Object-relational data model

Iztok Savnik, FAMNIT.

Outline

- Object-relational model of Informix
- Object-relational model of Oracle

Object-relational model of Informix

Object-oriented DBMS

- Considerable academic research on database technology over the past decade has been focused on new, post-relational data models
  - Merge the principles of object-oriented programming and design with traditional database characteristics
- Some large venture capital investments flowed into a group of startup software companies

Object-oriented DBMS

- early-1990s
  - Gemstone (Servio Logic, later renamed to Gemstone Systems), Gbase (Graphael), and Vbase (Ontologic)
- mid-1990s
  - ITASCA (Itasca Systems), Jasmine (Fujitsu, marketed by Computer Associates), Objectivity/DB (Objectivity, Inc.), ObjectStore (Progress Software, acquired by eXcelon, which was originally Object Design), Matisse (Matisse Software), O2 (O2 Technology, eventually acquired by Informix, which was acquired by IBM), ONTOS (Ontos, Inc., formerly Ontologic), POET (Poet Software, now FastObjects from Versant), Versant Object Database (Versant Corporation), and VOSS (Logic Arts)

OO-DBMS Characteristics

- Objects
  - Everything is an object and is manipulated as an object
  - The tabular, row/column organization of a relational database is replaced by the notion of collections of objects
- Classes
  - Object-oriented databases replace the relational notion of atomic data types with a hierarchical notion of classes and subclasses.
  - VEHICLES might be a class of object. CARS and BOATS …
- Inheritance
  - Objects inherit characteristics from their class and from all of the higher-level classes to which they belong.
  - Subclasses of VEHICLE, i.e., CARS, BOATS, and CONVERTIBLES classes have all properties defined in VEHICLE
  - CARS, BOATS, … can have specific attributes

OO-DBMS Characteristics

- Attributes
  - The characteristics that an object possesses are modeled by its attributes.
  - Color of an object, or number of doors that it has, …
  - Same way as columns of a table relate to its rows.
- Messages and methods
  - Objects communicate with one another by sending and receiving messages
  - When it receives a message, an object responds by executing a method
  - Thus, an object includes a set of behaviors described by its methods
- Encapsulation
  - The internal structure and data of objects is hidden from the outside world (encapsulated) behind a limited set of well-defined interfaces

OO-DBMS Characteristics

- Object identity
  - Objects can be distinguished from one another through unique object identifiers
    - Abstract pointer known as an object handle
  - Representation of the relationships among objects by the use object identifier

Pros and Cons of OO-DBMS

- Pros
  - Object-oriented databases have stirred up a storm of controversy in the database community
  - Multitable joins that are an integral part of the relational data model inherently create database overhead and make relational technology unsuitable for the ever-increasing performance demands of today's applications
- Cons
  - Object-oriented databases are unnecessary and offer no real, substantive advantages over the relational model
  - Handles (oid-s) of object-oriented databases are nothing more than the embedded database pointers of prerelational hierarchical and network databases
  - Object-oriented databases lack the strong underlying mathematical theory that forms the basis of relational databases
  -

Object-Relational Databases

- Object-relational databases typically begin with a relational database foundation and add selected features that provide object-oriented capabilities
- The object extensions that are commonly found in object-relational databases are:
  - Large data objects
  - Structured/abstract data types
  - User-defined data types

Object-Relational Databases

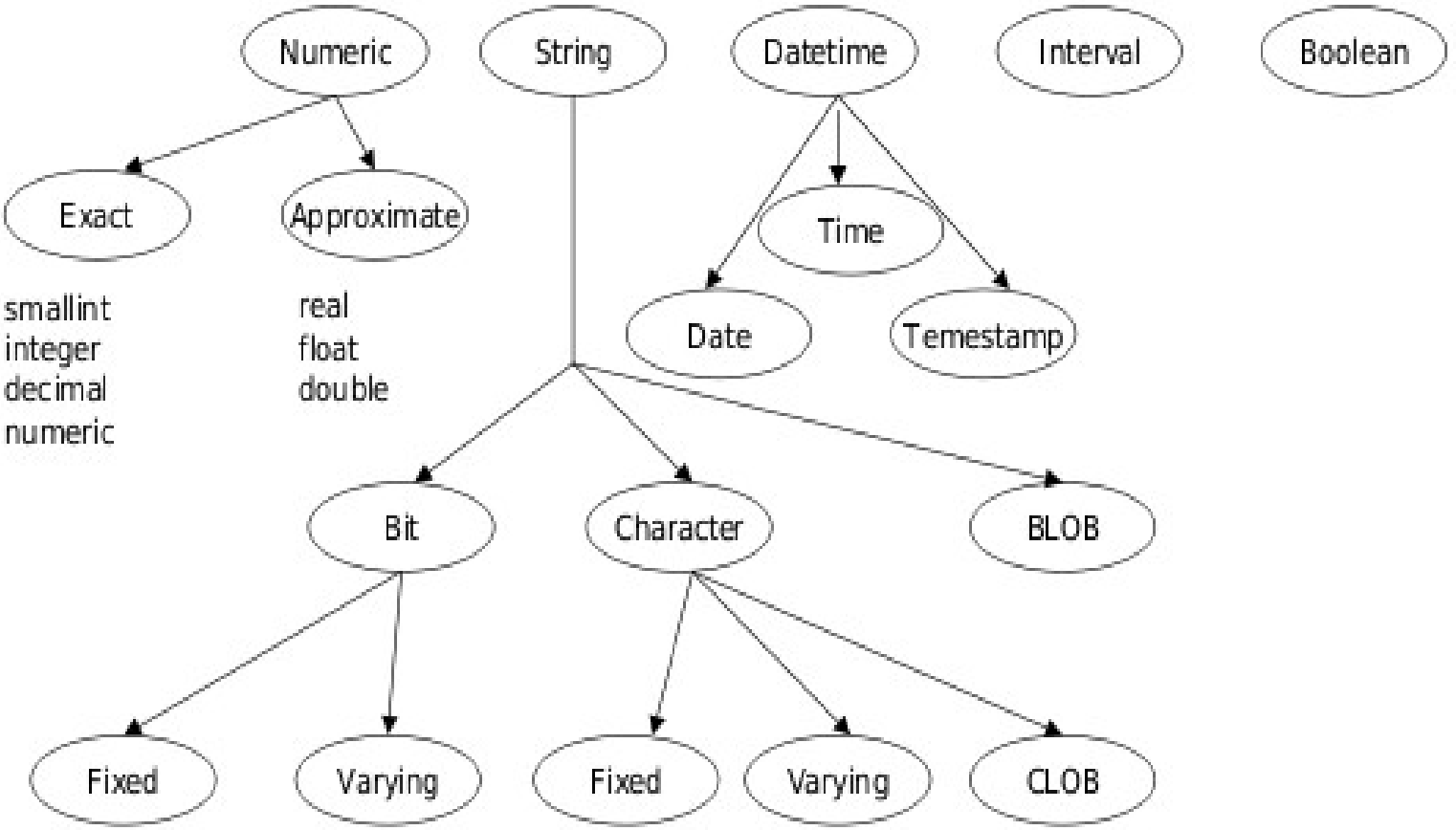
- Tables within tables
- Sequences, sets, and arrays
- Stored procedures
- Handles and object-ids

- State-of-the-art examples
  - IBM Informix
  - Oracle

Data types

- Predefined types
  - Numeric, String, BLOB, Boolean, Datetime, Interval
- User-defined types
  - Distinct type
  - Structured types
- Structured types
  - ROW, TABLE, ARRAY, REF, LIST, SET, MULTISET

Predefined types



Large Object Support

- LOBs – new data types
- LOB - byte sequences of many GB
- Two new types
  - BLOB - Binary Large Object
    - Audio, pictures, video
  - CLOB - Character Large Object
    - Text

LOB (1)

- Stored in a database
  - Not in “external files”
- The size of LOB is defined at creation
  - (KB, MB, GB)

```
CREATE TABLE  BookTable
(title        varchar(200),
book_id       INTEGER,
summary       CLOB(32K),
movie         BLOB(2G));
```

LOB: Example

```
CREATE TABLE employee
( emp#        INTEGER,
  name        VARCHAR(30),
  …
  abstract    CLOB(75K),
  signature   BLOB(1M),
  picture     BLOB(10M)
);
```

LOB: Functions

- LOB functions
  - CONCATENATION string1|| string2
  - SUBSTRING(string FROM start FOR length)
  - LENGTH(expression)
  - POSITION(search-string IN source-string)
  - NULLIF/COALESCE
  - TRIM, OVERLAY, Cast
  - User-defined functions
  - LIKE

```
EXEC SQL
  SELECT position(„Chapter 1“ IN book_text)
  INTO :int_variable
  FROM BookTable
  WHERE title=“Moby Dick“;
```

LOB: Updates

- LOBs can be read, updated, deleted as any other objects
- Large buffers for LOB
    - SQL99: locators

```
EXEC SQL
   SELECT summary, book_text, movie
   INTO :bigbuf,:biggerbuf, :massivebuf
   FROM BookTable
   WHERE title=“Moby Dick“;
```

Oracle locator functions

- **DBMS_LOB.READ(*locator, length, offset, buffer*)** Reads into the PL/SQL buffer the indicated number of bytes/characters from the LOB identified by the *locator*, starting at the *offset*.
- **DBMS_LOB.WRITE(*locator, length, offset, buffer*)** Writes the indicated number of bytes/characters from the PL/SQL buffer into the LOB identified by the *locator*, starting at the *offset*.
- **DBMS_LOB.APPEND(*locator1, locator2*)** Appends the entire contents of the LOB identified by *locator2* to the end of the contents of the LOB identified by *locator1*.
- **DBMS_LOB.ERASE(*locator, length, offset*)** Erases the contents of the LOB identified by the locator at *offset* for *length* bytes/characters; for character-based LOBs, spaces are inserted, and for binary LOBs, binary zeroes are inserted.
- **DBMS_LOB.COPY(*locator1, locator2, length, offset1, offset2*)** Copies *length* bytes/characters from the LOB identified by *locator2* at *offset2* into the LOB identified by *locator1* at *offset1*.
- **DBMS_LOB.TRIM(*locator, length*)** Trims the LOB identified by the *locator* to the indicated number of bytes/characters.
- **DBMS_LOB.SUBSTR(*locator, length, offset*)** Returns (as a text string return value) the indicated number of bytes/characters from the LOB identified by the locator, starting at the *offset*; the return value from this function may be assigned into a PL/SQL VARCHAR variable.
- **DBMS_LOB.GETLENGTH(*locator*)** Returns (as an integer value) the length in bytes/characters of the LOB identified by the *locator*.
- **DBMS_LOB.COMPARE(*locator1, locator2, length, offset1, offset2*)** Compares the LOB identified by *locator1* to the LOB identified by *locator2*, starting at *offset1* and *offset2*, respectively, for *length* bytes/characters; returns zero if they are the same and nonzero if they are not.
- **DBMS_LOB.INSTR(*locator, pattern, offset, i*)** Returns (as an integer value) the position within the LOB identified by the *locator* where the *i*th occurrence of *pattern* is matched; the returned value may be used as an offset in subsequent LOB processing calls.

Example CLOB (Oracle)

```
declare
   lob     CLOB;
   textbuf varchar(255);

begin
   /* Put text to be inserted into buffer /
   . . .

   /* Get lob locator and lock LOB for update */
   select document_lob into lob
     from documents
    where document_id = '34218'
      for update;

   /* Write new text 500 bytes into LOB */
   dbms_lob.write(lob,100,500,textbuf);

   commit;
end;
```

Distinct types

- Distinct types are used to define new type out of existing built-in types
- Before SQL99, columns could only be defined with the existing built-in data types

Distinct types

```
CREATE TYPE plan.roomtype
AS CHAR(10);

CREATE TYPE plan.meters
AS INTEGER;

CREATE TYPE plan.squaremeters
AS INTEGER;

CREATE TABLE RoomTable
(RoomID       plan.roomtype,
 RoomLength   plan.meters,
 RoomWidth    plan.meters,
 RoomPerimeter   plan.meters,
 RoomArea        plan.squaremeters);
```

```
UPDATE RoomTable
SET RoomArea=RoomLength;
```

Napaka v tipu !!!

SQL Routines

- Named persistent code
  - Activated from SQL
  - SQL procedure, function or method
- Created in schema or in a separate SQL module
- DDL
  - CREATE and DROP statements
  - ALTER statements – limited functionality
  - EXECUTE privileges GRANT and REVOKE statements

SQL Routine (1)

- Head and body
- Head consists of the name and a (possibly empty) set of parameters
  - Parameter types: IN, OUT, INOUT
  - Function parameters are always IN
  - Function return velue with RETURN statement
- SQL routines
  - Head and body are written in SQL
- External rutines
  - Head in SQL
  - Body in the host language

SQL Routine (2)

- Parameters have names and types
- Routine body is one SQL statemen
  - Can include: BEGIN ...END
  - Cannot include: DDL, CONNECT, DISCONNECT, dynamic SQL, COMMIT, ROLLBACK

```
CREATE PROCEDURE get_balance(IN acct_id INT, OUT bal
DECIMAL(15,2))
BEGIN
  SELECT balance INTO bal
    FROM accounts WHERE account_id =acct_id;
  IF bal <100
    THEN SIGNAL low_balance
  END IF;
END
```

SQL Routine (3)

- Routine body
  - RETURN allowed in functions
  - Exception is triggered if function does not include RETURN

```
CREATE FUNCTION get_balance(acct_id INT) RETURNS
DECIMAL(15,2)
BEGIN
  DECLARE bal DECIMAL(15,2);
  SELECT balance INTO bal
    FROM accounts
    WHERE account_id = acct_id;
  IF bal <100 THEN SIGNAL low_balance
  END IF;
  RETURN bal;
END
```

SQL Routine (4)

- Parameters
  - Names are optional
  - Not all SQL types are allowed (depends on host)
- LANGUAGE: host language
- NAME: file with the code

```
CREATE PROCEDURE get_balance (IN acct_id INT, OUT bal
DECIMAL(15,2))
LANGUAGE C
EXTERNAL NAME ‘bankSbalance_proc’

CREATE FUNCTION get_balance (IN INTEGER) RETURNS
DECIMAL(15,2)
LANGUAGE C
EXTERNAL NAME ‘usr/han/banking/balance’
```

SQL Routines: polymorphism

- More routines having the same name
  - S1.F (p1 INT, p2 REAL)
  - S1.F (p1 REAL, p2 INT)
  - S2.F (p1 INT, p2 REAL)
- Inside one schema we have one signature
  - S1.F (p1 INT, p2 REAL)
  - S1.F (p1 REAL, p2 INT)
- The same signature can be used in different schemas
  - S1.F (p1 INT, p2 REAL)
  - S2.F (p1 INT, p2 REAL)
- More later ...

SQL Routine: call

- Procedure is activated by CALL
  - `CALL get_balance(100, bal);`
- Functions can be called from expressions

```
SELECT account_id, get_balance (account_id)
FROM accounts
```

- Requires EXECUTE privilegies for a given routine
  - Otherwise routine is not found

Abstract (Structured) Data Types

- The data types envisioned by the relational data model are simple, indivisible, atomic data values
- Data item such as an address is actually composed of a street address, city, state, and postal code
  - Treat it as collumns
  - Treat it as a single unit
  - No middle ground

ADTs

- Many programming languages do provide such a middle ground
  - Including C and Pascal
  - They support compound data types or named data structures
  - Data structure is composed of individual data items, which can be accessed individually
  - Entire data structure can also be treated as a single unit when that is most convenient
- Structured or composite data types in object-relational databases provide this same capability in a DBMS context

ADTs – Informix

- Informix Universal Server supports abstract data types through its concept of row data types
  - Structured sequence of individual data items
- Example
  - Informix CREATE TABLE statement
  - Simple PERSONNEL table that uses a row data type to store both name and address information

Example: personnel

```
CREATE TABLE PERSONNEL (
    EMPL_NUM INTEGER,
        NAME ROW(
        F_NAME VARCHAR(15),
        M_INIT CHAR(1),
        L_NAME VARCHAR(20))
      ADDRESS ROW(
        STREET VARCHAR(35),
          CITY VARCHAR(15),
         STATE CHAR(2),
    POSTCODE ROW(
          MAIN INTEGER,
           SFX INTEGER));
```

ADTs (Informix)

- This table has three columns.
  - EMPL_NUM, has an integer data type.
  - NAME and ADDRESS, have row data type
    - ROW, followed by a parenthesized list of fields that make up the row
- NAME column’s row data type has three fields
- ADDRESS column’s row data type has four fields
  - POSTCODE itself has a row data type
    - consists of two fields
  - Hierarchy is only two levels deep

ADTs (Informix)

- Extension of the SQL dot notation
  - Already used to qualify column names with table names and user names
  - Adding a dot after a column name
    - Specify the names of individual fields within a column

```
SELECT EMPL_NUM, NAME.F_NAME, NAME.L_NAME
  FROM PERSONNEL
 WHERE ADDRESS.POSTCODE.MAIN = '12345';
```

ADTs (Informix)

- Query that retrieves the employee numbers of employees who are also managers
  - Suppose another table named MANAGERS, had the same NAME structure as one of its columns
  - Use the entire name column (all three fields) as the basis for comparison

```
SELECT EMPL_NUM
  FROM PERSONNEL, MANAGERS
 WHERE PERSONNEL.NAME = MANAGERS.NAME;
```

- Row data type allows access to the fields at any level of the hierarchy

ADTs (Informix)

- Special handling when you're inserting data into the database
  - Columns that have a row data type require a special ROW value-constructor

```
INSERT INTO PERSONNEL
     VALUES (1234,
             ROW('John', 'J', 'Jones'),
             ROW('197 Rose St.', 'Chicago', 'IL',
             ROW(12345, 6789)));
```

Defining Abstract Data Types

- If two tables need to use the same row data type structure, it is defined within each table
- Row data type should be defined once and then reused for the two columns
- Examples for the PERSONNEL table

Example (Informix)

```
CREATE ROW TYPE NAME_TYPE (
    F_NAME VARCHAR(15),
    M_INIT CHAR(1),
    L_NAME VARCHAR(20));

CREATE ROW TYPE POST_TYPE (
      MAIN INTEGER,
       SFX INTEGER);

CREATE ROW TYPE ADDR_TYPE (
    STREET VARCHAR(35),
      CITY VARCHAR(15),
     STATE CHAR(2),
  POSTCODE POST_TYPE);
```

```
CREATE TABLE PERSONNEL (
   EMPL_NUM INTEGER,
       NAME NAME_TYPE,
    ADDRESS ADDR_TYPE);
```

Example (Informix)

```
CREATE ROW TYPE PERS_TYPE (
  EMPL_NUM INTEGER,
      NAME NAME_TYPE,
   ADDRESS ADDR_TYPE);
```

```
CREATE TABLE PERSONNEL
      OF TYPE PERS_TYPE;
```

```
PERSONNEL.ADDRESS.POSTCODE.MAIN
```

```
SAM.PERSONNEL.ADDRESS.POSTCODE.MAIN
```

PERSONNEL Table

	NAME			ADDRESS				
							POSTCODE	
EMPL NUM	F NAME	M INIT	L NAME	STREET	CITY	STATE	MAIN	SFX
1234	Sue	J.	Marsh	1803 Main St.	Alamo	NJ	31948	4567
1374	Sam	F.	Wilson	564 Birch Rd.	Marion	KY	82942	3524
1421	Joe	P.	Jones	13 High St.	Delano	NM	13527	2394
1532	Rob	G.	Mason	9123 Plain Av.	Franklin	PA	83624	2643
			•					
			•					
			•					

Manipulating Abstract Data Types

- Informix Universal Server is fairly liberal in its data type conversion requirements for unnamed row types
  - Double-colon operator casts the constructed three-field row as a NAME_TYPE
- Oracle automatically defines a constructor method for the type
  - Constructor is used in the VALUES clause of the INSERT statement to glue the individual components together

Example: Manipulating ADTs

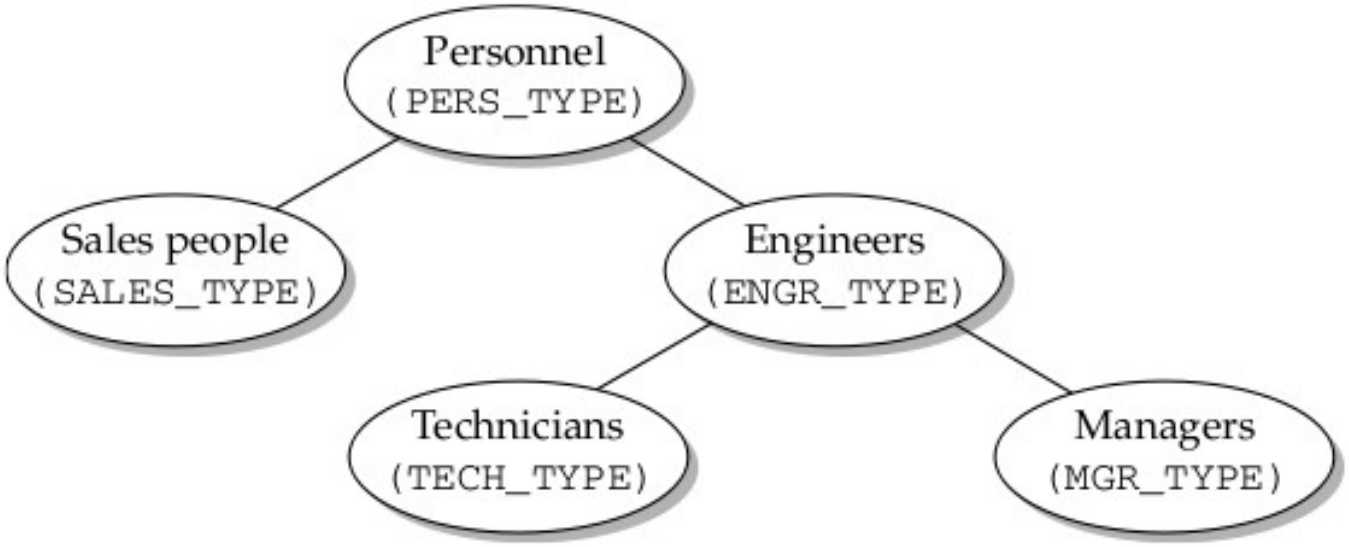
```
INSERT INTO PERSONNEL
     VALUES (1234,
             ROW('John', 'J', 'Jones')::NAME_TYPE,
             ROW('197 Rose St.', 'Chicago', 'IL',
             ROW(12345, 6789)));
```

```
INSERT INTO PERSONNEL
     VALUES (1234,
             NAME_TYPE('John', 'J', 'Jones'),
             ADDR_TYPE('197 Rose St.', 'Chicago', 'IL',
             POST_TYPE(12345, 6789)));
```

Inheritance

- Support for abstract data types gives the relational data model a foundation for object-based capabilities
  - Abstract data type can embody the representation of an object,
    - and the values of its individual fields are its attributes
- Another important feature of the object-oriented model is inheritance
  - Example of how inheritance might work in a model of a company's employee data

Example: Inheritance



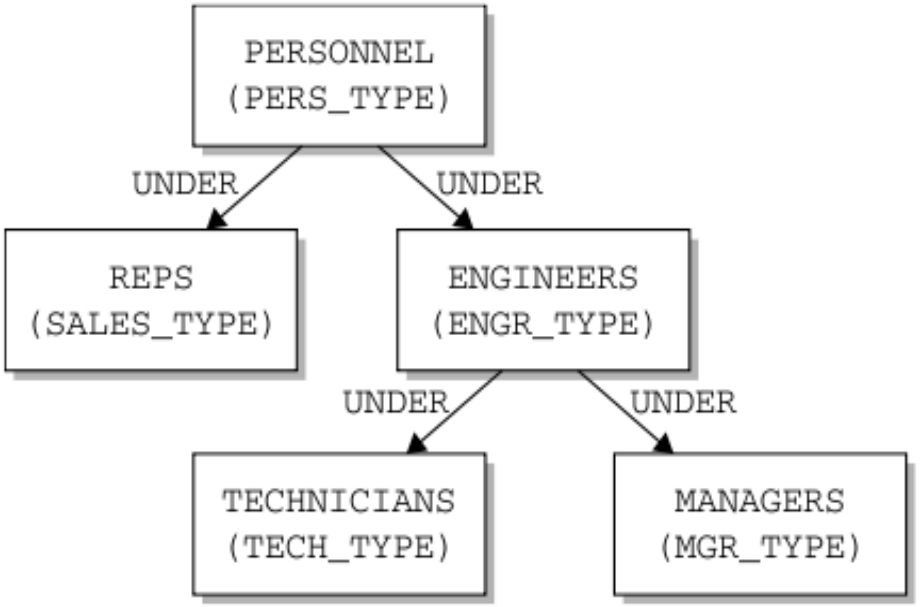
Example: Inheritance

```
CREATE ROW TYPE SALES_TYPE (
   SLS_MGR INTEGER,              /* employee number of sales mgr */
    SALARY DECIMAL(9,2),         /* annual salary */
     QUOTA DECIMAL(9,2))
     UNDER PERS_TYPE;

CREATE ROW TYPE ENGR_TYPE (
    SALARY DECIMAL(9,2),          /* annual salary */
 YRS_EXPER INTEGER                /* years of experience */
     UNDER PERS_TYPE;

CREATE ROW TYPE MGR_TYPE (
     BONUS DECIMAL(9,2))          /* annual bonus */
     UNDER ENGR_TYPE;

CREATE ROW TYPE TECH_TYPE (
 WAGE_RATE DECIMAL (5,2))         /* hourly wage rate */
     UNDER ENGR_TYPE;
```

Example: Inheritance

- The type hierarchy has pushed the complexity into the data type definitions and made the table structure very simple and easy to define
- All other characteristics of the table can (and must) still be defined within the table definition
  - REPS table includes a column that is actually a foreign key to the PERSONNEL table

Example: Creating tables




```
CREATE TABLE ENGINEERS
     OF TYPE ENGR_TYPE;
CREATE TABLE TECHNICIANS
     OF TYPE TECH_TYPE;
CREATE TABLE MANAGERS
     OF TYPE MGR_TYPE;
CREATE TABLE REPS
     OF TYPE SALES_TYPE;
```

```
CREATE TABLE REPS
     OF TYPE SALES_TYPE
 FOREIGN KEY (SLS_MGR)
  REFERENCES PERSONNEL(EMPL_NUM);
```

Example: Creating tables

- Type inheritance creates among the structure of the tables a relationship that is based on the defined row types
- Tables remain independent of one another in terms of the data that they contain
  - Rows inserted into the TECHNICIANS table don’t automatically appear in either the ENGINEERS table or in the PERSONNEL table
- A different kind of inheritance, table inheritance, provides a very different level of linkage between the table’s contents
  - Turning the tables into something much closer to object classes

Table Inheritance: Implementing Object Classes

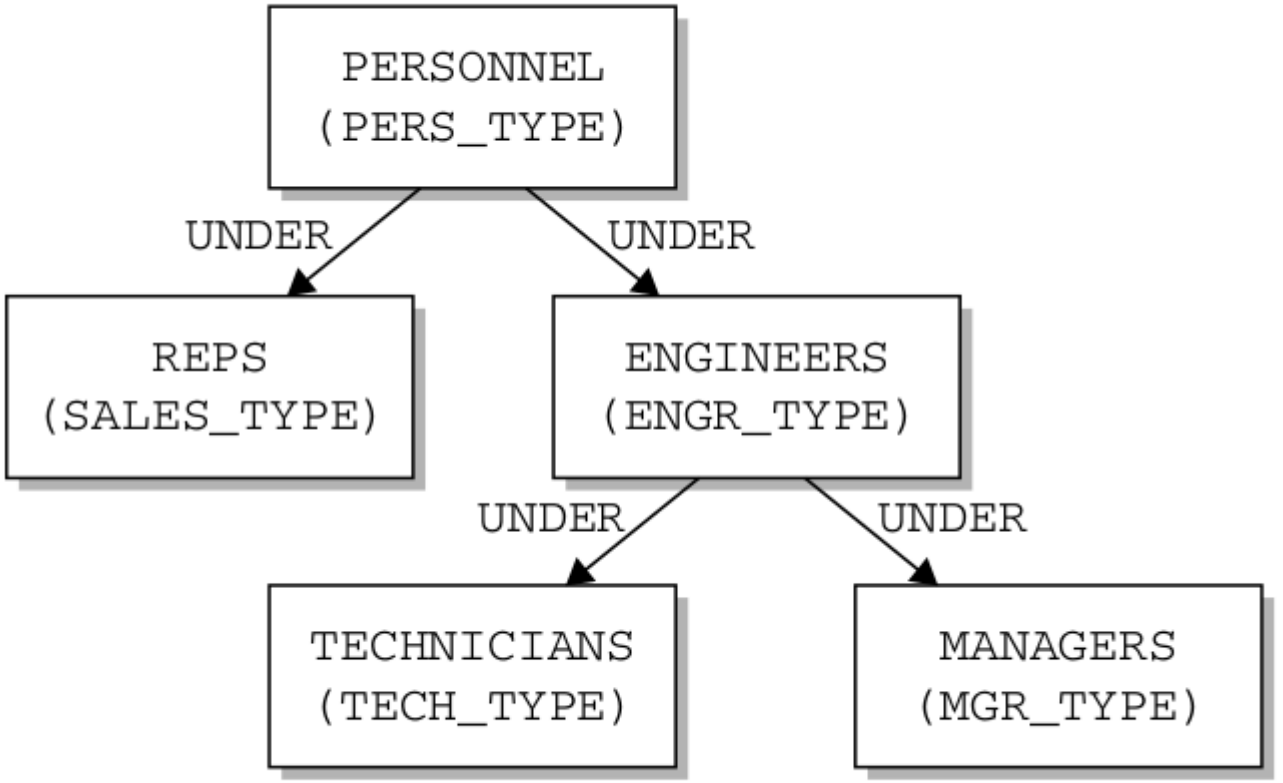
- Moves the table structure much closer to the concept of an object class
- Create a hierarchy of typed tables!
  - The tables are still based on a defined type hierarchy, but now the tables themselves have a parallel hierarchy

Example: Table inheritance

- Informix
- Table »under« does not inherits just attributes
  - Prim.key, foreign key, integrity constraints, indexes, ...
- Every type has its own table
- SQL queries work on sbtables

```
CREATE TABLE ENGINEERS
      OF TYPE ENGR_TYPE
        UNDER PERSONNEL;

CREATE TABLE TECHNICIANS
      OF TYPE TECH_TYPE
        UNDER ENGINEERS;

CREATE TABLE MANAGERS
      OF TYPE MGR_TYPE
        UNDER ENGINEERS;

CREATE TABLE REPS
      OF TYPE SALES_TYPE
        UNDER PERSONNEL;
```

Example: Table inheritance

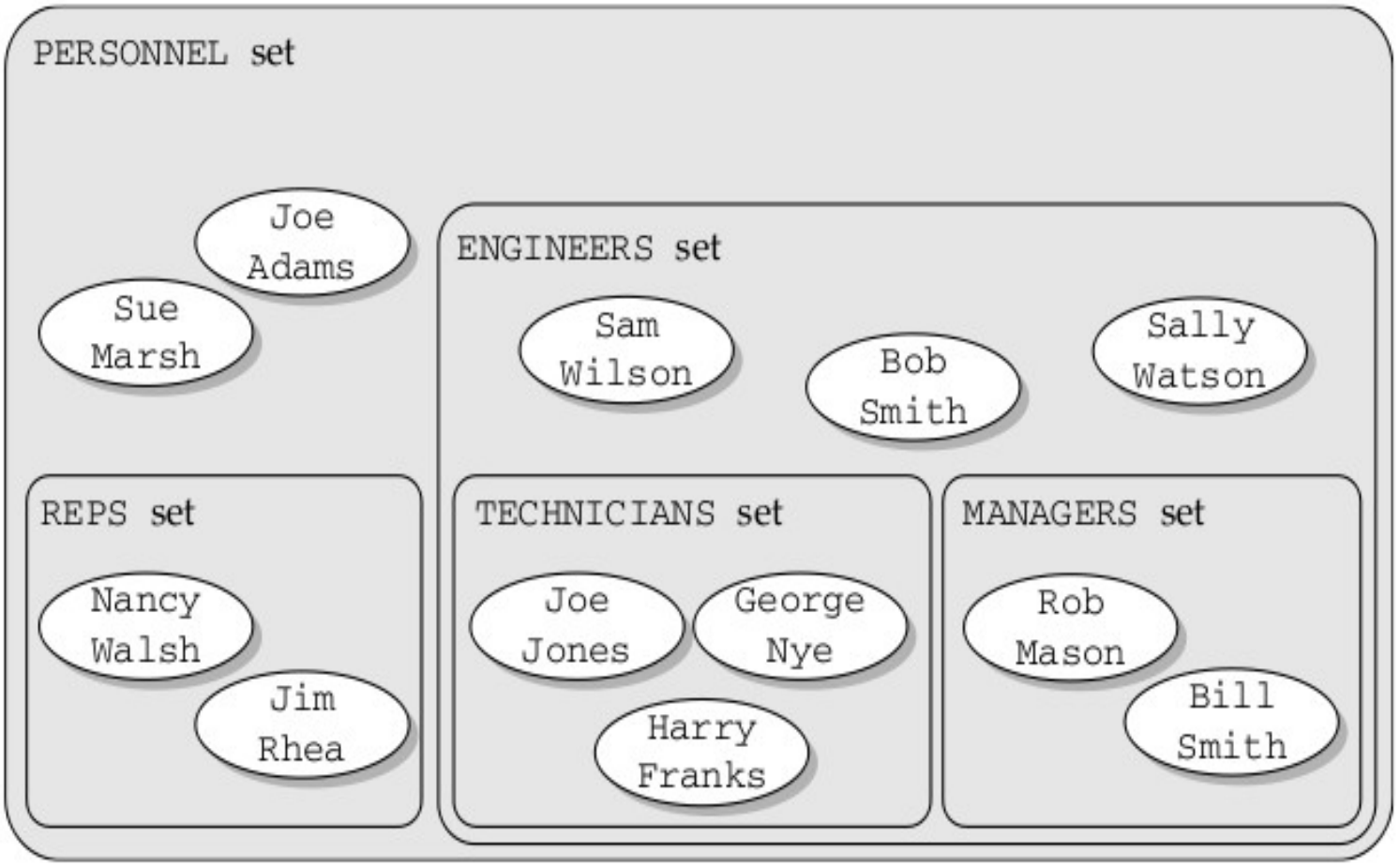


Example: Table inheritance

```
SELECT *
  FROM PERSONNEL;
```

```
SELECT *
  FROM ONLY(ENGINEERS);
```

```
SELECT *
  FROM ENGINEERS;
```



Example: Table inheritance

```
DELETE FROM PERSONNEL
 WHERE EMPL_NUM = 1234;
```

```
UPDATE PERSONNEL
   SET L_NAME = 'Harrison'
 WHERE EMPL_NUM = 1234;
```

```
DELETE FROM MANAGERS
 WHERE SALARY < 20000.00;
```

```
DELETE FROM ONLY(ENGINEERS)
 WHERE EMPL_NUM = 1234;
```

```
DELETE FROM PERSONNEL
 WHERE SALARY < 20000.00;
```

Sets, Arrays, and Collections

- Extend table engineers
  - Engineer has a set of academic degrees
  - Relational solution:

ENGINEERS Table

EMPL_NUM	NAME			ADDRESS					SALARY	YRS_EXPER
	F_NAME	M_INIT	L_NAME	STREET	CITY	STATE	POSTCODE MAIN	POSTCODE SFX		
1234	Bob	J.	Smith	956 Elm Rd.	Forest	NY	38294	4567	$45,000	6
1374	Sam	F.	Wilson	564 Birch Rd.	Marion	KY	82942	3524	$30,000	12
1421	Sally	P.	Watson	87 Dry Lane	Mt Erie	DL	73853	2394	$34,500	9
1532			•							
			•							
			•							

DEGREES Table

EMPL_NUM	DEGREE	SCHOOL
1245	BS	Michigan
1245	MS	Purdue
1374	BS	Lehigh
1439	BS	MIT
1436	BS	MIT
1439	MBA	Stanford
	•	
	•	
	•	

Foreign key

Defining Collections (Informix)

- **Lists** A *list* is an ordered collection of data items, all of which have the same type. Within a list is the concept of a first item, a last item, and the *n*th item. The items in the list are not required to be unique. For example, a list of the first names of the employees hired in the last year, in order of hire, might be {'Jim', 'Mary', 'Sam', 'Jim', 'John'}.
- **Multisets** A *multiset* is an unordered collection of data items, all of which have the same type. There is no concept of sequencing the items in a multiset; its items have no implied ordering. The items are not required to be unique. The list of employee first names could be considered a multiset if you didn't care about the order of hire: {'Jim', 'Sam', 'John', 'Jim', 'Mary'}.
- **Sets** A *set* is an unordered collection of unique data items, all of which have the same type. As in a multiset, there is no concept of first or last; the set has no implied order. The items must have unique values. The first names in the previous examples wouldn't qualify, but the last names might: {'Johnson', 'Samuels', 'Wright', 'Jones', 'Smith'}.

Example: Collections

- Extend previously defined tables with collections

```
ALTER TABLE REPS
        ADD QTR_TGT LIST(DECIMAL(9,2)); /* four quarterly targets */

ALTER TABLE TECHNICIANS
        ADD PROJECT SET(VARCHAR(15));   /* projects assigned */

ALTER TABLE ENGINEERS
        ADD DEGREES MULTISET(ROW(       /* degree info */
      DEGREE VARCHAR(3),
      SCHOOL VARCHAR(15)));
```

Querying Collection Data

- A limited set of SQL extensions or extend existing SQL concepts to provide simple queries involving collection data
- For more advanced queries, they require you to write stored procedure

```
SELECT EMPL_NUM, NAME
  FROM TECHNICIANS
 WHERE 'bingo' IN (PROJECTS);
```

```
SELECT T2.SCHOOL
  FROM ENGINEERS T1, TABLE(T1.DEGREES) T2
 WHERE EMPL_NUM = 1234;
```

Manipulating Collection Data

```
INSERT INTO TECHNICIANS
     VALUES (1279,
             ROW('Sam', 'R', 'Jones'),
             ROW('164 Elm St.', 'Highland', 'IL', ROW(12345, 6789)),
             SET{'atlas', 'checkmate', 'bingo'});

INSERT INTO ENGINEERS
     VALUES (1281,
             ROW('Jeff', 'R', 'Ames'),
             ROW('1648 Green St.', 'Elgin', 'IL', ROW(12345, 6789)),
             MULTISET{ROW('BS', 'Michigan'),
                      ROW('BS', 'Michigan'),
                      ROW('PhD','Stanford')});
```

Object-relational model of Oracle

Object-relational part of PL/SQL

- Oracle (12c Release) !
- Concepts
  - Abstract data types
    - User-defined named type
    - Internal structure and behaviour
  - Complex structures
  - Functions
  - Subtypes and inheritance
  - Encapsulation

User-defined types (1)

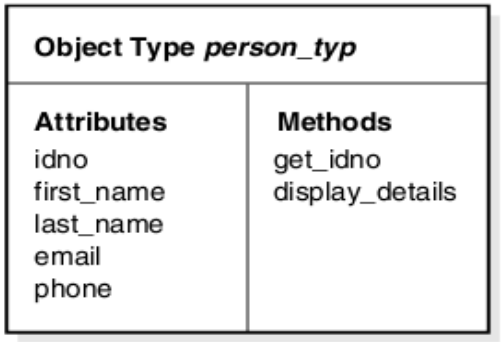
- There is no difference among atributes, functions and virtual attributed from »outside«
- Physical representation can change without affecting the external application
- Private/public attributes and functions
- Creation of instances using constructor

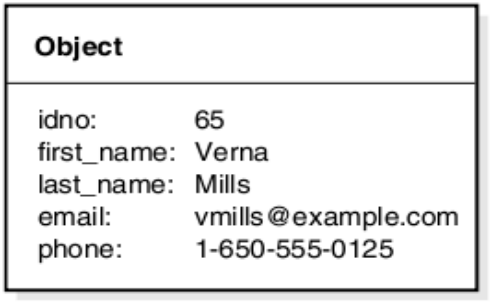
User-defined types (2)

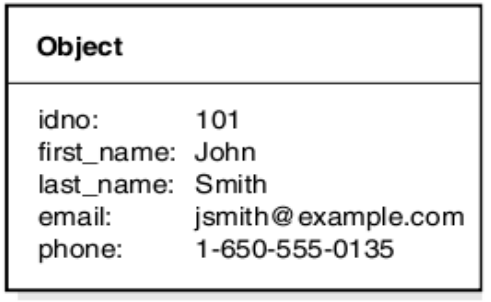
- Column type
  - Text, picture, audio, video, time series, ...
- Type constructors
  - REF
  - VARRAY, NESTED TABLE
- Orthogonality
- Querying complex structures

Object Types

- An object type is a kind of data type.
- You can specify an object ype as the data type of a column in a relational table, and you can declare variables of an object type







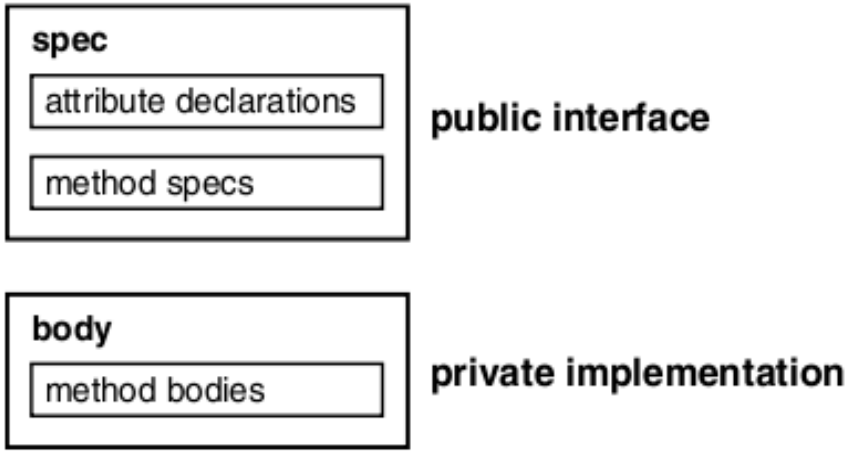
CREATE TYPE

- Object types serve as blueprints or templates that define both structure and behavior

```
CREATE TYPE person_typ AS OBJECT (
   idno NUMBER,
   first_name VARCHAR2(20),
   last_name VARCHAR2(25),
   email VARCHAR2(25),
   phone VARCHAR2(20),
   MAP MEMBER FUNCTION get_idno RETURN NUMBER,
   MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ));
/
CREATE TYPE BODY person_typ AS
   MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
   BEGIN
      RETURN idno;
   END;
   MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
   BEGIN
      -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' || first_name || ' ' || last_name);
      DBMS_OUTPUT.PUT_LINE(email || ' ' || phone);
   END;
END;
```

Object Attributes and Methods

- Object types are composed of attributes and methods:
  - Attributes hold the data about an object
  - Attributes have declared data types which can be other object types
  - Methods are procedures or functions that applications can use to perform operations on the attributes of the object type
  - Methods are optional
  - They define the behavior of objects of that type



Object instances

- Variable of an object type is an instance of the type, or an object
- Instance is a concrete thing, you can assign values to its attributes and call its methods

```
CREATE TABLE contacts (
    contact person_typ,
    contact_date DATE );

INSERT INTO contacts VALUES (
    person_typ (65, 'Verna', 'Mills', 'vmills@example.com', '1-650-555-0125'),
    to_date('24 Jun 2003', 'dd Mon YYYY'));
```

Object Methods

- Object methods are functions or procedures that you can declare in an object type definition to implement behavior that you want objects of that type to perform
- Methods:
  - Member methods
  - Static methods
  - Constructors

```
SELECT c.contact.get_idno() FROM contacts c;
```

How Objects are Stored in Tables

- Objects can be stored in two types of tables:
  - Object tables: store only objects
    - Each row represents an object, which is referred to as a row object
  - Relational tables: store objects with other table data
    - Objects that are stored as columns of a relational table, or are attributes of other objects, are called column objects

How Objects are Stored in Tables

```
CREATE TABLE person_obj_table OF person_typ
```

- Two views of object table:
  - Single-column table, in which each row is object, allowing you to perform object-oriented operations
  - Multi-column table, in which each attribute of the object type occupies a column

```
INSERT INTO person_obj_table VALUES (
person_typ(101, 'John', 'Smith', 'jsmith@example.com', '1-650-555-0135') );

SELECT VALUE(p) FROM person_obj_table p
WHERE p.last_name = 'Smith';

DECLARE
   person person_typ;
BEGIN -- PL/SQL block for selecting a person and displaying details
   SELECT VALUE(p) INTO person FROM person_obj_table p WHERE p.idno = 101;
   person.display_details();
END;
```

Using Object Identifiers to Identify Row Objects

- Object identifiers (OIDs) uniquely identify row objects in object tables
    - You cannot directly access object identifiers, but you can make references (REFs) to the object identifiers and directly access the REFs
- There are two types of object identifiers
    - System-Generated Object Identifiers (default)
    - Primary-Key Based Object Identifiers

References to Row Objects

- A REF is a logical pointer or reference to a row object that you can construct from an object identifier (OID)
  - You can use the REF to obtain, examine, or update the object
  - You can change a REF so that it points to a different object of the same object type hierarchy or assign it a null value

References to Row Objects →

Example

```
CREATE TYPE emp_person_typ AS OBJECT (
   name VARCHAR2(30),
   manager REF emp_person_typ );
/
CREATE TABLE emp_person_obj_table OF emp_person_typ;
/
INSERT INTO emp_person_obj_table VALUES (
    emp_person_typ ('John Smith', NULL));
INSERT INTO emp_person_obj_table
   SELECT emp_person_typ ('Bob Jones', REF(e))
   FROM emp_person_obj_table e
   WHERE e.name = 'John Smith';
```

```
select * from emp_person_obj_table e;

NAME       MANAGER
---------- ----------------------------------------------------
John Smith
Bob Jones 0000220208424E801067C2EABBE040578CE70A0707424E8010
                67C1EABBE040578CE70A0707
```

Topics

- Using Scoped REFs
- Checking for Dangling REFs (IS DANGLING)
- Dereferencing REFs

```
SELECT DEREF(e.manager) FROM emp_person_obj_table e;
```

- Obtaining a REF to a Row Object

```
DECLARE
person_ref REF person_typ;
person person_typ;
BEGIN
   SELECT REF(p) INTO person_ref
   FROM person_obj_table p
   WHERE p.idno = 101;
   select deref(person_ref) into person from dual;
   person.display_details();
END;
```

Collections

- Modeling multi-valued attributes and many-to-many relationships
- Two collection data types:
  - varrays (variable arrays) and
  - nested tables
- Orthogonality
  - use collection types anywhere other data types are used

Nested table

- Unordered set of data elements, all of the same data type
- select, insert, delete, and update in a nested table just as you do with ordinary tables
- To declare nested table types, use the CREATE TYPE ... AS TABLE OF statement
- Elements of a nested table are actually stored in a separate storage table


```
CREATE TYPE people_typ AS TABLE OF person_typ;
```

Nested table

- Using the Constructor Method to Insert Values into a Nested Table

```
CREATE TABLE people_tab (
   group_no NUMBER,
   people_column people_typ ) -- an instance of nested table
NESTED TABLE people_column STORE AS people_column_nt; -- storage table for NT

INSERT INTO people_tab VALUES (
   100,
   people_typ( person_typ(1, 'John Smith', '1-650-555-0135'),
                  person_typ(2, 'Diane Smith', NULL)));
```

Collections → Nested table → Example

```
-- nested table type
CREATE TYPE people_typ AS TABLE OF person_typ;
/
CREATE TCREATE TABLE students (
   graduation DATE,
   math_majors people_typ, -- nested tables (empty)
   chem_majors people_typ,
   physics_majors people_typ)
NESTED TABLE math_majors STORE AS math_majors_nt -- storage tables
NESTED TABLE chem_majors STORE AS chem_majors_nt
NESTED TABLE physics_majors STORE AS physics_majors_nt;
CREATE INDEX math_idno_idx ON math_majors_nt(idno);
CREATE INDEX chem_idno_idx ON chem_majors_nt(idno);
CREATE INDEX physics_idno_idx ON physics_majors_nt(idno);
```

```
iNSERT INTO students (graduation) VALUES ('01-JUN-03');
UPDATE students
SET math_majors =
      people_typ (person_typ(12, 'Bob Jones', '650-555-0130'),
                  person_typ(31, 'Sarah Chen', '415-555-0120'),
                  person_typ(45, 'Chris Woods', '415-555-0124')),
       chem_majors =
       people_typ (person_typ(51, 'Joe Lane', '650-555-0140'),
                   person_typ(31, 'Sarah Chen', '415-555-0120'),
                   person_typ(52, 'Kim Patel', '650-555-0135')),
        physics_majors =
        people_typ (person_typ(12, 'Bob Jones', '650-555-0130'),
        person_typ(45, 'Chris Woods', '415-555-0124'))
WHERE graduation = '01-JUN-03';
SELECT m.idno math_id, c.idno chem_id, p.idno physics_id
FROM students s, TABLE(s.math_majors) m, TABLE(s.chem_majors) c,
        TABLE(s.physics_majors) p;
```

Variable arrays

- A varray is an ordered set of data elements
- All elements of a given varray are of the same data type or a subtype of the declared one
- Each element has an index, which is anumber corresponding to the position of the element in the array
    - The index number is used to access a specific element
- Varrays are stored in columns either as raw values or LOBs

Example

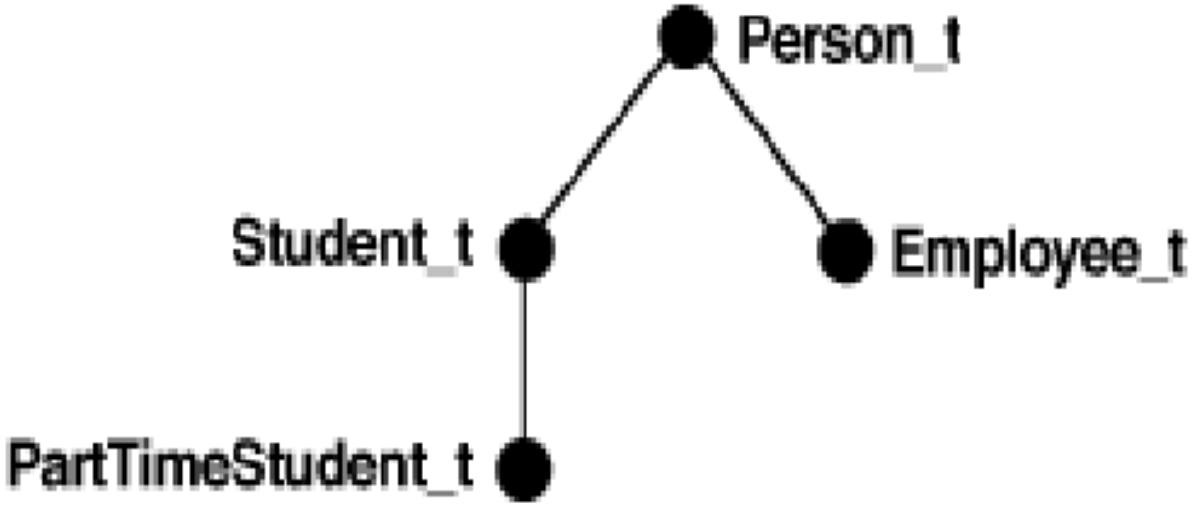
```
CREATE TYPE email_list_arr AS VARRAY(10) OF VARCHAR2(80);
/
CREATE TYPE phone_typ AS OBJECT (
   country_code VARCHAR2(2),
   area_code VARCHAR2(3),
   ph_number VARCHAR2(7));
/
CREATE TYPE phone_varray_typ AS VARRAY(5) OF phone_typ;
/
CREATE TABLE dept_phone_list (
   dept_no NUMBER(5),
   phone_list phone_varray_typ);
INSERT INTO dept_phone_list VALUES (
   100,
   phone_varray_typ( phone_typ ('01', '650', '5550123'),
                     phone_typ ('01', '650', '5550148'),
                     phone_typ ('01', '650', '5550192')));
```

Type Inheritance

- Type inheritance enables you to create type hierarchies
    - A set of successive levels of increasingly specialized subtypes that derive from a common ancestor object type, which is called a supertype
- Derived subtypes inherit the features of the parent object type and can extend the parent type definition
- Type hierarchy provides a higher level of abstraction for managing the complexity of an application model

Type Inheritance

- Two subtypes, Student_t and Employee_t, created under Person_t, and the PartTimeStudent_t, a subtype under Student_t



Inheritable properties

- For an object type to be inheritable, the object type definition must specify that it is inheritable
- Keywords FINAL or NOT FINAL are used for both types and methods

```
CREATE OR REPLACE TYPE person_typ AS OBJECT (
   idno NUMBER,
   name VARCHAR2(30),
   phone VARCHAR2(20),
   FINAL MAP MEMBER FUNCTION get_idno RETURN
NUMBER)
NOT FINAL;
```

Creating parent

Creating a parent or supertype object

```
CREATE OR REPLACE TYPE person_typ AS OBJECT (
   idno NUMBER,
   name VARCHAR2(30),
   phone VARCHAR2(20),
   MAP MEMBER FUNCTION get_idno RETURN NUMBER,
   MEMBER FUNCTION show RETURN VARCHAR2)
NOT FINAL;
/

CREATE OR REPLACE TYPE BODY person_typ AS
   MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
   BEGIN
      RETURN idno;
   END;
   -- function that can be overriden by subtypes
   MEMBER FUNCTION show RETURN VARCHAR2 IS
   BEGIN
      RETURN 'Id: ' || TO_CHAR(idno) || ', Name: ' || name;
   END;
END;
/
```

Creating subtype

- A subtype inherits the following:
  - attributes declared in or inherited by the supertype.
  - methods declared in or inherited by supertype.

- New dept_id, major
- Overrides the show method
- Generalized Invocation

```
CREATE TYPE student_typ UNDER person_typ (
   dept_id NUMBER,
   major VARCHAR2(30),
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2)
NOT FINAL;
/

CREATE TYPE BODY student_typ AS
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
   BEGIN
      RETURN (self AS person_typ).show || ' -- Major: ' || major ;
   END;
END;
/
```

Generalized Invocation

- Mechanism to invoke a method of a supertype or a parent type, rather than the specific subtype member method

```
(SELF AS person_typ).show
```

```
DECLARE
myvar student_typ := student_typ(100, 'Sam', '6505556666', 100, 'Math');
name VARCHAR2(100);
BEGIN
name := (myvar AS person_typ).show; --Generalized invocation
END;
/
```

```
DECLARE
myvar2 student_typ := student_typ(101, 'Sam', '6505556666', 100, 'Math');
name2 VARCHAR2(100);
BEGIN
name2 := person_typ.show((myvar2 AS person_typ)); -- Generalized expression
END;
/
```

Creating subtype

- Type can have multiple child subtypes, and these subtypes can also have subtypes

```
CREATE OR REPLACE TYPE employee_typ UNDER person_typ (
   emp_id NUMBER,
   mgr VARCHAR2(30),
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);
/

CREATE OR REPLACE TYPE BODY employee_typ AS
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
   BEGIN
      RETURN (SELF AS person_typ).show|| ' -- Employee Id: '
                  || TO_CHAR(emp_id) || ', Manager: ' || mgr ;
   END;
END;
/
```

Creating subtype

- Subtype can be defined under another subtype
  - New subtype inherits all the attributes and methods that its parent type has, both declared and inherited

```
CREATE TYPE part_time_student_typ UNDER student_typ (
   number_hours NUMBER,
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);
/

CREATE TYPE BODY part_time_student_typ AS
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
   BEGIN
      RETURN (SELF AS person_typ).show || ' -- Major: ' || major ||
                  ', Hours: ' || TO_CHAR(number_hours);
   END;

END;
/
```

Type Inheritance →

Creating table

```
CREATE TABLE person_obj_table OF person_typ;
INSERT INTO person_obj_table
   VALUES (person_typ(12, 'Bob Jones', '650-555-0130'));
INSERT INTO person_obj_table
   VALUES (student_typ(51, 'Joe Lane', '1-650-555-0140', 12, 'HISTORY'));
INSERT INTO person_obj_table
   VALUES (employee_typ(55, 'Jane Smith', '1-650-555-0144',100, 'Jennifer Nelson'));
INSERT INTO person_obj_table
   VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0135', 14, 'PHYSICS',
20));
```

Creating a table that contains supertype and subtype objects

```
SELECT p.show() FROM person_obj_table p;
The output is similar to:
Id: 12, Name: Bob Jones
Id: 51, Name: Joe Lane -- Major: HISTORY
Id: 55, Name: Jane Smith -- Employee Id: 100, Manager: Jennifer Nelson
Id: 52, Name: Kim Patel -- Major: PHYSICS, Hours: 20
```

Type Inheritance →

NOT INSTANTIABLE

- Type can be defined NOT INSTANTIABLE
  - There will be type hierarchy but we do not need instances
- Method can be defined NOT INSTANTIABLE
  - Method is just a placeholder
  - All subtypes will define their own

```
CREATE OR REPLACE TYPE person_typ AS OBJECT (
   idno NUMBER,
   name VARCHAR2(30),
   phone VARCHAR2(20),
   NOT INSTANTIABLE MEMBER FUNCTION get_idno RETURN NUMBER)
NOT INSTANTIABLE NOT FINAL;
```

Overloading and Overriding Methods

- Subtype can redefine methods it inherits, and it can also add new methods, including methods with the same name.
- Overloading Methods
  - Adding new methods that have the same names as inherited methods to the subtype is called overloading.
- Overriding and Hiding Methods
  - Redefining an inherited method to customize its behavior in a subtype is called overriding, in the case of member methods, or hiding, in the case of static methods
  -

Overloading

- Methods that have the same name, but different signatures are called overloads
- Overloading is useful when you want to provide a variety of ways of doing something
    - Compiler uses the method signatures to determine which method to call

```
CREATE TYPE ellipse_typ AS OBJECT (...,
    MEMBER PROCEDURE calculate(x NUMBER, x NUMBER),
) NOT FINAL;

CREATE TYPE circle_typ UNDER ellipse_typ (...,
    MEMBER PROCEDURE calculate(x NUMBER),
...);
```

Overriding

- Redefining an inherited method to customize its behavior in a subtype is called overriding, in the case of member methods, or hiding, in the case of static methods
  - Unlike overloading, you do not create a new method, just redefine an existing one
  - Methods preserve signatures
  - Method code determined dynamically (statically for static)

```
CREATE TYPE ellipse_typ AS OBJECT (...,
   MEMBER PROCEDURE calculate(),
   FINAL MEMBER FUNCTION function_mytype(x NUMBER)...
) NOT FINAL;

CREATE TYPE circle_typ UNDER ellipse_typ (...,
   OVERRIDING MEMBER PROCEDURE calculate(),
...);
```

Overriding

- Restrictions on Overriding Methods
  - Only methods that are not declared to be final in the supertype can be overridden
  - Order methods may appear only in the root type of a type hierarchy: they may not be redefined (overridden) in subtypes
  - A static method in a subtype may not redefine a member method in the supertype
  - A member method in a subtype may not redefine a static method in the supertype
  - If a method being overridden provides default values for any parameters, then the overriding method must provide the same default values for the same parameters

Dynamic Method Dispatch

- The way that method calls are dispatched to the nearest implementation at run time, working up the type hierarchy from the current or specified type
- ellipse_typ, circle_typ, and sphere_typ might define a calculate() method differently
- The method call works up the type hierarchy: never down
    - Type of object is dynamically determined
    - The nearest signature in type hierarchy is found
    - Method is invoked

Dynamic Method Dispatch

Substitutability →

- Supertype is substitutable if one of its subtypes can substitute or stand in for it in a variable or column whose declared type is the supertype
- In general, types are substitutable. Object attributes, collection elements and REFs are substitutable.

Substituting Types in a Type Hierarchy

- Work with types in a type hierarchy
  - Sometimes you need to work at the most general level, for example, to select or update all persons
  - Other times, you need to address only a specific subtype such as a student, e.g. persons who are not students
- Ability to select all persons and get back not only objects whose declared type is person_typ but also objects whose declared subtype is student_typ or employee_typ is called substitutability

Description

- Supertype is substitutable if one of its subtypes can substitute or stand in for it in a variable or column whose declared type is the supertype
    - Subtypes include all members of supertype
- In general, all types are substitutable
    - Object attributes
    - Collection elements
    - REFs
    - Also, object types

Column and Row Substitutability

- Object type columns and object-type rows in object tables are substitutable

```
CREATE OR REPLACE TYPE person_typ AS OBJECT (
   idno NUMBER,
   name VARCHAR2(30),
   phone VARCHAR2(20))
NOT FINAL;
/
CREATE TYPE student_typ UNDER person_typ (
   dept_id NUMBER,
   major VARCHAR2(30))
NOT FINAL;
/
CREATE TYPE part_time_student_typ UNDER student_typ (
   number_hours NUMBER);
/
CREATE TABLE contacts (
   contact person_typ,
   contact_date DATE );
```

Column and Row Substitutability

- A newly created subtype can be stored in any substitutable tables and columns of its supertype
- To access attributes of a subtype of a row or column's declared type, you can use the TREAT function.

```
INSERT INTO contacts
  VALUES (person_typ (12, 'Bob Jones', '650-555-0130'), '24 Jun 2003' );
INSERT INTO contacts
  VALUES (student_typ(51, 'Joe Lane', '1-650-555-0178', 12, 'HISTORY'), '24 Jun 2003' );
INSERT INTO contacts
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0190', 14, 'PHYSICS', 20),
                '24 Jun 2003' );
```

```
SELECT TREAT(contact AS student_typ).major FROM contacts;
```

Issues of substitutability

- Subtypes with Attributes of a Supertype
- Substitution of REF Columns and Attributes
- Substitution of Collection Elements
- Turning Off Substitutability in a New Table
- Constraining Substitutability
- Modifying Substitutability
- Restrictions on Modifying Substitutability
-

Literature

- Oracle® Database*, Object-Relational Developer's Guide,* 12c Release 1 (12.1), E53277-02
- *Information technology, Database languages, SQL* ISO/IEC JTC 1/SC 32, ANSI, 2007.
- P.Weinberg, J.Groff, A.Oppel. *SQL The Complete Reference 3rd Edition,* McGraw-Hill, 2010.
- Ki-Joon Han, *SQL3 Standardization.*
- H.Garcia-Molina, J.D.Ullman, J.Widom, *Database systems, The Complete Book,* 2nd Edition, McGraw-Hill, 2010.

Literatura (2)

- C.J.Date, H.Darwen, *A guide to the SQL standard* Addison-Wesley, 1994.
- P.Pistor, *SQL3 Standard Suite - An Overview,* DEXA, 1996.
- N.Mattos, *An overview of SQL3 standard*, 1996.
- N.Mattos, L.DeMichiel, *Recent Design Trade-offs in SQL3*, SIGMOD Record, 1996.
- L.Gallagher, *Influencing Database Language Standard,* 1994.