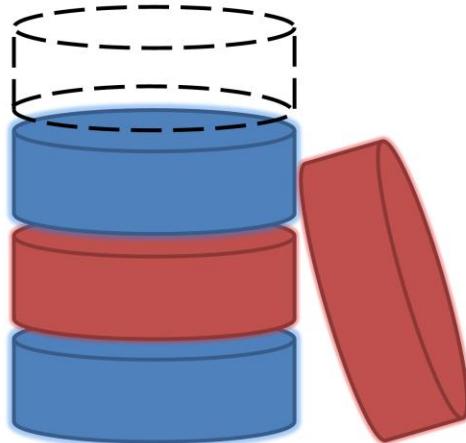


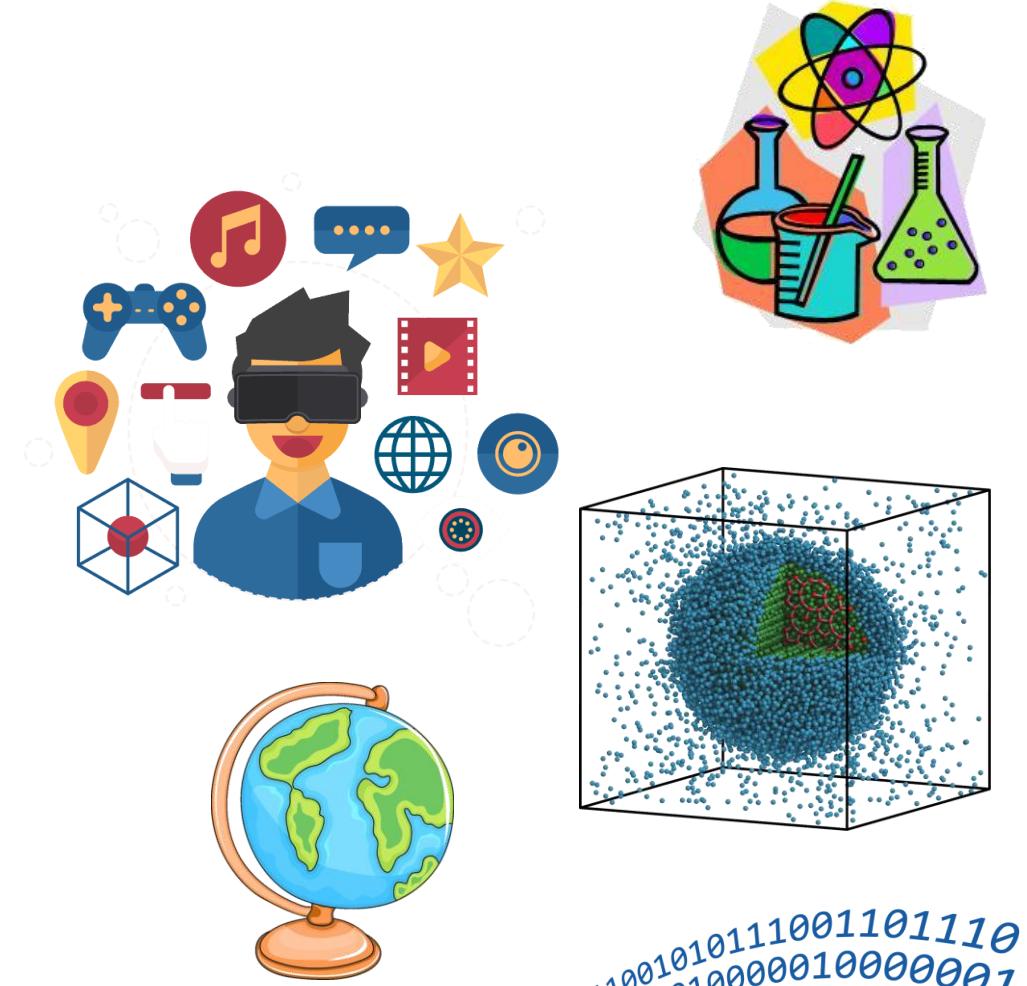
# Baze de Date Orientate-Obiect



11

# Nevoie unui mediu de stocare

- Informații multimedia (imagini, filme)
  - Date spațiale (GIS)
  - Date biologice
  - Proiecte tehnologice (date CAD)
  - Lumi virtuale
  - Jocuri
  - Fluxuri de date
  - Tipuri de date definite de utilizator



# Manipularea de categorii noi de date

- Un canal de televiziune necesită stocarea și accesarea rapidă a unor secvențe video, interviuri radio, documente multimedia, informații geografice etc.
- Un producător de filme dorește să stocheze filme întregi și secvențe, date despre actori și cinematografe etc
- Un laborator de cercetări biologice necesită stocarea de date complexe despre molecule, cromozomi și consultarea sau completarea anumitori părți din aceste date.
- Nevoi comerciale complexe.

# Nevoile pentru un SGBD

- Creșterea exponențială a cantității datelor accesate de aplicații în paralel cu reducerea timpului necesar de dezvoltarea a acestor aplicații
  - programare orientată-obiect
  - caracteristici SGBD: optimizare interogări, controlul accesului concurrent, recuperarea datelor, indexare etc.
- Subiect de cercetare anii '90: pot fi contopite cele două direcții?

# Dezavantajele bazelor de date relaționale

- Lipsesc atributele de tip colecție
- Lipsește moștenirea
- Lipsesc obiectele complexe, în afară de BLOB (*binary large object*)
- Diferență conceptuală între limbajul de acces la date (declarative: SQL) și limbajul de programare gazdă (procedural: C++, C#, Java etc).

⇒ Ce alte soluții pot fi implementate?

# Baze de date obiectuale

- Baza de date de obiecte – *depozit* de obiecte persistente:
  - Sisteme de baze de date orientate-obiect: alternativă la sistemele relaționale
  - Sisteme de baze de date relațional-obiectuale: extensie a sistemelor relaționale
- Sisteme de gestiune de baze de date OO: *ObjectStore, GemStone, Wakanda, Realm, ObjectDB, PostgreSQL(?)*

# Modelul de date obiectual

- *Modelul obiectual* reprezintă fundamentul bazelor de date orientate obiect, aşa cum *modelul relațional* reprezintă fundamentul pentru bazele de date relaționale.
- Baza de date conține o colecție de obiecte
- Un obiect are un ID unic (OID) iar colecția obiectelor ce au proprietăți similare se numește clasă.

# Proprietățile obiectelor

- **Atribute:** au tipuri atomice sau structurate (*set, bag, list, array*)
- **Relații:** referință către un obiect sau multime de obiecte
- **Metode:** funcții ce pot fi aplicate obiectelor unei clase

# Tipuri abstracte de date

- Funcționalitate cheie: crearea de noi tipuri de date arbitrară de către utilizatori.
- Un tip nou de date este însotit de metode de accesare corespunzătoare (tip + metode = tip abstract de date).
- De asemenea, un SGBD are tipuri predefinite.

# Încapsularea

- Încapsulare = structuri de date + operații
- Încapsularea permite ascunderea detaliilor interne unui tip abstract de date
- SGBD-ul nu trebuie să cunoască modul de stocare a datelor sau felul în care funcționează metodele unui tip abstract de date. Este necesară doar cunoașterea metodelor disponibile și a detaliilor de apelare a acestora (tipuri de intrare/iesire)

# Moștenire

O valoare are un tip  
Un obiect aparține unei clase

## ■ Ierarhie de tipuri

- Este permisă definirea de tipuri noi de date pe baza tipurilor existente
- Un *subtip* moștenește toate proprietățile *supertipului*

## ■ Ierarhie de clase

- O subclasă  $C'$  a unei clase  $C$  este o colecție de obiecte în care fiecare obiect al clasei  $C'$  este în același timp și obiect al clasei  $C$ .
- Un obiect al clasei  $C'$  moștenește toate proprietățile din  $C$

# Baze de date orientate obiect

- Scopul unui SGBD OO este integrarea “naturală” într-un limbaj de POO ca C++, C#, Java etc.
- ODL = *Object Description Language*, corespunzător DDL din SQL.
- OML = *Object Manipulation Language*, ce înlocuiește SQL DML într-un context orientat-obiect.

# ODL în SGBD-urile Orientate-Obiect

- ODL este utilizat pentru definirea de clase *persistente*, ale căror obiecte pot fi stocate permanent în baza de date.
  - Definirea claselor cu ODL reprezintă o extensie a limbajului orientat-obiect gazdă.

# ODL

- Declarația unei clase include:

- Numele clasei
- Declarație optională de chei
- Declarația unui *extent* = numele mulțimii tuturor obiectelor ce aparțin clasei.
- Declarații de elemente. Un *element* poate fi un atribut, o relație sau o metodă.

```
class <name> {  
    <list of element declarations,  
     separated by semicolons>}
```

# Declarații de atribute și metode

- Atributele sunt (de obicei) declarate prin nume și tip, unde tipul nu reprezintă o clasă.

```
attribute <type> <name>;
```

- Informațiile din declarația unei metode conțin:

- Tipul returnat (dacă este cazul)
- Numele metodei
- Categoria (*in*, *out*, *inout*) și tipul argumentelor (fără nume)
- Excepțiile ce pot fi aruncate de către metodă

```
real grade_avg(in string) raises (noGrades);
```

# Declarații de relații

- Relațiile conectează un obiect al unei clase cu unul sau mai multe obiecte ale unei alte clase.
- Relațiile sunt memorate ca perechi de pointeri inversați (A îl referă pe B și B îl referă pe A)
- Relațiile sunt întreținute automat de către sistem (dacă A este eliminat, pointerul lui B va fi automat inițializat cu NULL)
- Categorii de relații: *one-to-one, one-to-many, many-to-many*

**relationship <type> <name> **inverse** <relationship>;**

# Exemplu

```
class Movie{  
    attribute date start;  
    attribute date end;  
    attribute string movieName;  
    relationship Set<Cinema> shownAt inverse  
        Cinema::nowShowing;  
}  
  
class Cinema {  
    attribute string cinemaName;  
    attribute string address;  
    attribute integer ticketPrice;  
    relationship Set <Movie> nowShowing inverse  
        Movie::shownAt  
    float numshowing() raises(errorCountingMovies);  
}
```

*tipul relației*

*operatorul :: conectează un nume unui context*

*Movie::shownAt*

# Tipuri de relații

- Tipul unei relații poate să fie:
  - O clasă, ca *Movie*. În acest caz un obiect cu acest tip de relație poate fi conectat cu un singur obiect *Movie*.
  - **Set<Movie>**: obiectul este conectat cu o mulțime de obiecte *Movie*.
  - **Bag<Movie>, List<Movie>, Array<Movie>**: obiectul este conectat cu o mulțime cu duplicări, listă sau tablou de obiecte *Movie*.

# Multiplicitatea relațiilor

- Toate relațiile ODL sunt binare.
- Relațiile *many-to-many* au *Collection* ca tip al relației și sunt inversate.
- Relațiile *one-to-many* au *Collection<...>* în declarația relației în obiectul “*one*” și doar o clasă în declarația relației obiectului “*many*.”
- Relațiile *one-to-one* au tip de relație clasă în ambele direcții.

# Exemplu

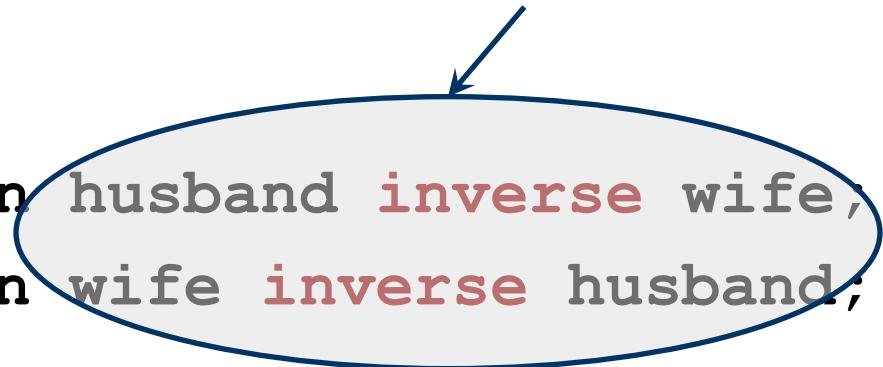
```
class Drinker { ...  
    relationship Set<Beer> likes inverse  
        Beer::fans;  
  
    relationship Beer favBeer inverse  
        Beer::superfans;  
}  
  
many-to-many  
  
class Beer { ...  
    relationship Set<Drinker> fans inverse  
        Drinker::likes;  
  
    relationship Set<Drinker> superfans inverse  
        Drinker::favBeer;  
}  
  
one-to-many
```

The diagram illustrates the relationships between the `Drinker` and `Beer` classes. It features two `Drinker` objects at the top and two `Beer` objects at the bottom. Four arrows connect them: a blue arrow from the first `Drinker` to the first `Beer` labeled "many-to-many"; a blue arrow from the second `Drinker` to the first `Beer` labeled "one-to-many"; a red arrow from the first `Beer` to the second `Drinker` labeled "inverse"; and a red arrow from the second `Beer` to the second `Drinker` labeled "inverse". The `Drinker` class definition includes a `Set<Beer>` relationship named `likes` with an inverse `Beer::fans`. The `Beer` class definition includes two `Set<Drinker>` relationships: `fans` with an inverse `Drinker::likes`, and `superfans` with an inverse `Drinker::favBeer`.

# Exemplu

```
class Person{  
    attribute ...;  
  
    relationship Person husband inverse wife;  
    relationship Person wife inverse husband;  
  
    relationship Set<Person> buddies  
};
```

*husband și wife* sunt relații *one-to-one* și una reprezintă inversa celeilalte



**inverse** buddies;

*buddies* este *many-to-many* și este propria sa inversă



# Conecțarea claselor

- Dacă se dorește conețarea claselor  $X$ ,  $Y$  și  $Z$  printr-o relație  $R$ :
  - Se crează o clasă  $C$ , a căror obiecte reprezintă un triplet de obiecte  $(x, y, z)$  din clasele  $X$ ,  $Y$  și  $Z$ .
  - Se vor crea trei relații *many-to-one* de la  $(x, y, z)$  la fiecare dintre  $x$ ,  $y$  și  $z$ .

# Exemplu: Conectarea claselor

- Fie clasele *BookStore* și *Book*. Dorim să memorăm prețul cu care fiecare librărie (obiect al *BookStore*) vinde o carte.
  - Acest lucru nu se poate modela cu o relație *many-to-many* între *BookStore* și *Book* deoarece nu se pot defini atribută conectate de o relație
- Soluția 1: se crează clasa *Price* și o clasă de conectare *BBP* ce reprezintă relația dintre librărie, carte și preț.
- Soluția 2: deoarece obiectele *Price* conțin doar un simplu număr este poate mai util să:
  - Adăugăm la clasa *BBP* un atribut *price*.
  - Folosim relații *many-to-one* între un obiect *BBP* și obiecte ale *BookStore* și *Book*.

# Exemplu: Conectarea claselor

- Definirea clasei BBP:

```
class BBP {  
    attribute real price;  
    relationship BookStore theBS inverse  
        BookStore::toBBP;  
    relationship Book theBook inverse  
        Book::toBBP; }
```

- *BookStore* și *Book* vor fi ambele modificate prin includerea relației numită *toBBP*, de tipul *Set<BBP>*.

# Tipurile ODL

- Tipuri de bază: *int, real/float, string, tipuri enumerare și clase.*
- Tipuri compuse:
  - *Struct* pentru structuri.
  - Tipuri colecții: *Set, Bag, List, Array* și *Dictionary*
- Tipurile de relații pot fi doar clase sau un tip colecție aplicat unei clase.

# Subclase ODL

- Corespund subclaszelor cunoscute din programarea orientată-obiect.

```
class Student:Person
{
    attribute string code;
    ...
}
```

# Chei și extensii în ODL

- Pentru o clasă se pot declara oricâte chei  
**(key <list of keys>)**
- Fiecare clasă are un *extent*, ce reprezintă mulțimea tuturor obiectelor clasei respective:
- O extensie se declară după numele clasei împreună cu cheile astfel:  
**(extent <extent name> ... )**
  - Convenție: se utilizează substantive comune la singular pentru numele claselor, și la plural *extensiile* corespunzătoare.

# Exemplu

**class Book**

(**key** name) { ... }

**class Course**

(**key** (dept, number),  
(room, hours)) { ... }

**class Student**

(**extent** Students **key** code) { ... }

# OML în SGBD OO

- Implementările OML nu sunt foarte eficiente (optimizările limbajului de interogare sunt modeste)
- Cel mai popular limbaj de interogare este OQL (*Object Query Language*) ce a fost proiectat pentru o sintaxă similară cu SQL.
- OQL poate fi privit ca o extensie a SQL
  - Include clauzele **select, from, where** și **group by**
  - S-au adăugat elemente ce accesează proprietățile obiectelor și operatori pentru tipuri de date complexe.

# Exemplu

```
class Movie (extent Movies key movieName) {  
    attribute date start;  
    attribute date end;  
    attribute string movieName;  
    relationship Set<Cinema> shownAt inverse  
        Cinema::nowShowing;  
}  
  
class Cinema (extent Cinemas key cinemaName) {  
    attribute string cinemaName;  
    attribute string address;  
    attribute integer ticketPrice;  
    relationship Set<Movie> nowShowing inverse  
        Movie::shownAt;  
    float numshowing() raises(errorCountingMovies);  
}
```

# Accesarea proprietăților obiectelor (expresii de cale)

■ Fie  $x$  un obiect al clasei  $C$ .

- Dacă  $a$  este un atribut al  $C$ , atunci  $x.a$  este valoarea acelui atribut.
- Dacă  $r$  este o relație a lui  $C$ , atunci  $x.r$  este obiectul sau colecția de obiecte cu care  $x$  este conectat prin  $r$ .
- Dacă  $m$  este o metodă a lui  $C$ , atunci  $x.m (...)$  este rezultatul aplicării lui  $m$  la  $x$ .

# OQL: Select-From-Where

- O frază OQL obișnuită are sintaxa:

```
SELECT <list of values>
      FROM <list of collections and
            names for typical members>
        WHERE <condition>
```

- Fiecare termen al clauzei FROM este:  
    <colecție> <nume membru>
- O colecție poate fi:
  - *Extensia unei clase, sau*
  - O expresie ce se evaluează la o colecție
- Pentru a schimba denumirea unui câmp, acesta va fi precedat de un nume si ":"

# Exemplu OQL

*Să se returneze cinematografele care proiectează mai mult decât un film și filmele proiectate în aceste cinematografe.*

```
SELECT mname: M.movieName ,  
       cname: C.cinemaName  
FROM Movies M, M.shownAt C  
WHERE C.numshowing() >1
```

# Tipul rezultatului unei interogări

- Implicit, tipul rezultatului unei structuri **select-from-where** este un *Bag de Struct*.
  - *Struct* are câte un câmp pentru fiecare termen al clauzei SELECT. Numele și tipul sunt preluate de la ultimul element al expresiei de cale.
- Dacă rezultatul interogării are un singur termen, acesta va fi de fapt o structură cu un singur câmp.

# Tipul rezultatului unei interogări

- Se poate adăuga DISTINCT după SELECT iar rezultatul va avea tipul *Set*, duplicatele fiind eliminate.
- La utilizarea clauzei ORDER BY rezultatul va fi o listă de structuri, ordonate după câmpurile enumerate în ORDER BY
  - Ordonarea se face crescător (ASC - implicit) sau descrescător (DESC).
  - Elementele listei pot fi accesate și utilizând indecsi ( [1], [2],... ), similar cursoarelor SQL.

# Subinterrogări

■ O expresie *select-from-where* poate fi utilizată ca subinterrogare în mai multe moduri:

- Într-o clauză FROM, ca o colecție.
- Într-o expresie logică folosită în clauza WHERE :

**FOR ALL  $x$  IN <collection> : <condition>**

**EXISTS  $x$  IN <collection> : <condition>**

## Exemplu

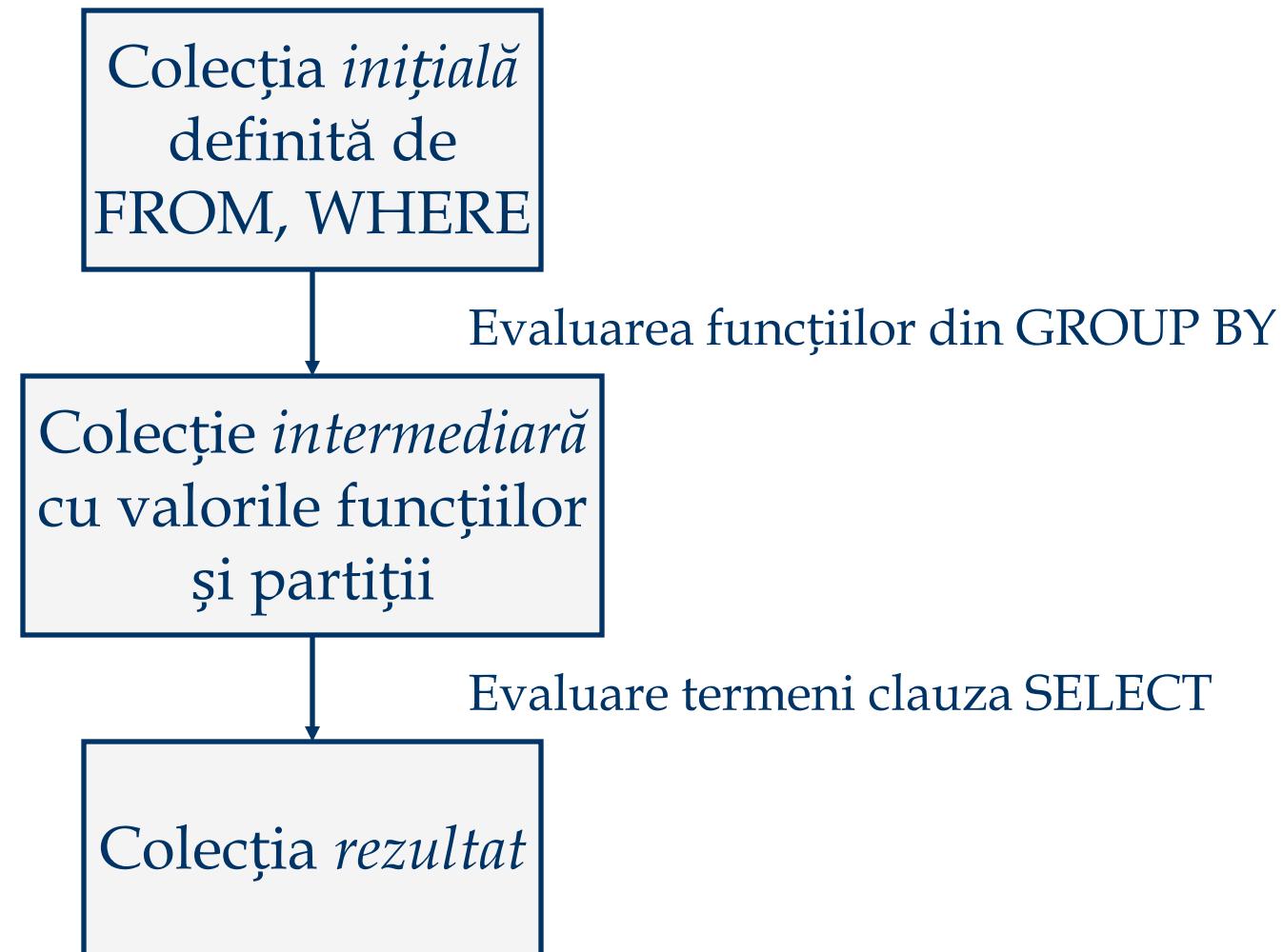
- *Să se returneze numele tuturor filmelor care sunt proiectate în cel puțin un cinematograf la un preț de bilet > 5*

```
SELECT m.name  
FROM Movies m  
WHERE  
    EXISTS c IN m.showAt:  
        c.ticketPrice > 5
```

# Gruparea datelor în OQL

- OQL extinde ideea grupării:
  - Toate colecțiile pot fi partiționate în grupuri.
  - Grupările se pot realiza având la bază orice funcție/functii ale obiectelor ce aparțin colecției initiale.
- AVG, SUM, MIN, MAX și COUNT se pot aplica tuturor colecțiilor (atunci când este cazul).

# Gruparea datelor în OQL



# Exemplu GROUP BY

*Să se returneze toate valorile distincte de preț utilizate de cinematografe și media numărului de filme proiectate cu bilete vândute la acel preț.*

```
SELECT C.ticketPrice,  
       avgNum:AVG(SELECT P.C.numshowing()  
FROM partition P)  
FROM Cinemas C  
GROUP BY C.ticketPrice
```

*Partitionarea în OQL*

## Exemplu GROUP BY: Colecția inițială

- Pe baza **FROM** și **WHERE** (care lipsește):

**FROM Cinemas C**

- Colecția inițială este un *Bag* de structuri cu un singur câmp pentru fiecare element din clauza **FROM**.
- În particular, colecția reprezintă un *Bag* de structuri de forma Struct(*c*: *obj* ), unde *obj* este un obiect *Cinema*.

# Exemplu GROUP BY: Colecția intermediară

- În general, este un *bag* de structuri cu o componentă pentru fiecare funcție din clauza GROUP BY, și o componentă suplimentară numită invariabil *partition*.
- Valoarea componentei *partition* este dată de mulțimea tuturor obiectelor din colecția inițială care aparțin grupului reprezentat de structură.

```
SELECT C.ticketPrice,  
    avgNum:AVG (  
        SELECT P.C.numshowing()  
        FROM partition P)  
FROM Cinemas C  
GROUP BY C.ticketPrice
```

O funcție de grupare:

- nume - *ticketPrice*,
- tip - *integer*.

Colecția intermediară este un *set* de structuri cu câmpurile

- *ticketPrice* : *integer*, și
- *partition*: *Set<Struct{c: Cinema}>*

## Exemplu GROUP BY: Colecția intermediară

- Un element al colecției intermediiare din exemplu este:

```
Struct(ticketPrice = 5,  
       partition = {c1, c2, ..., cn })
```

- Fiecare element al *partition* e un obiect *c<sub>i</sub>* al clasei *Cinema*, pentru care *c<sub>i</sub>.ticketPrice* = 5.

## Exemplu GROUP BY: Colecția finală

- Colecția rezultat e dată de clauza **SELECT** care este evaluată pe colecția intermediară.

# Exemplu GROUP BY: Colecția finală

```
SELECT C.ticketPrice, avgNum: AVG (  
    SELECT P.C.numshowing () FROM partition P)
```

Extrage câmpul *ticketPrice* din structura unui grup.

Pentru fiecare element *P* din *partition*, se accesează atributul *C* (obiect al *Cinema*), de unde accesează numărul de proiecții.

Media numerelor returnate de funcțiile *numshowing()* stocată în câmpul *avgNum* al structurilor din colecția finală.

Exemplu de element:  
Struct(ticketPrice = 5, avgNum = 9.5)

# Evoluția SGBD-urilor

- SGBD-urile orientate-obiect au eşuat deoarece nu au putut oferi eficiență obținută de SGBD-urile relationale.
- Extensiile relațional-obiectuale aplicate SGBD-urilor relationale captează o bună parte din avantajele OO, dar abstractizarea fundamentală rămâne relația.

# Clasificarea SGBD-urilor

