Chapter 12 Outline

- Overview of Object Database Concepts
- Object-Relational Features
- Object Database Extensions to SQL
- ODMG Object Model and the Object Definition Language ODL
- Object Database Conceptual Design
- The Object Query Language OQL
- Overview of the C++ Language Binding
Object and Object-Relational Databases

- **Object databases (ODB)**
  - Object data management systems (ODMS)
  - Meet some of the needs of more complex applications
- Specify:
  - Structure of complex objects
  - Operations that can be applied to these objects
Overview of Object Database Concepts

- Introduction to object-oriented concepts and features
  - Origins in OO programming languages
  - Object has two components:
    - State (value) and behavior (operations)
  - Instance variables (attributes)
    - Hold values that define internal state of object
  - Operation is defined in two parts:
    - Signature (interface) and implementation (method)
Overview of Object Database Concepts (cont’d.)

- Inheritance
  - Permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes

- Operator overloading
  - Operation’s ability to be applied to different types of objects
  - Operation name may refer to several distinct implementations
Object Identity, and Objects versus Literals

- **Object has Unique identity**
  - Implemented via a unique, system-generated object identifier (OID)
- **Immutable**
- Most OO database systems allow for the representation of both objects and literals (simple or complex values)
Complex Type Structures for Objects and Literals

- Structure of arbitrary complexity
  - Contain all necessary information that describes object or literal
- Nesting **type constructors**
  - Generate complex type from other types
- Type constructors (type generators):
  - Atom (basic data type – int, string, etc.)
  - Struct (or tuple)
  - Collection
Complex Type Structures for Objects and Literals (cont’d.)

- Collection types:
  - Set
  - Bag
  - List
  - Array
  - Dictionary

- Object definition language (ODL)
  - Used to define object types for a particular database application
**Figure 12.1** Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

```plaintext
define type EMPLOYEE
tuple (  Fname: string;
        Minit: char;
        Lname: string;
        Ssn: string;
        Birth_date: DATE;
        Address: string;
        Sex: char;
        Salary: float;
        Supervisor: EMPLOYEE;
        Dept: DEPARTMENT;
)

define type DATE
tuple (  Year: integer;
        Month: integer;
        Day: integer;
)

define type DEPARTMENT
tuple (  Dname: string;
        Dnumber: integer;
        Mgr: tuple (  Manager: EMPLOYEE;
                       Start_date: DATE;
                       )
        Locations: set(string);
        Employees: set(EMPLOYEE);
        Projects: set(PROJECT);
)
**Figure 12.2** Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

```plaintext
define class EMPLOYEE
type tuple {
    Fname: string;
    Minit: char;
    Lname: string;
    Ssn: string;
    Birth_date: DATE;
    Address: string;
    Sex: char;
    Salary: float;
    Supervisor: EMPLOYEE;
    Dept: DEPARTMENT;
};

operations
    age: integer;
    create_emp: EMPLOYEE;
    destroy_emp: boolean;
end EMPLOYEE;

define class DEPARTMENT
type tuple {
    Dname: string;
    Dnumber: integer;
    Mgr: tuple {  
        Manager: EMPLOYEE;
        Start_date: DATE;
    };
    Locations: set (string);
    Employees: set (EMPLOYEE);
    Projects: set (PROJECT);
};

operations
    no_of_emps: integer;
    create_dept: DEPARTMENT;
    destroy_dept: boolean;
    assign_emp(e: EMPLOYEE): boolean;
    (* adds an employee to the department *)
    remove_emp(e: EMPLOYEE): boolean;
    (* removes an employee from the department *)
end DEPARTMENT;
```
Encapsulation of Operations

- **Encapsulation**
  - Related to abstract data types
  - Define *behavior* of a class of object based on operations that can be externally applied
  - External users only aware of interface of the operations
  - Can divide structure of object into visible and hidden attributes
Encapsulation of Operations

- **Constructor** operation
  - Used to create a new object

- **Destructor** operation
  - Used to destroy (delete) an object

- **Modifier** operations
  - Modify the state of an object

- **Retrieve** operation

- *Dot notation* to apply operations to object
Persistence of Objects

- **Transient objects**
  - Exist in executing program
  - Disappear once program terminates

- **Persistent objects**
  - Stored in database, persist after program termination
  - **Naming mechanism**: object assigned a unique name in object base, user finds object by its name
  - **Reachability**: object referenced from other persistent objects, object located through references
define class DEPARTMENT_SET

type set (DEPARTMENT);

operations add_dept(d: DEPARTMENT): boolean;
    (* adds a department to the DEPARTMENT_SET object *)
    remove_dept(d: DEPARTMENT): boolean;
    (* removes a department from the DEPARTMENT_SET object *)
    create_dept_set: DEPARTMENT_SET;
    destroy_dept_set: boolean;

end Department_Set;

...

persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
(* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
...

d:= create_dept;
(* create a new DEPARTMENT object in the variable d *)
...

b:= ALL_DEPARTMENTS.add_dept(d);
(* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
Type (Class) Hierarchies and Inheritance

- **Inheritance**
  - Definition of new types based on other predefined types
  - Leads to *type* (or *class*) hierarchy

- **Type**: *type name* and list of visible (public) functions (attributes or operations)

- **Format**:
  - `TYPE_NAME: function, function, ..., function`
Type (Class) Hierarchies and
Inheritance (cont’d.)

- **Subtype**
  - Useful when creating a new type that is similar but not identical to an already defined type
  - Subtype inherits functions
  - Additional (local or specific) functions in subtype
  - Example:
    - EMPLOYEE subtype-of PERSON: Salary, Hire_date, Seniority
    - STUDENT subtype-of PERSON: Major, Gpa
Type (Class) Hierarchies and Inheritance (cont’d.)

- **Extent**
  - A *named persistent object* to hold collection of all persistent objects for a class

- **Persistent collection**
  - Stored permanently in the database

- **Transient collection**
  - Exists temporarily during the execution of a program (e.g. query result)
Other Object-Oriented Concepts

- **Polymorphism** of operations
  - Also known as **operator overloading**
  - Allows same operator name or symbol to be bound to two or more different implementations
  - Type of objects determines which operator is applied

- **Multiple inheritance**
  - Subtype inherits functions (attributes and operations) of more than one supertype
Summary of Object Database Concepts

- Object identity
- Type constructors (type generators)
- Encapsulation of operations
- Programming language compatibility
- Type (class) hierarchies and inheritance
- Extents
- Polymorphism and operator overloading
Object-Relational Features: Object DB Extensions to SQL

- **Type constructors (generators)**
  - Specify complex types using UDT
- **Mechanism for specifying object identity**
- **Encapsulation of operations**
  - Provided through user-defined types (UDTs)
- **Inheritance mechanisms**
  - Provided using keyword `UNDER`
User-Defined Types (UDTs) and Complex Structures for Objects

- **UDT syntax:**
  - CREATE TYPE <type name> AS
    (<component declarations>);
  - Can be used to create a complex type for an attribute (similar to `struct` – no operations)
  - Or: can be used to create a type as a basis for a table of objects (similar to `class` – can have operations)
User-Defined Types and Complex Structures for Objects (cont’d.)

- Array type – to specify collections
  - Reference array elements using []
- **CARDINALITY** function
  - Return the current number of elements in an array
- Early SQL had only array for collections
  - Later versions of SQL added other collection types (set, list, bag, array, etc.)
Object Identifiers Using Reference Types

- **Reference type**
  - Create unique object identifiers (OIDs)
  - Can specify system-generated object identifiers
  - Alternatively can use primary key as OID as in traditional relational model

- Examples:
  - `REF IS SYSTEM GENERATED`
  - `REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;`
Creating Tables Based on the UDTs

- **INSTANTIABLE**
  - Specify that UDT is instantiable
  - The user can then create one or more tables based on the UDT
  - If keyword INSTANTIABLE is left out, can use UDT only as attribute data type – not as a basis for a table of objects
Encapsulation of Operations

- **User-defined type**
  - Specify methods (or operations) in addition to the attributes
  - Format:
    ```sql
    CREATE TYPE <TYPE-NAME> (  
    <LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>  
    <DECLARATION OF FUNCTIONS (METHODS)>  
    ) ;
    ```
Figure 12.4a  Illustrating some of the object features of SQL. Using UDTs as types for attributes such as Address and Phone.

(a)  CREATE TYPE STREET_ADDR_TYPE AS (  
    NUMBER         VARCHAR (5),  
    STREET         NAME VARCHAR (25),  
    APT_NO         VARCHAR (5),  
    SUITE_NO       VARCHAR (5)   
);  

CREATE TYPE USA_ADDR_TYPE AS (  
    STREET_ADDR STREET_ADDR_TYPE,  
    CITY          VARCHAR (25),  
    ZIP           VARCHAR (10)   
);  

CREATE TYPE USA_PHONE_TYPE AS (  
    PHONE_TYPE VARCHAR (5),  
    AREA_CODE   CHAR (3),  
    PHONE_NUM   CHAR (7)   
);  

continued on next slide
Figure 12.4b  Illustrating some of the object features of SQL. Specifying UDT for PERSON_TYPE.

**(b)** `CREATE TYPE PERSON_TYPE AS (
  NAME VARCHAR(35),
  SEX CHAR,
  BIRTH_DATE DATE,
  PHONES USA_PHONE_TYPE ARRAY [4],
  ADDR USA_ADDR_TYPE

INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD `AGE()` RETURNS INTEGER;
CREATE INSTANCE METHOD `AGE()` RETURNS INTEGER
  FOR PERSON_TYPE
BEGIN
  RETURN /* CODE TO CALCULATE A PERSON’S AGE FROM TODAY’S DATE AND SELF.BIRTH_DATE */
END;
);

continued on next slide
Specifying Type Inheritance

- **NOT FINAL:**
  - The keyword NOT FINAL indicates that subtypes can be created for that type

- **UNDER**
  - The keyword UNDER is used to create a subtype
Figure 12.4c Illustrating some of the object features of SQL. Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE.

(c) CREATE TYPE GRADE_TYPE AS (  
    COURSENO    CHAR (8),  
    SEMESTER   VARCHAR (8),  
    YEAR       CHAR (4),  
    GRADE      CHAR  
);

CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (  
    MAJOR_CODE  CHAR (4),  
    STUDENT_ID  CHAR (12),  
    DEGREE      VARCHAR (5),  
    TRANSCRIPT  GRADE_TYPE ARRAY [100]

continued on next slide
Figure 12.4c (continued) Illustrating some of the object features of SQL. Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE.

```
INSTANTIABLE
NOT FINAL
INSTANCE METHOD GPA() RETURNS FLOAT;
CREATE INSTANCE METHOD GPA() RETURNS FLOAT
    FOR STUDENT_TYPE
    BEGIN
        RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                  SELF.TRANSCRIPT */
    END;
);
CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (  
    JOB_CODE    CHAR (4),
    SALARY      FLOAT,
    SSN         CHAR (11)
    INSTANTIABLE
    NOT FINAL
    );
CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (  
    DEPT_MANAGED CHAR (20)
    INSTANTIABLE
    );
```

continued on next slide
Specifying Type Inheritance

- **Type inheritance rules:**
  - All attributes/operations are inherited
  - Order of supertypes in UNDER clause determines inheritance hierarchy
  - Instance (object) of a subtype can be used in every context in which a supertype instance used
  - Subtype can redefine any function defined in supertype
Creating Tables based on UDT

- UDT must be INSTANTIABLE
- One or more tables can be created
- Table inheritance:
  - UNDER keyword can also be used to specify supertable/subtable inheritance
  - Objects in subtable must be a subset of the objects in the supertable
Figure 12.4d  Illustrating some of the object features of SQL. Creating tables based on some of the UDTs, and illustrating table inheritance.

(d) `CREATE TABLE PERSON OF PERSON_TYPE
    REF IS PERSON_ID SYSTEM GENERATED;
CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
    UNDER PERSON;
CREATE TABLE MANAGER OF MANAGER_TYPE
    UNDER EMPLOYEE;
CREATE TABLE STUDENT OF STUDENT_TYPE
    UNDER PERSON;`
Specifying Relationships via Reference

- Component attribute of one tuple may be a **reference** to a tuple of another table
  - Specified using keyword **REF**

- **Keyword SCOPE**
  - Specify name of table whose tuples referenced

- **Dot notation**
  - Build path expressions
  - \( \rightarrow \)
  - Used for dereferencing
Figure 12.4e  Illustrating some of the object features of SQL. Specifying relationships using REF and SCOPE.

(e) CREATE TYPE COMPANY_TYPE AS (
    COMP_NAME VARCHAR(20),
    LOCATION VARCHAR(20));
CREATE TYPE EMPLOYMENT_TYPE AS (
    Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
    Company REF (COMPANY_TYPE) SCOPE (COMPANY)));
CREATE TABLE COMPANY OF COMPANY_TYPE ( 
    REF IS COMP_ID SYSTEM GENERATED, 
    PRIMARY KEY (COMP_NAME));
CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
Summary of SQL Object Extensions

- UDT to specify complex types
  - INSTANTIABLE specifies if UDT can be used to create tables; NOT FINAL specifies if UDT can be inherited by a subtype
- REF for specifying object identity and inter-object references
- Encapsulation of operations in UDT
- Keyword UNDER to specify type inheritance and table inheritance
ODMG Object Model and Object Definition Language ODL

- ODMG object model
  - Data model for **object definition language (ODL)** and **object query language (OQL)**

- Objects and Literals
  - Basic building blocks of the object model

- Object has five aspects:
  - **Identifier**, name, lifetime, **structure**, and creation

- Literal
  - Value that does not have an object identifier
Behavior refers to operations

State refers to properties (attributes)

Interface

- Specifies only behavior of an object type
- Typically noninstantiable

Class

- Specifies both state (attributes) and behavior (operations) of an object type
- Instantiable
Inheritance in the Object Model of ODMG

- Behavior inheritance
  - Also known as IS-A or interface inheritance
  - Specified by the colon (:) notation

- EXTENDS inheritance
  - Specified by keyword `extends`
  - Inherit both state and behavior strictly among classes
  - Multiple inheritance via extends not permitted
Built-in Interfaces and Classes in the Object Model

- **Collection objects**
  - Inherit the basic Collection interface
  - \( i = o\.create\_iterator() \)
    - Creates an iterator object for the collection
    - To loop over each object in a collection
- Collection objects further specialized into:
  - set, list, bag, array, and dictionary
Figure 12.6  Inheritance hierarchy for the built-in interfaces of the object model.
Atomic (User-Defined) Objects

- Specified using keyword `class` in ODL

- **Attribute**
  - Property; describes data in an object

- **Relationship**
  - Specifies inter-object references
  - Keyword `inverse`
    - Single conceptual relationship in inverse directions

- **Operation signature**: 
  - Operation name, argument types, return value
Figure 12.7 The attributes, relationships, and operations in a class definition.

class EMPLOYEE
{
    extent ALL_EMPLOYEES
    key Ssn

    
    attribute string Name;
    attribute string Ssn;
    attribute date Birth_date;
    attribute enum Gender{M, F} Sex;
    attribute short Age;
    relationship DEPARTMENT Works_for
        inverse DEPARTMENT::Has_emps;
    void reassign_emp(in string New_dname)
        raises(dname_not_valid);
};

class DEPARTMENT
{
    extent ALL_DEPARTMENTS
    key Dname, Dnumber

    
    attribute string Dname;
    attribute short Dnumber;
    attribute struct Dept_mgr {EMPLOYEE Manager, date Start_date}
        Mgr;
    attribute set<string> Locations;
    attribute struct Projs {string Proj_name, time Weekly_hours}
        Projs;
    relationship set<EMPLOYEE> Has_emps inverse EMPLOYEE::Works_for;
    void add_emp(in string New_ename)
        raises(ename_not_valid);
    void change_manager(in string New_mgr_name; in date
        Start_date);
};
Extents, Keys, and Factory Objects

- **Extent**
  - A persistent named collection object that contains all persistent objects of class

- **Key**
  - One or more properties whose values are unique for each object in extent of a class

- **Factory object**
  - Used to generate or create individual objects via its operations
Object Definition Language ODL

- Support semantic constructs of ODMG object model
- Independent of any particular programming language
- Example on next slides of a UNIVERSITY database
- Graphical diagrammatic notation is a variation of EER diagrams
Figure 12.9a  An example of a database schema. Graphical notation for representing ODL schemas.
Figure 12.9b An example of a database schema. A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).
Figure 12.10  Possible ODL schema for the UNIVERSITY database in Figure 12.9(b).

```
class PERSON
  (extent PERSONS)
  (key Ssn)
  {attribute struct Pname { string Fname, string Mname, string Lname } Name;}
  attribute string Ssn;
  attribute date Birth_date;
  attribute enum Gender[M, F] Sex;
  attribute struct Address { short No, string Street, short Apt_no, string City, string State, short Zip } Address;
  short Age();
}

class FACULTY extends PERSON
  (extent FACULTY)
  {attribute string Rank;}
  attribute float Salary;
  attribute string Office;
  attribute string Phone;
  relationship DEPARTMENT Works_in inverse DEPARTMENT::Has_faculty;
  relationship set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
  relationship set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
  void give_raise(in float raise);
  void promote(in string new_rank);}

class GRADE
  (extent GRADES)
  {attribute enum GradeValues[A,B,C,D,F,i] Grade;}
  relationship SECTION Section inverse SECTION::Students;
  relationship STUDENT Student inverse STUDENT::Completed_sections;}

class STUDENT extends PERSON
  (extent STUDENTS)
  {attribute string Class;
  attribute string Department, Minors_in;
  relationship Department Majors_in inverse DEPARTMENT::Has_majors;
  relationship set<GRADE> Completed_sections inverse GRADE::Student;
  relationship set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
  void change_major(in string dname) raises(dname_not_valid);
  void assign_grade(in short secno, in GradeValue grade) raises(section_not_valid, grade_not_valid);}
```
Possible ODL schema for the UNIVERSITY database in Figure 12.9(b).
Interface Inheritance in ODL

- Next example illustrates interface inheritance in ODL
Figure 12.11a An illustration of interface inheritance via ":". Graphical schema representation.

(a)

![Diagram showing interface inheritance with GeometryObject as the root and branches to Rectangle, Triangle, Circle, and more.](image-url)

continued on next slide
Figure 12.11b  An illustration of interface inheritance via "::". Corresponding interface and class definitions in ODL.

```java
(b) interface GeometryObject
{
    attribute enum Shape{RECTANGLE, TRIANGLE, CIRCLE, ...}
    Shape;
    attribute struct Point {short x, short y} Reference_point;
    float perimeter();
    float area();
    void translate(in short x_translation; in short y_translation);
    void rotate(in float angle_of_rotation); }

class RECTANGLE : GeometryObject
{
    extent RECTANGLES
    {
        attribute struct Point {short x, short y} Reference_point;
        attribute short Length;
        attribute short Height;
        attribute float Orientation_angle;
    }

class TRIANGLE : GeometryObject
{
    extent TRIANGLES
    {
        attribute struct Point {short x, short y} Reference_point;
        attribute short Side_1;
        attribute short Side_2;
        attribute float Side1_side2_angle;
        attribute float Side1_orientation_angle;
    }

class CIRCLE : GeometryObject
{
    extent CIRCLES
    {
        attribute struct Point {short x, short y} Reference_point;
        attribute short Radius;
    }
...
```
Object Database Conceptual Design

- Differences between conceptual design of ODB and RDB, handling of:
  - Relationships
  - Inheritance
- Philosophical difference between relational model and object model of data
  - In terms of behavioral specification
Mapping an EER Schema to an ODB Schema

- Create ODL class for each EER entity type
- Add relationship properties for each binary relationship
- Include appropriate operations for each class
- ODL class that corresponds to a subclass in the EER schema
  - Inherits type and methods of its superclass in ODL schema
Mapping an EER Schema to an ODB Schema (cont’d.)

- Weak entity types
  - Mapped same as regular entity types
- Categories (union types)
  - Difficult to map to ODL
- An $n$-ary relationship with degree $n > 2$
  - Map into a separate class, with appropriate references to each participating class
The Object Query Language OQL

- Query language proposed for ODMG object model
- Simple OQL queries, database entry points, and iterator variables
  - Syntax: select ... from ... where ... structure
  - Entry point: named persistent object
  - Iterator variable: define whenever a collection is referenced in an OQL query
Query Results and Path Expressions

- Result of a query
  - Any type that can be expressed in ODMG object model

- OQL orthogonal with respect to specifying path expressions
  - Attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions
Other Features of OQL

- **Named query**
  - Specify identifier of named query
- **OQL query will return collection as its result**
  - If user requires that a query only return a single element use `element` operator
- **Aggregate operators**
- **Membership and quantification over a collection**
Other Features of OQL (cont’d.)

- Special operations for ordered collections
- **Group by** clause in OQL
  - Similar to the corresponding clause in SQL
  - Provides explicit reference to the collection of objects within each group or **partition**
- **Having clause**
  - Used to filter partitioned sets
Overview of the C++ Language Binding in the ODMG Standard

- Specifies how ODL constructs are mapped to C++ constructs
- Uses prefix `d_` for class declarations that deal with database concepts
- Template classes
  - Specified in library binding
  - Overloads operation new so that it can be used to create either persistent or transient objects
Summary

- Overview of concepts utilized in object databases
  - Object identity and identifiers; encapsulation of operations; inheritance; complex structure of objects through nesting of type constructors; and how objects are made persistent
- Description of the ODMG object model and object query language (OQL)
- Overview of the C++ language binding