul iteratorilor
- Unele containere au funcții comune și împart aceleași funcționalități
- Alegerea unui anumit tip de container depinde de:
  - funcționalitățile oferite de container
  - eficiența (complexitatea) acestor funcționalități

# Clasificarea containerelor

---

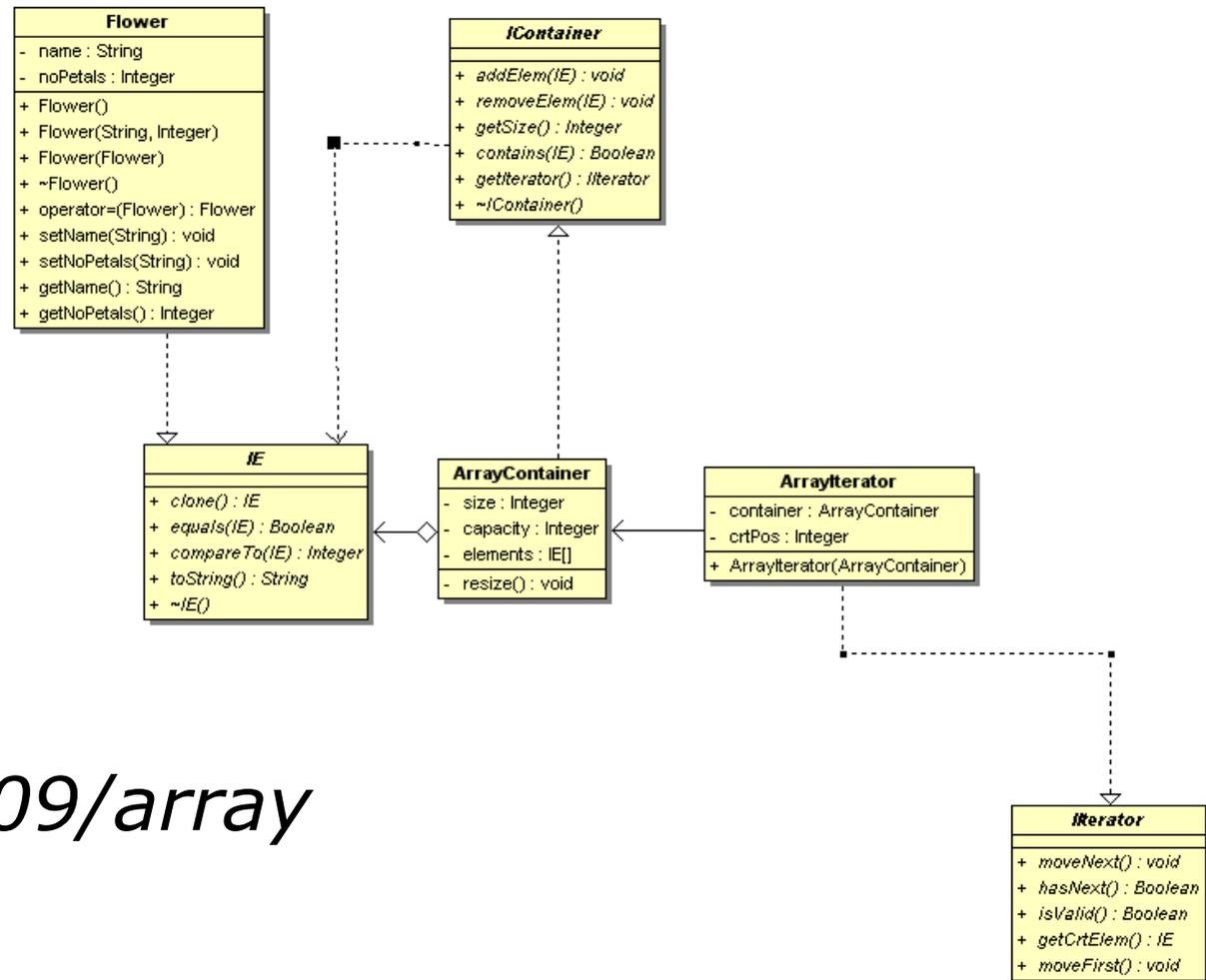
- Containerere secvențiale
  - Vector
  - Listă
  - *Deque* (coadă cu 2 capete)
- Containerere adaptate
  - Stivă
  - Coadă
  - Coadă cu priorități
- Containerere asociative
  - Multime
  - Mulțime multiplă
  - Dicționar
  - Dicționar multiplu
  - Tabelă de dispersie

# Structuri de date polimorfice



# Structuri de date polimorfice

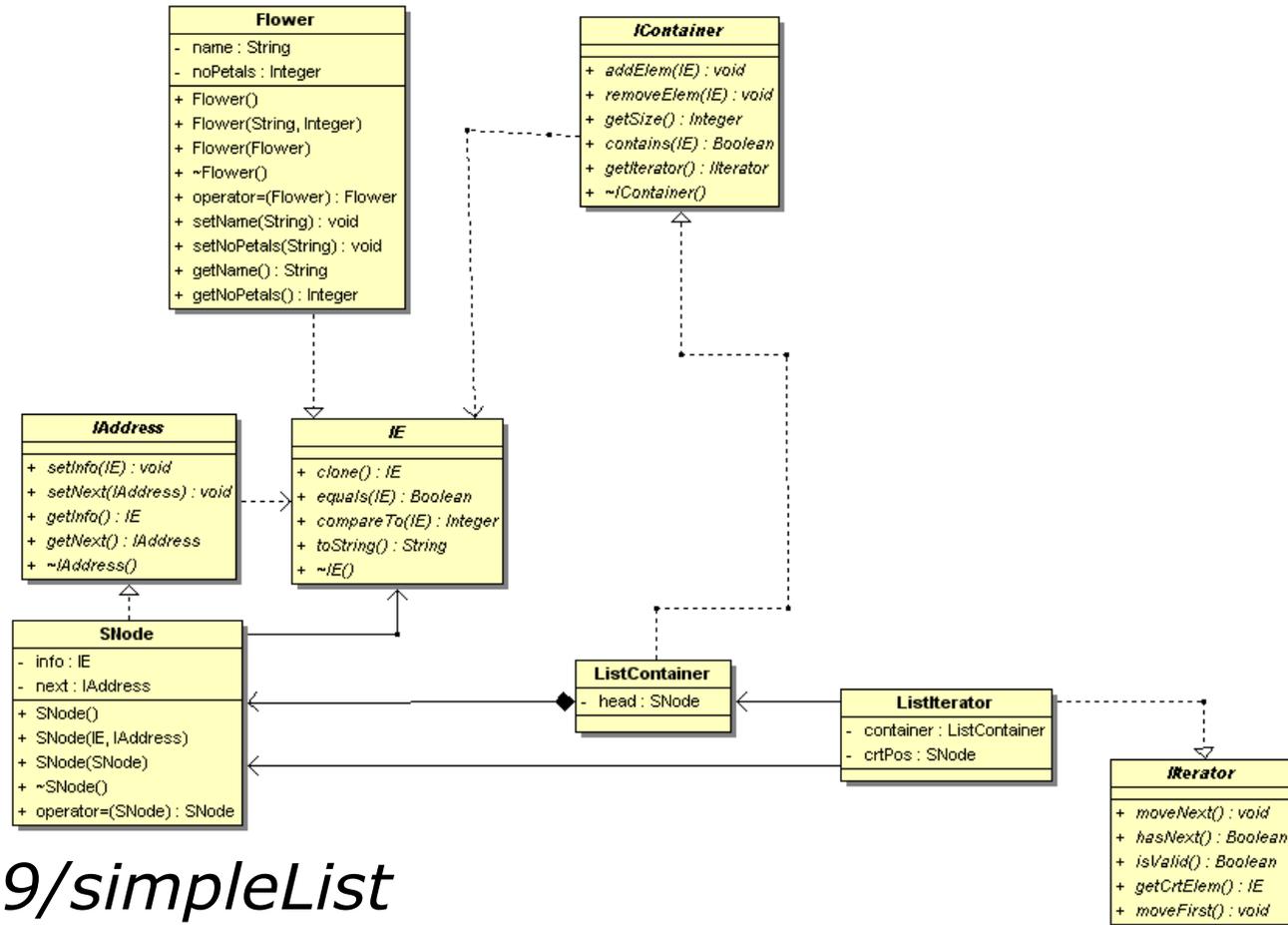
## Şiruri polimorfice



## exemplu în 09/array

# Structuri de date polimorfice

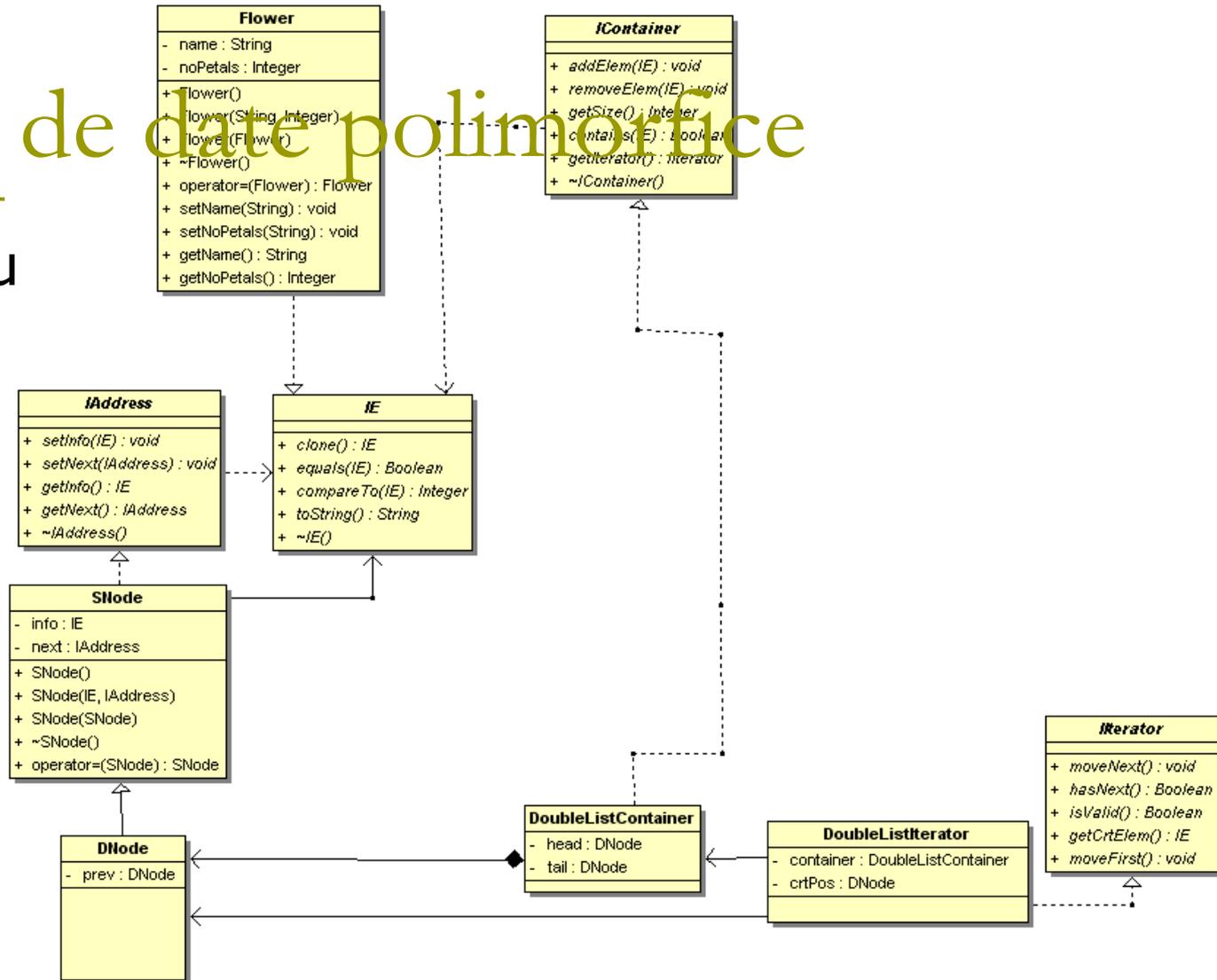
## ▣ Liste simplu înlanțuite polimorfice



## ▣ exemplu în *09/simpleList*

# Structuri de date polimorfice

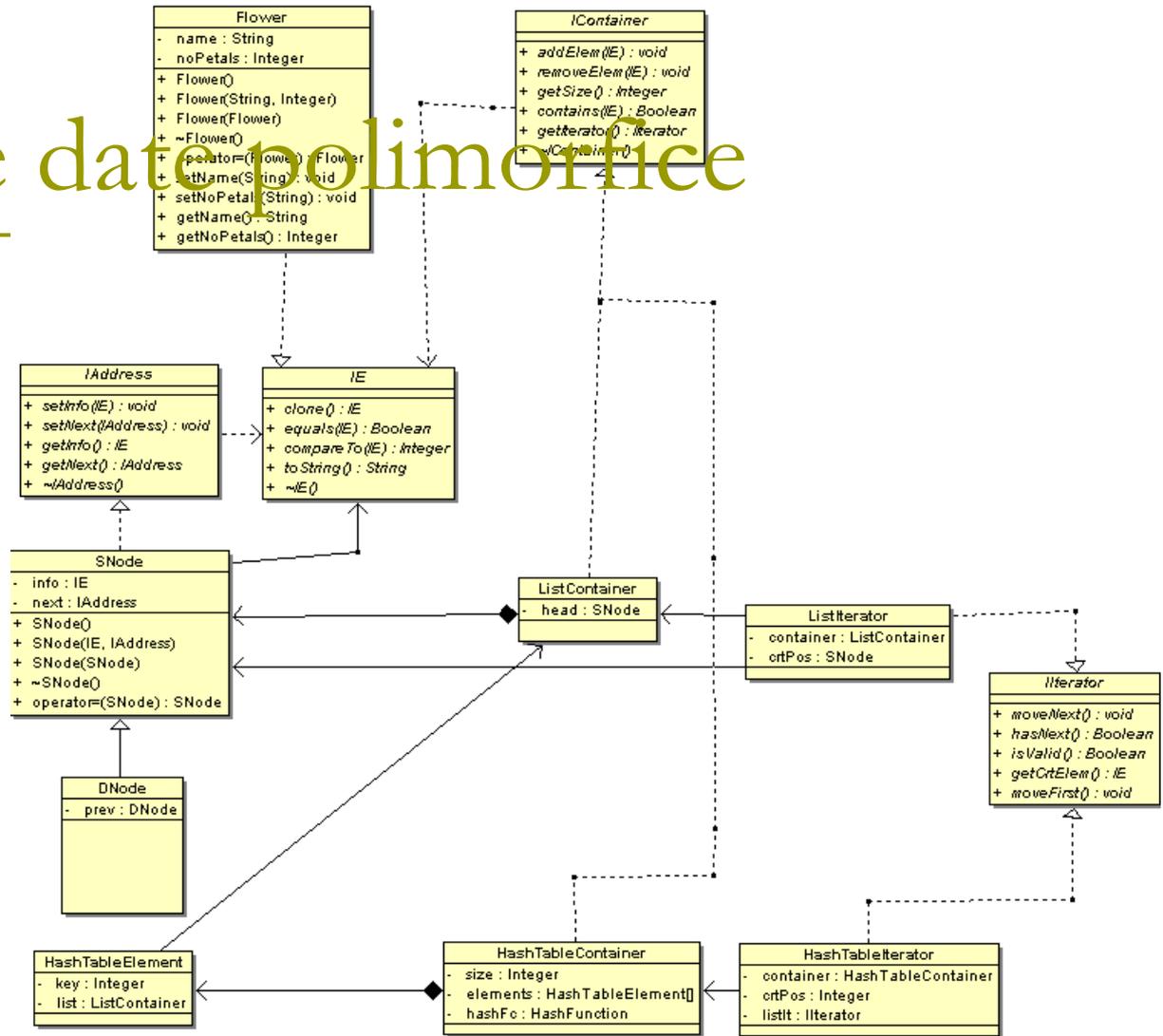
▣ Liste dublu  
înlănțuite  
polimorfice



▣ exemplu în *09/doubleList*

# Structuri de date polimorfice

▣ Tabele de dispersie polimorfice



▣ exemplu în 09/hashTable

# Structuri de date polimorfice

---

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

# Temă

---

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

# POO

---

- Excepții
- Spații de nume

# Prinderea erorilor

---

## □ Abordarea tradițională

### ■ Cum?

- funcții speciale (pentru programe mici)
  - *assert()*
  - *assure()*
  - *require()*
- Stegulețe de eroare (ca variabile globale)
- Coduri de retur

### ■ Dezavantaje

- codul nu este clar
- anumite erori pot fi ignorate – a se vedea funcția *printf*
- codurile de erorare și cele logice pot coincide
- Dificultăți de înlănțuire

# Prinderea erorilor

---

- Abordarea modernă
  - Prinderea excepțiilor
  - Excepția
    - O situație anormală care apare în timpul execuției
      - accesarea unui indice în afara limitelor
      - folosirea unui pointer nul
      - împărțire la 0
    - Oferă posibilitatea tratării situațiilor excepționale prin transferul controlului unor funcții speciale numite *handlers*

# Aruncarea unei excepții

---

- Când apare o situație specială
  - Se aruncă o excepție = se pot trimite informații despre respectiva eroare într-un context mai larg prin crearea unui obiect care să conțină acele informații și aruncarea lui în contextul curent
  - A se consulta exemplul din directorul *09/simpleException*
- Ce se întâmplă când o funcție apelată aruncă o excepție?
  - Se creează o copie a excepției și se întoarce această copie (chiar dacă funcția curentă întoarce un alt tip de dată)

# Elementele componente ale unei excepții

---

- **try**
  - marchează codul susceptibil de a arunca excepții
- **catch**
  - *Handler*-ul excepției (locul în care excepția ajunge și este tratată)
  - are un parametru → tipul excepției
  - Câte un *handler* pentru fiecare tip de excepție
    - mai multe clauze **catch**
    - mecanismul de prindere a excepțiilor caută primul *handler* al cărui parametru se potrivește cu excepția 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 excepția aruncată, atunci programul se termină (anormal: este apelată funcția *terminate()* sau *abort()*)
- **throw**
  - aruncă o excepție (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 *handler*-ului excepției
  - În cazul mai multor *handler*-e, ele pot avea parametri de tipuri diferite
  
- Tipuri de excepții
  - Tipuri predefinite (int, char, double, char\*)
    - A se consulta exemplul din directorul 09/*simpleException*
  - tipuri definite de utilizator (obiecte)
    - obiectul excepție este transmis ca:
      - referință
      - pointer
    - pentru a evita apelul constructorului de copiere al clasei excepție

# Excepții ca obiecte

---

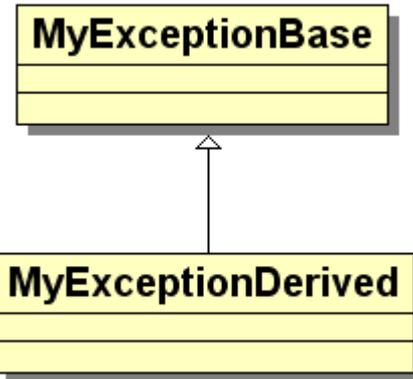
- ❑ Obiectul excepție
  - Transmiterea informațiilor despre situația apărută
- ❑ Obiectul excepție este distrus abia apoi după ce excepția a fost tratată
  - ultimul bloc catch s-a terminat
- ❑ Transmiterea excepțiilor
  - Aruncarea unei excepții prin valoare
  - Prinderea unei excepții prin referință
    - ❑ 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 excepții, prinderea unei singure excepții
  - funcția *oneException()*
- Aruncarea mai multor excepții, prinderea primei excepții aruncate
  - funcția *moreThrownExceptions*
- Aruncarea unei excepții, mai mulți handleri catch, prinderea primului tip potrivit
  - funcția *morePossibleExceptions*
- Prinderea oricărei excepții
  - clauza *catch(...)*
    - trebuie să fie plasată la sfârșit (altfel, va prinde orice fel de excepție)
  - funcția *anyException*
- Nici una dintre clauzele **catch** nu se potrivește cu excepția aruncată
  - funcția *exceptionThatDoesntMatchAnyCatchClause*

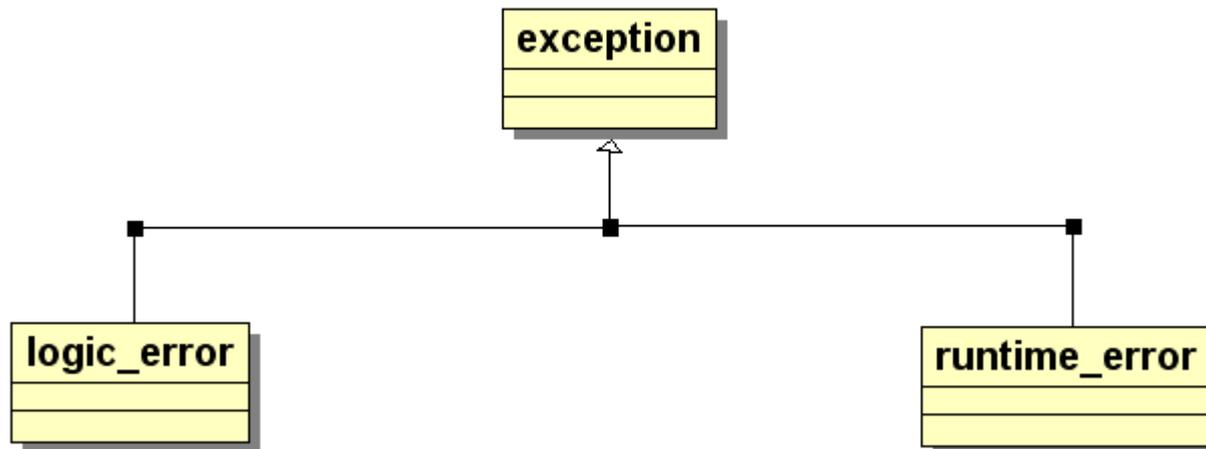
# Polimorfismul excepțiilor



- Presupune că avem relația:
- Dacă un cod este susceptibil de a arunca o excepție de bază, atunci:
  - Dacă se prind doar excepții de tip părinte (bază), compilatorul va face potrivirea cu clauza **catch** respectivă
  - Dacă se prind ambele tipuri de excepții (bază și derivată), compilatorul va face potrivirea cu clauza catch de tip bază
- Dacă un cod este susceptibil de a arunca o excepție derivată, atunci:
  - Dacă se prind doar excepții de tip derivat, compilatorul va face potrivirea cu clauza **catch** respectivă
  - Dacă se prind ambele tipuri de excepții (bază și derivată), compilatorul va face potrivirea cu prima clauza **catch** care se potrivește (bază sau derivată)
  - Dacă se prind doar excepții de bază, compilatorul va face potrivirea cu clauza catch de tip bază (excepția derivată este convertită la excepție 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 excepții 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 inspecție
  - **runtime\_error** – apar ca rezultat a unor cauze neașteptate
    - e.g. probleme hardware sau de memorie
    - pot fi detectate doar în momentul execuției



# 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 *handlers*
  - Prinderea orientată-obiect a excepțiilor
    - Același *handler* este folosit pentru prinderea mai multor tipuri de excepții – *dynamic binding*

# Specificarea excepțiilor

---

- Anunțarea utilizatorului despre excepțiile pe care le poate arunca o funcție
  - → utilizatorul poate trata aceste excepții
  - a se consulta exemplul din directorul *09/excepSpecif*
- Reutilizarea cuvântului rezervat **throw** urmat, în paranteză, de lista tuturor tipurilor de potențiale excepții pe care le aruncă funcția respectivă
  - *void fc() noexcept;* - funcția nu aruncă nici o excepție
  - *void fc() noexcept(false);* - funcția s-ar putea să arunce o excepție
  - *void fc();* - funcția poate arunca orice fel de excepție
    - funcțiile *division1()* și *run1()*
  - *void fc() throw(Exc1);* - funcția aruncă excepții de tipul *Exc1*
  - *void fc() throw(Exc1, Exc2);* - funcția aruncă excepții de tipul *Exc1* și *Exc2*
    - funcțiile *division2()* și *run2()*
  - *void fc() throw();* - funcția nu aruncă nici o excepție
    - funcțiile *division3()* and *run3()*
  - Dacă o funcție aruncă o excepție nespecificată → se apelează automat metoda *unexpected()* (*terminate()* → *abort()*)
    - funcțiile *division4()* and *run4()*
- Dacă nu se cunosc excepțiile care pot apărea, nu se specifică excepțiile
- Specificarea excepțiilor se folosește mai ales pentru clasele ne-șablonate (*non-template classes*)

# Rearuncarea unei excepții

---

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

# Crearea și distrugerea obiectelor

---

- Realizarea “*curățeniei în cămară*”
  
- Managementul resurselor --> *exception-safe code*
  - Dacă un cod aruncă excepții, 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 execuția se mută în alt scop
    - utilizați RAI (Resource Acquisition Is Initialisation)

# *Exception-safe code*

---

## □ RAII

- 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();
}
```

### ■ *Smart pointers*

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

# Excepții

---

- Se evită folosirea excepțiilor atunci când se lucrează:
  - cu evenimente asincron
  - cu condiții pentru erori ne-progresive
  - cu scheme de control
- Avantajele excepțiilor
  - se pot evita situațiile de eroare
  - codurile excepțiilor sunt independente de logica aplicației
  - Se pot prinde mai multe tipuri de erori într-un singur loc (moștenire)
  - Funcțiile necesită mai puțini parametri (in&out) – mai ușor de înțeles

# Spații de nume

---

- ❑ Introduc un domeniu de vizibilitate care nu poate conține 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;  
}
```

# Spații de nume

---

- ❑ Declaraire (definire, creare)

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

- ❑ Declarația este similară declarării unei clase, cu următoarele excepții:
  - Nu există protecție (toate elementele sunt publice)
  - Nu se pune ";" la sfârșit
  - Spațiile de nume nu se pot instanția (nu se pot crea variabile de tip spațiu)
  - Definirea unui spațiu de nume se poate face doar în scop global sau imbricat în alt spațiu de nume
  - Se poate împărți un spațiu de nume în mai multe fișiere
    - ❑ a se consulta exemplul din directorul *09/nameSpace*

# Spații de nume

---

- Se pot declara spații de nume anonime
  - Compilatorul va genera automat un nume unic pentru un astfel de spațiu

File *Library.h*

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

# Spații de nume

---

- Accesul la elementele unui spațiu de nume
  - Prin folosirea operatorului de rezoluție

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 spațiu de nume (în cadrul declarării unor clase)

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

- funcția ***fcFriend()*** este membră a spațiului de nume ***Library***.

# Spații de nume

## □ Directiva **using**

- Importă un întreg spațiu de nume o dată
- Permite accesul la elementele spațiului
- Expune toate elementele spațiului
- 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***

- Injectează un nume la un moment dat în cadrul scopului curent
- Introduce un element al unui spațiu 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;  
}
```

# Cursul următor

---

- Standard Template Library (STL) – librăria standard
  - Containers
  - Algorithms