Introduction to OQL

History

- Object-oriented DBMS (OODBMS) vendors hoped to take market share from traditional relational DBMS (RDBMS) vendors by offering object-based data management
  - Extend OO languages (C++, SmallTalk) with support for persistent objects
- RDBMS vendors responded by adding object support to relational systems (i.e., ORDBMS) and largely kept their customers
- OODBMS vendors have survived in another market niche: software systems that need some of their data to be persistent (e.g., CAD)

Recall:
- ODMG: Object Database Management Group
- ODL: Object Definition Language
- OQL: Object Query Language

Query-Related Features of ODL

Example: a student can take many courses but may TA at most one

```java
interface Student (extent Students, key SID) {
    attribute integer SID;
    attribute string name;
    attribute integer age;
    attribute float GPA;
    relationship Set<Course> takeCourses
        inverse Course::students;
    relationship Course assistCourse
        inverse Course::TAs;
};
interface Course (extent Courses, key CID) {
    attribute string CID;
    attribute string title;
    relationship Set<Student> students
        inverse Student::takeCourses;
    relationship Set<Student> TAs
        inverse Student::assistCourse;
};
```
For every class we can declare an extent, which is used to refer to the current collection of all objects of that class.

We can also declare methods written in the host language.

**Basic `SELECT` Statement in OQL**

Example: find CID and title of the course assisted by Lisa

```sql
SELECT s.assistCourse.CID, s.assistCourse.title
FROM Students s
WHERE s.name = "Lisa";
```

~ In the `FROM` clause, remember to refer to the extent `Students`, not the class name `Student`,

~ “s” is a variable that ranges over the objects in `Students`

~ In *path expressions*, “.” is used to access any property (either an attribute or a relationship) of an object

Example: find CID and title of the courses taken by Lisa

```java
/* WRONG! */
SELECT s.takeCourses.CID, s.takeCourses.title
FROM Students s
WHERE s.name = "Lisa";
```

~ Problem: “.” must be applied to a single object, never to a collection of objects

~ Solution: use correlated variables in the `FROM` clause

Example: find CID and title of courses taken by either Bart or Lisa; order the result by CID and rename the result attributes to `CourseID` and `CourseTitle`

```sql
~ Without DISTINCT, the query result has type:
Bag<Struct {integer CourseID, string CourseTitle}>

~ With DISTINCT, the query result has type:
Set<Struct {integer CourseID, string CourseTitle}>

~ ORDER BY works just like in SQL
```
Operational semantics of the above SELECT query:

For each $c$ in Courses, for each $s$ in $c$.students:
  If $s$.name is Bart or Lisa, add to the output bag:
    Struct(CourseID:$c$.CID,CourseTitle:$c$.title);
Sort the output bag according to CourseID:
Eliminate duplicates from the bag and output the result set

Subqueries in OQL

Subqueries in FROM Clause

Example: classmates of CS145 students

Subqueries in WHERE Clause

EXISTS objectvar IN collection: condition
~⇒ Returns true if condition is true for at least one object in collection
Example: find courses that enroll some student with GPA higher than 4.0

FOR ALL objectvar IN collection: condition
~⇒ Returns true if condition is true for all objects in collection
Example: find students with higher GPA than all their TA's

Other Features of OQL

- SQL-style EXISTS, IN subqueries
- SQL-style quantifiers: ALL, ANY (= SOME in OQL)
- Aggregates, GROUP BY, and HAVING
- Set/bag operations: UNION, EXCEPT, and INTERSECT
- Set/bag inclusion tests: e.g., Set(1,2,3)<Set(3,4,2,1)
Interacting With an OODBMS

- “Navigational access” directly through the host language
  - Database classes are also classes in the host language
  - Database objects are manipulated in the usual way (including via methods) through the host language
  - Data and changes are persistent
- “Declarative access” through OQL
  - Similar to embedded SQL only much less awkward
  - OQL does not have data modification statements, so all modifications must be navigational

Example:

```cpp
// processing collection results:
Bag<Student> cs145Students =
    SELECT s
    FROM Students s
    WHERE EXISTS c IN s.takeCourses:
        c.CID = "CS145"
    ORDER BY s.name;
    cout << "CS145 Students:"
    for (int i=1; i<=COUNT(cs145Students); i++) {
        cout << cs145Students[i].SID << "\n" << cs145Students[i].name << "\n";
    }

// processing singleton results:
string student123Name =
    ELEMENT(SELECT s.name
            FROM Students s
            WHERE s.SID = 123);
```

 throwError In reality, the syntax could be much more complicated