Database Systems

Department of Information Technology,

Faculty of Computing,

Sri Lanka Institute of Information Technology.

Course Introduction

- Introduction to Unit
  - Contents
    - Design and development of object-relational databases
    - Understanding of indexing techniques
    - Understanding of query optimization
    - Understanding and application of database tuning techniques.
    - Understanding the concepts and techniques used in distributed databases.

Course Introduction… (contd.)

- Contact hours
  - 2 hours lecture/week
  - 2 hours practical/week
  - 1 hour tutorial/week

- Recommended References
  - Oracle Documentation
  - Ramakrishnan, R. and Gehrke, J., *Database Management Systems*, 3nd Editon, McGraw-Hill,2003.
  - Garcia-Molina, H., Ullman, J.D. and Widom, J., *Database Systems: The Complete Book*,Prentice-Hall, 2002.

Course Introduction… (contd.)

- Grading

Midterm test	-	20%
Practical examinations	-	20%
Final Exam	-	60%

Course Introduction… (contd.)

- All related course materials on CourseWeb page
  - http://courseweb.sliit.lk
  - Enrolment key: **IT3020-DBS**
- Contact persons:
  - Prasanna S. Haddela (Lecturer-in-charge)
    - Email: prasanna@sliit.lk (preferred for appointments)
    - Office: 7th Floor - Malabe Campus
  - Ms. Thamali Dassanayake
    - Office: 7th Floor, Malabe Campus
    - Email: thamali.d@sliit.lk

Database Management Systems III

Introduction to Object Relational Database Systems

Lecture - 1

Background

- Relational model (70’s):
  - Clean and simple representation and access.
    - Tables, primary and foreign keys, SQL.
    - Good foundation of relational structure, algebra, and normal forms.
  - Swept the DB market in the 1980s.
    - Continues to dominate the DB applications even now.
  - Right for business and administrative data.
    - Tables of atomic attributes adequate to represent information.

Background

- Relational model (70's):
  - Not as good for other kinds of data (e.g., multimedia, networks, CAD).
    - Cumbersome to manage such data with Binary Large Objects (BLOBs).
  - RDB has limitations even in its core application areas.
    - Set valued attributes (e.g., academic qualifications, children of employees).
    - Some logical identifiers replaced by artificial keys (e.g., addresses of properties, multiple attribute keys).
    - ISA Hierarchies of entity sets (e.g., student is a person).

Background

- Object-Oriented models (80’s):
  - Proposed as an alternative to relational model.
    - To overcome its limitations.
  - Complex objects were supported.
    - nested relations.
  - Added DBMS functionality to OO programming environments.
    - Object ID, inheritance, methods.
  - ODMG standards: Object data and query languages
    - ODL & OQL.
  - Made limited inroads in the 1990s, then faded away.

Background

- Object-relational DBMS (mid-90's):
  - Extends relational model to support a broader class of applications.
  - Bridge between relational and OO models.
  - SQL:99 and SQL:2003 standards extended SQL to support the OO model.
  - DBMS vendors (e.g., Oracle, IBM, Informix) have added OO functionality.
    - Adherence to the standard varies.

Limitations of relational model

- No support for set valued attributes
  - Example:
    - Person (ID: string, Name: string, PhoneN: string, childID: string)
    - Key: (ID, PhoneN, childID)
    - Not in 3NF (FD: ID -> Name)

ID	Name	PhoneN	ChildID
111	Joe	4576	222
111	Joe	6798	222
222	Bob	5162	333

1NF to 3NF

ID	Name	PhoneN	ChildID
111	Joe	4576	222
111	Joe	6798	222
222	Bob	5162	333

ID	Name
111	Joe
111	Joe
222	Bob

ID	PhoneN
111	4576
111	6798
222	5162

ID	ChildID
111	222
111	222
222	333

Limitations of RDB: Example

- 1NF relation:
  - Person (ID: string, Name: string, PhoneN: string, childID: string)
- 3NF relations:
  - Person(ID, Name)
  - Phone (ID, PhoneN)
  - ChildOf (ID, childID)
- Query: Find the phone numbers of all of Joe’s grandchildren.
- SQL on both schema need multiple joins.

Set valued attributes

ID	Name	PhoneN	ChildID
111	Joe	{4576, 6798}	{222}
222	Bob	{5162}	{333}

- A more appropriate schema:
  - Person (ID: string, Name: string, PhoneN: {string}, child: {string})
  - Query: Find the phone numbers of all of Joe’s grandchildren.
  - Ideally, the query could be written as:
    - Select p.child.child.PhoneN
    - From Person p
    - Where p.Name = ‘Joe’
  - Note: Oracle implementation is different from this.

SQL extensions for ORDB

- Add OO features to the type system of SQL.
  - columns can be of new user defined types (UDTs)
  - user-defined methods on UDTs
  - reference types and “deref”
  - inheritance
  - old SQL schemas still work! (backwards compatible)

User Defined Types

- A user-defined type, or UDT, is essentially a class definition, with data fields and methods.
- Two uses:

1. As a rowtype, that is, the type of a relation.
2. As the type of an attribute of a relation.

Object types in Oracle

- Uses object types for both columns and rows of relations.
- Example:

```
CREATE TYPE BarType AS OBJECT (
 name CHAR(20),
 addr CHAR(20) )
 /
```

- Note: in Oracle, type definitions must be followed by a slash (/) to store the type.

Object Type

- An object type has 3 components:
  - A name identifies the object type uniquely within the schema.
  - Attributes model the structure and state of the real-world entity.
    - Attributes can be built-in types or object types.
  - Methods, functions or procedures implement operations.
    - (To be covered later.)

Creating Row Objects

- Type declarations do not create tables.
- Object types used in place of attribute lists in CREATE TABLE statements.
  - Example:

```
CREATE TABLE bars OF BarType;
```

- Each row of the table represents an object of given row type.

Inserting Values

- As a multi-column table:

```
INSERT INTO bars VALUES ('Sally''s', 'River
   Rd');
```

- As a single column/object value:
  - Each object type (type defined with AS OBJECT) has a type constructor of the same name.
  - Example:

```
INSERT INTO Bars VALUES(
   BarType('Joe''s Bar', 'Maple St.') );
```

Normal select

- Retrieve as a multi-column table:

```
SELECT * FROM bars;
```

```
NAME                ADDR
---------------------------------------
Joe's Bar           Maple St.
Sally's             River Rd
```

Select as a Single Column

- Select from bars as a single column table.

```
SELECT VALUE(b) FROM bars b;
VALUE(B)(NAME, ADDR)
---------------------------------------------
BARTYPE('Joe''s Bar  ', 'Maple St.   ')
BARTYPE('Sally''s    ', 'River Rd    ')
```

Column Objects

- Objects that occupy table columns in a relational table.
- Example:

```
CREATE TYPE BeerType AS OBJECT (
 name CHAR(20),
 manf CHAR(20) )
 /
CREATE TABLE menu
 (bar bartype,
  beer beertype,
  price real);
```

  - Menu is a table with two attributes of object types.

Alternative: Object table

```
CREATE TYPE MenuType AS OBJECT (
    bar BarType,
    beer BeerType,
    price real)
/
CREATE TABLE Menu2 OF MenuType;
```

- Menu2 is a table of object type.
- Rows of Menu2 are objects (not so in table Menu).
- Methods can be defined on MenuType and invoked on rows of Menu2.

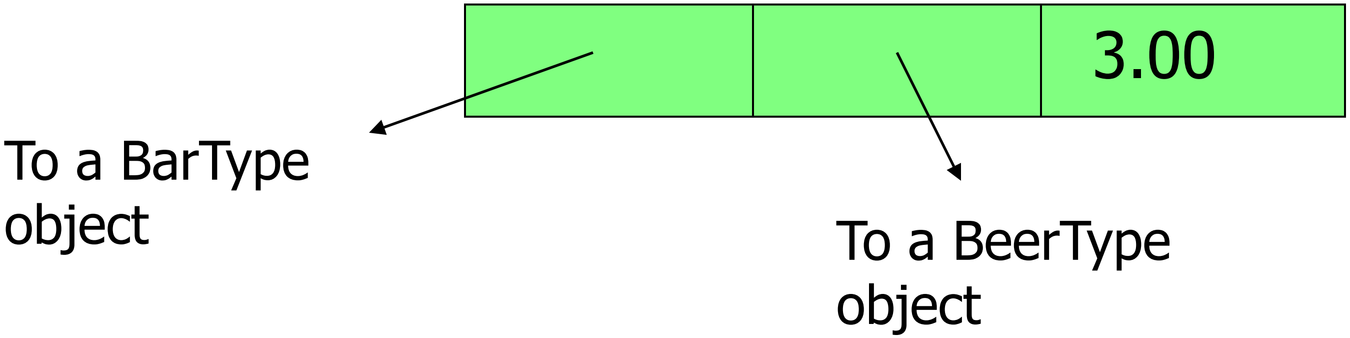
Object References using REF

- If T is a type, then REF T is the type of a reference to T, that is, a pointer to an object of type T.
- Often called an "object ID" in OO systems.
- REF is a built-in data type in Oracle.
- Unlike object ID's, a REF is visible, although it is usually gibberish.

Example: REF

```
CREATE TYPE MenuType2 AS OBJECT (
    bar         REF BarType,
    beer        REF BeerType,
    price       FLOAT)
/
```

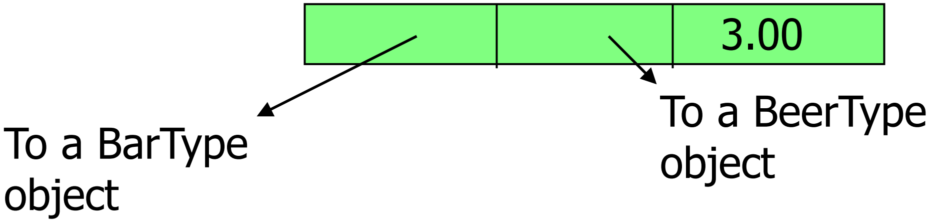
- MenuType2 objects look like:



Obtaining REFs

- To get the REF to a row object,
  - select the object from its object table applying the REF operator.
- Example

```
CREATE TABLE Sells OF MenuType2;
```



```
INSERT INTO sells VALUES(
    (SELECT REF(b) FROM bars b WHERE name='Jim''s'),
    (SELECT REF(e) FROM beers e WHERE name='Swan'),
    2.40);
```

Use aliases to retrieve objects

- To access an attribute of object type, you must use an alias for the relation.
  - Example:

```
SELECT s.Beer.name
   FROM Sells s;
```

- This will not work:

```
SELECT Beer.name
   FROM Sells;
```

- Neither will:

```
SELECT Sells.Beer.name
   FROM Sells;
```

alias-අන්වර්ථ නාමයක්

Retrieving with REF Datatype

```
SELECT s.bar.name, s.beer.name, price
    FROM sells s;
    BAR.NAME     BEER.NAME          PRICE
    -------------------------------------------
    Jim's               Swan                 2.40
    Jim's               Bud                  3.00
    Sally's             Fosters              2.65
    Sally's             Miller               2.75
```

Dereferencing

- Accessing the object referred to by a REF is called dereferencing the REF.
  - Dereferencing is automatic, using the dot operator.
- Example

```
SELECT s.beer.name
   FROM Sells s
   WHERE s.bar.name = 'Joe''s Bar';
```

Another Example

```
CREATE TYPE person AS OBJECT (
   name VARCHAR2(30),
   manager REF person );
/
CREATE TABLE person_table OF person;
SELECT x.name, x.manager.name
   FROM person_table x;
```

- x.manager.name follows the pointer from the person x to x's manager (who is another person in the table), and retrieves the manager's name.

Scoped REFs

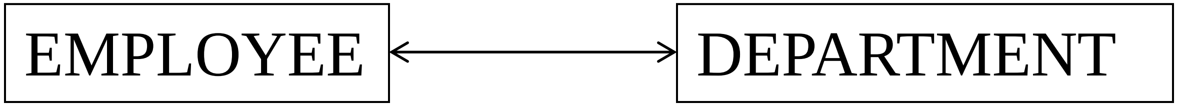
- Example:

```
CREATE type dept_t as object (dno integer, dname
   varchar(12))
/
CREATE TABLE dept_table OF dept_t;
CREATE TABLE dept_loc (
   dept REF dept_t references dept_table,
   loc VARCHAR(20));
```

  - Only objects of dept_table can be values of dept column.

Co-dependent Types

- Types can depend upon each other for their definitions.
  - Example: Object types EMPLOYEE and DEPARTMENT



EMPLOYEE

DEPARTMENT

  - one attribute of EMPLOYEE is the department the employee belongs to and
  - one attribute of DEPARTMENT is the employee who manages the department.

Incomplete Types

```
CREATE TYPE department;
/
CREATE TYPE employee AS
    OBJECT (
    name VARCHAR2(30),
    dept REF department,
    supv REF employee );
/
CREATE TYPE department AS
    OBJECT ( name
    VARCHAR2(30),
    mgr REF employee);
/
```

- CREATE TYPE department; is optional.
- DEPARTMENT is now an *incomplete object type*.
- A REF to an incomplete object type is accepted, so EMPLOYEE type is stored without error.
- Department type is then completed.

Constraints on Object Tables

- Define constraints on an object table just as on other tables
- Example:

```
CREATE TYPE person AS OBJECT (
    pid NUMBER,
    name VARCHAR(25),
    address VARCHAR(50));
/
CREATE TABLE person_table OF person
    ( pid PRIMARY KEY,
     name NOT NULL
   );
```

REF columns

- Oracle does not allow Unique or PRIMARY KEY constraints on REF columns.
- A REF column can be assigned a null value.
- NOT NULL constraint can be specified on such columns.

Null Objects and Attributes

- As in the relational model, a NULL represents an unknown value.
- The following can be NULL:
  - A column value in a table
  - Object
  - Object attribute value
  - A collection, or collection element
- A NULL can be replaced by an actual value later on.

Summary

- ORDB: introduction.
- Object type definitions.
- Creation of row and column objects.
- REF and DEREF operations.
- Constraints on object tables.

Database Systems

ORDB: Collections

Last Lecture

- Object types
  - Declaring
  - Row objects/ column objects
  - References
    - Dereferencing (implicit join)
- Constraints on object tables

- Any questions?

NAME	ADDRESS	INVESTMENTS			
		COMPANY	PURCHASE PRICE	DATE	QTY
John Smith	3 East Av Bentley WA 6102	BHP	12.00	02/10/01	1000
		BHP	10.50	08/06/02	2000
		IBM	58.00	12/02/00	500
		IBM	65.00	10/04/01	1200
		INFOSYS	64.00	11/08/01	1000
Jill Brody	42 Bent St Perth WA 6001	INTEL	35.00	30/01/00	300
		INTEL	54.00	30/01/01	400
		INTEL	60.00	02/10/01	200
		FORD	40.00	05/10/99	300
		GM	55.50	12/12/00	500

COMPANY	CURRENT PRICE	EXCHANGES TRADED	LAST DIVIDEND	EARNING PER SHARE
BHP	10.50	Sydney New York	1.50	3.20
IBM	70.00	New York London Tokyo	4.25	10.00
INTEL	76.50	New York London	5.00	12.40
FORD	40.00	New York	2.00	8.50
GM	60.00	New York	2.50	9.20
INFOSYS	45.00	New York	3.00	7.80

Collection Types

- Useful for modelling one-to-many relationships.
  - Example: An investor makes many share purchases.
- Collection datatypes in Oracle:
  - varrays
  - nested tables.

VARRAYs

- Arrays of variable size.
  - Specify a maximum size when you declare the array type.
  - Creating an array type does not allocate space.
    - Since it is only a type definition.
- Examples:

```
CREATE TYPE price_arr AS VARRAY(10) OF
    NUMBER(12,2);
```

  - The VARRAYs of type PRICES have no more than ten elements, each of data type NUMBER(12,2).

VARRAYs

- A varray type can be used as:
  - the data-type of a column of a relational table;
  - an attribute data-type in an object type definition;
- Example:
  - Create type excharray as varray(5) of varchar(12)

```
Create type excharray as varray(5) of varchar(12)
/
Create type share_t as object(
cname varchar(12),
cprice number(6,2),
exchanges excharray,
dividend number(4,2),
earnings number(6,2))
/
```

Creating a VARRAY

- To insert a collection use its type constructor method.
  - The type constructor method has same name as the type.
  - Its argument is a comma-separated list of collection elements.
- Example:
  - create table shares of share_t(

    cname primary key);
  - insert into shares values(‘BHP’, 10.50,

    excharray(‘Sydney’ ,’New York’), 1.50, 3.20);

VARRAY Example

```
CREATE TYPE price_arr AS
   VARRAY(10) OF NUMBER(12,2)
/
CREATE TABLE pricelist (
   pno integer,
   prices price_arr);

INSERT INTO pricelist
   VALUES(1, price_arr(2.50,3.75,4.25));
```

Retrieving from a VARRAY

- SELECT * FROM pricelist;

```
PNO   PRICES
 --------------------------------
 1     PRICE_ARR(2.5, 3.75, 4.25)
```

- SELECT pno, s.COLUMN_VALUE price
  FROM pricelist p, TABLE(p.prices) s;

```
PNO    PRICE
--------------
 1      2.5
 1      3.75
 1      4.25
```

Nested Tables

- Allow values of tuple components to be whole relations.
- If *T* is a UDT, we can create a table type *S:*

  CREATE TYPE S AS TABLE OF *T* ;

  - Values of type S are relations with rowtype *T*.
  - S can be the type of an attribute in another UDT or in a relation.
  - See the example on next slide.

Example: Nested Table Type

```
CREATE TYPE BeerType AS OBJECT (
   name   CHAR(20),
   kind   CHAR(10),
   colour CHAR(10))
/
CREATE TYPE BeerTableType AS
   TABLE OF BeerType
/
```

Example - Continued

-
  ```
  CREATE TABLE Manfs (
                      name CHAR(30),
                      addr   CHAR(50),
                      beers beerTableType)
  NESTED TABLE beers STORE AS beer_table;
  ```
- BeerTableType used in Manfs relation to store the set of beers by each manufacturer in one tuple.

Storing Nested Tables

- Oracle doesn’t really store each nested table as a separate relation
  - it just makes it look that way.
  - tuples of all the nested tables for one attribute *A* are stored in one relation *R*.
- Declare a storage of nested tuples in CREATE TABLE by:

  NESTED TABLE *A* STORE AS *R*
- In previous example,

  NESTED TABLE beers STORE AS beer_table;

Querying a Nested Table

- We can retrieve the value of a nested table like any other value.
- But these values have two type constructors:
  - For the table.
  - For the type of tuples in the table.
- Find the beers by Anheuser-Busch:

```
SELECT beers FROM Manfs
WHERE name = ’Anheuser-Busch’;
```

- Produces one value like:

```
BeerTableType(
   BeerType(’Bud’, ’lager’, ’yellow’),
   Beertype(’Lite’, ’malt’, ’pale’),
…)
```

Querying Within a Nested Table

- A nested table can be converted to an ordinary relation by applying TABLE(…).
- This relation can be used in FROM clauses like any other relation.
- Find the ales made by Anheuser-Busch:

```
SELECT b.name
FROM TABLE( SELECT beers
            FROM Manfs
            WHERE name = ’Anheuser-Busch’) b
WHERE b.kind = ’ale’;
```

Nested Table Example 2

- -- ‘/’ after each type definition omitted to save space

```
CREATE TYPE proj_t AS OBJECT (
        projno NUMBER,
        projname VARCHAR (15));
CREATE TYPE proj_list AS TABLE OF proj_t;
CREATE TYPE employee_t AS OBJECT (
        eno number,
        projects proj_list);
CREATE TABLE employees of employee_t (eno primary key)
NESTED TABLE projects STORE AS employees_proj_table;
```

Inserting and Retrieving

- Insert a row into employees table:

```
INSERT INTO employees VALUES(1000, proj_list(
   proj_t(101, 'Avionics'),
   proj_t(102, 'Cruise control')
));
```

- To retrieve the projects of eno 1000:

```
SELECT *
FROM TABLE(SELECT t.projects FROM employees t
WHERE t.eno = 1000);
```

```
PROJNO  PROJNAME
----------------------
 101     Avionics
 102     Cruise control
```

Collection Unnesting

- Unnest or flatten the collection attribute of a row
  - by joining each row of the nested table with the row that contains the nested table.
  - Example:

```
SELECT e.eno, p.*
FROM employees e, TABLE (e.projects) p;
```

```
 ENO       PROJNO    PROJNAME
---- ---------- ----------------
1000        101     Avionics
1000        102     Cruise control
2000        100     Autopilot
```

DML on Collections:

- Use a TABLE expression to identify the nested table values.

```
INSERT INTO TABLE(SELECT e.projects
                    FROM employees e
                    WHERE e.eno = 1000)
  VALUES (103, 'Project Neptune');
UPDATE TABLE(SELECT e.projects
               FROM …) p
  SET p.projname = 'Project Pluto'
  WHERE p.projno = 103;
DELETE TABLE(SELECT e.projects
               FROM …) p
  WHERE p.projno = 103;
```

DML on Nested Tuples

- To drop a particular nested table, set the nested table column in the parent row to NULL.

```
UPDATE employees e
    SET e.projects = NULL
    WHERE e.eno = 1000;
```

- To add back a nested table row:

```
UPDATE employees e
    SET e.projects = proj_list(proj_t(103, 'Project Pluto'))
    WHERE e.eno=1000;
```

DML on Nested Tuples

- There is a difference between a NULL value and an empty constructor. To add back a nested table row, we could have done it in two steps as follows:

```
UPDATE employees e
   SET e.projects = proj_list() // Creates a nested table w/o any rows
   WHERE e.eno=1000;

INSERT INTO TABLE
   (SELECT e.projects FROM employees e WHERE e,eno = 1000)
   VALUES (proj_t(102, ‘Project Pluto’));
```

Multilevel Collection Types

- Collection types whose elements are themselves another collection type.

- Possible multilevel collection types are:
  - Nested table of nested table type
  - Nested table of varray type
  - Varray of nested table type
  - Varray of varray type
  - Nested table or varray of a user-defined type that has an attribute that is a nested table or varray type

Multilevel Collection Types Example

- Models a system of stars in which each star has a collection of the planets revolving around it, and each planet has a collection of its satellites.
  - CREATE TYPE sat_t AS OBJECT (name VARCHAR2(20), orbit NUMBER);

    /
  - CREATE TYPE sat_ntt AS TABLE OF sat_t

    /
  - CREATE TYPE planet_t AS OBJECT (name VARCHAR2(20), mass NUMBER, satellites sat_ntt)

    /

Multilevel Collection Types Example… (contd.)

- CREATE TYPE planet_ntt AS TABLE OF planet_t;
  /
- CREATE TYPE star_t AS OBJECT (name VARCHAR2(20), age NUMBER, planets planet_ntt)
  /
- CREATE TABLE stars_tab of start_t (name PRIMARY KEY)
  NESTED TABLE planets STORE AS planets_nttab
  (NESTED TABLE satellites STORE AS satellites_nttab);
- Separate nested table clauses are provided for the outer planets nested table and for the inner satellites one.

Multilevel Collection Types Example… (contd.)

- Inserting a new star called ‘Sun’…

-
  ```
  INSERT INTO stars VALUES('Sun',23,
   nt_pl_t( planet_t( 'Neptune',10,
                 nt_sat_t(satellite_t('Proteus',67),
                          satellite_t('Triton',82))),
             planet_t('Jupiter',189,
                 nt_sat_t(satellite_t('Callisto',97),
                          satellite_t('Ganymede', 22)) ) ));
  ```

Multilevel Collection Types Example… (contd.)

- Inserting a planet called ‘Saturn’ to the star ‘Sun’…

- INSERT INTO TABLE(SELECT planets FROM stars WHERE name = 'Sun')

```
 VALUES ('Saturn', 56,
  nt_sat_t(
   satellite_t('Rhea', 83)
  )
 );
```

Multilevel Collection Types Example… (contd.)

- Inserting a satellite called ‘Miranda’ to planet ‘Uranus’ of the star ‘Sun’…

-
  ```
  INSERT INTO TABLE(
  SELECT p.satellites
  FROM TABLE( SELECT s.planets
                         FROM stars s
                         WHERE s.name = 'Sun') p
  WHERE p.name = 'Uranus')
  VALUES ('Miranda', 31);
  ```

Summary

- Collection Types
  - VARRAYs
  - Nested Tables
- DDL, DML and SELECTs on Collection Types
- Multilevel collection types

Database Systems

ORDB: Methods and Inheritance

Last Week

- Nested Collections
  - VARRAYs and Nested tables
    - Storing
    - Querying
    - Manipulating

- Any questions?

Encapsulation and UDTs: Methods

- Functions or procedures declared in an object type definition to implement behaviour of objects.
  - Declared in a CREATE TYPE statement and Defined in a CREATE TYPE BODY statement.
- Methods written in PL/SQL or Java are stored in the database.
  - preferable for data-intensive procedures and short procedures that are called frequently.
- Procedures in other languages, such as C, are stored externally.
  - preferable for computationally intensive procedures that are called less frequently.

Member Method

- Define a member method in the object type for each operation an object of that type should perform.
- Example: Add a method priceInYen to MenuType.
- CREATE TYPE MenuType AS OBJECT (

```
CREATE TYPE MenuType AS OBJECT (
   bar REF BarType,
   beer REF BeerType,
   price FLOAT,
   MEMBER FUNCTION priceInYen(rate IN FLOAT)
      RETURN FLOAT
)
/
```

Example: Type Body

```
CREATE TYPE BODY MenuType AS
MEMBER FUNCTION
priceInYen(rate FLOAT)
RETURN FLOAT IS
   BEGIN
     RETURN rate * SELF.price;
   END;
END;
/
CREATE TABLE Sells OF MenuType;
```

Some Points to Remember

- SELF is a built-in parameter that denotes the object instance on which the method is currently being invoked.
- Member methods can reference the attributes and methods of SELF without using a qualifier.
  - The SELF bit in SELF.price is optional.
- Many methods will take no arguments.
  - In that case, do not use parentheses after the function name.
- The body can have any number of function definitions, separated by semicolons.
  - The body must include all the functions;

Adding a new method

- Use ALTER TYPE to add a method:

```
ALTER TYPE MenuType
    ADD MEMBER
    FUNCTION
    priceInUSD(rate FLOAT)
    RETURN FLOAT
    CASCADE;
```

```
CREATE OR REPLACE TYPE BODY
    MenuType AS
    MEMBER FUNCTION
    priceInYen(rate FLOAT)
    RETURN FLOAT IS
        BEGIN
            RETURN rate * SELF.price;
        END priceInYen;
    MEMBER FUNCTION
    priceInUSD(rate FLOAT)
    RETURN FLOAT IS
        BEGIN
            RETURN rate * SELF.price;
        END priceInUSD;
END;
/
```

Example of Method Use

- Use an alias for the object followed by a dot, the name of the method, and argument(s) if any.
- EXAMPLE:

```
SELECT s.beer.name, s.priceInYen(106.0)
FROM Sells s
WHERE s.bar.name = 'Joe''s Bar';
```

- Use parentheses, even if a method has no arguments.
  - E.g., select e.ename, e.age()
    ```
    from oremp e;
    ```
  - Assume age() is computed from attribute birthdate of the object type.

Object Comparison

- The values of scalar data types such as CHAR or REAL have a predefined order.
- But, instances of an object type have no predefined order.
- To compare two items of a user-defined type, define an order relationship using a *map* or an *order* method.
  - At most one map method (or one order method) for an object type.

Map Methods

- Compare objects by mapping object instances to a scalar type.
  - DATE, NUMBER, VARCHAR, etc.
- Example: For an object type called RECTANGLE, the map method AREA can return its (HEIGHT * WIDTH) .
  - Then two rectangles can be compared by their areas.

Map Method

- A parameter-less member function that uses the MAP keyword.
- If an object type defines one, the method is called automatically to evaluate
  - comparisons such as obj_1 > obj_2 and
  - comparisons implied by the DISTINCT, GROUP BY, and ORDER BY clauses.

Example

```
CREATE TYPE Rectangle_type AS OBJECT
( length NUMBER,
  width NUMBER,
 MAP MEMBER FUNCTION area RETURN NUMBER
);
CREATE TYPE BODY Rectangle_type AS MAP MEMBER
   FUNCTION area RETURN NUMBER IS
   BEGIN
     RETURN length * width;
  END area;
END;
```

Example

```
CREATE TABLE rectangles OF Rectangle_type;
INSERT INTO rectangles VALUES (1,2);
INSERT INTO rectangles VALUES (2,1);
INSERT INTO rectangles VALUES (2,2);
SELECT DISTINCT VALUE(r) FROM rectangles r;
```

```
   VALUE(R)(LEN, WID)
--------------------
  RECTANGLE_TYP(1, 2)
  RECTANGLE_TYP(2, 2)
```

Order Methods

- Order methods make direct object-to-object comparisons.
- A function with one declared parameter for another object of the same type.
- Definition of this method must return
  - < 0 if "self " is less than the argument object.
  - 0   if "self " is equal to the argument object.
  - > 0 if "self " is greater than the argument object.

Order Methods

- Called automatically whenever two objects need to be compared.

- Useful where comparison semantics may be too complex to use a map method.
  - E.g., to compare images, create an order method to compare by their brightness or number of pixels.

Example

- An order method that compares customers by customer ID:

-
  ```
  CREATE TYPE Customer_typ AS OBJECT
  ( id NUMBER,
    name VARCHAR2(20),
    addr VARCHAR2(30),
    ORDER MEMBER FUNCTION match (c Customer_typ) RETURN INTEGER );
  /
  ```

Example

```
CREATE TYPE BODY Customer_typ AS
ORDER MEMBER FUNCTION match (c Customer_typ)
RETURN INTEGER IS
BEGIN
        IF id < c.id THEN RETURN -1; -- any num <0
        ELSIF id > c.id THEN RETURN 1; -- any num >0
        ELSE RETURN 0;
        END IF;
END;
END;
 /
```

On Comparison Methods

- In defining an object type, you can specify either a map method or an order method for it, but not both.

- If an object type has no comparison method, Oracle can compare two objects of that type only for equality or inequality.
  - Two objects of the same type count as equal only if the values of their corresponding attributes are equal.

On Comparison Methods

- When sorting or merging a large number of objects, use a map method.
  - One call maps all the objects into scalars, then sorts the scalars.
  - An order method is less efficient because it must be called repeatedly (it can compare only two objects at a time).

Methods on Nested Tables

```
CREATE TYPE proj_t AS OBJECT (projno number,
    Projname varchar(15));
CREATE TYPE proj_list AS TABLE OF proj_t;
CREATE TYPE emp_t AS OBJECT
   ( eno number,
       projects proj_list,
     MEMBER FUNCTION projcnt RETURN INTEGER
   );
```

Methods on Nested Tables

```
CREATE OR REPLACE TYPE BODY emp_t AS MEMBER
    FUNCTION projcnt RETURN INTEGER IS
       pcount INTEGER;
       BEGIN
         SELECT count(p.projno) INTO pcount
         FROM TABLE(self.projects) p;
         RETURN pcount;
       END;
    END;
    /
```

Methods on Nested Tables

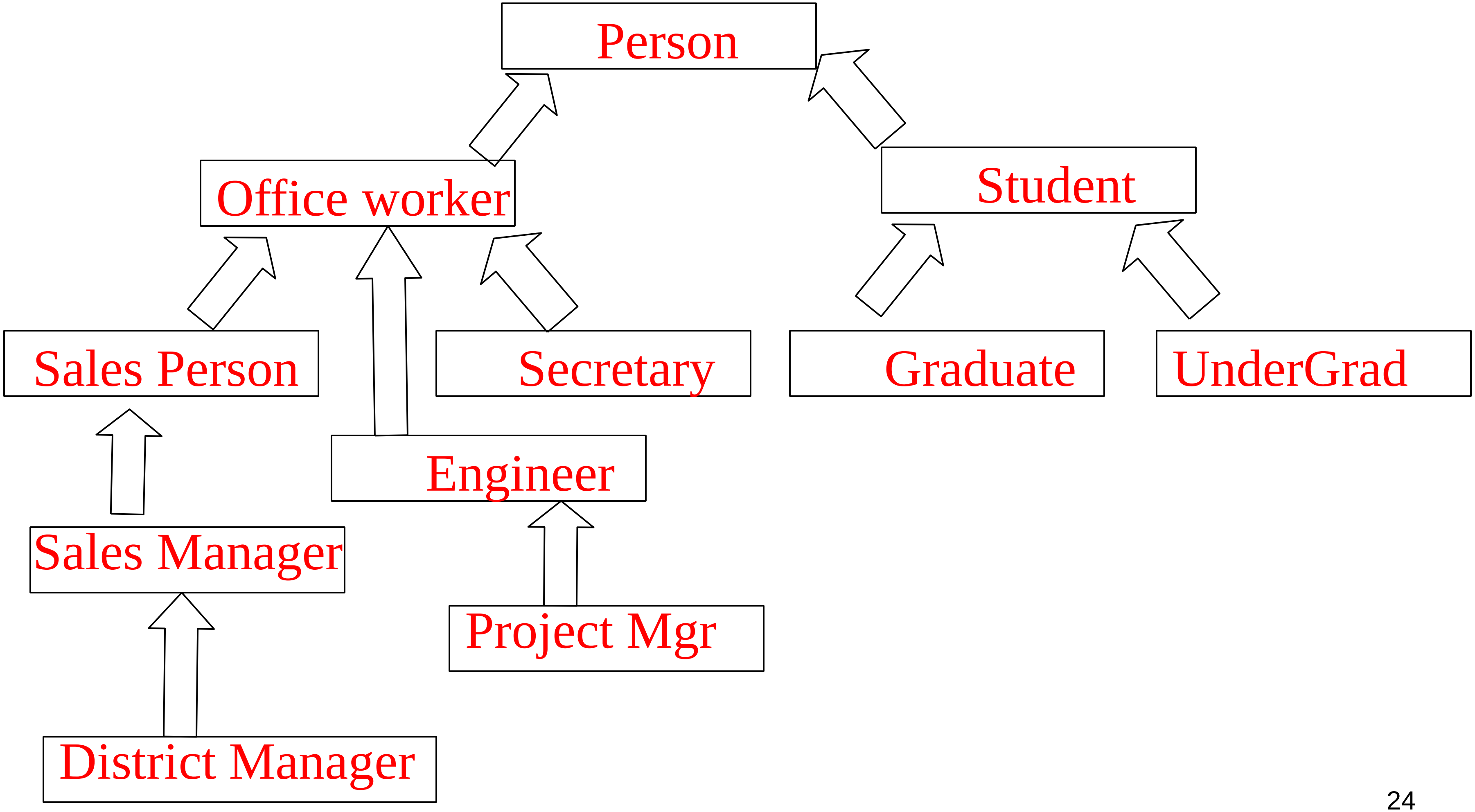
```
CREATE TABLE emptab OF emp_t
  (Eno PRIMARY KEY)
   NESTED TABLE projects STORE AS emp_proj_tab;

SELECT e.eno, e.projcnt() projcount
FROM emptab e;
```

Inheritance

- A natural model for organising information.
  - e.g. captures the fact that sales managers are also salespeople.
- Methods and representation can be shared.
  - Reduces redundancy.
- New types and objects can be defined in existing hierarchies rather than from scratch.
  - Increases flexibility and extensibility.

Example: Person hierarchy



Inheritance in Oracle

- It consists of a parent base type, or supertype, and one or more levels of child object types, or subtypes.
- Subtypes in a hierarchy are connected to their supertypes by inheritance.
    - subtypes automatically acquire the attributes and methods of their parent type.
    - any attribute or method updated in a supertype is automatically updated in subtypes also.

Specializing Subtypes

- Add new attributes the supertype does not have.
- A subtype cannot drop or change the type of an attribute it inherits from its parent.
- Add new methods that the parent does not have.
- Override the implementation of a parent method.

Specializing Subtypes

- Change the implementation of some methods a subtype inherits.
  - E.g., a shape object type might define a method calculate_area().
  - Two subtypes, rectangular and circular, might implement this method in a different way.

FINAL and NOT FINAL Types

- To permit subtypes, the object type must be defined as not final.
    - By including the keyword NOT FINAL in the type declaration.
    - By default, an object type is final.

- Example
    - CREATE TYPE Person_type AS OBJECT

      (pid NUMBER,

        name VARCHAR2(30),

        address VARCHAR2(100))   NOT FINAL;
    - Subtypes of Person_type can be defined.

Altering object type

- You can change a final type to a not final type and vice versa with an ALTER TYPE statement.
  - If a NOT FINAL type has no current subtypes.

- For example,
  - ALTER TYPE Person_type FINAL;

Creating Subtypes

- Use a CREATE TYPE statement with an UNDER parameter to specify the parent type:

```
CREATE TYPE Student_type UNDER Person_type
   ( deptid NUMBER,
      major VARCHAR2(30)) NOT FINAL;
/
```

  - Student_type inherits all the attributes and methods declared in or inherited by Person_type.
- New attributes in a subtype must have different names from the attributes or methods in all its supertypes in the type hierarchy.

Multiple child types

- A type can have multiple child subtypes, and these can also have subtypes.
- Example:

```
CREATE TYPE Employee_type UNDER Person_type
   ( empid NUMBER,
        mgr VARCHAR2(30)
   );
/
```

  - In addition to student_typ under person_type given earlier

Subtype under another subtype

- The new subtype inherits all the attributes and methods of its parent type, both declared and inherited.
- Example:

```
CREATE TYPE PartTimeStudent_type UNDER
    Student_type
 (numhours NUMBER);
```

Table of supertype

- Creating a supertype table

```
Create table person_tab of person_type
   (pid primary key);
```

- Inserting a subtype object/row

```
Insert into person_tab values
   ( student_type(4, 'Edward Learn',
          '65 Marina Blvd, Ocean Surf, WA, 6725',
          40, 'CS')
  );
```

Selecting all instances

- Using VALUE() function to select all instances of a super type:
  - Select all persons such as employees, students, etc. in the table:
- SELECT VALUE(p) FROM person_tab p;

```
VALUE(P)(PID, NAME, ADDRESS)
-----------------------------------------------
Person_type(21937, ‘Fred’, ‘4 Ambrose Street’)
Student_type(27362, `Peter’, … , 21, ‘Oragami’)
PartTimeStudent_type(2134, ‘Jack’,…, 13, ‘Physics’,
  5)
Person_type(21362, ‘Mary’, …)
Student_type(18437, `Susan’, … , 13, ‘Maths’)
PartTimeStudent_type(4318, ‘Jill’,…, 21, ‘Pottery’,
  2)
Person_type(39374, ‘George’, …)
```

Selecting instances

- From student type and its subtypes

```
SELECT VALUE(s)
   FROM person_tab s
   WHERE VALUE(s) IS OF (Student_type);

VALUE(P)(PID, NAME, ADDRESS)
-------------------------------------------
Student_typ(27362, `Peter’, … , 21, ‘Oragami’)
PartTimeStudent_type(2134,’Jack’,…,13,‘Physics’,
   5)
Student_typ(18437, `Susan’, … , 13, ‘Maths’)
PartTimeStudent_type(4318,’Jill’,…,21,‘Pottery’,
   2)
```

Selecting instances

- From student type but not from subtypes

```
SELECT VALUE(s)
FROM person_tab s
WHERE VALUE(s) IS OF (ONLY student_type);
```

```
VALUE(P)(PID, NAME, ADDRESS)
------------------------------------------------
Student_typ(27362, `Peter’, … , 21, ‘Oragami’)
Student_typ(18437, `Susan’, … , 13, ‘Maths’)
```

Selecting a Subtype Attribute

- TREAT() function to make the system treat each person as a part-time student to access the subtype attribute numhours:

```
SELECT Name, TREAT(VALUE(p) AS
PartTimeStudent_type).numhours hours
FROM person_tab p
WHERE VALUE(p) IS OF (ONLY PartTimeStudent_type);
```

```
NAME      hours
-----------------------------------------------
Jack      5
Jill      2
```

NOT INSTANTIABLE Types

- Use this option with types intended solely as supertypes of specialized subtypes.
  - CREATE TYPE Address_typ AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;*
  - CREATE TYPE AusAddress_typ UNDER Address_typ(...);
  - CREATE TYPE IntlAddress_typ UNDER Address_typ(...);

* You cannot create instances of the Address_typ only (similar to “*abstract classes*” in OO)

NOT INSTANTIABLE Methods

- Use this option to declare a method in a type without implementing it there.
  - A type that contains a non-instantiable method must itself be declared not instantiable.

  -
    ```
    CREATE TYPE T AS OBJECT (
    x NUMBER,
    NOT INSTANTIABLE MEMBER FUNCTION func1()*
    RETURN NUMBER ) NOT INSTANTIABLE NOT FINAL;
    ```

* The type body for T does not contain a definition for func1

NOT INSTANTIABLE Methods

- Define a method as non-instantiable if every subtype is to override the method in a different way.

- If a subtype does not implement every inherited non-instantiable method, the subtype must be declared not instantiable.
  - A non-instantiable subtype can be defined under an instantiable supertype.

FINAL and NOT FINAL Methods

- If a method is declared to be final, subtypes cannot override it by providing their own implementation.
  - Unlike types, methods are not final by default.
  - They must be explicitly declared to be final.
- An overriding method is specified in a CREATE TYPE BODY statement.

Example

```
CREATE TYPE MyType AS OBJECT
    (  ...,
       MEMBER PROCEDURE Print,
       FINAL MEMBER FUNCTION foo(x NUMBER) …, ...
    )   NOT FINAL;
/
CREATE TYPE MySubType UNDER MyType
    ( ...,
      OVERRIDING MEMBER PROCEDURE Print,
       ...);
/
```

Overloading Methods

- A subtype can add new methods that have the same names as methods it inherits.
  - Methods that have the same name but different signatures in a type are called overloads.
  - The compiler uses the methods' signatures to tell them apart.

Example: Overloading Methods

```
CREATE TYPE MyType AS OBJECT
  ( ...,
    MEMBER FUNCTION fun(x NUMBER)…,
    ...) NOT FINAL;
/
CREATE TYPE MySubType UNDER MyType
  ( ...,
     MEMBER FUNCTION fun(x DATE) …,
    ...);
/
```

- Same function name, different signature

Summary

- Nested Collections
  - Varrays and nested tables
    - Storing
    - Querying
    - Manipulating
- Inheritance in Oracle
  - FINAL/NOT FINAL
  - Subtypes (UNDER)
  - Getting at particular subtypes
  - INSTANTIABLE/NOT INSTANTIABLE (Types and methods)
  - Overriding & Overloading

Database Systems

ORDB: Triggers and ER-ORDB Mapping

Last Week

- Methods
  - Member methods
  - Comparison methods (MAP & ORDER)
  - Methods on nested tables
- Inheritance
  - Type inheritance (UNDER, FINAL, NOT FINAL)
  - INSTANTIABLE/NOT INSTANTIABLE Types and Methods
  - Methods (Overriding, Overloading)
- Any questions?

Triggers

- Procedures stored in the database and implicitly run, or *fired*, when something happens.
  - INSERT, UPDATE, or DELETE on a table or view (DML Triggers).
  - System and other data events on DATABASE and SCHEMA (System Triggers).

Triggers on Object Tables

- Triggers can be defined on an object table just as on other tables.

- Example:
  - CREATE TYPE location as OBJECT (campus VARCHAR2(20), building NUMBER, room NUMBER);
  - CREATE TYPE staff as OBJECT(sid NUMBER, name VARCHAR2(100), address VARCHAR2(100), office location);
  - CREATE TABLE staff_tab OF staff (sid PRIMARY KEY);

Example

-
  ```
  CREATE TABLE movement ( sid NUMBER, old_office location,
  new_office location );
  ```

-
  ```
  CREATE TRIGGER trig1
  BEFORE UPDATE OF office ON staff_tab
  FOR EACH ROW
  WHEN (new.office.campus = 'Bentley' )
  BEGIN
    IF :new.office.building = 314 THEN
     INSERT INTO movement VALUES (:old.sid,:old.office,
  :new.office);
   END IF;
  END;
  ```

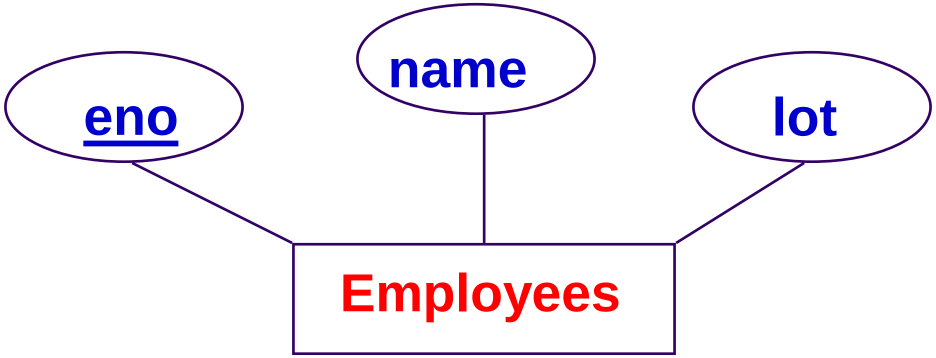
Database design for ORDB

- ER model may be extended with more constructs such as:
  - Multi-valued attributes (e.g., locations of a store)
  - Structured-type attributes (e.g., address made up of house no, street, suburb, etc.)
  - Collection-type attributes (e.g., list of dates)

Mapping ERD to ORDB

- Map each ER entity type to ORDB type
  - Include all attributes of ER type in ORDB type
  - Declare a table for each ORDB type and specify key attributes as (primary) keys
  - Sub-types in ISA hierarchy inherit attributes of super type

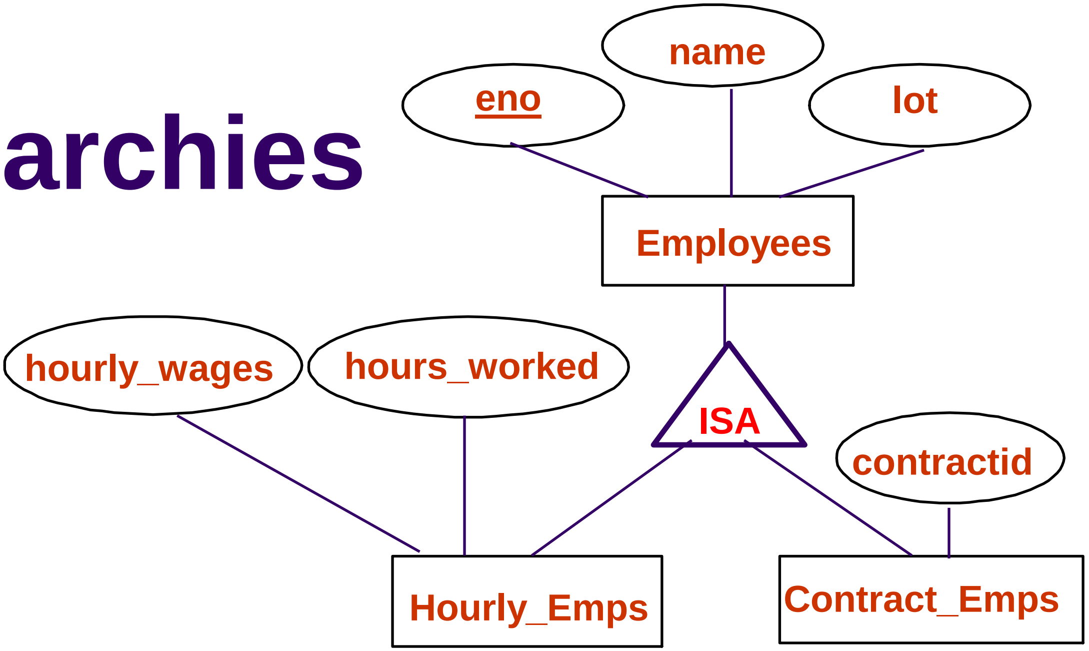
Entity Types



- Entity type can be mapped to a type easily.

```
CREATE TYPE Emp_t
          (eno CHAR(11),
          name CHAR(20),
          lot  INTEGER);
Create table employees of Emp_t(
    eno primary key);
```

ISA Hierarchies

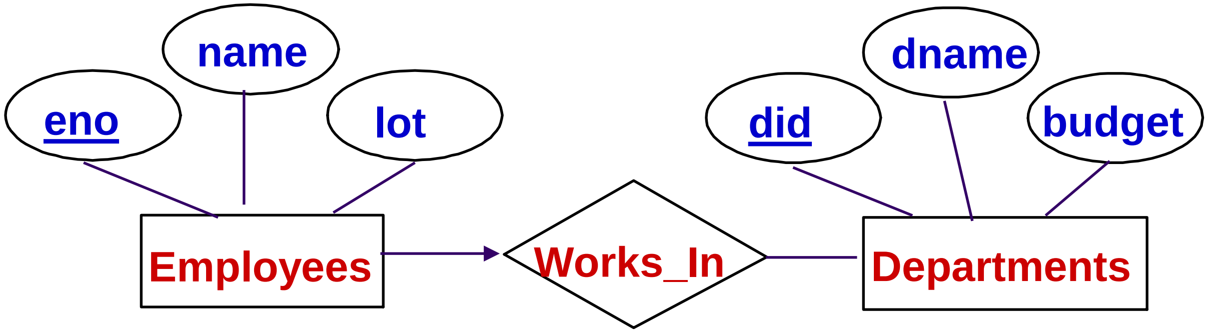


```
CREATE TYPE emp_t AS OBJECT (…) NOT FINAL;
CREATE TYPE hourly_emps_t UNDER emp_t
    (hourly_wages number(6,2),
     hours_wkd number(4,2));
CREATE TYPE contract_emps_t UNDER emp_t
    (contractid char(10));
```

Mapping ERD to ORDB

- Binary relationship types
  - Cardinality 1:1, N:1, or M:N
- Weak entity types
  - Similar to regular entity types
  - Alternative mapping possible if the weak entity does not participate in any other relationship
- N-ary relationships with n>2
  - Mapped as a separate type
  - Appropriate references to each participating class
- Methods for types (not in ER)
  - UML?

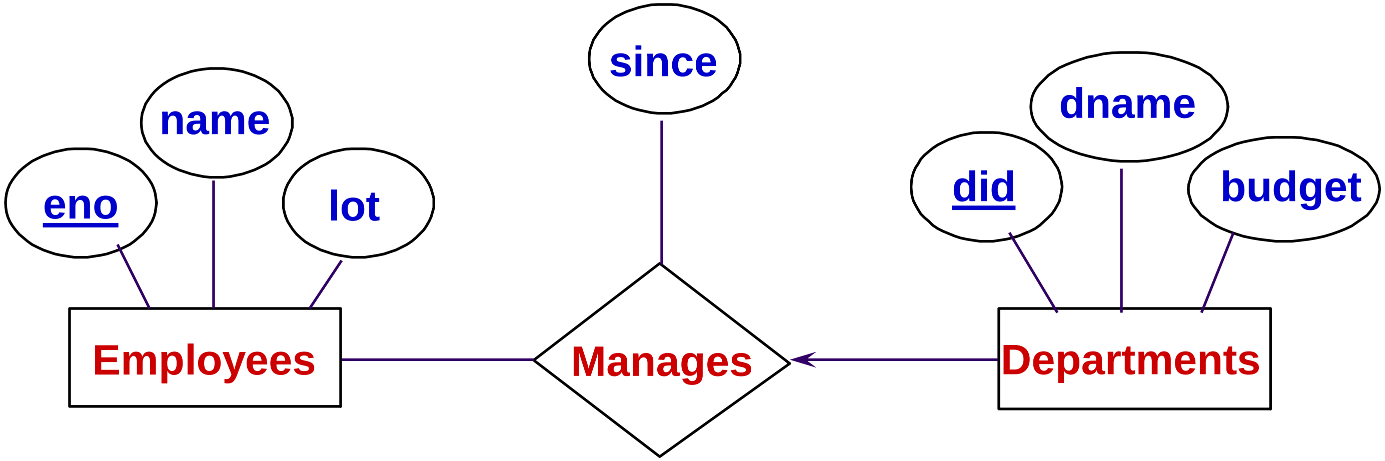
Relationship sets



- Add a reference attribute for a binary relationship with key constraint into the corresponding object type.
  - CREATE TYPE Emp_t

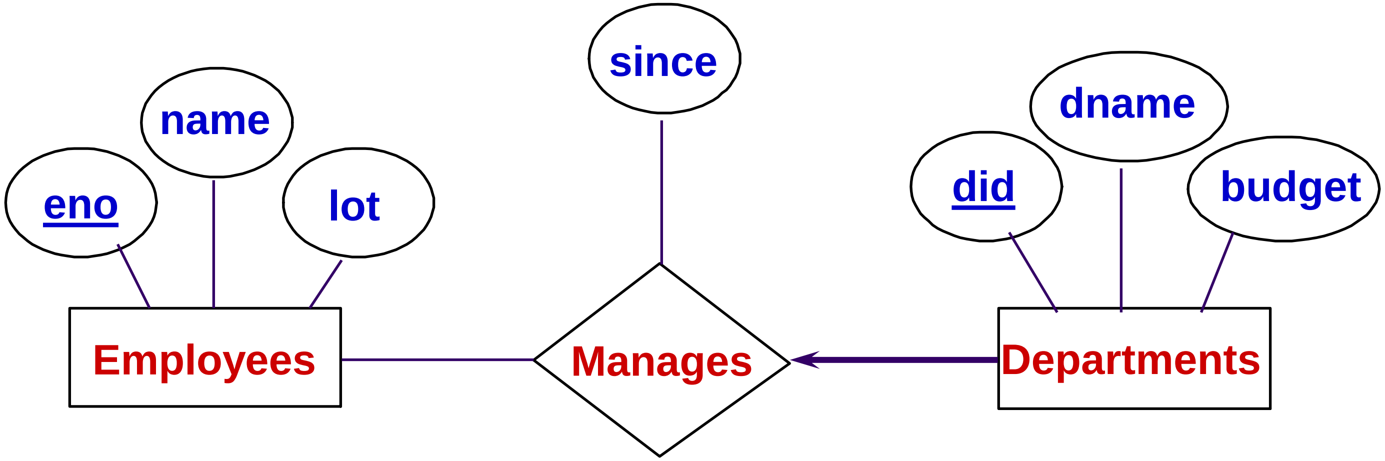
    ```
    (eno CHAR(11),
    name CHAR(20),
    lot  INTEGER,
    workdept ref dept_t);
    ```

Translating Key Constraints



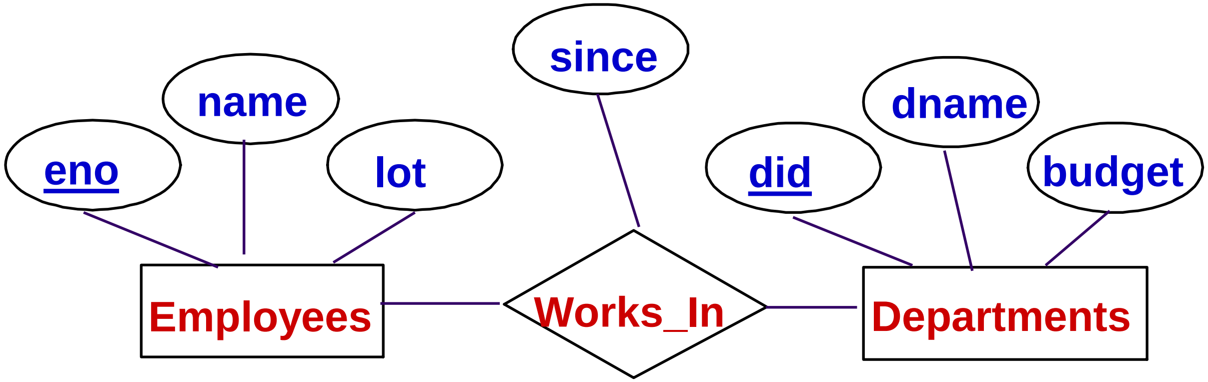
```
CREATE TYPE  Dept_t as object
   ( did  INTEGER,
     dname  CHAR(20),
     budget  REAL,
     mgr  ref emp_t
     since  DATE);
```

Participation Constraints



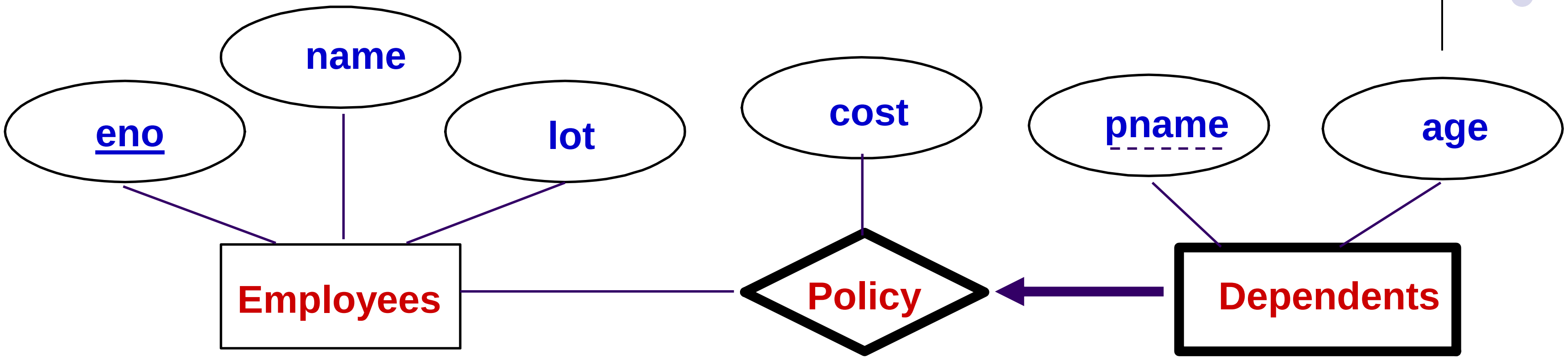
```
CREATE TABLE  Department of Dept_t
   ( did  primary key,
     mgr NOT NULL REFERENCES Employees
   );
```

M:N Relationship set



```
CREATE TYPE WorksIn_t AS OBJECT(
    emp   REF emp_t,
    dept  REF dept_t,
    since  DATE);
CREATE TABLE worksIn OF WorksIn_t(
    emp REFERENCES Employees,
    dept REFERENCES Departments);
```

Weak Entities

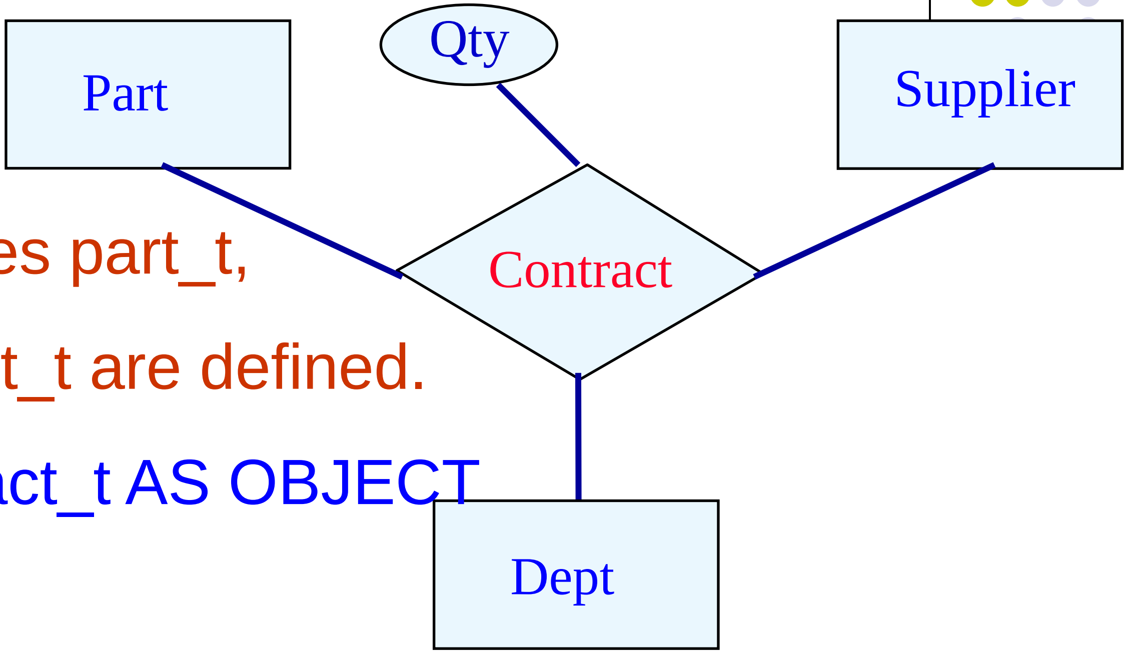


- Option 1: Map like a regular entity
- Option 2: Weak entity types that do not participate in any relationships except their identifying relationship with owning entity.
  - Use nested collection type for the weak entity information.
- Option 3: Weak entity type objects are not to be referenced by attributes of any other object.
  - Use nested collection type for the weak entity information.

Weak Entity Sets: Option 2

```
Create type depend_t as object(
    Pname varchar(12),
    Age integer,
    policyCost number (6,2));
Create type policy_t as table of depend_t;
Create type emp_t as object( …,
    dependents policy_t)
```

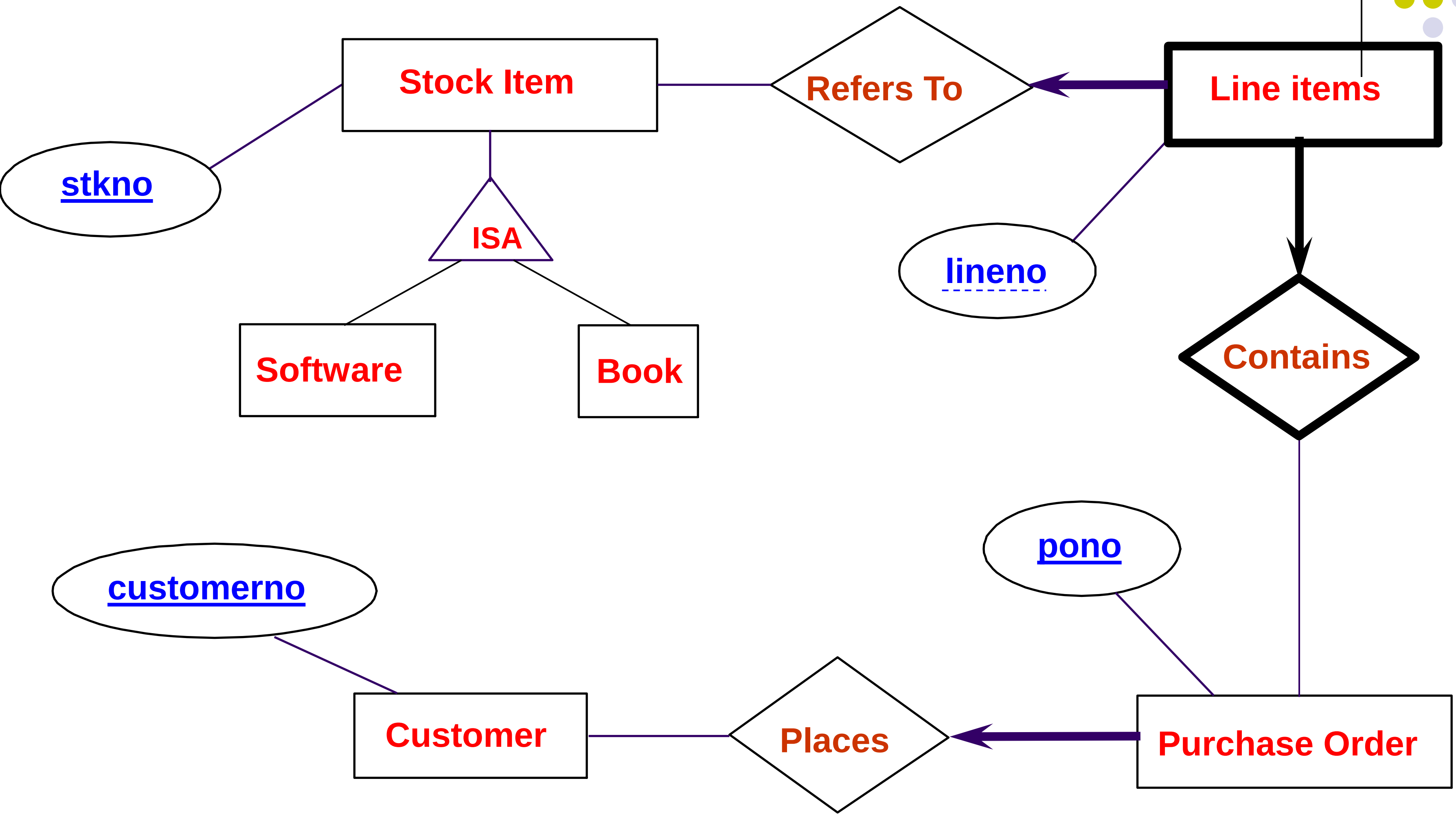
N-ary relationship



- Assuming object types part_t, supplier_t, and dept_t are defined.

```
CREATE TYPE Contract_t AS OBJECT
  (part REF part_t,
   supplier REF supplier_t,
   dept REF dept_t,
   qty NUMBER(6,2)
  );
```

Cyber Shop Example



Mapping to OR Schema

- Regular entities to types
  - Customer_typ (custNo, …)
  - StockItem_typ (StockNo, …)
  - PurchaseOrder_typ (PONo, …)
- Inheritance
  - StockItem_typ (StockNo, …) NOT FINAL
  - Book_typ(…) under StockItem_typ
  - Software_typ(…) under StockItem_typ

Mapping to OR Schema

- Weak entity type
  - LineItem_typ (LineItemNo, …)
  - LineItemList_ntabtyp AS TABLE OF LineItem_typ
  - PurchaseOrder_typ (PONo, …, LineItemList_ntab LineItemList_ntabtyp)
- Binary relationship with key constraint
  - PurchaseOrder_typ (PONo, …, LineItemList_ntab LineItemList_ntabtyp, Cust_ref REF Customer_typ)
  - LineItem_typ (LineItemNo, …, Stock_ref REF StockItem_typ)

Final types

- Customer_typ (custNo, …)
- StockItem_typ (StockNo, …)
- Book_typ(…) under StockItem_typ
- Software_typ(…) StockItem_typ
- LineItem_typ (LineItemNo, …, Stock_ref REF StockItem_typ)
- LineItemList_ntabtyp AS TABLE OF LineItem_typ
- PurchaseOrder_typ (PONo, …, LineItemList_ntab LineItemList_ntabtyp, Cust_ref REF Customer_typ)

Tables and constraints

- Customers of Customer_typ (custNo primary key)
- Stocks of StockItem_typ (StockNo primary key)
- Orders of PurchaseOrder_typ (PONo primary key,

  Cust_ref references Customers)

  nested table LineItemlist

Summary

- Trigger Example
- Mapping ER to ORDB
  - Mapping various ER features to ORDB
  - Example