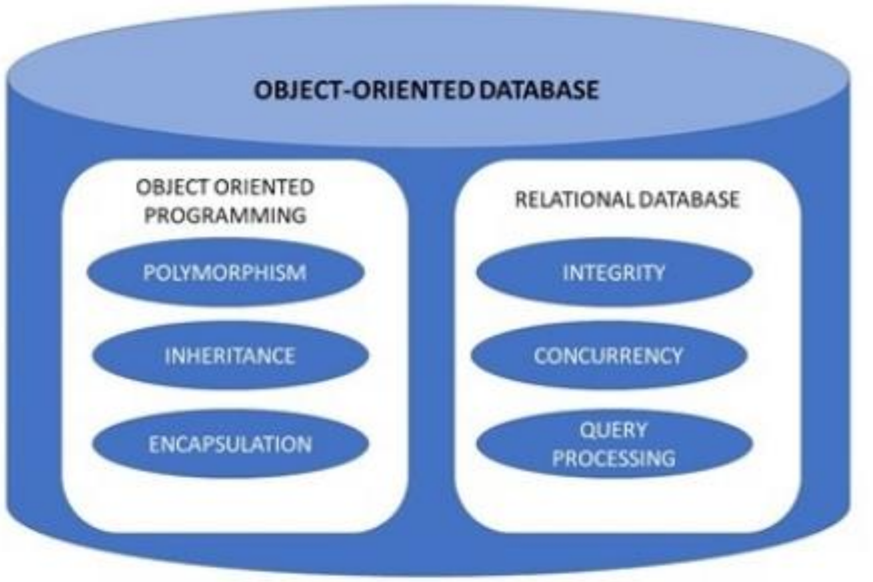
Object Oriented Databases

Overview of Object Oriented Concepts

An object-oriented database is a collection of object-oriented programming and relational database. There are various items which are created using object-oriented programming languages like C++, Java which can be stored in relational databases, but object-oriented databases are well-suited for those items.

An object-oriented database is organized around objects rather than actions, and data rather than logic. For example, a multimedia record in a relational database can be a definable data object, as opposed to an alphanumeric value.



Advantages and disadvantages of the object-oriented database model

The choice of database type heavily depends on the individual application. When working with **object-oriented programming languages**, like Java for example, an object database is advantageous. The objects of the source code can simply be incorporated into the database. If we turn to a relational database, which is fairly common, complex objects are more difficult to integrate into the table construct.

One disadvantage of OODBs is **poor adoption among the community**. Although the model has been established since the 1980s, up to now, very few database management systems have taken to object databases. The community that works with the model is correspondingly small. Most developers, therefore, prefer to turn to the more widely spread, well-documented and highly developed relational databases.

Although complexity of OODBs is one of its advantages, it can present a few disadvantages in certain situations. The complexity of the objects means that complex queries and operations can be undertaken much more quickly than in relational models. However, if the operations are simple, the complex structure still has to be used resulting in a **loss of speed**.

Advantages	Disadvantages
Complex data sets can be saved and retrieved quickly and easily.	Object databases are not widely adopted.
Object IDs are assigned automatically.	In some situations, the high complexity can cause performance problems.
Works well with object-oriented programming languages.	

Object Identity

Object Structure and Type constructions

• OODBS concepts have their origin in object-oriented programming languages (abstract data types, encapsulation, information hiding, methods, class/type hierarchies, inheritance . . .)

• Object =ˆ state (value) and behavior (operations)

• In OODBs objects are persistent (comp. transient objects in object-oriented programming languages)

• From a DB design perspective, the object-oriented model is a logical model (like ER model)

• Essentially, an object corresponds to an entity in the ER model (but many issues are missing in the ER model, e.g., methods)

Object-Identity

• OO database systems provide unique identifier (OID) for each object stored in DB; used to manage inter-object references

• Object retains identifier even if some or all of the values of variables or definitions of methods change, i.e., OIDs are immutable.

• OIDs are not based on physical representation/storage of object (i.e., = ROWID or TID)

• Object-identity is a stronger notion of identity than in programming languages or data models not based on objectorientation (❀ concept does not apply to tuples of a relational database)

• Some OO models require that everything from simple value to complex object has an OID.

• Object identifiers are used to uniquely identify objects – stored as a field of an object – Used to refer to another object, e.g., the spouse field of a person object may be the identifier of another person object – can be system generated (built-in; created by the DBS) or external (user-supplied)

Object Structure In OODBS

The state (current value) of a complex object may be constructed from other objects (or other values) using type constructors.

• Basic type constructors are atom, tuple, and set; can also include list, bag, and array.

• Formal representation of an object as a triple (i, c, v), with i being OID, c a type constructor, and v is the object state.

• Examples:

o0 = (i0, atom, 815)

o1 = (i1, atom, UC Davis)

o2 = (i2, atom, Computer Science)

o3 = (i3, atom, Art)

o4 = (i4, set, {i1, i2})

o5 = (i5, tuple,

University: i1, Major: i2)

• Model allows arbitrary nesting of constructors (e.g., bag of lists of sets of numbers)

• Object can be represented as graph with each object oi being a node labeled with OID ii and type constructor ci. If object value is constructed, graph includes arc from object node to node representing constructed value. For each atomic value, graph includes direct arc from object node to node representing the value.

• Graph representation allows to compare states of two objects for equality: – identical states (deep equality): graphs must be identical, including OIDs at every level – equal states (shallow equality): graph structure must be the same, but some nodes may have different OIDs

Example of shallow equality:

o1 = (i1, tuple,

---

a1 : i4, a2 : i6)

o2 = (i2, tuple,

---

a1 : i5, a2 : i6)

Type Constructors

Object Definition Language (ODL) that includes type constructors can be used to define object types for particular DB application.

Type constructors in ODL are used to define data structures for OO database schema.

Example (using a generic notation):

define type Employee tuple(fname: string; lname: string; salary: float; hiredate: Date; supervisor: Employee; dept: Department);

define type Date tuple(year: integer; month: integer; day: integer);

d efine type Department tuple(dname: string; mgr: tuple (manager: Employee; startdate: Date); locations: set(string); employees: set(Employee));

Encapsulation of Operations, Method and Persistence

The concept of *encapsulation* is one of the main characteristics of OO languages and systems. It is also related to the concepts of *abstract data types* and *information hiding* in programming languages. In traditional database models and systems this concept was not applied, since it is customary to make the structure of database objects visible to users and external programs. In these traditional models, a number of generic database operations are applicable to objects *of all types*. For example, in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to *any relation* in the database.

The relation and its attributes are visible to users and to external programs that access the relation by using these operations. The concepts of encapsulation is applied to database objects in ODBs by defining the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations.

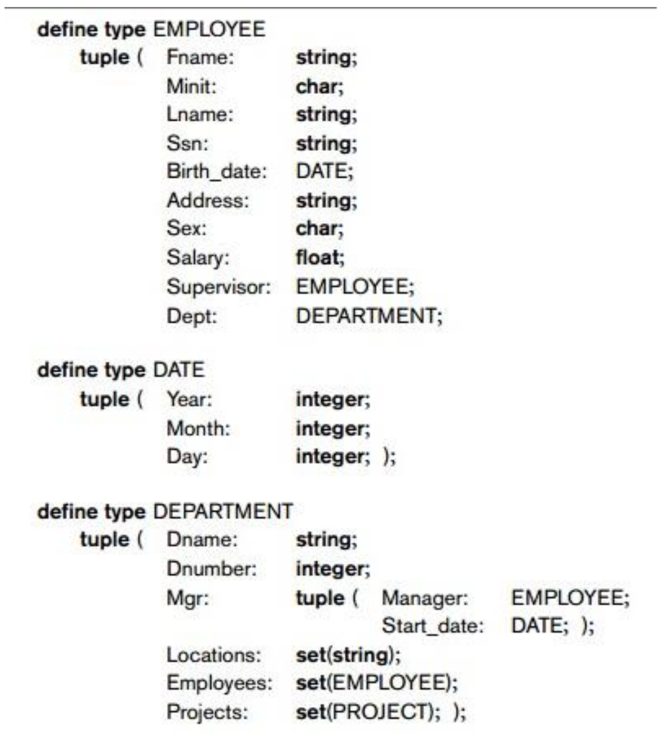
Still other operations may perform a combination of retrieval, calculation, and update. In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.

The external users of the object are only made aware of the **interface** of the operations, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of any hidden internal data structures of the object and the implementation of the operations that access these structures. The interface part of an operation is sometimes called the **signature**, and the operation implementation is sometimes called the **method**.

For database applications, the requirement that all objects be completely encapsulated is too stringent. One way to relax this requirement is to divide the structure of an object into **visible** and **hidden** attributes (instance variables). Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language.

The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most ODMSs employ high-level query languages for accessing visible attributes.

The term **class** is often used to refer to a type definition, along with the definitions of the operations for that type. Figure 2 shows how the type definitions in Figure 1 can be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition.

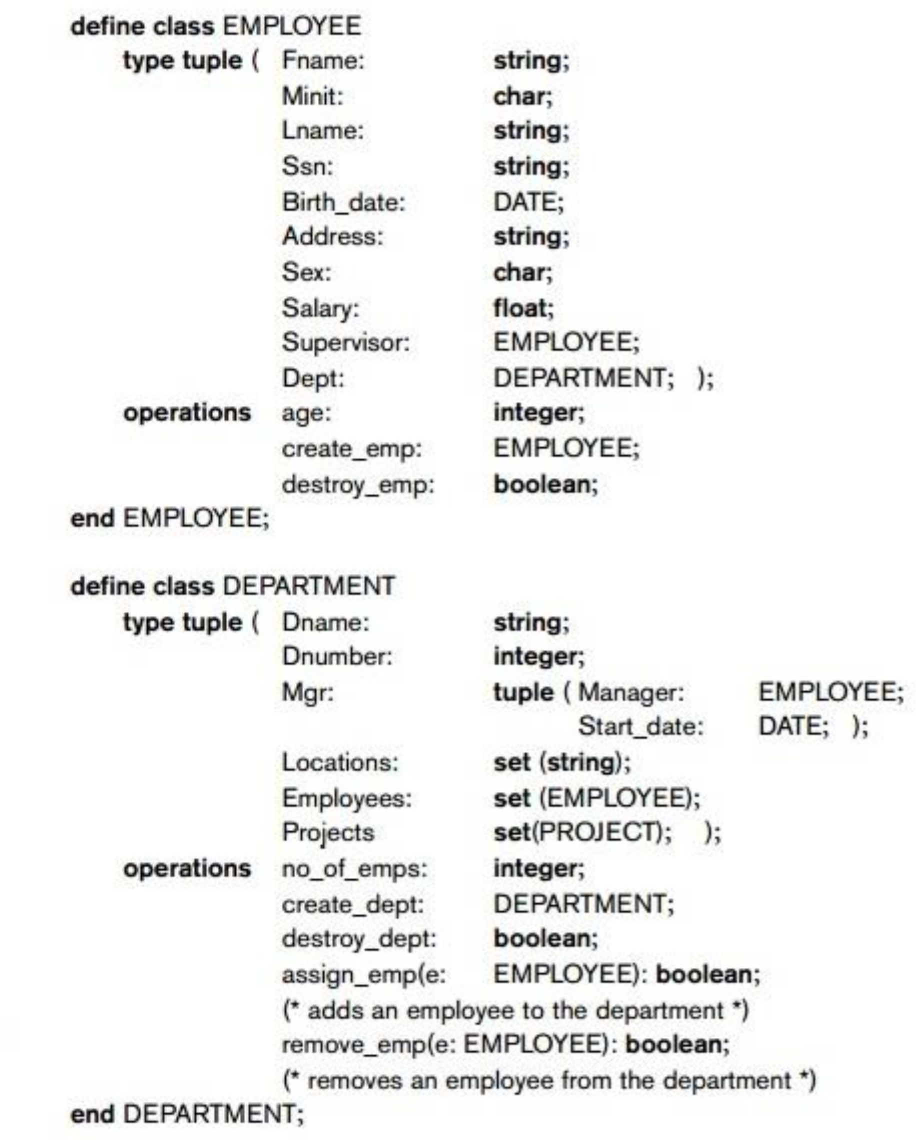


```
define type EMPLOYEE
    tuple (  Fname:        string;
             Minit:        char;
             Lname:        string;
             Ssn:          string;
             Birth_date:   DATE;
             Address:      string;
             Sex:          char;
             Salary:       float;
             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 1



```
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;
```

Figure 2

A method (implementation) for each operation must be defined elsewhere using a programming language. Typical operations include the **object constructor** operation (often called *new*), which is used to create a new object, and the **destructor** operation, which is used to destroy (delete) an object. A number of **object modifier** operations can also be declared to modify the states (values) of various attributes of an object. Additional operations can **retrieve** information about the object.

An operation is typically applied to an object by using the **dot notation**.

For exam-ple, if d is a reference to a DEPARTMENT object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted). The only exception is the constructor opera-tion, which returns a reference to a new DEPARTMENT object. Hence, it is customary in some OO models to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure 2. The dot nota-tion is also used to refer to attributes of an object—for example, by writing d.Dnumber or d.Mgr_Start_date.

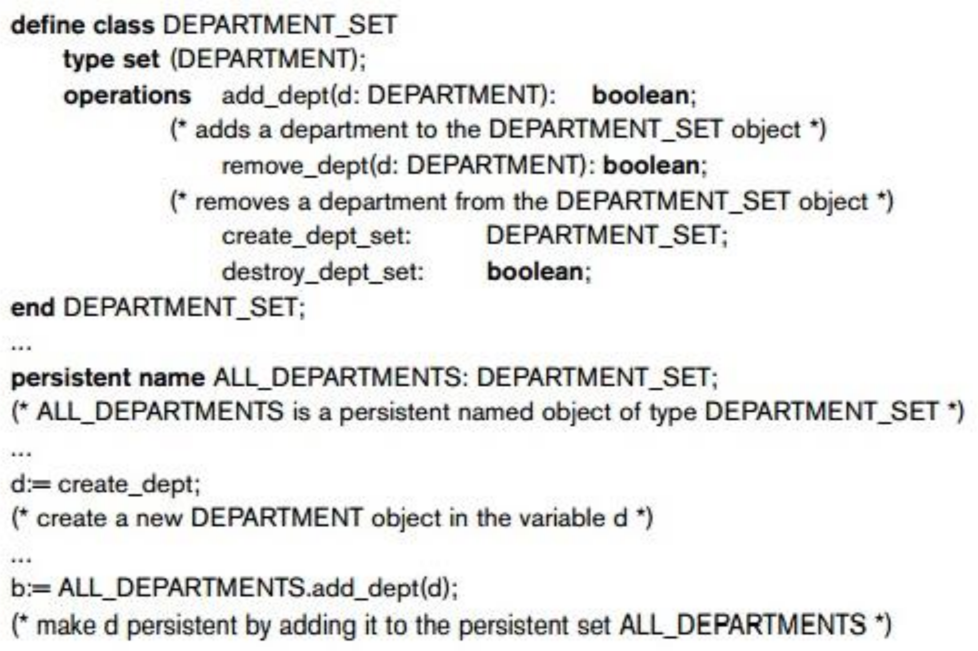
Specifying Object Persistence via Naming and Reachability. An ODBS is often closely coupled with an object-oriented programming language (OOPL). The OOPL is used to specify the method (operation) implementations as well as other application code. Not all objects are meant to be stored permanently in the data-base. **Transient objects** exist in the executing program and disappear once the pro-gram terminates. **Persistent objects** are stored in the database and persist after program termination. The typical mechanisms for making an object persistent are *naming* and *reachability.*

The **naming mechanism** involves giving an object a unique persistent name within a particular database. This persistent **object name** can be given via a specific statement or operation in the program, as shown in Figure 3. The named persistent objects are used as **entry points** to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability**. The reachability mechanism works by making the object reachable from some other persistent object. An object *B* is said to be **reachable** from an object *A* if a sequence of references in the database lead from object *A* to object *B*.

If we first create a named persistent object *N*, whose state is a *set* (or possibly a *bag*) of objects of some class *C*, we can make objects of *C* persistent by *adding them* to the set, thus making them reachable from *N*. Hence, *N* is a named object that defines a **persistent collection** of objects of class *C*. In the object model standard, *N* is called the **extent** of *C*

For example, we can define a class DEPARTMENT_SET (see Figure 3) whose objects are of type set(DEPARTMENT).14 We can create an object of type DEPARTMENT_SET, and give it a persistent name ALL_DEPARTMENTS, as shown in Figure 11.3. Any DEPARTMENT object that is added to the set of ALL_DEPARTMENTS by using the add_dept operation becomes persistent by virtue of its being reachable from ALL_DEPARTMENTS.

Notice the difference between traditional database models and ODBs in this respect.



```
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 *)
```

Figure 3

In traditional database models, such as the relational model, *all* objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the *type declaration* for EMPLOYEE and a *persistent set* of *all* EMPLOYEE records (tuples). In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) or bag(EMPLOYEE) whose value is the *collection of references* (OIDs) to all persistent EMPLOYEE objects, if this is desired, as shown in Figure 3. This allows transient and persistent objects to follow the same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

Type hierarchies and Inheritance

Type Extents and Queries Complex Objects

Database schema design for OODBMS

OQL

Object Query Language (**OQL**) is a query language standard for object-oriented databases modeled after SQL. OQL was developed by the Object Data Management Group (ODMG). Because of its overall complexity nobody has ever fully implemented the complete OQL. OQL has influenced the design of some of the newer query languages like JDOQL and EJB QL, but they can't be considered as different flavors of OQL.

General rules

The following rules apply to OQL statements:

- All complete statements must be terminated by a semi-colon.
- A list of entries in OQL is usually separated by commas but not terminated by a comma(,).
- Strings of text are enclosed by matching quotation marks.

Examples

The following example illustrates how one might retrieve the CPU-speed of all PCs with more than 64MB of RAM from a fictional PC database:

SELECT pc.cpuspeed

FROM PCs pc

WHERE pc.ram > 64;

Query with grouping and aggregation

The following example illustrates how one might retrieve the average amount of RAM on a PC, grouped by manufacturer:

SELECT manufacturer, **AVG**(**SELECT** part.pc.ram **FROM** partition part)

FROM PCs pc

GROUP BY manufacturer: pc.manufacturer;

Note the use of the keyword partition, as opposed to aggregation in traditional SQL

Persistent Programming Language

Programming languages that natively and seamlessly allow objects to continue existing after the program has been closed down are called **persistent programming languages**. JADE is one such language.

A persistent programming language is a programming language extended with constructs to handle persistent data. It is distinguished from embedded SQL in at least two ways:

In a persistent programming language:

- The query language is fully integrated with the host language and both share the same type system.
- Any format changes required between the host language and the database are carried out transparently.

In Embedded SQL:

- Where the host language and data manipulation language have different type systems, code conversion operates outside of the OO type system, and hence has a higher chance of having undetected errors.
- Format conversion must be handled explicitly and takes a substantial amount of code.

Using Embedded SQL, a programmer is responsible for writing explicit code to fetch data into memory or store data back to the database. In a persistent programming language, a programmer can manipulate persistent data without having to write such code explicitly.

The drawbacks of persistent programming languages include:

- While they are powerful, it is easy to make programming errors that damage the database.
- It is harder to do automatic high-level optimization.
- They do not support declarative querying well.

OODBMS architecture and Storage Issues

Example of ODBMS

In OOP, an entity is represented as an object and objects are stored in memory. Objects have members such as fields, properties, and methods. Objects also have a life cycle that includes the creation of an object, use of an object, and deletion of an object. OOP has key characteristics, encapsulation, inheritance, and polymorphism.

Today, there are many popular OOP languages such as C++, Java, C#, Ruby, Python, JavaScript, and Perl.

The idea of **object databases** was originated in 1985 and today has become common for various common OOP languages, such as C++, Java, C#, Smalltalk, and LISP. Common examples are Smalltalk is used in GemStone, LISP is used in Gbase, and COP is used in Vbase.

Object databases are commonly used in applications that require high performance, calculations, and faster results. Some of the common applications that use object databases are real-time systems, architectural & engineering for 3D modeling, telecommunications, and scientific products, molecular science, and astronomy