Traditional Data Models

- Traditional Data Models:
  - Hierarchical
  - Network (since mid-60's)
  - Relational (since 1970 and commercially since 1982)
- These traditional models have been quite successful in developing the database technology required for many traditional business database applications.

RELATIONAL MODEL

- Relations are the key concept, everything else is around relations
- Primitive data types, e.g., strings, integer, date, etc.
- Great normalization, query optimization, and theory
- What is missing?
  - Handling of complex objects
  - Handling of complex data types
  - Code is not coupled with data
  - No inheritance, encapsulation, etc.

Object-Oriented Database

- Relational model has certain limitations when more complex database applications must be designed and implemented, like CAD/CAM, CIM, scientific experiments, telecommunications, GIS, and multimedia.
- Object Oriented (OO) Data Models since mid-90’s, to address the limitations of traditional relational database models.
- The key feature of object-oriented databases is the power they give the designer to specify both the structure of complex objects and the operations that can be applied to these objects.

Why Object-Oriented Databases?

- Reasons for creation of Object Oriented Databases
  - Need for more complex applications
  - Need for additional data modeling features
  - Increased use of object-oriented programming languages
- Commercial OO Database products -
  - Several in the 1990's, but did not make much impact on mainstream data management

Object-Orientation Concepts

- Class
- Object
- Abstraction
- Encapsulation
- Interface
- Methods and their Signatures
- Attributes
- Object State

History of OO Models and Systems

- Languages:
  - Simula (1960’s)
  - Smalltalk (1970’s)
  - C++ (late 1980’s)
  - Java (1990’s and 2000’s)

History of OO Models and Systems (contd.)

- Experimental Systems:
  - Orion at MCC (Microelectronics and Computer Technology Corporation) in Austin.
  - IRIS at H-P labs
  - Open-OODB at T.I.
  - ODE at ATT Bell labs
  - Postgres - Montage - Illustra at UC/B
  - Encore/ObServer at Brown

History of OO Models and Systems (contd.)

- Commercial OO Database products:
  - Ontos
  - Gemstone
  - O2
  - Objectivity
  - Objectstore
  - Versant
  - Poet
  - Jasmine (Fujitsu - GM)

Object-Oriented Database

- ODBMS is a database management system that implements object-oriented data model
- The Object-oriented Database System Manifesto, *Atkinson et all, 1989*. – in scientific paper described the properties which ODBMS must satisfy:
  - Object-oriented concepts
  - Database management system concepts
- Implementation of object-oriented database management system (ODBMS) is generally programmed for a specific programming language, and differ quite with each other.

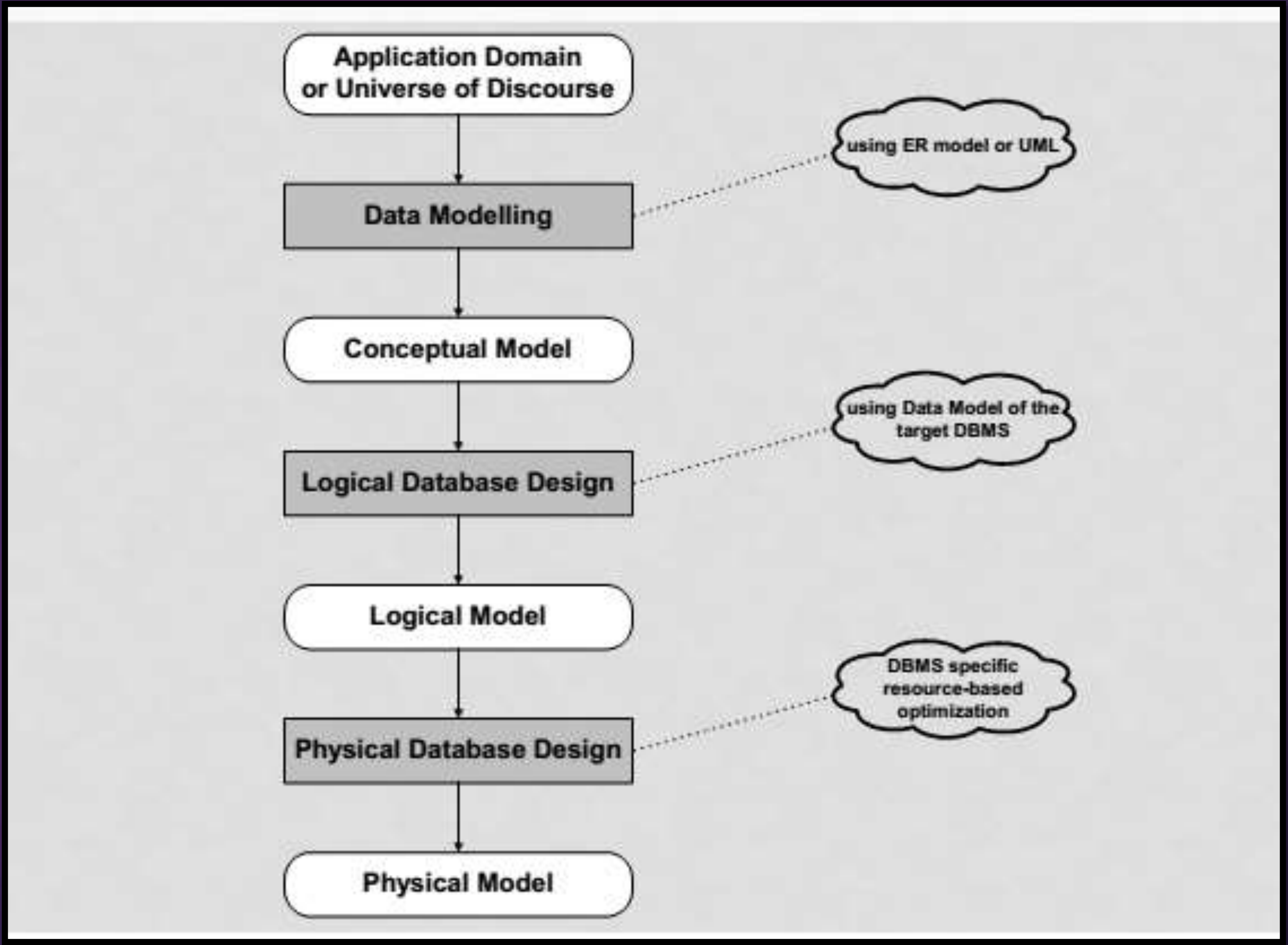
Limitations of ODBMS

- No logical data independence : Modifications to the database (schema evolution) require changes to the application and vice versa
- Lack of agreed standards, the existing standard (ODMG) is not fully implemented
- Dependence on a single programming language. Typical OODBMS is tied to a single programming language with it's programming interface
- Lack of interoperability with a large number of tools and features that are used in SQL
- Lack of Ad-Hoc queries (queries on the new tables that are obtained by joining new tables with the existing ones)

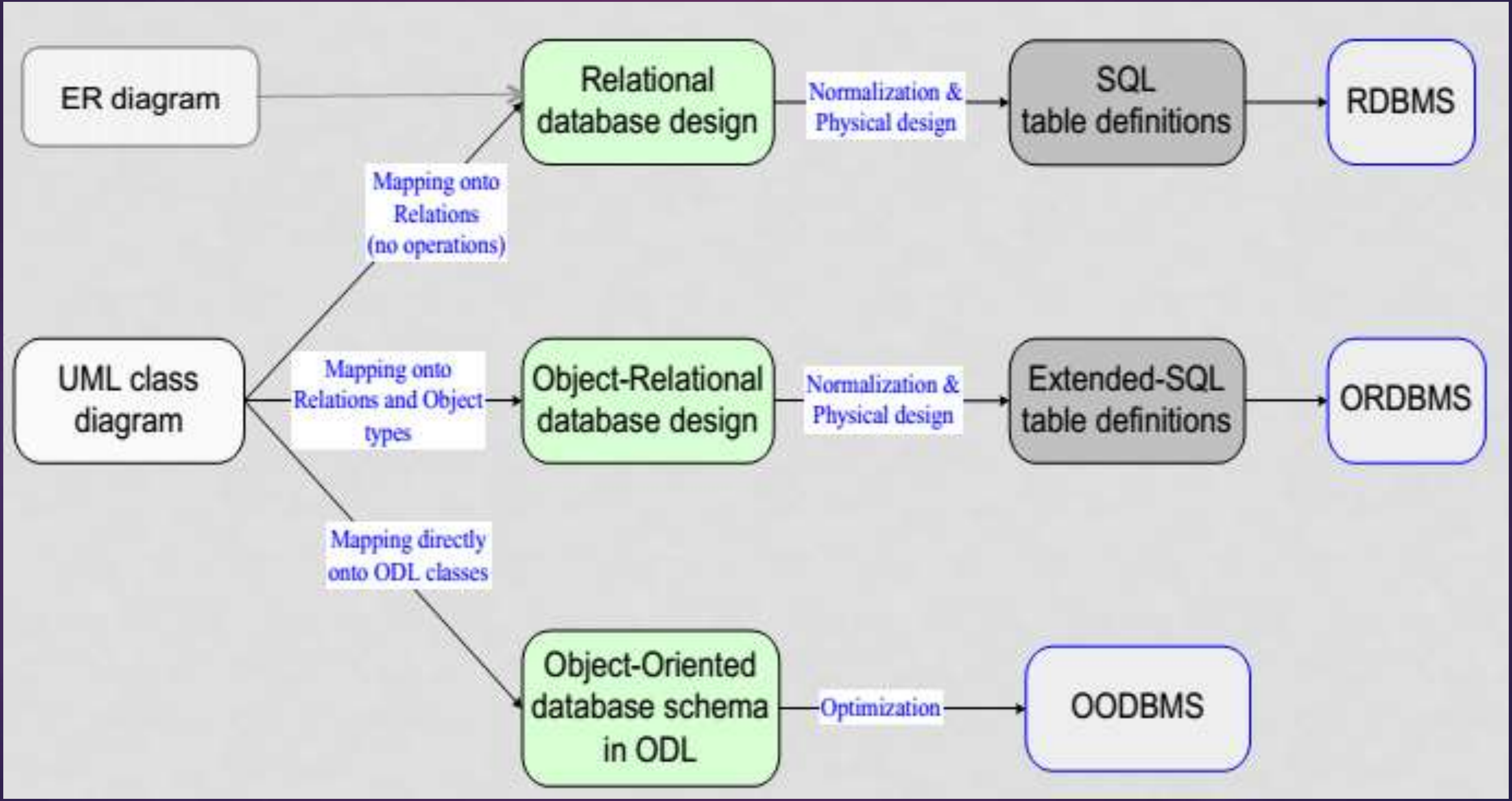
ODBMS in real world

- Chicago Stock Exchange - management in stocks trade (Versant)
- Radio Computing Services – automation of radio stations (POET)
- Ajou University Medical Canter in South Korea – all functions of the hospital, including those critical such as pathology, laboratory, blood bank, pharmacy and radiology
- CERN – big scientific data sets (Objectivity/DB)
- Federal Aviation Authority – simulation of passengers and baggage
- Electricite de France – management in electric power networks

Database Design Process



LOGICAL & PHYSICAL LAYERS


ER diagram
Relational database design
Normalization & Physical design
SQL table definitions
RDBMS
Mapping onto Relations (no operations)
UML class diagram
Mapping onto Relations and Object types
Object-Relational database design
Normalization & Physical design
Extended-SQL table definitions
ORDBMS
Mapping directly onto ODL classes
Object-Oriented database schema in ODL
Optimization
OODBMS

What is Object-Oriented Data Model?

- OO databases try to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon.
- Relations are not the central concept, classes and objects are the main concept.
- Main Features:
  - Powerful type system
  - Classes
  - Object Identity
  - Inheritance

FEATURE 1: POWERFUL TYPE SYSTEM

- **Primitive types :** Integer, string, date, Boolean, float, etc.
- **Structure type** : Attribute can be a record with a schema, struct { integer x, string y}
- **Collection type** : Attribute can be a Set, Bag, List, Array of other types
- **Reference type**: Attribute can be a Pointer to another object
- OODBMS are capable of storing complex objects, I.e., objects that are composed of other objects, and/or multi-valued attributes.

FEATURE 2: CLASSES

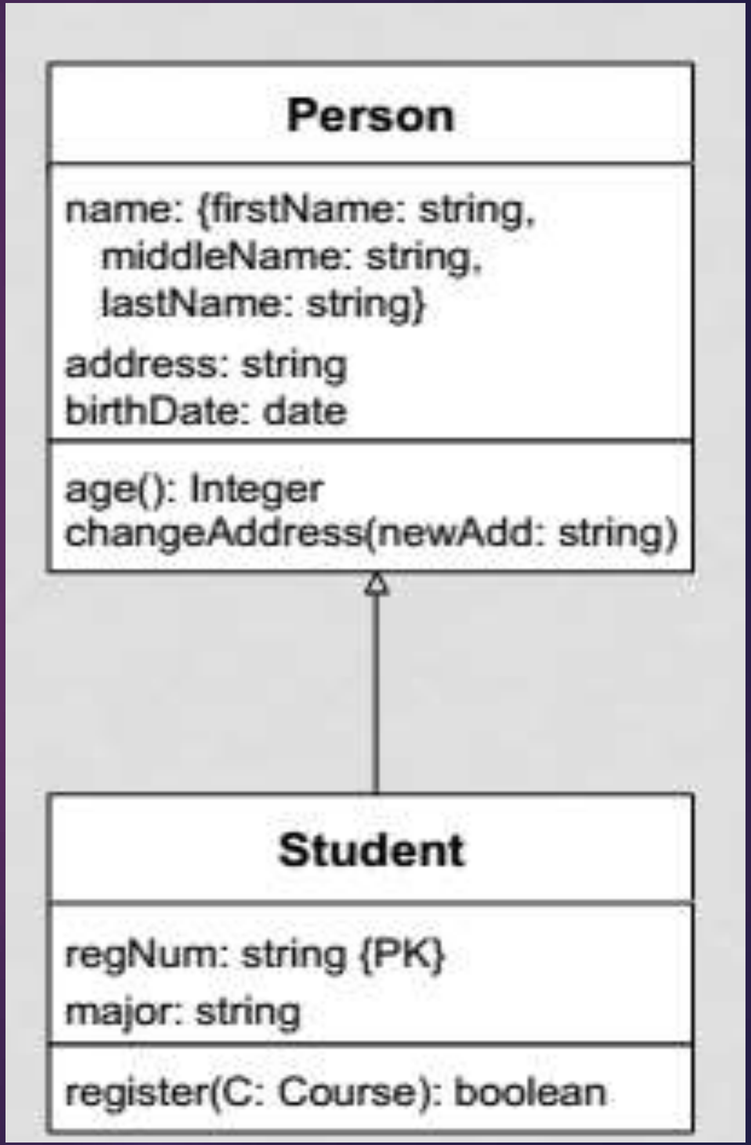
- A ‘class’ is in replacement of ‘relation’
- Same concept as in OO programming languages
  - All objects belonging to a same class share the same properties and behavior
- An ‘object’ can be thought of as ‘tuple’ (but richer content)
- An object is made of two things:
  - State: attributes (name, address, birthDate of a person)
  - Behavior: operations (age of a person is computed from birthDate and current date)
- Classes encapsulate data + methods + relationships
  - Unlike relations that contain data only
- In OODBMSs objects are persistent (unlike OO programming languages)

FEATURE 3: OBJECT IDENTITY

- OID is a unique, unchangeable object identifier generated by the OO system regardless of its content
- Independent of the values of the object attributes
- Invisible to the user
- Used for referencing objects
- Two objects are identical if they have the same identity of the object - property that uniquely identifies them
- Even if all attributes are the same, still objects have different OIDs

FEATURE 4: INHERITANCE

- A class can be defined in terms of another one.
- Person is super-class and Student is sub-class.
- Student class inherits attributes and operations of Person.



Encapsulation

- Related to the concepts of **abstract data types** and **information hiding** in programming languages
- The encapsulation is achieved through objects and its operations.
- Some OO models insist that all operations a user can apply to an object must be predefined. This forces a complete encapsulation of objects.
- To encourage **encapsulation**, an operation is defined in two parts:
  - **signature** or **interface** of the operation, specifies the operation name and arguments (or parameters).
  - **method** or **body**, specifies the implementation of the operation.

Encapsulation

- Operations can be invoked by passing a message to an object, which includes the **operation name** and the **parameters**. The object then executes the method for that operation.
- This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations.

Polymorphism

- This refers to an operation's ability to be applied to different types of objects; in such a situation, an operation name may refer to several distinct implementations, depending on the type of objects it is applied to.
- Operator overloading: It allows the same **operator name or symbol** to be bound to two or more **different implementations** of the operator, depending on the type of objects to which the operator is applied
  - For example + can be:
    - Addition in integers
    - Concatenation in strings (of characters)

Object Structure

- Every instance of an object is characterized by its state.
- Structure of an object instance is represented by a triple: (i, c, v)

where	i	:	object identifier
	c	:	type constructor
	v	:	object state or value

Type Constructors

- In OO databases, the state (current value) of a complex object may be constructed from other objects (or other values) by using certain type constructors.
- The three most basic constructors are **atom**, **tuple** and **set**.
- Other commonly used constructors include **list**, **bag**, and **array**.
- The atom constructor is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly.

Type Constructors

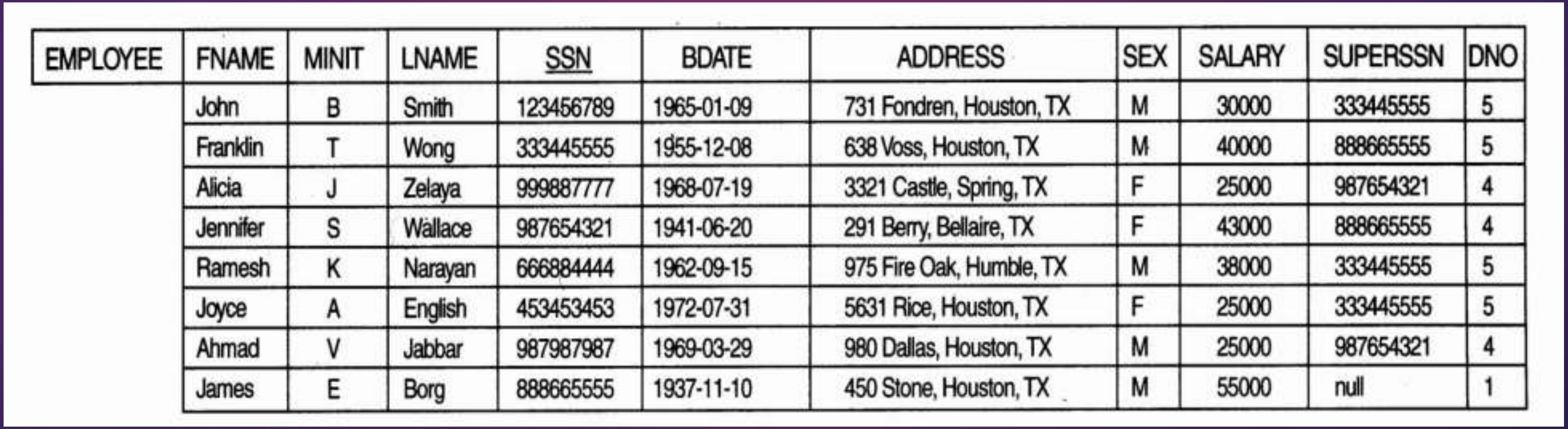
- The tuple type constructor is often called a structured type, since corresponds to struct construct in C and C++. It represents a tuple of the form $<a_1:i_1, a_2:i_2,..., a_n:i_n>$ where a_j is attribute name and each i_j is an OID.
- The set type constructor represents a set of OIDs { i1,i2,...,in}, which are the OIDs of objects of typically of same type.
- List type constructor represent the ordered list of OIDs of the same type.

Type Constructors

- Array type constructor is single dimension array of OIDs.
- The main difference between list and array is that a list can have an arbitrary number of element whereas the array has a maximum size.
- The bag is similar same as set but bag has duplicate elements.

Object Identity, Object Structure, and Type Constructors

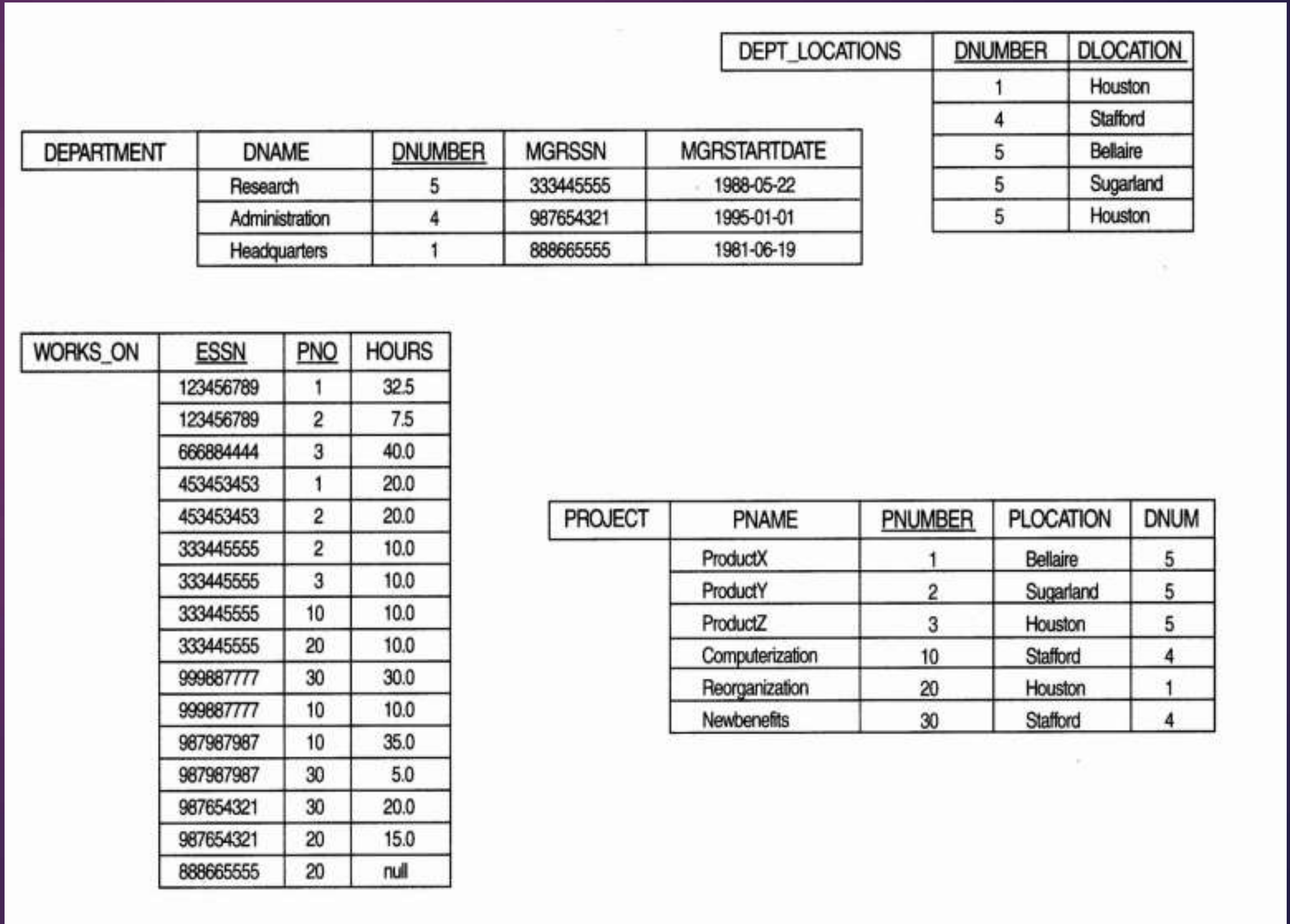
- Example 1
  - One possible relational database state corresponding to COMPANY schema



EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

Object Identity, Object Structure, and Type Constructors

- Example 1 (contd.):



DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston

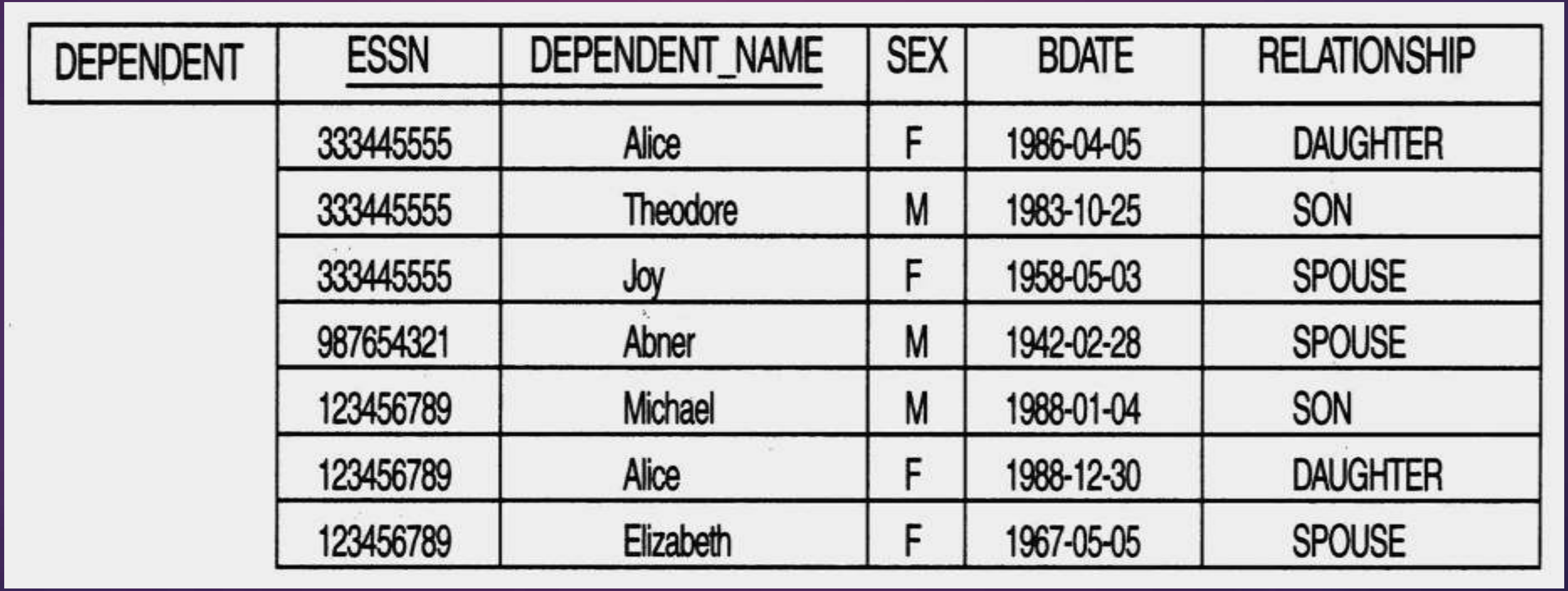
DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

Object Identity, Object Structure, and Type Constructors

- Example 1 (contd.)



DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

Object Identity, Object Structure, and Type Constructors

- Example 1 (contd.)
  We use i_1, i_2, i_3, . . . to stand for unique system-generated object identifiers. Consider the following objects:
    - o_1 = (i_1, atom, 'Houston')
    - o_2 = (i_2, atom, 'Bellaire')
    - o_3 = (i_3, atom, 'Sugarland')
    - o_4 = (i_4, atom, 5)
    - o_5 = (i_5, atom, 'Research')
    - o_6 = (i_6, atom, '1988-05-22')
    - o_7 = (i_7, set, {i1, i2, i3})

Object Identity, Object Structure, and Type Constructors

- Example 1(contd.)
  - o_8 = (i_8, tuple, <dname:i_5, dnumber:i_4, mgr:i_9, locations:i_7, employees:i_{10}, projects:i_{11}>)
  - o_9 = (i_9, tuple, <manager:$i_1$2, manager_start_date:i_6>)
  - o_{10} = (i_{10}, set, {i_{12}, i_{13}, i_{14}})
  - o_{11} = (i_{11}, set {i_{15}, i_{16}, i_{17}})
  - o_{12} = (i_{12}, tuple, <fname:i_{18}, minit:i_{19}, lname:i_{20}, ssn:i_{21}, . . ., salary:i_{26}, supervisor:i_{27}, dept:i_8>)
  - . . .

Object Identity, Object Structure, and Type Constructors

- Example 1 (contd.)
  - The first six objects listed in this example represent atomic values.
  - Object seven is a set-valued object that represents the set of locations for department 5; the set refers to the atomic objects with values {'Houston', 'Bellaire', 'Sugarland'}.
  - Object 8 is a tuple-valued object that represents department 5 itself, and has the attributes DNAME, DNUMBER, MGR, LOCATIONS, and so on.

Object Identity, Object Structure, and Type Constructors

- Example 2:
  - This example illustrates the difference between the two definitions for comparing object states for equality.
  - $o_1 = (i_1, \text{tuple}, <a_1:i_4, a_2:i_6>)$
  - $o_2 = (i_2, \text{tuple}, <a_1:i_5, a_2:i_6>)$
  - $o_3 = (i_3, \text{tuple}, <a_1:i_4, a_2:i_6>)$
  - $o_4 = (i_4, \text{atom}, 10)$
  - $o_5 = (i_5, \text{atom}, 10)$
  - $o_6 = (i_6, \text{atom}, 20)$

Object Identity, Object Structure, and Type Constructors

- Example 2 (contd.):
  - In this example, The objects o1 and o2 have equal states, since their states at the atomic level are the same but the values are reached through distinct objects o4 and o5.
  - However, the states of objects o1 and o3 are identical, even though the objects themselves are not because they have distinct OIDs.
  - Similarly, although the states of o4 and o5 are identical, the actual objects o4 and o5 are equal but not identical, because they have distinct OIDs.

Object Identity, Object Structure, and Type Constructors

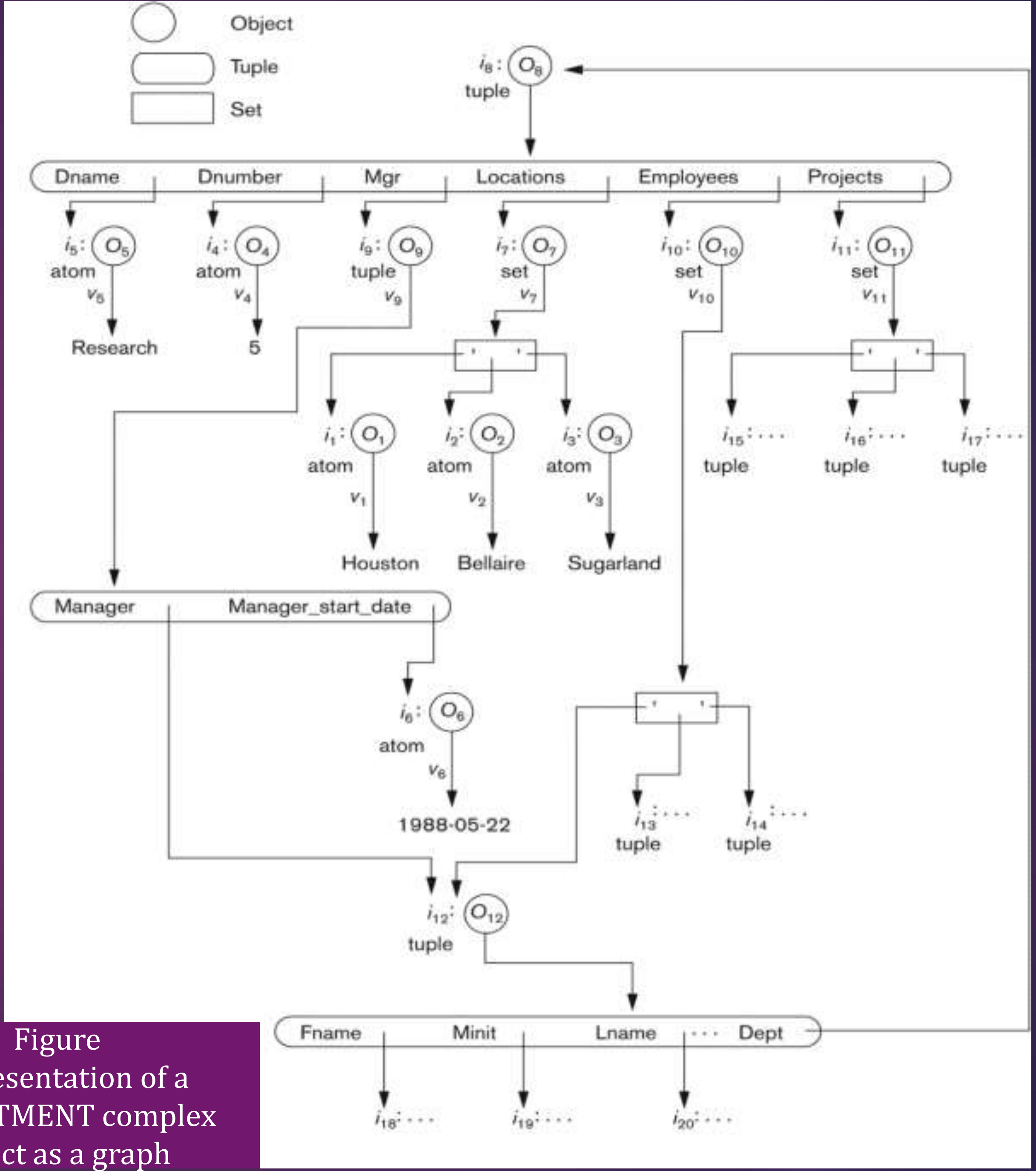
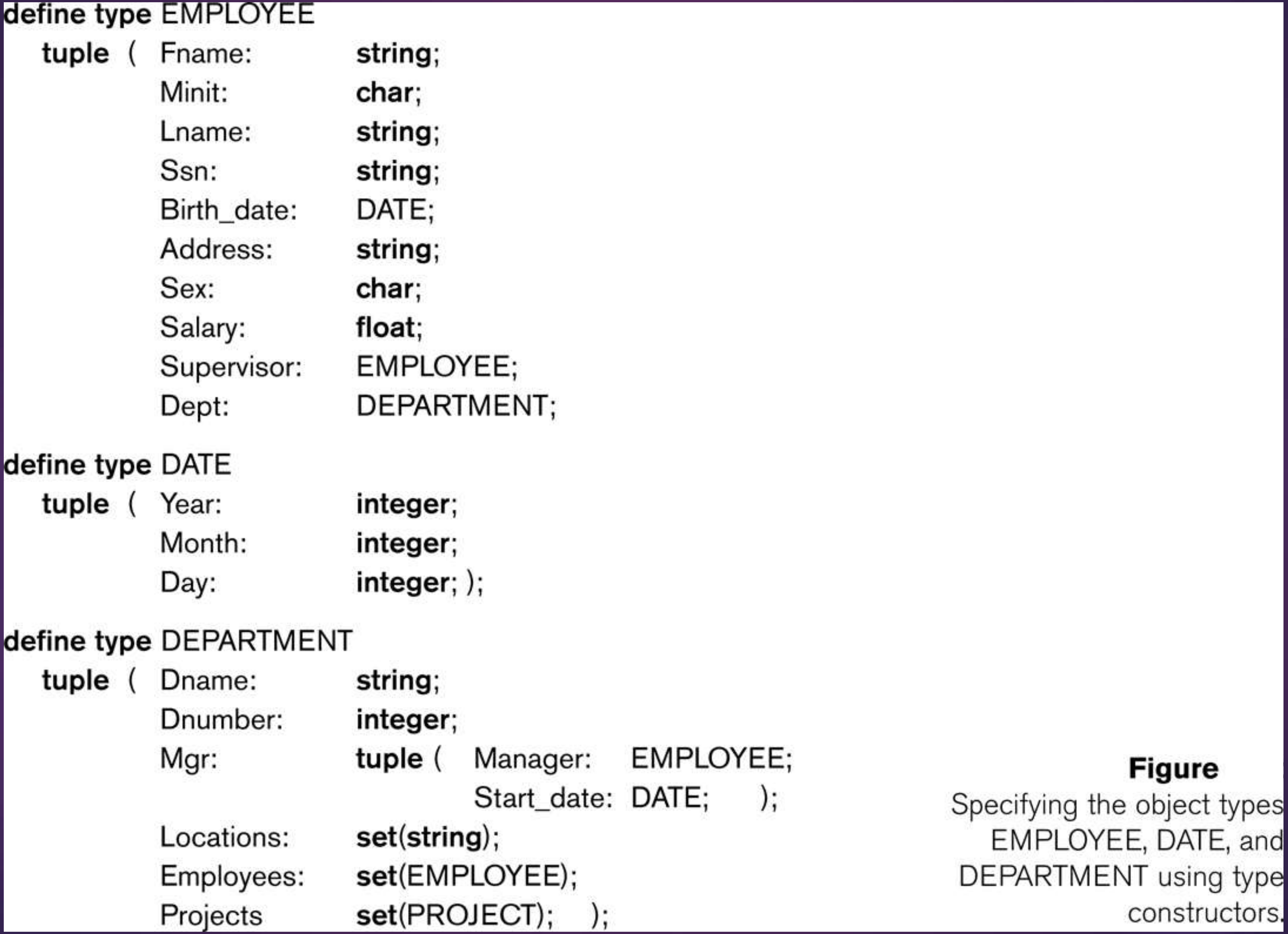


Figure
Representation of a DEPARTMENT complex object as a graph

Object Structure and Type Constructors



```
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
Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

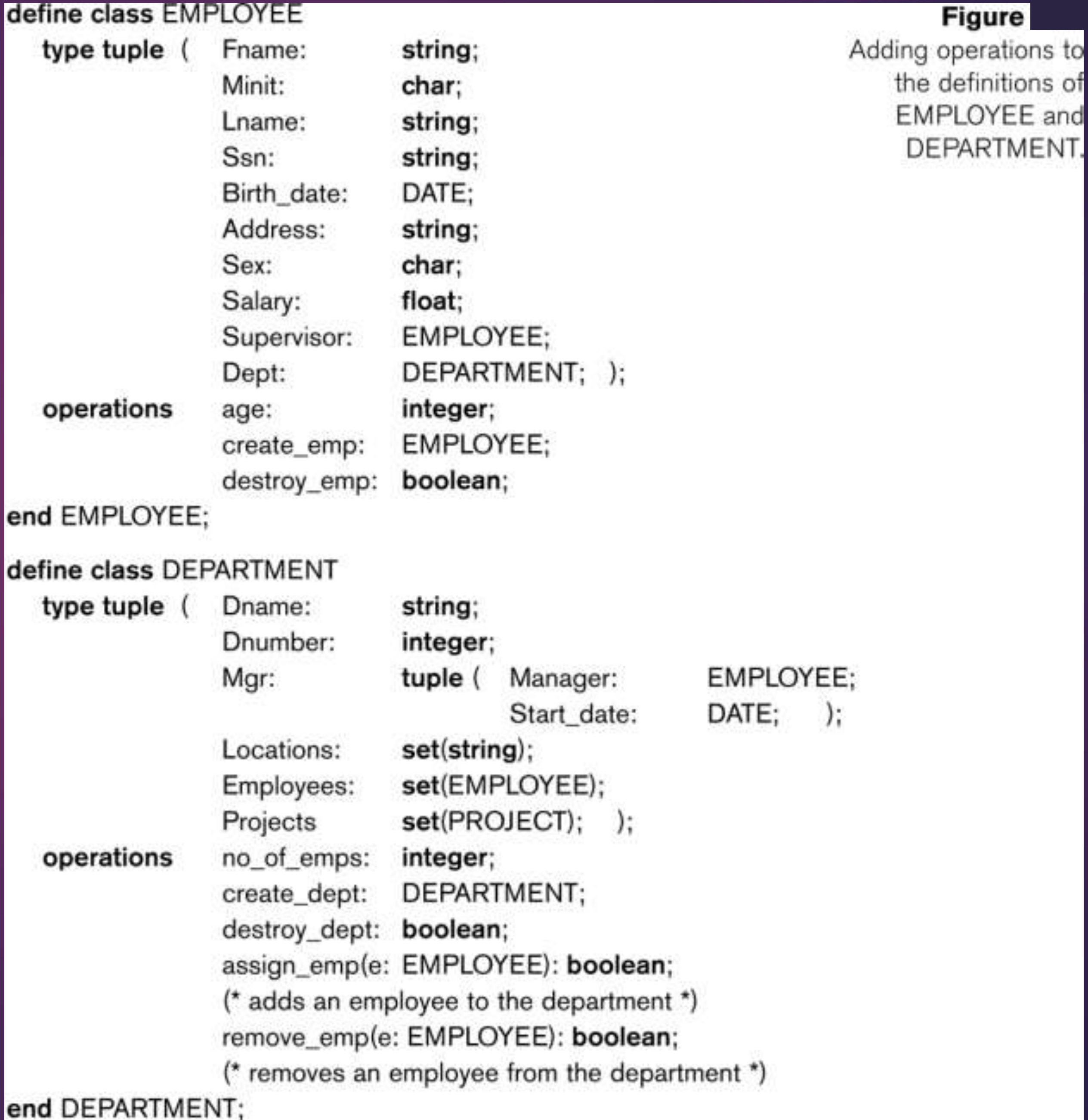
Specifying Object Behavior via Class Operations

- The main idea is to define the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type.
- 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.

Specifying Object Behavior via Class Operations (contd.):

- For database applications, the requirement that all objects be completely encapsulated is too stringent.
- One way of relaxing this requirement is to divide the structure of an object into visible and **hidden attributes** (**instance variables**).
- The visible attributes may be directly accessed for reading by external operators of by high level query language.
- The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations.

Encapsulation of Operations



```
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
Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

Persistence of Objects

- **Persistent objects** are the objects that are created and stored in database and exist even after the program termination.
- Typical mechanisms for making an object persistence are: Naming and Reachability.
  - **Naming Mechanism**: Assign an object a unique persistent name through which it can be retrieved by this and other programs.
  - **Reachability Mechanism**: Make the object reachable from some persistent object. An object B is said to be **reachable** from an object A if a sequence of references in the object graph lead from object A to object B.

Inheritance

- **Type (class) Hierarchy**
  - A type in its simplest form can be defined by giving it a type name and then listing the names of its visible (**public**) functions
  - When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:
    - TYPE_NAME: function, function, . . . , function
  - Example:
    - PERSON: Name, Address, Birthdate, Age, SSN

Inheritance

- **Subtype:**
  - When the designer or user must create a new type that is similar but not identical to an already defined type **supertype.** Subtype inherits all the functions of the Supertype.
  - Example: 1
    - PERSON: Name, Address, Birthdate, Age, SSN
    - EMPLOYEE **subtype-of** PERSON: Salary, HireDate, Seniority
    - STUDENT **subtype-of** PERSON: Major, GPA

Inheritance: Example (2)

- Consider a type that describes objects in plane geometry, which may be defined as follows:
  - GEOMETRY_OBJECT: Shape, Area, ReferencePoint
- Now suppose that we want to define a number of subtypes for the GEOMETRY_OBJECT type, as follows:
  - RECTANGLE **subtype-of** GEOMETRY_OBJECT: Width, Height
  - TRIANGLE **subtype-of** GEOMETRY_OBJECT: Side1, Side2, Angle
  - CIRCLE **subtype-of** GEOMETRY_OBJECT: Radius

Inheritance: Example (2) (contd.)

- An alternative way of declaring these three subtypes is to specify the value of the Shape attribute as a condition that must be satisfied for objects of each subtype:
  - RECTANGLE **subtype-of** GEOMETRY_OBJECT (Shape='rectangle'): Width, Height
  - TRIANGLE **subtype-of** GEOMETRY_OBJECT (Shape='triangle'): Side1, Side2, Angle
  - CIRCLE **subtype-of** GEOMETRY_OBJECT (Shape='circle'): Radius

Inheritance

- Multiple Inheritance and Selective Inheritance
  - Multiple inheritance in a type hierarchy occurs when a certain subtype T is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes.
  - For example, we may create a subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER.
    - This leads to the creation of a type lattice rather than a type hierarchy.

Extents

- **Extents**:
  - The collection of objects of the same type in the database is known as an extent of the type.
  - Extent is the current set of objects belonging to the class.
  - Queries in OQL refer to the extent of a class and not the class directly.
- **Persistent Collection**:
  - This holds a collection of objects that is stored permanently in the database and hence can be accessed and shared by multiple programs
- **Transient Collection**:
  - This exists temporarily during the execution of a program but is not kept when the program terminates

Complex Objects

Unstructured complex object:

- These is provided by a DBMS and permits the storage and retrieval of large objects that are needed by the database application.
- Typical examples of such objects are bitmap images and long text strings (such as documents); they are also known as binary large objects, or BLOBs for short.
- This has been the standard way by which Relational DBMSs have dealt with supporting complex objects, leaving the operations on those objects outside the RDBMS.

Complex Objects

Structured complex object:

- This differs from an unstructured complex object in that the object's structure is defined by repeated application of the type constructors provided by the OODBMS.
- Hence, the object structure is defined and known to the OODBMS.
- The OODBMS also defines methods or operations on it.
- For example: The DEPARTMENT object is structured object.

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

Versions and Configurations

- Versions
  - Some OO systems provide capabilities for dealing with multiple versions of the same object (a feature that is essential in design and engineering applications).
  - For example, an old version of an object that represents a tested and verified design should be retained until the new version is tested and verified: very crucial for designs in manufacturing process control, architecture , software systems …..
- **Configuration**:
  - A **configuration** of the complex object is a collection consisting of one version of each module arranged in such a way that the module versions in the configuration are compatible and together form a valid version of the complex object.

Current Status

- OODB market growing very slowly these days.
- O-O ideas are being used in a large number of applications, without explicitly using the OODB platform to store data.
- Growth:
  - O-O tools for modeling and analysis, O-O Programming Languages like Java and C++
- Compromise Solution Proposed:
  - Object Relational DB Management (Informix Universal Server, Oracle 11g, IBM’s UDB, DB2/II …)

Exercise

1. What is OID? How persistent objects are maintained in OO Database?
2. Define state of an object. Distinguish between persistent and transient objects.
3. Distinguish multiple inheritance and selective inheritance in OO concepts.
4. What is the difference between structured and unstructured complex object? Differentiate identical versus equal objects with examples.
5. What are the advantages and disadvantages of OODBMS?

THANK YOU !