

# 9장. 객체 지향 질의 언어

---

- ▣ ODL의 질의 관련 특징
- ▣ OQL의 소개
- ▣ 추가적인 OQL 수식
- ▣ OQL에서 객체 배정과 생성
- ▣ SQL3에서의 튜플 객체
- ▣ SQL3에서의 추상 데이터 타입



# OQL과 SQL3

---

## ㄴ OQL(Object Query Language)

- SQL의 장점을 객체 지향 영역으로 도입
- 릴레이션은 그다지 중요하지 않음

cf. Persistent programming language

## ㄴ SQL3

- 객체 지향의 장점을 관계 데이터베이스 영역으로 도입
- 릴레이션은 데이터를 구성하는 기본적인 개념
  - » 객체-관계(object-relational) : 객체와 클래스는 항상 릴레이션 내에서 정의되고 사용된다.

# ODL의 질의 관련 특징

---

## u ODL 객체에 대한 연산

- ODL과 호스트 언어(예를 들어 C++)는 밀착 결합(tightly coupled)
  - » ODL은 호스트 언어로 직접 변환 가능
  - » 호스트 언어의 변수는 객체를 쉽게 표현 가능
  - » 메소드는 이 둘 사이의 결합을 좀 더 편리하게 해준다.

## u 메소드(method) : 클래스와 연관된 함수

- ODL 객체에 대한 연산
- OQL에서 클래스의 애트리뷰트처럼 사용될 수 있다.

## u ODL의 질의 관련 특징

- 메소드 시그너처(signature), 클래스의 익스텐트(extent)

# ODL의 질의 관련 특징 (계속)

---

## u 시그너처(signature)

- 메소드의 이름과 매개변수의 입출력 타입을 선언한 것
- 메소드의 실제 코드는 호스트 언어로 작성
  - » 메소드의 코드는 ODL의 일부가 아님

## u 메소드 선언(즉, 시그너처)의 구문

- 아래 두가지 사항을 제외하고는 C의 함수 선언과 유사
  - n 매개변수의 명시 : `in`, `out`, `inout`
  - n 예외 사건(exception)의 발생
    - » 비정상적이거나 예상치 못한 상태에 대한 특별한 응답 방식
    - » 키워드 : `raises`

# ODL의 질의 관련 특징 (계속)

---

[예] Movie 클래스에 메소드 시그니처들을 추가

```
class Movie
    (extent Movies
     key (title, year))
{
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enumeration(color,blackAndWhite) filmType;
    relationship Set<Star> stars
        inverse Star::starredIn;
    relationship Studio ownedBy
        inverse Studio::owns;
    float lengthInHours() raises(noLengthFound);
    starNames(out Set<String>);
    otherMovies(in Star, out Set<Movie>) raises(noSuchStar);
};
```

# ODL의 질의 관련 특징 (계속)

---

## u 시그너처를 사용하는 이유

- 구현이 설계 사양과 일치하는 지를 검사
  - » 연산의 의미까지 정확하게 구현되었는지는 검사할 수 없다.
  - » 입출력 매개변수들의 수와 타입이 올바르게 사용되었는지는 검사할 수 있다.

## u 익스텐트(extent)

- 해당 클래스에 현재 존재하는 모든 객체들의 집합
- OQL 질의는 (클래스 이름이 아니라) 클래스의 익스텐트를 참조

[예] 영화 “Gone With the Wind”의 제작 연도를 검색하라.

```
SELECT m.year
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

# OQL의 소개

---

- u SQL과 유사한 표기법
- u C++, Smalltalk, Java와 같은 객체 지향 호스트 언어의 확장 언어로 사용될 수 있도록 하기위한 의도로 만들어 졌다.
  - 호스트 언어의 문장들과 OQL 질의가 서로 명시적(explicit)인 값의 전송 없이 혼합되어 사용될 수 있는 기능을 제공

# OQL의 소개 (계속)

## ODL로 작성한 영화 데이터베이스

```
class Movie
  (extent Movies
   key (title, year))
{
  attribute string title;
  attribute integer year;
  attribute integer length;
  attribute enumeration
    (color,blackAndWhite) filmType;
  relationship Set<Star> stars
    inverse Star::starredIn;
  relationship Studio ownedBy
    inverse Studio::owns;
  float lengthInHours()
    raises(noLengthFound);
  starNames(out Set<String>);
  otherMovies(in Star,out Set<Movie>)
    raises(noSuchStar);
};
```

```
class Star
  (extent Stars
   key name)
{
  attribute string name;
  attribute Struct Addr
    {string street, string city} address;
  relationship Set<Movie> starredIn
    inverse Movie::stars;
}
```

```
class Studio
  (extent Studios
   key name)
{
  attribute string name;
  attribute string address;
  relationship Set<Movie> owns
    inverse Movie:: ownedBy;
}
```





# OQL의 소개 (계속)

---

## u OQL 타입 시스템

### (1) 기본 (basic) 타입

- 원자적 타입
  - » 정수, 실수, 문자, 문자열, 부울함수
- 열거(eunmeration type)

### (2) 복합(complex) 타입

- 컬렉션 타입
  - » Set, Bag, List, Array
- 구조 : Struct
  - » `bag(2,1,2) : value`
  - » `struct(foo: bag(2,1,2), bar: "baz")`

필드 이름

필드 값

# OQL의 소개 (계속)

## u 경로 수식(path expression)

- 복합 타입 변수의 요소에 접근
- $a.p$ : 객체  $a$ 에 특성  $p$ 를 적용 한 결과

$p$ 의 특성	$a.p$ 의 값
애트리뷰트	객체 $a$ 에 있는 해당 애트리뷰트 값.
관계성	관계성 $p$ 에 의해 $a$ 와 연관된 객체 (또는 객체들의 컬렉션)
메소드	$p$ 를 $a$ 에 적용시킨 결과

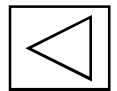
## ☀ 점(dot) 대신 화살표 사용

- OQL은 점과 같은 의미로 화살표를 사용한다. 즉,  $a \rightarrow p \circ a.p$

# OQL의 소개 (계속)

---

- u **myMovie : Movie** 객체를 값으로 가지는 호스트 언어 변수
  - **myMovie.length** : myMovie가 나타내는 **Movie** 객체의 **length**의 값
  - **myMovie.lengthInHours()** : 영화의 상영시간을 시간으로 나타낸 값으로 메소드 **lengthInHours**를 myMovie 객체에 적용시켜서 계산한 실수값
  - **myMovie.stars** : 관계성 **stars**에 의해 영화 myMovie에 연관된 **Star** 객체들의 집합
  - **myMovie.starNames(myStars)** : 영화에 출연한 스타들의 이름들을 문자열의 집합으로 myStars로 출력
  - **myMovie.ownedBy.name** : 영화를 소유하고 있는 영화사의 이름



# OQL의 소개 (계속)

---

## ⌞ SELECT-FROM-WHERE 수식

1. 키워드 **SELECT**와 그 뒤에 오는 수식 리스트
  2. 키워드 **FROM**과 그 뒤에 오는 하나 이상의 변수 선언들의 리스트
    - (a) 집합이나 백과 같은, 컬렉션 타입의 값을 갖는 수식
      - 보통, 클래스의 익스텐트가 나옴.
    - (b) 선택사항인 키워드 **AS**
    - (c) 변수의 이름
  3. 키워드 **WHERE**와 부울 값을 갖는 수식
    - » 논리 연산자 : **SQL**에서와 같이 **AND**, **OR**, **NOT**
    - » 비교 연산자 : **SQL**에서 사용한 것과 동일
      - ⌞ 단, “같지 않음”은 **!=** 로 나타냄 (<>가 아님)
- 질의의 결과는 **Bag**

# OQL의 소개 (계속)

---

[예] *Casablanca*에 나오는 스타들의 이름을 검색하라.

```
SELECT s.name
FROM   Movies m, m.Stars s
WHERE  m.title = "Casablanca"
```

OQL에서는 이중 인용부호(“”)가  
사용된다는 것에 주의하라.  
SQL에서는 단일 인용부호(“) 사용.

✓ 결과는 중첩 루프의 의미로 계산된다.

For each m in Movies DO

For each s in m.Stars DO

IF m.title = “Casablanca” THEN

add s.name to the output bag

# OQL의 소개 (계속)

- u 중복 제거 : **DISTINCT**

- u 복합 출력 타입

- 출력으로 구조(structure)를 사용할 수도 있다.

[예] 주소가 같은 스타 쌍들을 구하라.

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM   Stars s1, Stars s2
WHERE  s1.addr = s2.addr AND s1.name < s2.name
```

키워드 **SELECT** 뒤에 필드 이름과 변수를 나열하면 같은 효과를 얻을 수 있다. 즉, 첫 번째 줄을 다음과 같이 기술해도 된다.

```
SELECT star1: s1, star2: s2
```

# OQL의 소개 (계속)

---

## U SELECT 절에서 구조 표기법

- Struct (필드-이름: 값, ...)

  - » 값(value)의 구성 : 둥근 괄호 ( )

  - » 타입 정의 : 각진 괄호 < > 나 중괄호 { }

- 이전 질의어의 결과는 **Set<Struct N {star1 : Star, star2 : Star}>** 로 나타난다

: OQL 프로그램에서 나타나는 타입이지만 ODL 선언에서는 나타나지 않는다.

## U SELECT-FROM-WHERE 절에서 SQL과의 다른 점

- 문자열 상수는 이중 인용부호(" ") 사용

- “같지 않음”은 != 를 사용 (<>가 아님)


# OQL의 소개 (계속)

## ㄴ 부질의 (subquery)

- WHERE 절과 FROM 절 모두에 사용 가능
  - » SQL3 는 FROM 절에서도 부질의를 사용가능

[예] 디즈니 영화에 나오는 스타들을 찾아라.

```
SELECT DISTINCT s.name  
FROM Movies m, m.stars s  
WHERE m.ownedBy.name = "Disney"
```



```
SELECT DISTINCT s.name  
FROM (SELECT m  
      FROM Movies m  
      WHERE m.ownedBy.name = "Disney") d,  
      d.stars s
```



# OQL의 소개 (계속)

---

## ↳ 결과의 순서화 : ORDER BY

[예] 디즈니 영화들의 리스트를 상영 시간의 순서로 검색하라. (만약 상영 시간의 길이가 같으면 알파벳 순으로 순서를 정한다.)

```
SELECT m
FROM   Movies m
WHERE  m.ownedBy.name = "Disney"
ORDER BY m.length, m.title [ASC/DESC]
```

# 추가적인 OQL 수식

---

## └ 정량자 수식 (quantifier expression)

■ 범용(universal) 정량자:  $\text{FOR ALL } x \text{ IN } S : C(x)$

– 집합  $S$ 의 모든 원소가 조건  $C(x)$ 를 만족하면, 이 식의 결과는 참이다.

■ 존재(existential) 정량자:  $\text{EXISTS } x \text{ IN } S : C(x)$

– 집합  $S$ 에 있는 적어도 하나의 원소가 조건  $C(x)$ 를 만족하면, 이 식의 결과는 참이다.

# 추가적인 OQL 수식 (계속)

---

[예] 디즈니 영화에만 나오는 스타들을 찾아라.

```
SELECT  s
FROM    Stars s
WHERE   FOR ALL m IN s.starredIn :
        m.ownedBy.name = "Disney"
```

이 질의의 결과는 어떤 영화에도  
출연하지 않은 스타들도 포함

[예] 디즈니 영화에 나오는 모든 스타들을 찾아라.

```
SELECT  s
FROM    Stars s
WHERE   EXISTS m IN s.starredIn :
        m.ownedBy.name = "Disney"
```



# OQL에서 집단화 (계속)

---

## u 집단화 수식

- COUNT : 모든 컬렉션에 적용 가능
- AVG, SUM : 정수와 같은 숫자 타입에 적용
- MIN, MAX : 숫자나 문자열과 같이 비교가 가능한 타입에 적용

**[예] 모든 영화의 평균 상영시간을 계산하라.**

```
AVG(SELECT m.length FROM Movies m)
```

# OQL에서 집단화 (계속)

---

## u GROUP BY 수식

GROUP BY <분할 애트리뷰트>

- 분할(partition) 애트리뷰트들 :  $f_1:e_1, f_2:e_2, \dots, f_n:e_n$ 
  - F SQL에서 그룹화(grouping) 애트리뷰트와 같은 개념
  - »  $f_i$  : 필드 이름 (분할 애트리뷰트에 이름을 붙인 것)
  - »  $e_i$  : 수식 (예를 들어 경로 수식)

## 🍌 GROUP BY의 결과

- 다음 형태를 가진 구조들의 집합 (한 구조가 한 그룹을 나타냄)

**Struct** ( $f_1:v_1, f_2:v_2, \dots, f_n:v_n, \text{partition}:P$ )

- 처음  $n$  개의 필드가 특정 그룹을 표시
- $v_1, \dots, v_n$  는 **WHERE** 절의 조건을 만족시키는 컬렉션에 있는 적어도 하나의  $i$  에 대해  $e_1(i), \dots, e_n(i)$  를 계산한 값들이다. (각 그룹에 적어도 하나의 객체는 있다는 것을 말한다.)

# OQL에서 집 단화(계속)

---

## ■ **partition : GROUP BY** 결과의 마지막 필드

- **partition** 이라는 특별한 이름을 가지며, `struct(x:i)` 형태의 구조를 멤버로 가지는 백이다. 여기서  $x$  는 **FROM**절의 변수다. (즉, **FROM**절에 단지 한 개의 변수를 갖는 경우다.)
- **partition**을 통하여 백  $P$ 의 멤버인 구조에 있는 필드  $x$  를 참조할 수 있다.
  - ◆  $P$ 의 멤버가 (객체가 아니라) 구조인 이유는, **FROM**절에 둘 이상의 변수들이 있을 수 있기 때문이다.

## ▼ **SELECT**문에서는 **GROUP BY**에 있는 필드 즉, $f_1, f_2, \dots, f_n$ 과 **partition** 만을 참조할 수 있다.

- **SQL**에서 그룹화 애트리뷰트들만 **SELECT**절에 나올 수 있는 것과 유사한 개념이다.

# OQL에서 집 단화 (계속)

---

[예] 각 스튜디오의 연도별 영화들의 총 상영시간에 대한 표를 만들어라.

```
SELECT std, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY std: m.ownedBy.name, yr: m.year
```

– **GROUP BY**가 생성하는 집합의 멤버는 다음과 같은 형태다.

Struct(std: "Disney", yr: 1990, partition: P). 예를 들어,

("Disney", 1990, { $m_j$ ,  $m_{j+1}$ , ...})

("MGM", 1980, { $m_k$ ,  $m_{k+1}$ , ...})

$m_i$  는 구조 Struct( $m$ :  $m_{PW}$ )이며,  $m_{PW}$ 는  
Pretty Woman에 대한 Movie객체며,  
 $m$ 은 FROM 절의 변수이다.

# OQL에서 집 단화 (계속)

## ⚙ FROM 절에 변수가 하나 이상 있을 경우

- » FROM 절에 변수  $x_1, x_2, \dots, x_k$  들이 있다고 하자.
- GROUP BY 절에 있는 수식  $e_1, e_2, \dots, e_n$ 에 변수  $x_1, x_2, \dots, x_k$ 가 모두 사용될 수 있다.
- partition 필드의 값인 백에 있는 구조는 필드  $x_1, x_2, \dots, x_k$  를 가진다.
- $i_1, i_2, \dots, i_k$ 가 WHERE 절을 참으로 만드는 변수  $x_1, x_2, \dots, x_k$ 의 값이라고 하자. 그러면 GROUP BY의 결과로 생성되는 집합에는 다음과 같은 형태의 구조들이 만들어진다.

**Struct**( $f_1:e_1(i_1, \dots, i_k), \dots, f_n:e_n(i_1, \dots, i_k), \text{partition:P}$ )

» 백 P는 **Struct**( $x_1:i_1, x_2:i_2, \dots, x_k:i_k$ )인 구조들의 집합



# OQL에서 집 단화 (계속)

---

[예] 각 스튜디오의 연도별 영화들의 총 상영시간의 합과,  
출연했던 배우들중에서 가장 나이가 어린 배우의 생년월일을  
구하라.

```
SELECT std, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p),
        maxBirthday: MAX(SELECT p.s.birthday
                           FROM partition p)
FROM Movies m, m.stars s
GROUP BY std: m.ownedBy.Name, yr: m.year;
```

# OQL에서 집단화 (계속)

---

## u 그룹에 대한 조건: HAVING

HAVING <조건>

- 이 조건은 **partition** 필드(즉, 특정 그룹)에 있는 값들의 집단화 특성에 적용된다.

**[예]** 상영시간이 120분 이상인 영화를 적어도 하나 이상 만든 스튜디오에 대해서만, 각 스튜디오의 연도별 영화들의 총 상영시간의 합을 구하라.

```
SELECT std, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY std: m.ownedBy.Name, yr: m.year
HAVING MAX(SELECT p.m.length FROM partition p) > 120
```

# OQL에서 집합 연산

---

## ⌋ 집합 연산자: UNION, INTERSECT, EXCEPT

- 결과는 피연산자의 타입에 따라 집합이거나 백이 된다.
- $B_1$ 과  $B_2$ 에 멤버  $x$ 가 각각  $n_1$ 번과  $n_2$ 번씩 있다고 하자. 그리고  $B_1$ 과  $B_2$  중 적어도 하나는 백이라 하자.
  - $B_1 \dot{\cup} B_2$  :  $x$ 가  $n_1 + n_2$  번 나타난다.
  - $B_1 \cap B_2$  :  $x$ 가  $\min(n_1, n_2)$  번 나타난다.
  - $B_1 - B_2$  :  $n_1 \geq n_2$  이면  $x$ 는 0 번 나타나고,  
그렇지 않으면  $n_1 - n_2$  번 나타난다.

# OQL에서 집합 연산 (계속)

---

↳ **DISTINCT** 를 사용하면 집합이 된다.

```
( SELECT DISTINCT m
  FROM Movies m, m.Stars s
  WHERE s.name = "Harrison Ford" )
EXCEPT
( SELECT DISTINCT m
  FROM Movies m
  WHERE m.OwnedBy.name = "Disney" )
```

# OQL에서 객체 배정

---

## u 호스트 언어 변수에 객체 배정(assignment)

- 호스트 언어 변수에 **OQL** 수식의 결과를 배정할 수 있다.

[예] `oldMovies = SELECT DISTINCT m  
FROM Movies m  
WHERE m.year < 1920;`

- `oldMovies : Set<Movie>` 타입의 호스트 언어 변수

## u ELEMENT 연산자

- 원소가 하나인 집합이나 백을 하나의 멤버로 바꾸어 준다.

[예] 변수 `gwtw`의 타입이 `Movie`라고 하자.

```
gwtw = ELEMENT( SELECT m  
FROM Movies m  
WHERE m.title = "Gone With the Wind"  
);
```

# OQL에서 객체 배정 (계속)

---

## u 컬렉션의 각 멤버를 접근

1. 집합이나 백을 리스트의 형태로 변환

» **ORDER BY**

2. 결과를 리스트 타입을 갖는 호스트 언어, 예를 들어 L에 배정

3. 리스트의 각 원소를 접근

»  $L[i-1]$  : 리스트  $L$ 의  $i$ 번째 원소

# OQL에서 객체 배정 (계속)

---

[예] 각 영화의 제목, 제작연도, 상영 시간을 출력하라.

```
movieList = SELECT m
              FROM Movies m
              ORDER BY m.title,m.year;
numberOfMovies = COUNT(Movies);
for (i=0; i < numberOfMovies; i++)
{
    movie = movieList[i];
    cout << movie.title << " " << movie.year << " "
         << movie.length << "\n";
}
```

movieList의 타입은  
List<Movie> 이다.

# 새로운 객체의 생성

---

- u 존재하는 객체로부터 **SELECT-FROM-WHERE** 수식을 통해 새로운 값을 생성

- 값을 생성할 때는 둥근 괄호를 사용. (각진 괄호나 중괄호가 아님)

- [예] `SELECT DISTINCT Struct(star1: s1, star2: s2)`

- u 명시적(**explicit**)인 생성

- 상수나 수식을 구조나 컬렉션으로 명시적으로 조합

- » `x = Struct(a:1, b:2);`

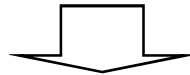
- » `y = Bag(x, x, Struct(a:3, b:4));`



# 새로운 객체의 생성 (계속)

- ▣ 새로운 객체를 생성할 때, 명시적인 타입 수식 대신 타입 이름을 사용할 수 있다.

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.addr = s2.addr AND s1.name < s2.name
```



```
Struct StarPair {star1: Star, star2: Star}

SELECT DISTINCT StarPair(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.addr = s2.addr AND s1.name < s1.name

• 결과는 Set<StarPair> 타입을 갖는다.
```

# 새로운 객체의 생성 (계속)

- ◆ 생성자(constructor function) : 타입 이름이 클래스일 때, 타입 이름을 인수들에 적용

[예] Movie 객체들의 생성자를 생각해 보자.

```
gwtw = Movie(title: "Gone With the Wind",  
              year: 1939,  
              length: 239,  
              ownedBy: mgm);
```


mgm 은 MGM Studio  
객체를 값으로 갖는 변수

- Movie 클래스에 속하는 새로운 객체를 생성한다.
- 이 객체를 호스트 언어 변수 gwtw의 값으로 만든다.



# SQL3에서의 사용자 정의 타입

---

 OQL에서는 릴레이션에 대한 명시적인 개념이 없다. 단지, 구조들의 집합이나 백의 개념만 있다. SQL3에서는 릴레이션이 객체의 중심 개념이다.

## u SQL3에서의 타입

### – 사용자 정의 타입(User-Defined Types:UDT)

- » 테이블의 타입이 될 수 있다.
- » 어떤 테이블에 속하는 애트리뷰트의 타입이 될 수 있다.

# SQL3에서의 사용자 정의 타입 (계속)

---

## u 타입 정의

- 사용자 정의 타입(UDT)은 클래스와 대체로 유사하다.

`CREATE TYPE <타입 이름> AS (<구성 요소 선언>);`

**[예] 영화 스타를 나타내는 타입을 생성하라.**

```
CREATE TYPE AddressType AS (  
    street    CHAR(50),  
    city      CHAR(20));
```

```
CREATE TYPE StarType AS (  
    name      CHAR(30),  
    address   AddressType);
```

# 사용자 정의 타입에 대한 메소드

---

## u SQL3에서 메소드

- 어떤 타입의 값을 반환한다.
- 타입 정의에서 메소드를 선언하고 메소드의 내용은 분리된 **CREATE METHOD**문에서 정의된다.

## u 메소드의 선언

```
CREATE TYPE AddressType AS (  
    street    CHAR(50),  
    city      CHAR(20)  
)  
METHOD houseNumber() RETURN CHAR(10);
```

- 주소의 일부분인 **street**로 부터 집번호를 반환하는 메소드

# 사용자 정의 타입에 대한 메소드 (계속)

---

## u 메소드의 정의

1. 키워드 **CREATE METHOD**
2. 메소드 이름, 인수들과 그들의 타입들, 반환 타입
3. 키워드 **FOR**와 메소드가 선언된 **UDT** 이름
4. 메소드의 몸체

```
CREATE METHOD houseNumber() RETURNS CHAR(10)  
FOR AddressType  
BEGIN  
    ...  
END;
```

# 사용자 정의 타입 릴레이션의 선언

---

## u 타입을 이용한 릴레이션 선언

`CREATE TABLE <릴레이션 이름> OF <타입 이름>`

[예] MovieStar를 타입이름이 StarType인 릴레이션으로 선언하라.

`CREATE TABLE MovieStar OF StarType;`

- ◆ 하나의 타입이 여러 릴레이션들의 정의에 사용될 수도 있으며, 또는 어떤 릴레이션의 정의에서도 사용되지 않아도 된다.

## u 참조(reference)

- 행 타입의 요소가 다른 행 타입을 참조할 수도 있다.
- T가 행 타입이면, **REF(T)**는 T 타입 튜플을 참조하는 참조 타입이다.
- 튜플을 객체로 생각하면, 객체의 참조는 그 객체의 객체 식별자(ID)이다.

[ex] `int sum, *ptr; ... ptr = &sum;`

# 사용자 정의 타입 릴레이션의 선언 (계속)

[예] MovieStar 릴레이션이 애트리뷰트 bestMovie를 포함하도록 하라.

```
CREATE TYPE MovieType AS(  
    title      CHAR(30),  
    year       INTEGER,  
    inColor    BIT(1)  
);
```

```
CREATE TABLE Movie OF MovieType;  
  
CREATE TYPE StarType AS(  
    name        CHAR(30),  
    address     AddressType,  
    bestMovie    REF(MovieType)  
);
```

- ❖ **SQL3**는 애트리뷰트에 컬렉션 타입을 허용하지 않는다.
- ❖ **SQL3**에는 기존의 행 타입을 변경하는 **ALTER TYPE** 등과 같은 문장이 없다. 따라서 행 타입과 그 행 타입을 갖는 테이블들을 삭제(drop)한 다음, 그 타입을 재정의하고 테이블들을 다시 생성한다.



# 사용자 정의 타입 릴레이션의 선언 (계속)

📖 스타, 영화, 그리고 이들 사이의 관계성

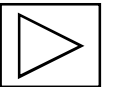
```
CREATE TYPE MovieType AS (  
    title    CHAR(30),  
    year     INTEGER,  
    inColor  BIT(1)  
);
```

```
CREATE TYPE StarType AS (  
    name      CHAR(30),  
    address   AddressType,  
);
```

```
CREATE TYPE AddressType AS (  
    street CHAR(50),  
    city   CHAR(20)  
);
```

```
CREATE TYPE StarsInType AS (  
    star      REF(StarType),  
    movie     REF(MovieType)  
);
```

```
CREATE TABLE Movie OF MovieType  
CREATE TABLE MovieStar OF StarType  
CREATE TABLE StarsIn OF StarsInType
```



# 참조 연산자들

---

## u 연산자 : ->

- $x$ 가 튜플  $t$ 에 대한 참조이고  $a$ 가  $t$ 의 애트리뷰트 일 때,  $x \rightarrow a$ 는 튜플  $t$ 에서 애트리뷰트  $a$ 의 값이다.

[cf.] `int sum = 10, *ptr; ... ptr = &sum; printf("%d", *ptr);`

[예] “Mel Gibson” 이 출연한 모든 영화의 제목과 상영시간을 검색하라.

```
SELECT movie->title, movie->length
FROM StarsIn
WHERE star->name = 'Mel Gibson';
```

## u 연산자 : Deref

- 참조에 적용하고 참조된 튜플을 생성한다.

# 참조 연산자들 (계속)

---

## ↳ Deref의 활용

```
SELECT Deref(movie)
FROM StarsIn
WHERE star->name = 'Mel Gibson';
```

- **Deref(movie)** : movie가 참조하는 영화 튜플 그 자체를 생성
- 만일 **SELECT movie**로 사용한다면
  - » 튜플들에 대한 알 수 없는 식별자 리스트를 얻게 된다.

# 참조의 영역

## ↳ 참조의 영역(scope)

» 여러 릴레이션들이 같은 행 타입으로 선언될 수 있다.

[예] 다음 질의를 계산해 보자.

```
SELECT movie->title
FROM   StarsIn
WHERE  star->name = 'Mel Gibson';
```



- 각 **StarsIn** 튜플에 대해, **star**의 참조를 따라간 다음 참조된 튜플의 **name** 애트리뷰트가 “**Mel Gibson**”인지 확인한다.
    - 시간이 많이 소요 (**time-consuming**)
  - **MovieStar** 릴레이션의 **name**에 대한 다음과 같은 색인을 이용
    - 애트리뷰트 **name**값에 대해, 이 값을 갖는 **MovieStar**의 튜플을 참조하는 **StarsIn**의 튜플을 가리키는 색인
- F Note:** **star** 는 **StarType** 타입 튜플의 참조이며, **StarType** 타입으로 선언된 릴레이션들이 다수 있을 수 있다.

# 참조의 영역 (계속)

---

- n 참조 애트리뷰트가 어떤 릴레이션을 참조하는지 명시

<애트리뷰트> REF(타입) SCOPE <릴레이션>

```
CREATE TYPE StarType AS (  
    name          CHAR(30),  
    address       AddressType,  
    bestMovie     REF(MovieType) SCOPE Movie  
);
```

```
CREATE TABLE MovieStar OF StarType (  
    REF IS starID SYSTEM GENERATED,  
    PRIMARY KEY (name)  
);
```

- starID값은 새로운 튜플이 생성될 때마다 DBMS가 생성한다.
- 'SYSTEM GENERATE'대신 'DERIVED'로 바뀌면 starID 값은 사용자가 생성한다.

# UDT 튜플들의 애트리뷰트 접근

---

## u 튜플의 의미

- UDT로 정의된 테이블에서 튜플들은 단일 객체들로 인식
- **Movie** 테이블의 튜플 **t**는 구성 요소로 하나의 객체를 가진다.
  - » 튜플 **t**를 위한 UDT는 (**title**, **year**, **inColor**) 등의 세개의 구성 요소를 가진다.

## u 관찰자(observer) 메소드 : 애트리뷰트를 접근

- **t**가 타입 **T**를 가지는 변수이고 **x**가 **T**의 애트리뷰트라면 **t.x()**는 튜플(객체)에 있는 **x**의 값이다.

**SELECT m.year()**

**FROM Movie m**

**WHERE m.title() = 'King Kong';**

- **Movie** 테이블의 각 객체 **m**에 대해서 관찰자 메소드를 적용

# 생성자와 변경자 함수들

---

## u 의미

- UDT가 정의될 때마다 관찰자 메소드들과 함께 자동적으로 생성된다.

## u 함수들

- 생성자(generator) 메소드 : 타입 이름이 생성자 메소드가 된다. T가 UDT면 T()는 타입 T의 빈 객체를 반환한다.
- 변경자(mutator) 메소드 : UDT T의 각 애트리뷰트 x에 대해서 변경자 메소드 (v)가 있다.
  - » 해당 객체의 애트리뷰트 x에 대한 값을 v로 변경한다.

# 생성자와 변경자 함수들 (계속)

## ❏ 프로시저어 InsertStar

- **street, city, name**을 인수로 가지고 새로운 **StarType**의 객체를 **MovieStar** 테이블에 삽입한다.

```
CREATE PROCEDURE InsertStar (  
    IN s CHAR(50), IN c CHAR(20), IN n CHAR(30)  
)  
DECLARE newAddr AddressType;  
DECLARE newStar StarType;  
BEGIN  
    SET newAddr = AddressType();  
    SET newStar = StarType();  
    newAddr.street(s); newAddr.city(c);  
    newStar.name(n); newStar.address(newAddr);  
    INSERT INTO MovieStar VALUES(newStar);  
END;
```

```
InsertStar('345 Spruce St.', 'Glendale',  
          'Gwyneth Paltrow');
```

또는

```
INSERT INTO MovieStar VALUES(  
    StarType('Gwyneth Paltrow',  
    AddressType('345 Spruce St.', 'Glendale')));
```



# UDT에 대한 비교 및 정렬

---

## u UDT에서 비교 연산

- 기본적으로 UDT인 객체들에 대한 비교는 불가능하다.
  - » 같은 값을 가지는 두 UDT 객체들의 동등(equality) 비교는 불가능
- 따라서, 객체들의 정렬, 중복 제거, 그룹화 등의 연산도 불가능하다.

## u 비교 연산 정의

- 동등 비교

**CREATE ORDERING FOR <타입> EQUALS ONLY BY STATE;**

» 객체의 구성요소의 값이 전부 같을 때 동등하다고 판단, 대소비교는 지원되지 않는다.

- 대소 비교

**CREATE ORDERING FOR <타입>**

**ORDERING FULL BY RELATIVE WITH <비교함수>**

» 비교함수  $F$ 는  $x_1 = x_2$ 인 경우  $F(x_1, x_2) = 0$ ,  $x_1 > x_2$ 면  $F(x_1, x_2) > 0$ 을 반환하는 사용자 정의 함수이다.

» 'ORDERING FULL'을 'EQUALS ONLY'로 바꾸면  $F(x_1, x_2) = 0$ 은  $x_1 = x_2$ 를 의미하고 나머지  $F(x_1, x_2)$  값은  $x_1 \neq x_2$ 를 의미한다.

# UDT에 대한 비교 및 정렬 (계속)

---

## u UDT StarType에 대한 가능한 정렬의 정의

```
CREATE ORDERING FOR StarType EQUALS ONLY BY STATE;  
CREATE ORDERING FOR AddressType EQUALS ONLY BY STATE;
```

```
CREATE ORDERING FOR AddressType  
ORDERING FULL BY RELATIVE WITH AddrLEG
```

```
CREATE FUNCTION AddrLEG (  
    x1 AddressType, x2 AddressType ) RETURN INTEGER  
IF x1.city() < x2.city() THEN RETURN (-1)  
ELSEIF x1.city() > x2.city() THEN RETURN (1)  
ELSEIF x1.street() < x2.street() THEN RETURN (-1)  
ELSEIF x1.street() > x2.stree() THEN RETURN (1)  
ELSE RETURN (0)  
END IF;
```