Le système O_2

langage objet
+
fonctions BD (persistance, concurrence, intégrité malgré pannes)

Les 3 langages

- OQL : langage d 'interrogation des données persistantes
- O2shell : langage de description des données
- O2C : langage de manipulation des données
  - programmation des méthodes
  - programmes de gestion des données (création, modification,...)

Types et classes

- objets atomiques
  - **integer , real, char, string** ("toto"), **boolean, bits**("0010010")
- classes prédéfinies (O_2Kit)
  - Date, Text, Bitmap
- classes structurées et objets composites
  - 4 constructeurs : **structures, listes, ensembles** et **tas**

Les objets persistants

- Persistance par référence :
  - un objet persiste s 'il est relié à une racine de persistance
  - sinon c 'est un objet temporaire
- Racines de persistance :
  - objets déclarés comme persistant dans le schéma,
  - par l 'intermédiaire de la commande *name*
  - objets interrogeables par des requêtes,
  - à partir de ces objets, on peut atteindre toutes les données persistantes

Le langage OQL

Les objets persistants

- Persistance par référence :
  - un objet persiste s 'il est relié à une racine de persistance
  - sinon c 'est un objet temporaire
- Racines de persistance :
  - objets déclarés comme persistant dans le schéma,
  - par l 'intermédiaire de la commande *name*
  - objets interrogeables par des requêtes,
  - à partir de ces objets, on peut atteindre toutes les données persistantes

Schéma agence_s
Définition des classes


```
class Pays inherit Object public type
   tuple( nom: string,
          capitale : Ville,
          population: integer,
          villes: set(Ville))
end;
```

```
class Hotel inherit Object public type
   tuple( nom: string,
          etoiles: integer,
          petit_dejeuner: integer,
          demi_pension: integer,
          pension_complete: integer,
          adresse : string,
          ville: Ville)
end;
```

```
class Ville inherit Object public type
   tuple( nom: string,
          population: integer,
          pays: Pays,
          lieux_a_visiter : set(Place),
          hotels: set(Hotel),
          tours: set(Tour))
end;
```

```
class Tour inherit Object public type
   tuple( nom: string,
          places: list(Place))
   method public tarif:integer
/* retourne le tarif cumulé des places*/
end;
```


Schéma agence_s (2)


```
class Place inherit Object private type
 tuple( public   nom: string,
        read     description: string,
        read     photo: Bitmap,
        public   adresse:
                 tuple( rue: string,
                        ville: Ville),
        public   tarif: integer)
end;
```

```
class Monument inherit Place public type
  tuple(siecle: integer,
        jours_fermeture : list(string))
end;
```

```
class Musee inherit Place public type
  tuple( specialite: string,
         jours_visite: list(string))
end;
```

Racines de persistance

```
name La_France :Pays;
name Les_Monuments :set(Monument);
name Les_Musees :set(Musee);
name Les_Pays :list(Pays);
name Les_Places :set(Place);
name Les_Villes :list(Ville);
name Monuments_Paris :set(Monument);
name Paris :Ville;
name Rome :Ville;
name Tour_Eiffel :Monument;
```


Requêtes élémentaires

- Nom d'objet : renvoie l'objet associé
  - La_France ⇒ résultat de type Pays
- Requêtes sur les atomes
  - atome : renvoie cet atome
    - 2 ⇒ 2
  - opérations arithmétiques : **+,-,*,/,mod, abs**
    - 2 + 2 ⇒ 4
  - opérations sur les chaînes : +, x**[i]**, x**[i:j]**, **count**(x)

  !! la numérotation commence à 0
  - opérations booléennes : x **and** y, x **or** y, **not**(x)

Requêtes élémentaires (2)

- Projection : accès à un attribut d'un n-uplet
  - attribut atomique : nom.attribut
    - La_France.nom ⇒ "France" (type **string**)
  - attribut non-atomique : possibilité de reprojeter
    - Tour_Eiffel.adresse ⇒ type **tuple**(rue:...)
    - Tour_Eiffel.adresse.ville ⇒ type Ville
    - Tour_Eiffel.adresse.ville.nom ⇒ "Paris"
    - La_France.villes ⇒ ensemble de villes

  !!!! La_France.villes.nom est incorrect !!!
  car nom n'est pas un attribut de la classe ensemble

Requêtes élémentaires (3)

- application d'une méthode :
  - nom.méthode[(paramètres)]
- construire un n-uplet
  - struct() renvoie n-uplet vide
  - struct(a_1:q_1, ..,a_n:q_n) a_i =nom q_i=requête
  - ↳ tuple
  - ↳ attributs de noms a_i,
  - ↳ valeurs : résultat de q_i

ex : **struct**(nom: Paris.nom, pays:Paris.pays.nom)
=> tuple(nom:Paris, pays:France)

Requêtes sur des collections

- les collections doivent être homogènes!
- opérations
  - **element(**C**)** : renvoie l'unique élément de C
  - **count, sum, max, avg, in, distinct**
  - + (concaténation)
  - **flatten** : élimination d 'un niveau de décomposition

  ex: flatten(list(list(1,2),list(3,4,5),list(6))) =>list(1,2,3,4,5,6)
  (n 'enlève qu 'un seul niveau)

Opérations sur les listes

soit a une liste

- ième élément : a**[**i**]**
  - Les_Villes[2] : 3ième ville
- sous-liste : a**[**i**:**j**]**
  - Les_Villes[0:10] : les 11 premières villes
- **first(**a**)**, **last(**a**)**
- construction : **list(**x1**,**x2**,** ... xn**)**
- transformation en ensemble : **listtoset(**a**)**

Opérations sur les ensembles et les tas

Soit a et b deux ensembles

- différence : a **except** b
- union : a **union** b
- intersection : a **intersect** b
- inclusion : <,<=,>,>=
- construction :
  - ensemble/tas vide : **set()** ou **bag()**
  - **set(**x1**,** x2**,** ..**,** xn**)** ou **bag**(x1, x2, …, xn)

Prédicats et quantificateurs

- comparaison : **=, !=, <, >, <=, >=, in, like**
  ex : nom**like "**Du*****"
- quantificateurs
  exists x **in** y **:** p(x)
  forall x **in** y **:** p(x)
  - x nom de variable,
  - y collection,
  - p prédicat = requête faisant référence à x

Select-from-where

Select [**distinct**] R
from v_1 **in** c_1, v_2 **in** c_2, ... v_n **in** c_n
[**where** e]

- c_1, c_2, ... expressions de type collection
- v_1, v_2, ... noms de variables
- e expression booléenne

↳ produit carthésien de tous les c_i, filtré par e
collection de n-uplets (v_1: type des él de c_1, v_2: ..., v_n: ..)
↳ requête R ensuite exécutée sur chaque n-uplet

Partition

select R
from v_1 **in** c_1, v_2 **in** c_2, ..., v_n **in** c_n
where e
group by att_1**:**e_1, att_2**:**e_2 ... att_m**:**e_m
att_i : noms, e_i: expressions
groupe les éléments de la collection select-from-where
selon même valeur de (e_1,...e_m)
↳ résultat : ensemble de m+1uplets
(att_1:val_1, ... att_m:val_m, **partition**:collection de n-uplets)
↳ la partition est de la forme (v1:..., v2 : ..., ..., vn:...)
↳ la requête R est ensuite exécutée sur chaque m+1 uplet.

Partition : exemple

select struct(nompays : v.pays.nom, listeville :**partition**)
from v **in** Les_Villes
group by v.pays.nom
groupe les éléments de la collection Les_Villes
selon le nom du pays
↳ résultat : ensemble couples
(nompays : ..., listeville :set(tuple(v:..),tuple(v:…),tuple(v:…)))

- **having** e'
  ↳ résultat filtré selon e'

Tri

select-from-[where]-[group-by-[having]]
order by e1 [**desc**], e2 [**desc**], ... en [**desc**]
ex :

select p **from** p **in** Les_Places
order by p.tarif **desc**

↳ collection obtenue : **liste**

Le langage O_2shell
Définition des données

Définition d'une classe

La structure d'une classe peut être:
- atomique
- complexe (tuple, list, set , etc)

La définition des méthodes est optionnelle :

[**create**] **class** <nom classe>
inherit <nom classe>, <nom classe>
<visibilité> **type** <type>
method
<visibilité> <nom méthode> <paramètres> : <type résultat>,
<visibilité> <nom méthode> <paramètres> : <type résultat>
end;

Classe : visibilité

mode de visibilité sur :
- type de la classe
- attribut de tuple
- méthode

attribut privé : les méthodes de C et des sous-classes de C
méthode privée : les méthodes de C
attribut read : lecture tous, modif méthodes de C et des sous-classes
propriété public : aucune restriction d'accès

Méthode

- Spécification

[**create**] **method** <visibilité> <nom> <paramètres> **:** <type résultat>
in class <nom classe>**;**
<paramètres> = (<nom>:<type O2> , <nom>:<type O2>,...)

- Implémentation

method body <nom> <paramètres> **in class** <nom classe> { /* code O2C */};

method public tarif(tva:real) **:** real **in class** Tour;
method body tarif(tva : real) **in class** Tour
{/* corps de la méthode : commentaires***/**
/*déclarations O2 et déclarations de variables de type **C*/**
o2 real x; ...
return(x) **; };**

Le langage O_2C
gestion des données

Syntaxe O_2C

Langage C étendu avec :

- types O_2 : (toujours précédés de "O2" !!)
  - integer, real, char, string, bits, boolean
  - [unique] set, tuple, list
  - classes
- casting comme en C incluant les types O_2
- manipulation d'objets O_2
- intégration d'OQL
- gestion de transactions

Manipulation d'objets atomiques

- **o2 integer**, **o2 real**, **o2 char** : comme en C
- **o2 boolean** : true (ou <>0), false (ou 0), &&, ||
- **o2 string** (et **o2 bits**) :
  déclaration : **o2 string** a,b="ceci est une chaîne";
  opérations : a=b; **count**(s); a==b; a!=b; a=b+c; a**[i]**; a**[i:j]**;
  a **in** b; **for (**c **in** x [**where** conditions]**){**instructions**}**

- la numération commence à 0 !

Manipulation d'objets structurés

- tuples
  - décl : **o2 tuple (**attr1 **:** type1**, ...,** attrn**:** typen**)** nomdevartuple;
  - accès aux attributs : **.**
- set/unique set :
  - décl : **o2 [unique] set (**typedeselements**)** nomvarens;
  - op : =, = = , !=, +, *,-, count(x), in, +=, -=, for (a in x) {}, element(x)
- list : comme les chaînes

Manipulation d'instances

init : méthode d'initialisation appelée implicitement par **new**
méthode init implicite : initialise l'objet aux valeurs par défaut de sa classe
méthode init utilisateur :

```
method init ( nomp: string) in class Place;
method body init ( nomp: string) in class Place
{self->nom= nomp;
};
```

new reçoit paramètres transmis à **init.**

```
o2 Place p; /*déclaration*/
p = new Place ( "Parc Astérix" ); /* affectation*/
```


Opérations sur les instances

Soient x1, x2 des instances d'une même classe

- affectation
  x1=x2 : 2 variables => même objet
- comparaison
  x1 = = x2, x1 !=x2 : compare les OID
- accès aux méthodes et propriétés : ->

```
o2 Personne p;
p->nom="Jean";
```


La classe Object

classe **Object** ⇨ méthodes génériques héritées par toutes les classes d'un schéma.

les méthodes de copie
copy et *deep_copy*
les méthodes de comparaison
equal et *deep_equal*
les méthodes de comparaison de structure
class_of et *type_of : renvoie un code*

equal et deep_equal

equal (teste les valeurs des attributs)
deep_equal (teste en profondeur)

```
Ex :      o2 Ville v, w, v1, v2 ;
     v= new Ville("Angers") ;
     w = new Ville("Angers");
/*v = = w : faux          v->equal(w) : vrai        v->deep_equal(w) : vrai */
     v->pays= new Pays("France");
     w->pays=new Pays("France");
/* v = = w : faux         v->equal(w) : faux        v->deep_equal(w) : vrai */
```

Remarque : equal => deep_equal

copy et deep_copy

copy: crée objet avec valeurs identiques à l'objet source renvoie un OID

deep_copy: idem mais crée les sous-objets

```
Ex : (suite)
v1=(o2 Ville)v->copy;
v2=(o2 Ville)v->deep_copy;
/*   v1->equal(v) : vrai
     v2->equal(v) : faux
     v2->deep_equal(v) : vrai */
```


Manipulation des objets nommés

```
name  Les_Musees_Paris : list(Musee);
```

en O_2C : racines = **variables globales**.

```
run body{
o2 Musee x;
for( x in Les_Musees_Paris) x->afficher;
display( count(Les_Musees_Paris) );
}
```

modification d'une racine : en mode transactionnel !

OQL en O_2C

fonction **o2query**.
code = o2query(resultat, requete, [<parametre>,<parametre>]);

code != 0 si requête échoue
resultat = variable O_2 contenant le résultat de la requête
requete = requête OQL sous forme de string
parametre = expression O_2 C se substituant à référence $ dans requete

```
o2 list( Pays ) resultat;
o2 integer  pop = 500;
o2query(      resultat,
   "select  x from x in Pays where x.population > $1 order by x.nom", pop );
```


Transaction

instructions O_2C pour entrer/sortir du mode transactionnel :

```
transaction;
validate;
commit;
abort;
```

par défaut, une application est en mode lecture seule