PROGRAMARE ORIENTATĂ OBIECT (C++)

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

cristina.dascalescu@prof.utm.ro

Universitatea Titu Maiorescu

2018 - 2019

Conținutul thematic – Curs 2

- Tipuri abstracte definite de către programator
- Sintaxa unei clase
- Obiecte. Ciclul de viață al unui obiect
- Pointer this
- Metode setter/getter

Concepte de bază POO

- Încapsularea
- Abstractizarea
- Ierarhizarea
- Polimorfismul

Abstractizarea

- Procesul prin care se identifică datele și operațiile relevante pentru un concept din lumea reală.

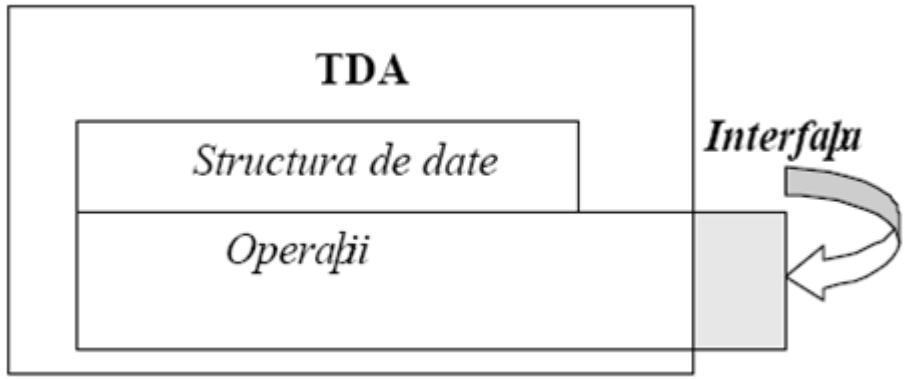
Tipul Persoana - pentru aplicația Recensământ	Tipul Persoana – pentru aplicația Calcul Intreținere
▪ Nume ▪ prenume ▪ varsta ▪ localitate	▪ nume ▪ prenume ▪ suprafataLocuita ▪ NrPersoaneIntretinere
▪ numara ▪ afisare ▪ statictica	▪ calculIntretinere ▪ deduceri ▪ Afisare

Abstractizarea

- **Date și operații (funcții) sunt incapsulate**

```
Date
nume;
facultatea;
anStudii;

Funcții
inițializare
calculmedie
afisare
```

Tip Abstract de date (TAD)



Implementarea unui tip nou de date

- **Clasa** definește un nou tip de dată determinat prin mecanismul de abstractizare.

- Terminologie:
  - **setul de date** -> **date membre ale unui clase**
  - **setul de operații** -> **metode membre ale unei clase**

Sintaxa unei clase

```
class NumeClasa
{
      private:
            date și metode membre
      public:
             date și metode membre
      protected:
            date și metode membre
}
```

Modificatorii de acces

➢ **Modificatorul public**

- Datele și metodele membre pot fi accesate și din afara clasei
- Este denumită și interfața clasei
- Încapsulează, de regulă, metodele membre ale clasei

➢ **Modificatorul private**

- Datele și metodele membre pot fi accesate doar din interiorul clasei
- Încapsulează, de regulă, stuctura de date membre
- Pentru a accesa datele membre private se pot utiliza metode membre publice

➢ **Modificatorul protected**

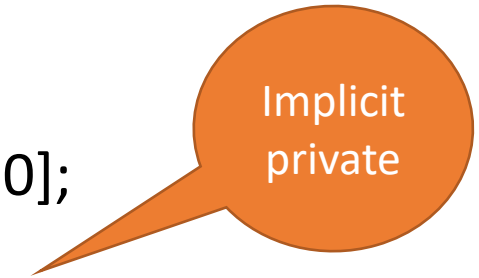
- Datele și metodele membre pot fi accesate din interiorul clasei, respectiv de către clasele derivate (clasele care se află în aceeași ierarhie)

Observații

- **Dacă denumirea unei clase este urmată de blocul {…} , atunci clasa se consideră definită, altfel este doar declarată!!!**
- **Dacă nu se definește niciun modificator de acces, atunci clasa este implicit privată!!!**
- **Odată definită o clasă, compilatorul recunoște ca tip de dată numele clasei**

Exemple

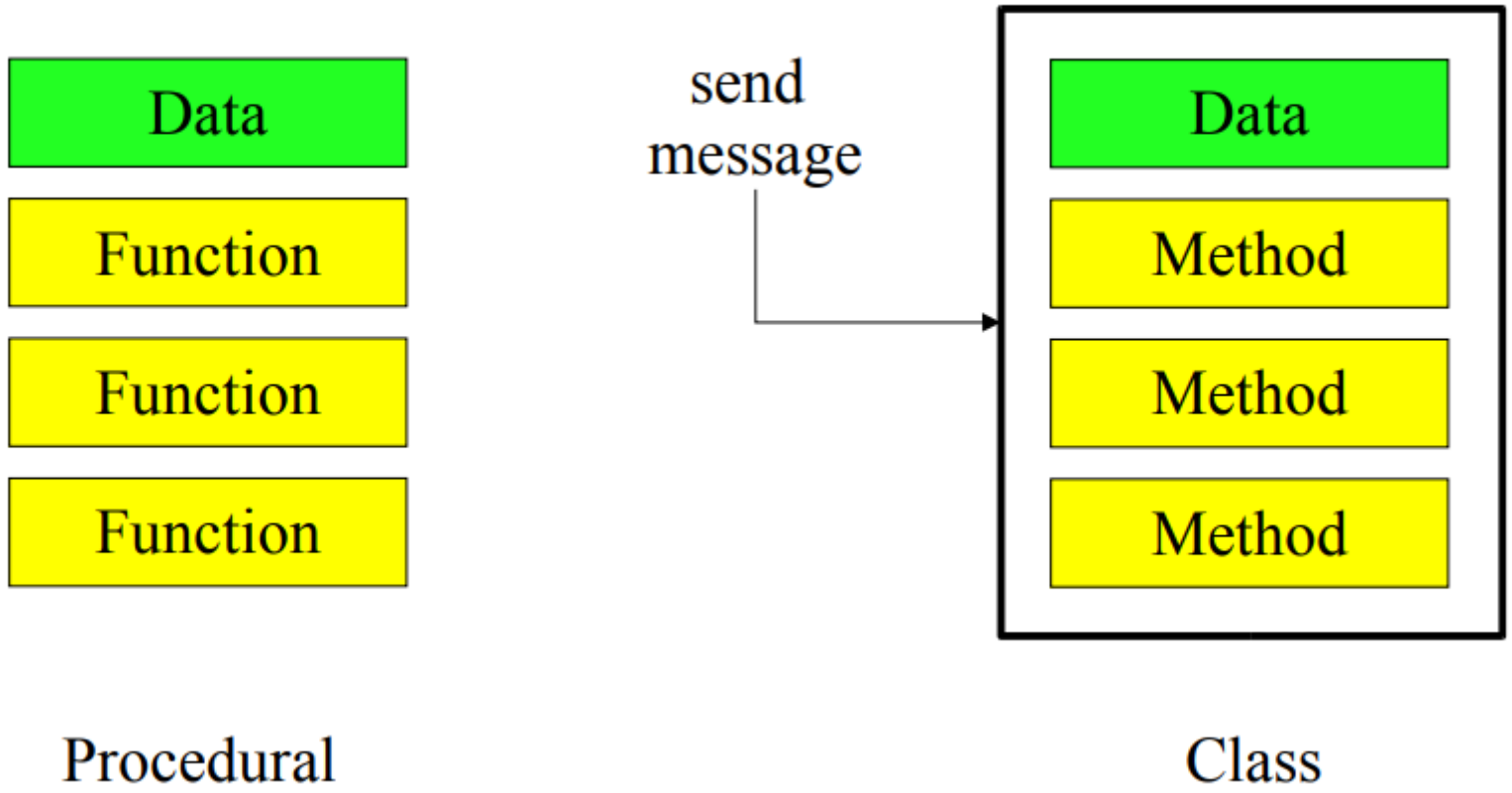
```
class ContBancar
{
   char titular[50];
   char numarCont[30];
   double sold;

public:
   void depunere(double suma)
   {
      sold+=suma;
   }
```




```
void retragere(double suma)
   {
      sold-=suma;
   }
   void afisare()
   {
      cout<<titular<<" "<<numarCont
      <<"    "<<sold<<endl;
   }
};
```

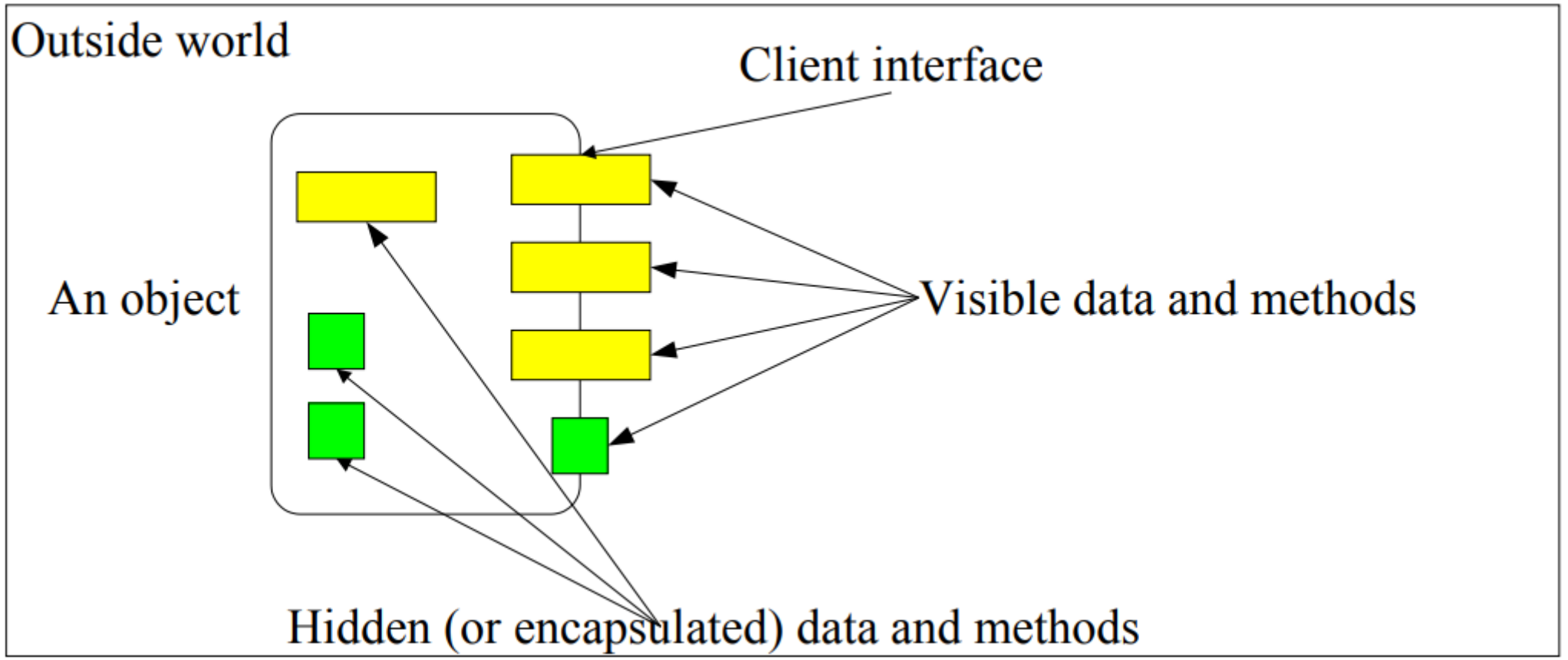
Încapsularea

- Mecanismul prin care datele și operațiile sunt înglobate sub forma unui tot unitar (obiect)



Încapsularea

- Datele pot fi accesate din afara entității (obiectului) numai prin intermediul operațiilor (funcțiilor/metodelor) publice!!!



Definirea obiectelor

➢ **Instanța unei clase se numește obiect**

- Un obiect este determinat în mod unic prin numele său
- **Starea** obiectului este definită de către valorile datelor membre

Exemplu:

```
ContBancar cont1; //valoare reziduală
ContBancar cont2;
cont2.init(“Popescu”, “RO123ING”, 67008);
```

- **Comportamentul** unui obiect este definit strict de metodele sale publice

Exemplu: pentru un obiect de tip ContBancar se pot apela funcționalități precum: inițializare, afișare, retragere, depunere

Declararea unui obiect

- Se consideră definită o clasă **`class C {…}`**

```
C ob; // obiect de tip C

C Tablou[20]; // tablou de obiecte de tip C

C* ptrC; // pointer la un obiect de tip C

C &tC = ob; // referinta la un obiect de tip C
```

Accesarea membrilor publici

- Se consideră definită o clasă **`class C {…}`**
- Se consideră declarat obiectul **`C`** `ob;`

- Accesarea unui membru public se realizează prin operatorul **`.`**

`ob.membruPublic`

- Se consideră declarat obiectul **`C`** `*pob = new` **`C;`**

- Accesarea unui membru public se realizează prin operatorul **`->`**

`pob->membruPublic`

Metode setter și getter

- **Metodele setter sunt metode publice care** au rolul de a modifica valoarea unei date membre a unui obiect (pentru care se apeleaza metoda setter)

- Uzual, o metoda de tip setter are signatura:

```
void setDataMembra(tipDM valoare)
{
    dataMembra=valoare;
}
```

Exemple:

```
void setSuma(double val){suma=val;}
void setNume(char[] sirNume){strcpy(nume, sirNume);}
```

Metode setter și getter

- **Metodele getter sunt metode publice care** au rolul de returna valoarea currentă a unei date membre pentru un obiect(pentru care se apeleaza metoda setter)

- Uzual, o metoda de tip getter are signatura:

```
tipDataMembra getDataMembra()
{
    return dataMembra;
}
```

Exemple:

```
double getSuma(){return sumaCont}
char* getNume(){return nume};
```

Pointer this

Observații

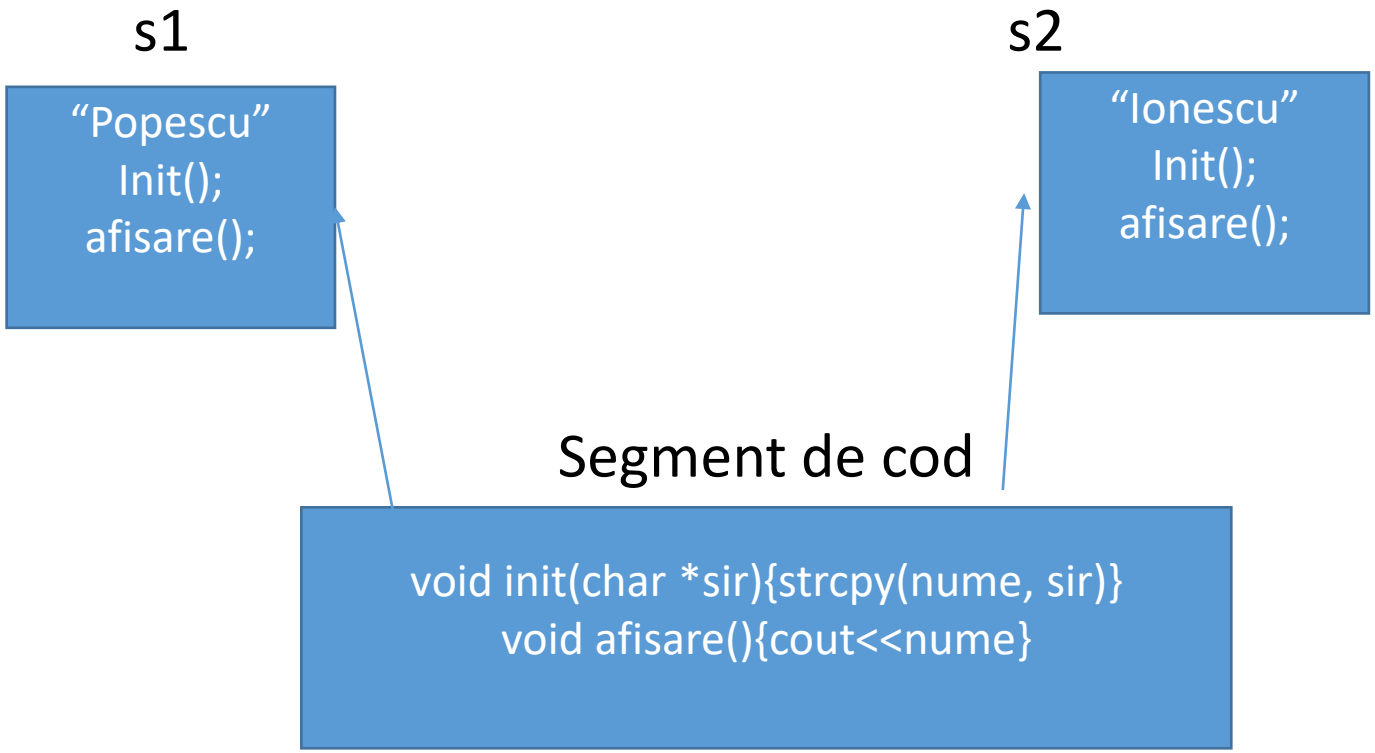
- Metodele unei clase invocă datele membre din clasă fără ca acestea să fie transmise ca parametrii

- Fiecare obiect al unei clase are propriul său set de date

```
class Student {char nume[50];…}
 Student s1;
 s1.init(“Popescu”);
 Student s2;
 s2.init(“Ionescu”);
```

Pointer this

- Toate obiectele accesează același set de funcții, salvat pe segmentul de cod



Pointer this

- Accesul unei metode la datele corespunzatoare unui obiect se realizează prin adresa obiectului respectiv, reținută de către pointerul **this**.

- **Pointerul this** = adresa obiectului curent

- Orice metodă a clasei primește implicit ca argument al clasei pointerul this

```
void afisare(){cout<<this->nume;}
s1.afisare();
s2.afisare();
```

this = adresa obiectului s1

this = adresa obiectului s2

Pointer this

➢Utilitate

- Accesarea corectă a datelor membre
- Eliminarea ambiguității provocată în cazul în care un argument al unei metode are același denumire cu cea a clasei

```
void init(char *nume)
{
 strcpy(this->nume, nume);
}
```