Object-Oriented Databases

Old data application의 특성
. Uniformity : 동일한 크기를 갖는 구조를 갖는 대량의 data item들이 존재
. Record Orientation : 고정 길이 record들로 구성됨
. Small data items : record 크기가 작음(수백 바이트이상이 드물다)
. Atomic fields : record 내의 field들은 길이가 짧고 고정 길이임. Field 내에 어떠한 구조도 존재하지 않음

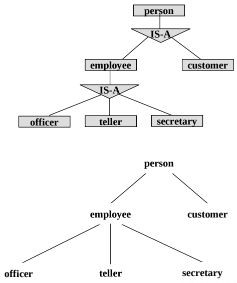
Old data application 특성들을 갖지 못하는 New Data applications
. CAD
. CASE(Computer-aided software engineering)
. Multimedia Database(대량의 비구조적인 data)
. OIS(Office information system) : OIS DB는 스케쥴, document, document 내용에 적절한 질의를 지원해야 함
. Hyper Text database : 링크에 근거한 질의와 hyper text 구조에 근거한 질의를 지원해야 함

ER model과 Relation Model은 이러한 new application들에 대한 data 요구사항에 적합한 모델이 아님
. Business application도 시간에 따라 더욱 복잡해지고 있기 때문에 relational model로 몇몇 요구사항들은 표현하기 어려움(image 데이터나 hyper text 데이터베이스를 다루는 경우)

Object-Oriented Data Model

Inheritance

Specialization Hierarchy



```
class person {
   stirng name;
   string address;
};
class customer isa person {
   int credit-rating;
};
class employee isa person {
   date start-date;
   int salary;
};
class officer isa employee {
   int office-number;
   int expense-account-number;
};
class teller isa employee {
   int hours-per-week;
   int station-number;
};
class secretary isa employee {
   int hours-per-week;
   string manager;
};
```

Object-Oriented Data Model

Object Structure
. *Object* : ER model의 entity set 또는 entity에 대응됨.
Object들간의 인터페이스는 message들의 집합에 의해 정의됨
. *Variable(attribute)* : Object에 대한 data를 포함함
. *Message* : 0개 이상의 파라메터를 가질 수 있다.
read-only message
update message
. *Method* : message를 구현하는 코드 부분
read-only method
update method
. *Encapsulation*(데이터와 코드를 encapsulation)

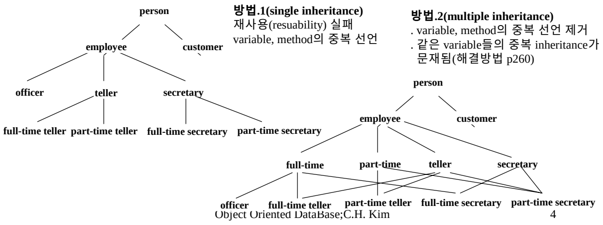
```
Class employee {
/*Variables */
    string name;
    string address;
    date start-date;
    int salary;
/* Messages */
    int annual-salary();
    string get-name();
    string get-address();
    int set-address(string new address);
    int employee-length()
```

Object Class
. *Class*
. *Instance*

class object의 특성
. 그 값이 해당 class의 instance인 모든 object들의 set인 variable(**set-valued variable**)
. 새로운 instance를 구현하기 위해 message new 에 대한 method 구현

Object(Instance)를 class에 관련시키는 방법
1. officer, teller, secretary의 instance들을 employee class에 관련시키는 방법
2. officer, teller, secretary의 instance가 아닌 employee object들만을 employee class에 관련시키는 방법
(이 방법이 대부분의 object-oriented system에서 채택되고 있음)

Multiple Inheritance
. Full-time employee와 part-time employee 를 구분해서 full-time employee와 part-time employee를 나타내는 다른 variable와 method를 나타내기 위한 방법(아래 그림 참조)



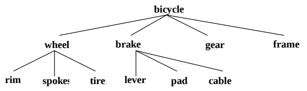
방법.1(single inheritance)
재사용(resuability) 실패
variable, method의 중복 선언

방법.2(multiple inheritance)
. variable, method의 중복 선언 제거
. 같은 variable들의 중복 inheritance가 문제됨(해결방법 p260)

Object Identity(Oid)
- object가 생성될 때 object-oriented system에 의해 자동으로 생성됨
- object마다 unique하게 할당됨
- object간의 relationship을 나타내기 위해 attribute 값으로 object의 identifier를 이용함
- Sytem-generated ID는 대부분 시스템 고유이기 때문에 다른 database system으로 옮기기 위해서는 변환되어야 한다. 동일한 메커니즘을 이용하는 경우 ID들이 중복될 가능성이 발생한다.

Object Containment(Aggregation)
- **Complex Object or Composite Object** : object를 포함하는 object
- **Containment Hierarchy(is-part-of 관계)**

Object Containment Hierarchy



Object-Oriented Languages

Object-Oriented model concepts을 이용하는 방법
1. Object-orientation concepts을 설계 도구로서만 사용한 다음 relational database로 encode한다.
(즉, conceptual model 로서만 사용하는 방법)
2. Object-oriemtation concepts이 database를 조작하기 위한 언어로 구체화하는 방법
- SQL과 같은 DML을 complex type과 orientation 개념을 첨가하여 확장하는 방법
relation scheme에 대해 object-oriented 확장을 제공하는 시스템을 object-relational system이라 함
- 기존의 object-oriented programming language를 database를 처리할 수 있도록 확장하는 방법.
이러한 언어를 persistent programming language라 함

Persistent Programming Language

Data persistency : 데이터를 생성한 프로그램이 종료된 후에도 그 데이터가 계속 존재하는 성질
Persistent Object
Persistent Programming Language : persistent data를 조작하는 구성요소들로 확장된 programming language

Persistent programming language 와 embedded SQL과의 차이점

1. Embedded language에서 host language의 type system과 DML의 type system`은 서로 다르기 때문에 프로그래머가 이들간의 type 변환을 수행해야 한다.(검출되지 않은 오류가능성과 application code가 매우 커지는 경향이 있다)
   반면, Persistent PL은 query language와 동일한 type system을 공유하므로 type 변환없이 object를 생성 및 저장할 수 있다.
2. Embedded query language를 사용하는 프로그래머는 DB에서 주기억장치로 데이터를 fetch하고 기록하기 위한 명확한 code를 작성해야 한다.
   반면, Persistent PL의 경우 프로그래머는 DB에서 주기억장치로 데이터를 fetch하고 기록하기 위한 명확한 code를 작성할 필요가 없다.

C++, Smalltalk와 같은 persistent programming language 버전이 소개됨
프로그래머는 프로그래밍언어를 이용하여 DB 데이터를 직접 조작 가능

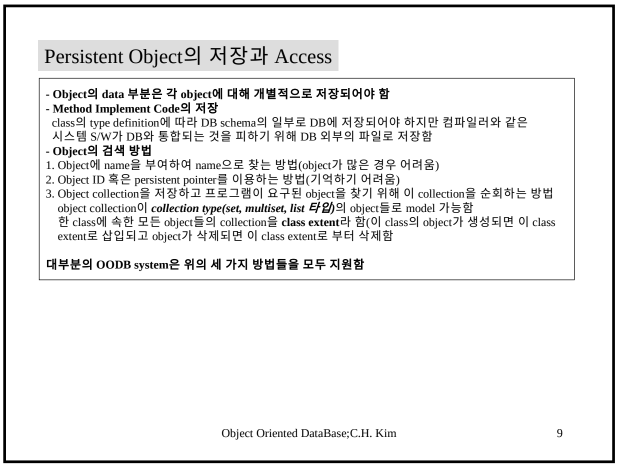
Persistent programming language의 단점 : DB에 손상을 가할 수 있는 프로그래밍 오류를 범할 가능성이 존재

Object Persistence

Object-oriented programming language는 ***transient***함
Object persistence를 제공하기 위한 접근 방법
- **Persistence by class**
  class가 persistent함을 선언하거나 default로 처리하는 방법. Nonpersistent object는 모두 transient (flexible 하지 않음, 한 class 내의 object들의 일부는 persistent, 다른 일부는 transient해야 함) persistable classes 선언 가능
- **Persistence by creation**
  persistent object를 생성하기 위한 new syntax를 도입하는 방법(여러 OODB 시스템에서 사용함)
- **Persistence by marking**
  object가 생성된 후 persistent object임을 mark 하는 방법
- **Persistence by reference**
  하나 이상의 object들이 (root) persistent objects로 명시적으로 선언. Root persistent object로부터 직접,간접으로 참조된 모든 object는 persistent함. 또한 이 object들로 부터 참조된 object들 또한 persistent함(계속적으로)

Object Identity and Pointers

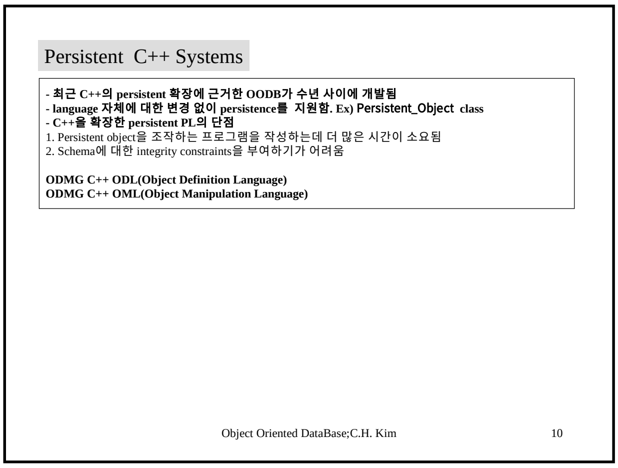
Identity permanence의 degrees
- **Intraprocedure** : ex) procedure내의 local variable
- **Intraprogram** : ex) PL에서의 global variable
- **Interprogram** : 프로그램들 사이에서 persistent한 identifier. Ex) 디스크상의 파일 시스템 데이터에 대한 pointer(재구성시 변경됨)
- **Persistent** : 프로그램 사이 뿐만 아니라 데이터에 대한 구조적 재구성에서도 persistent한 Identifier Object-oriented system에서 필요함



Persistent Object의 저장과 Access

- Object의 data 부분은 각 object에 대해 개별적으로 저장되어야 함
- Method Implement Code의 저장
class의 type definition에 따라 DB schema의 일부로 DB에 저장되어야 하지만 컴파일러와 같은 시스템 S/W가 DB와 통합되는 것을 피하기 위해 DB 외부의 파일로 저장함
- Object의 검색 방법
1. Object에 name을 부여하여 name으로 찾는 방법(object가 많은 경우 어려움)
2. Object ID 혹은 persistent pointer를 이용하는 방법(기억하기 어려움)
3. Object collection을 저장하고 프로그램이 요구된 object을 찾기 위해 이 collection을 순회하는 방법
object collection이 ***collection type(set, multiset, list 타입)***의 object들로 model 가능함
한 class에 속한 모든 object들의 collection을 **class extent**라 함(이 class의 object가 생성되면 이 class extent로 삽입되고 object가 삭제되면 이 class extent로 부터 삭제함

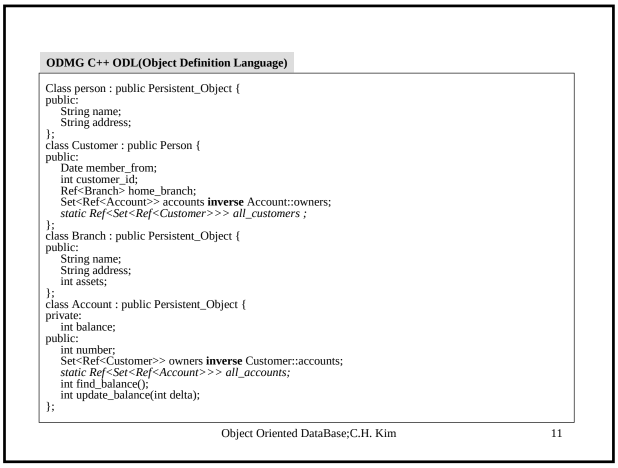
대부분의 OODB system은 위의 세 가지 방법들을 모두 지원함



Persistent C++ Systems

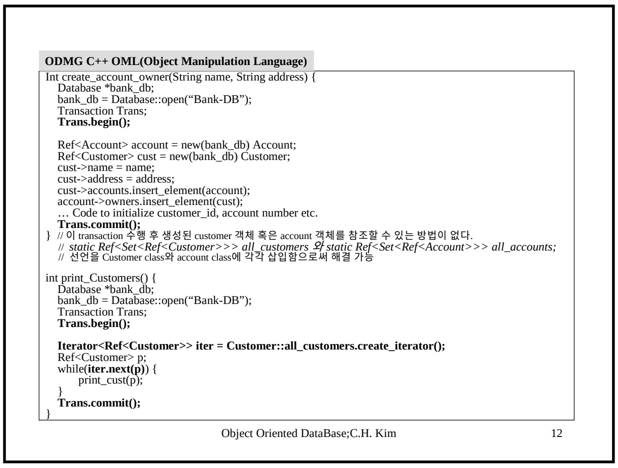
- 최근 C++의 persistent 확장에 근거한 OODB가 수년 사이에 개발됨
- language 자체에 대한 변경 없이 persistence를 지원함. Ex) Persistent_Object class
- C++을 확장한 persistent PL의 단점
1. Persistent object을 조작하는 프로그램을 작성하는데 더 많은 시간이 소요됨
2. Schema에 대한 integrity constraints을 부여하기가 어려움

ODMG C++ ODL(Object Definition Language)
ODMG C++ OML(Object Manipulation Language)



ODMG C++ ODL(Object Definition Language)

```
Class person : public Persistent_Object {
public:
    String name;
    String address;
};
class Customer : public Person {
public:
    Date member_from;
    int customer_id;
    Ref<Branch> home_branch;
    Set<Ref<Account>> accounts inverse Account::owners;
    static Ref<Set<Ref<Customer>>> all_customers ;
};
class Branch : public Persistent_Object {
public:
    String name;
    String address;
    int assets;
};
class Account : public Persistent_Object {
private:
    int balance;
public:
    int number;
    Set<Ref<Customer>> owners inverse Customer::accounts;
    static Ref<Set<Ref<Account>>> all_accounts;
    int find_balance();
    int update_balance(int delta);
};
```



ODMG C++ OML(Object Manipulation Language)

```
Int create_account_owner(String name, String address) {
    Database *bank_db;
    bank_db = Database::open("Bank-DB");
    Transaction Trans;
    Trans.begin();

    Ref<Account> account = new(bank_db) Account;
    Ref<Customer> cust = new(bank_db) Customer;
    cust->name = name;
    cust->address = address;
    cust->accounts.insert_element(account);
    account->owners.insert_element(cust);
    … Code to initialize customer_id, account number etc.
    Trans.commit();
}  // 이 transaction 수행 후 생성된 customer 객체 혹은 account 객체를 참조할 수 있는 방법이 없다.
    //  static Ref<Set<Ref<Customer>>> all_customers 와 static Ref<Set<Ref<Account>>> all_accounts;
    //  선언을 Customer class와 account class에 각각 삽입함으로써 해결 가능

int print_Customers() {
    Database *bank_db;
    bank_db = Database::open("Bank-DB");
    Transaction Trans;
    Trans.begin();

    Iterator<Ref<Customer>> iter = Customer::all_customers.create_iterator();
    Ref<Customer> p;
    while(iter.next(p)) {
          print_cust(p);
    }
    Trans.commit();
}
```