Extended Relational Databases (Object-Relational Databases)

ACS 575: Database Systems

Instructor: Dr. Jin Soung Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

References

- W. Lemanhieu, et al., Principles of Database Management, Ch 9. Extended Relational Databases
- Oracle Object-Relational Developer’s Guide

NOTE

- Extended SQL syntax might vary among different ORDBMSs. but, the core principles are generally similar across systems.
- For specific syntax and features in Oracle ORBMS, consult the Object-Relational Developer’s Guide
  - https://docs.oracle.com/database/121/ADOBJ/toc.htm
  - https://docs.oracle.com/en/database/oracle/oracle-database/19/adobj/index.html

Outline

- Success of the relational model
- Limitations of the Relational Model
- Active RDBMS Extensions
- Object-Relational DBMS

Success of the Relational Model

- Relational model and the RDBMS implements have been very successful for managing well-structured numerical and alphanumerical data
- Mathematical simplicity and soundness of relational model
- The two building blocks are tuples and relations
  - A tuple is a composition of values of attribute types
  - A relation is a mathematical set of tuples describing similar entities.
- The relational model is also referred to as a value based model (↔ identity based OO model)
  - It uses the values within tables to establish relationships among entities.
- SQL is an easy to learn, descriptive and non-navigational data manipulation language (DML)
  - It allows users to specify what data to operate on (e.g., retrieve, insert, update, delete) without needing to define the specific path or navigation through the data structure to perform these operations.

Limitations of the Relational Model

- Relational model requires all relations to be normalized.
  - Data about entities can be fragmented across multiple relations
  - Relations can only be connected by using primary-foreign key relationships. - A value based model
  - Expensive joins make a heavy burden on the performance of database application
- Specialization, categorization and aggregation cannot be directly supported.
  - Implementing these concepts often requires additional tables and complex queries

Limitations of the Relational Model (cont.)

- Relations can only store atomic values. Composite attribute type cannot be directly modeled in a relation. Multi-valued attribute types cannot be also supported.
- Not possible to model behavior or stored functions which enable reducing network traffic and unnecessary replication of code
- Complex objects (e.g., multimedia, geospatial information systems, genomics, time series, the internet of things, etc.) are difficult to handle
  - Poor support for audio, video, text which are often encountered in modern-day applications

Outline

□ Success of the relational model

□ Limitations of the Relational Model

☞ **Active RDBMS Extensions**

□ Object-Relational DBMS

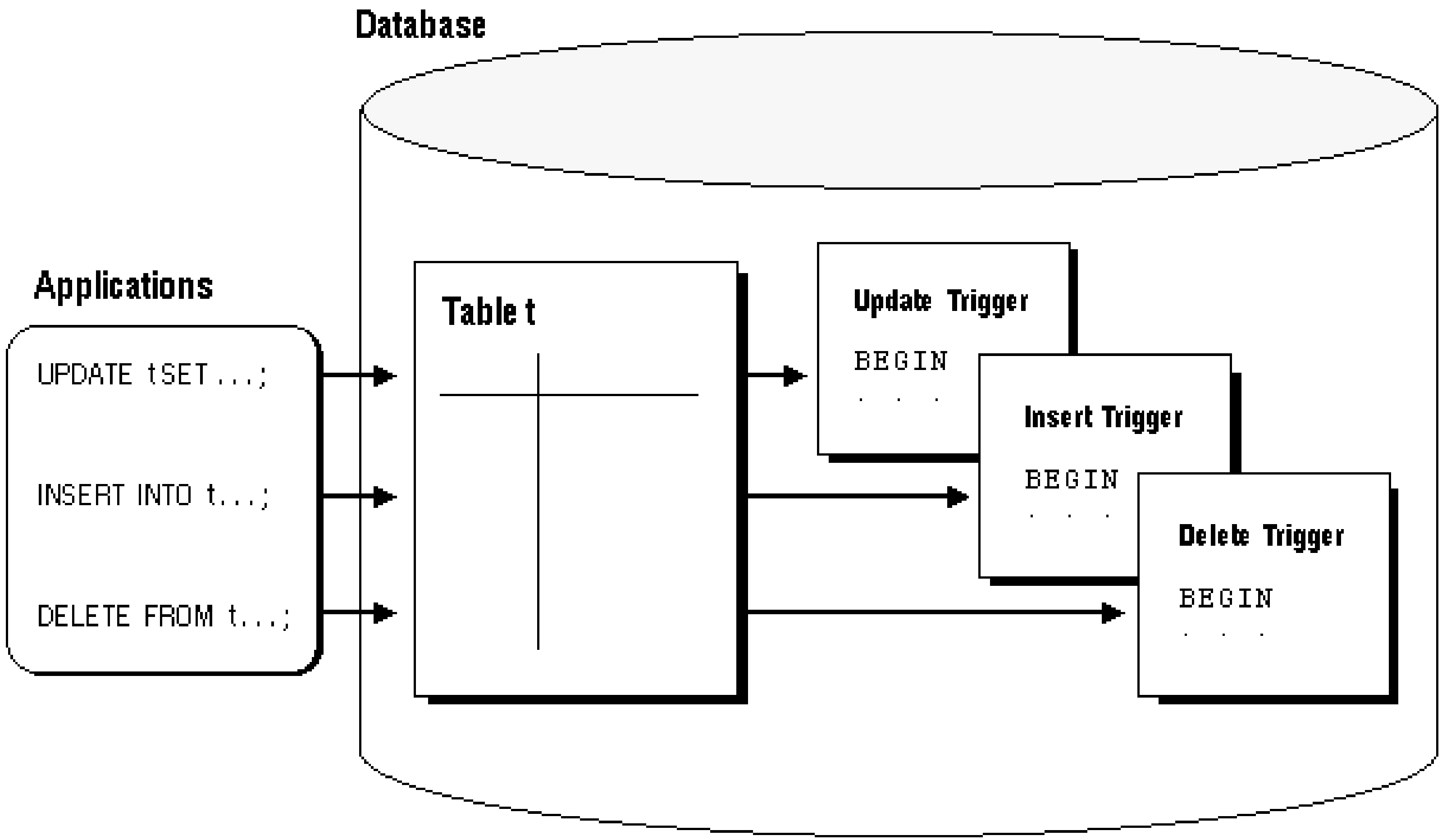
Active RDBMS Extensions

- Traditional RDBMSs are **passive**
  - They only execute transactions explicitly invoked by users and/or applications
- Modern day RDBMSs are **active**
  - They can autonomously take initiative for action if specific situation occur.
- Two key components of active RDBMSs are triggers and stored procedures

Triggers

- A **trigger** is a piece of SQL code consisting of declarative and/or procedural instructions and stored in the catalog of the RDBMS
- A trigger is automatically activated and run by the RDBMS whenever a specific event (e.g., insert, update, delete) occurs and a specific condition is evaluated as true.
- It is used to enforce complex semantic constrains that cannot be captured in the basic relational model.
- In contrast to CHECK constraints, triggers can also reference attribute types in other tables

Triggers


Database
Applications
UPDATE t SET . . . ;
INSERT INTO t . . . ;
DELETE FROM t . . . ;
Table t
Update Trigger
BEGIN
. . .
Insert Trigger
BEGIN
. . .
Delete Trigger
BEGIN
. . .

TRIGGER General Syntax

- **CREATE TRIGGER** trigger_name

  {BEFORE|AFTER|INSTEAD OF} {INSERT|DELETE|UPDATE} ON

  table_name

  {FOR EACH {ROW|STATEMENT} [WHEN (search condition**)]**

  <triggered statements here usually using procedure SQL language>;

- Useful if there is a trigger with that name and you want to modify the trigger

  CREATE OR REPLACE TRIGGER trigger_name

  ….

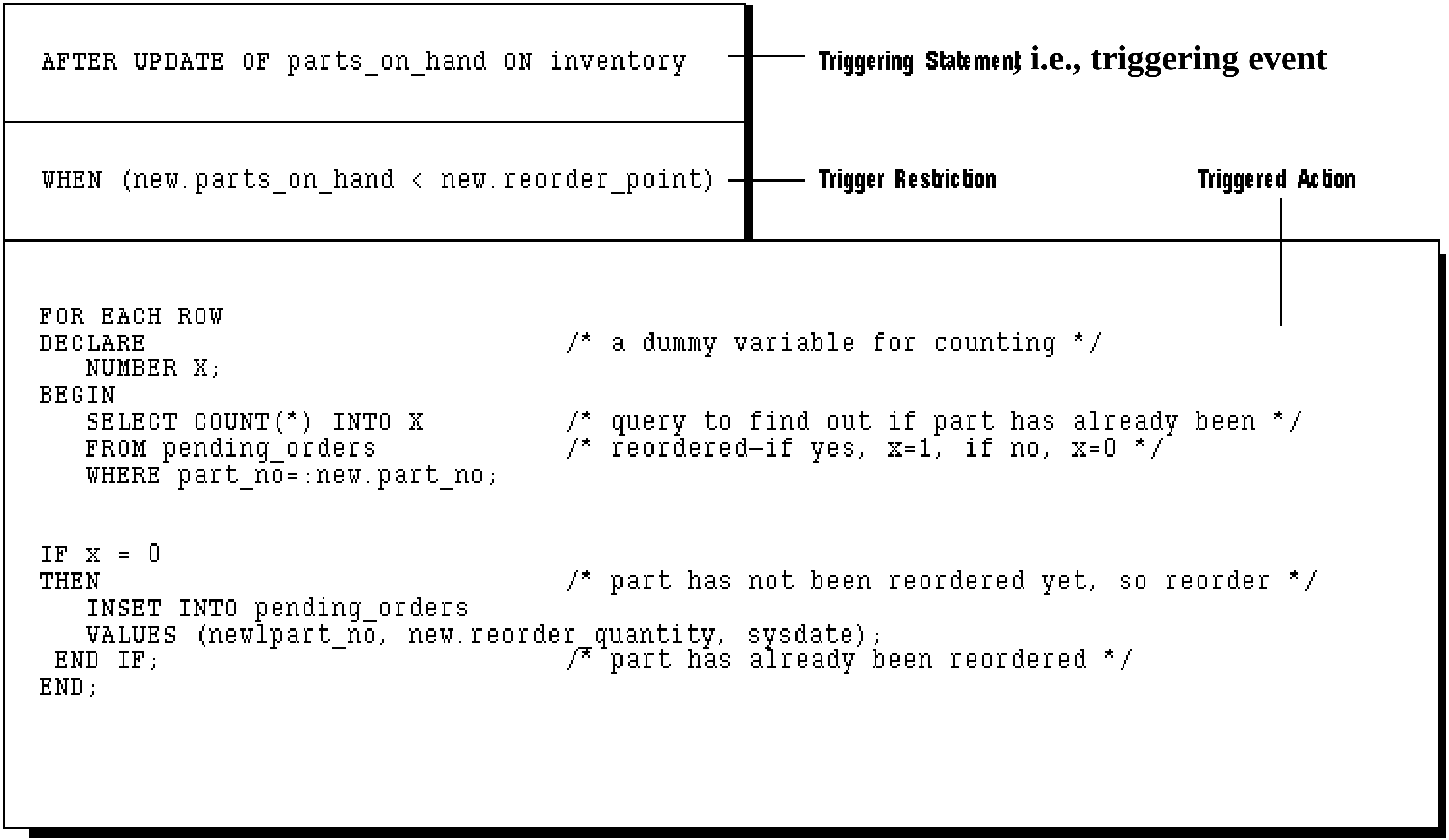
- **DROP TRIGGER** trigger_name**;**

Trigger Parts: Event-Condition-Action Rules

- Another name for “trigger” is *ECA rule*, or event-condition-action rule.
- **Event** (Triggering event)
  - typically a type of database modification such as INSERT, UPDATE or DELETE
  - activates the trigger
- **Condition** (Trigger restriction)
  - tests whether the trigger should run
  - Any SQL boolean-valued expression.
- **Action** (Trigger action) what happens if the trigger runs
  - Any SQL (procedure SQL) statements.

Example

REORDER Trigger



```
AFTER UPDATE OF parts_on_hand ON inventory
```

```
WHEN (new.parts_on_hand < new.reorder_point)
```

```
FOR EACH ROW
DECLARE                           /* a dummy variable for counting */
   NUMBER X;
BEGIN
   SELECT COUNT(*) INTO X         /* query to find out if part has already been */
   FROM pending_orders            /* reordered-if yes, x=1, if no, x=0 */
   WHERE part_no=:new.part_no;

IF x = 0
THEN                              /* part has not been reordered yet, so reorder */
   INSET INTO pending_orders
   VALUES (newlpart_no, new.reorder_quantity, sysdate);
 END IF;                          /* part has already been reordered */
END;
```

BEFORE vs. AFTER Triggers

- You can specify the *trigger timing*.
- **BEFORE Triggers**
  - execute the trigger action before the triggering statement.
- **AFTER Triggers**
  - execute the trigger action after the triggering statement is executed.

Example

EMPLOYEE(SSN, ENAME, SALARY, BONUS, JOBCODE, *DNR*)

DEPARTMENT(DNR, DNAME, TOTAL-SALARY, *MGNR*)

```
CREATE TRIGGER SALARYTOTAL
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
WHEN (NEW.DNR IS NOT NULL)
UPDATE DEPARTMENT
SET TOTAL-SALARY = TOTAL-SALARY + NEW.SALARY
WHERE DNR = NEW.DNR
```

After Trigger!

(First insert the employee tuple(s) and then execute the trigger body to update the department’s total salary)

Triggering event:

```
INSERT INTO employee (ssn, salary, dnr)
VALUES (7777, 10000,  10);
```

Example

EMPLOYEE(SSN, ENAME, SALARY, BONUS, JOBCODE, *DNR*)

DEPARTMENT(DNR, DNAME, TOTAL-SALARY, *MGNR*)

```
CREATE TRIGGER SALARYTOTAL
AFTER DELETE ON EMPLOYEE
FOR EACH ROW
WHEN (OLD.DNR IS NOT NULL)
UPDATE DEPARTMENT
SET TOTAL-SALARY = TOTAL-SALARY - OLD.SALARY
WHERE DNR = OLD.DNR
```

Triggering event:

```
DELETE FROM employee
WHERE ssn = 7777;
```

First delete the employee tuple(s) and then execute the trigger body to update the department's total salary

Example

EMPLOYEE(SSN, ENAME, SALARY, BONUS, JOBCODE, *DNR*)

WAGE(JOBCODE, BASE_SALARY, BASE_BONUS)

```
CREATE TRIGGER WAGEDEFAULT
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEWROW
FOR EACH ROW
SET (SALARY, BONUS) = (SELECT BASE_SALARY, BASE_BONUS
                       FROM WAGE
                       WHERE JOBCODE = NEWROW.JOBCODE)
```

Before Trigger!

(First retrieve the BASE_SALARY and BASE_BONUS value for each new employee tuple and then insert the entire tuple in the EMPLOYEE Table.)

Advantages of Triggers

- Automatic monitoring and verification in case of specific events or situations
- Modelling extra semantics and/or integrity rules without changing the user front-end or application code
- Assign default values to attribute types for new tuples
- Synchronic updates in case of data replication;
- Automatic auditing and logging which may be hard to accomplish in any other application layer
- Automatic exporting of data

Disadvantages of Triggers

- Hidden functionality, which may be hard to follow-up and manage
- Cascade effects leading up to an infinite loop of a trigger triggering another trigger etc.
- Uncertain outcomes if multiple triggers for the same database object and event are defined
- Deadlock situations
- Debugging complexities since they don’t reside in an application environment
- Maintainability and performance problems

Reference for Database Trigger

- For triggers in Oracle, refer to
  - https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/CREATE-TRIGGER-statement.html
  - https://docs.oracle.com/en/database/oracle/oracle-database/21/tdddg/using-triggers.html#GUID-3744214A-861D-4C59-AD2D-95840B5B0871
- For triggers in MS SQL Sever, refer to

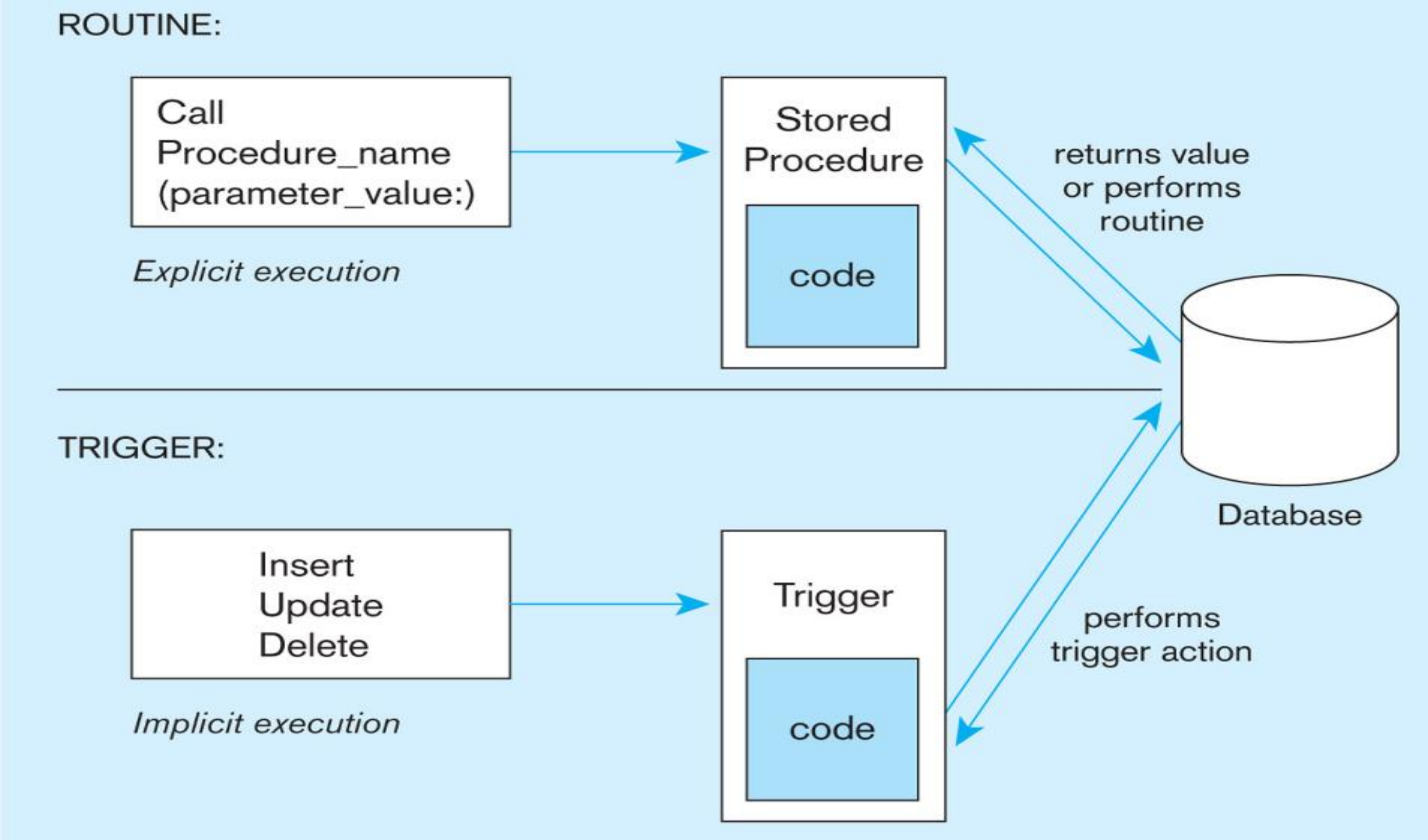
  https://learn.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16
- Database venders use their own procedure SQL. Their syntax is slightly different.
  - Oracle's PL/SQL: https://docs.oracle.com/database/121/LNPLS/toc.htm
  - MS SQL Server's Transact-SQL:

    https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16

Stored Procedures

- A **stored procedure** is a piece of SQL code consisting of declarative and/or procedural instructions and stored in the catalog of the RDBMS
- Needs to be invoked explicitly

Procedure vs. Trigger



CREATE Stored Procedure/Functions Syntax

Stored procedure:

```
CREATE PROCEDURE name (p1, p2, …, pn)
    local variable declarations
    procedure code; -- usually implemented using procedure SQL language
```

Stored Function:

```
CREATE FUNCTION name (p1, …, pn)
RETURN sqlDataType
    local variable declarations
    function code;
    return value;
```

Example of Oracle Stored Function

A function `rateSailor` returns the rating of a given sailor. If the number of reservations is greater than 10, return rating 1, otherwise, rating 0.

```
CREATE FUNCTION rateSailor (IN sailorId NUMBER)
RETURN NUMBER
AS
     rating NUMBER;
     numRes NUMBER;
BEGIN
    SELECT COUNT(*)  INTO numRes
    FROM Reserves R
    WHERE R.sid = sailorId;
    IF (numRes > 10) THEN rating =1;   -- Oracle PL/SQL syntax
    ELSE rating = 0;
    END IF;
    RETURN rating;
END;
```

Calling Stored Procedures

```
CREATE PROCEDURE REMOVE-EMPLOYEES
(DNR-VAR IN CHAR(4), JOBCODE-VAR IN CHAR(6)) AS
BEGIN
  DELETE FROM EMPLOYEE
  WHERE DNR = DNR-VAR AND JOBCODE = JOBCODE-VAR;
END
```

In application program

```
import java.sql.CallableStatement;
…
CallableStatement cStmt
   = conn.prepareCall("{call REMOVE-EMPLOYEES(?, ?)}");
cStmt.setString(1, "D112");
cStmt.setString(2, "JOB124");
```

Advantages and Disadvantages of Stored Procedures

- **Advantages**
  - Similar to OODBMSs, they store behavior in the database
  - They can reduce network traffic
  - They can be implemented in an application-independent way
  - They improve data and functional independence
  - They can be used as a container for several SQL instructions that logically belong together
  - They are easier to debug in comparison to triggers
- **Disadvantages**
  - Like triggers, the main disadvantage is their maintainability.

Outline

- □ Success of the relational model
- □ Limitations of the Relational Model
- □ Active RDBMS Extensions
- ☞ **Object-Relational DBMS**

DBMS and OO Paradigm

- The object-oriented (OO) paradigm was first introduced by programming languages such as C++ and Smalltalk.
- **Object-Oriented DBMSs (OODBMSs)** originated as extensions of OO programming languages
- They were then gradually extended with database facilities such as querying, concurrency control and transaction management.
  - OODBMSs store persistent objects in a transparent way with no need to map to an underlying relational structure
  - OODBMSs guarantee the ACID (Atomicity, Consistency, Isolation, Durability) properties of DBMS
- However, OODBMSs are perceived as very complex to work with (e.g., no good standard DML such as SQL), although offer several advantages such as storing complex objects and relationships in a transparent way

Object-Relational RDBMS Extensions

- **Object-Relational DBMSs (ORDBMSs)** keep the relation as the fundamental building block and SQL as the core DDL/DML, but with the following OO extensions:
  - User-Defined Types (UDTs)
  - User-Defined Functions (UDFs)
  - Inheritance
  - Behavior
  - Polymorphism
  - Collection types
  - Large objects (LOBs)

Outline

- Success of the relational model
- Limitations of the Relational Model
- Active RDBMS Extensions
- Object-Relational DBMS
  - **UDTs**
  - UDFs
  - Inheritance
  - Polymorphism, Overloading, Overriding
  - Collection Types

User-Defined Types (UDTs)

- Standard SQL data types are `CHAR, VARCHAR, INT, FLOAT, DOUBLE, DATE, TIME, BOOLEAN`, etc.
- These standard data types are insufficient to model complex objects
- **User-Defined Types (UDT)** define customized data types with specific properties

UDTs

- General Syntax

```
CREATE TYPE <typename> AS (
  <list of elements, as in CREATE TABLE> );
```

- Example

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

- In Oracle, add "**OBJECT**" as in CREATE … AS OBJECT.

```
CREATE TYPE <typename> AS OBJECT (
        <list of elements, as in CREATE TABLE>
);
```

Constructor, Generator & Mutator

- Each UDT has a *type constructor* of the same name that wraps objects of that type.
- In SQL-99, each attribute of a UDT has *generator* method (get the value) and *mutator* method (change the value) of the same name as the attribute.
  - The generator for A takes no argument, as A().
  - The mutator for A takes a new value as argument, as A(*v*) where *v* is a value.

UDT General Usage as Table Type

- **UDT usage as table type**

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

CREATE TABLE Bars OF BarType;

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

SELECT b.name, b.addr  -- generators (/ get functions)
FROM Bars b;
```

UDT General Usage as Column Type

- **UDT usage as column type**

```
CREATE TYPE BeerType AS OBJECT(
        name    CHAR(20),
        manf    CHAR(20)
);

CREATE TABLE Drinkers (
        name      CHAR(30),
        addr      CHAR(50),
        favBeer   BeerType
);

INSERT INTO Drinkers
VALUES ('Smith', '2101 Coliseum Blvd, Chicago,
        4444', Beertype ('Miler', 'OBBBB') );
```

Different Types of UDT

- **Distinct data types** extend existing SQL data types
- **Opaque data types** define entirely new data types
- **Unnamed row types** use unnamed tuples as attribute values
- **Named row types** use named tuples as attribute values
- **Table data types** define tables as instances of table types

NOTE: Not all ORDMBS products support every type of User-Defined Type (UDT). Verify which UDTs are supported by your specific ORDBMS.

UDT - Distinct Data Type

- A **distinct data type** is a user-defined data type which specializes a standard, built-in SQL data type.
- Define data types

```
CREATE DISTINCT TYPE us-dollar AS DECIMAL(8,2)
CREATE DISTINCT TYPE euro AS DECIMAL(8,2)
```

- Usage of data types

```
CREATE TABLE account(
 accountno SMALLINT PRIMARY KEY NOT NULL,
 …
 amount-in-dollar us-dollar,
 amount-in-euro euro
);
```

UDT - Distinct Data Type

- Once a distinct data type has been defined, the ORDBMS will automatically create two casting functions:
  - A function to cast or map the values of the user-defined type to the underlying, built-in type
  - A function to cast or map the built-in type to the user-defined type
- Example

```
CREATE TABLE account(
   accountno SMALLINT,
    …
   amount-in-euro euro);
```

```
SELECT *
FROM account
WHERE amount-in-euro > 1000
```

ERROR! – A type incompatibility

```
SELECT *
FROM account
WHERE amount-in-euro > euro(1000)
```

UDT - Opaque Data Type

- An **opaque data type** is an entirely new, user-defined data type, not based upon any existing SQL data type.
  - E.g., data types for image, audio, video, fingerprints, text, spatial data, RFID tags, QR codes, etc.
- Define data types

```
CREATE OPAQUE TYPE image AS <…>
CREATE OPAQUE TYPE fingerprint AS <…>
```

- Usage of data type

```
CREATE TABLE employee
( ssn SMALLINT NOT NULL,
  name CHAR(25) NOT NULL,
  …
  empFingerprint fingerprint,
  photograph image);
```

UDT - Unnamed Row Type

- An **unnamed row type** includes a composite data type in a table by using the keyword `ROW`.
- It consists of a combination of data types such as built-in types, distinct types, opaque types, etc.

```
CREATE TABLE employee
( ssn  SMALLINT NOT NULL,
  name ROW(fname CHAR(25), lname CHAR(25)),
  address ROW(streetAddress CHAR(20) NOT NULL,
              zipCode CHAR(8),
              city    CHAR(15) NOT NULL),
  …
  empFingerprint fingerprint,
  photograph image)
```

UDT - Named Row Type

- A **named row type** is a user-defined data type which groups a coherent set of data types into a new composite data type and assigns a meaningful name to it
- The type can be used in table definitions, queries, or anywhere else a standard SQL data type can be used

Example: Named Row Type

General syntax

```
CREATE ROW TYPE address AS
(streetAddress CHAR(20),
 zipCode CHAR(8),
 city CHAR(15) NOT NULL);

CREATE TABLE employee
 (ssn SMALLINT NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress address,
  …
  empFingerprint fingerprint,
  photograph image);
```

Oracle syntax

```
CREATE TYPE address AS OBJECT
(streetAddress CHAR(20),
 zipCode CHAR(8),
 city CHAR(15));

CREATE TABLE employee
 (ssn NUMBER NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress address,
 …
  empFingerprint fingerprint,
  photograph image);
```

Example: Named Row Type (cont.)

```
SELECT lname, empaddress
FROM employee
WHERE empaddress.city = 'LEUVEN'
```

```
SELECT lname, empaddress
FROM employee E1
WHERE E1.empaddress.city = 'LEUVEN'
```

Oracle requires a table alias.

```
SELECT E1.lname, E1.empaddress
FROM employee E1, employee E2
WHERE E1.empaddress.city = E2.empaddress.city
AND E2.ssn = '123456789'
```

UDT - Table Data Type

- A **table data type** (or typed table) defines the type of a table.
  - Similar to a class in OO

```
CREATE TYPE employeetype AS OBJECT
( ssn SMALLINT,
  fname CHAR(25),
  lname CHAR(25),
  empAddress address
  …
  empFingerprint fingerprint,
  photograph image
)

CREATE TABLE employee OF TYPE employeetype;
```

Examples: Usage of Table Data Type

- Example

```
CREATE TABLE employee OF TYPE employeetype
PRIMARY KEY (ssn);

CREATE TABLE ex-employee OF TYPE employeetype
PRIMARY KEY (ssn);
```

- Oracle Examples

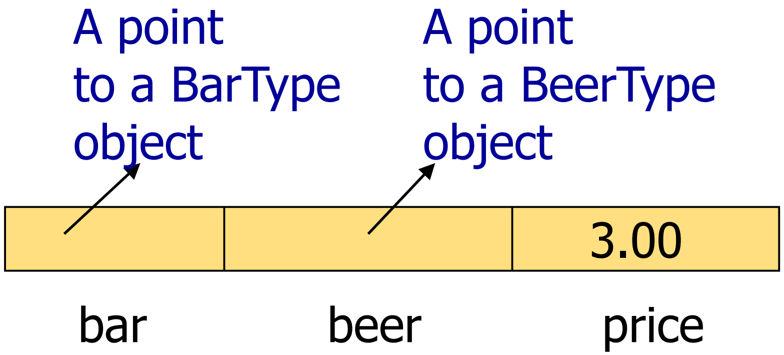
```
CREATE TABLE employee OF employeetype
(ssn PRIMARY KEY);

CREATE TABLE ex-employee OF employeetype
(ssn PRIMARY KEY);
```

Object Identifiers Using Reference Types

- 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*.
- The **reference** represents the ORDBMS counterpart of OIDs in OODBMSs
- Unlike OIDs, the reference can be explicitly requested and visualized to the user.

```
CREATE TYPE MenuType AS
( bar     REF BarType,
  beer    REF BeerType,
  price   float
);
```

MenuType objects look like:



Example

General syntax

```
CREATE TYPE departmenttype
( dnr smallint not null,
  dname char(25) not null,
  dlocation address
  manager REF(employeetype));
```

Oracle syntax

```
CREATE TYPE departmenttype
( dnr number not null,
  dname char(25) not null,
  dlocation address
  manager REF employeetype );
```

- The manager attribute type will contain a reference or pointer to an employee tuple of a table whose type is EMPLOYEE.
- This assumes the ORDBMS supports row identifications.

Following REF

- The reference can be replaced by the actual data it refers to by means of the **DEREF**(from dereferencing) function
- Oracle example:

```
SELECT DEREF(ss.beer).name,
   DEREF(ss.beer).manufacture
FROM Sells ss
WHERE DEREF(ss.bar).name
    = 'Joe''s Bar';
```

Simply, a dot can be used for dereferencing

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

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

CREATE TYPE BeerType AS OBJECT (
        name        CHAR(20),
        manufacture       CHAR(20));

CREATE TYPE MenuType AS OBJECT(
        bar       REF  BarType,
        beer      REF  BeerType,
        price   FLOAT);

CREATE TABLE Sells OF MenuType;
```

Examples with ORACLE

```
CREATE TYPE address_obj AS OBJECT
(street varchar2 (20),   city varchar2 (20),   state char(2),  zip char(5) );
CREATE TABLE address_table OF address_obj;
INSERT INTO address_table VALUES (address_obj('1 A St.', 'Mobile', 'AL', '36608'));

CREATE TABLE client (name varchar2(20),
        address    REF address_obj SCOPE IS address_table, -- to reference a specific object table
        Permanent_add    address_obj );
INSERT INTO client(name, address)
SELECT 'Walsh', ref(aa) FROM address_table aa WHERE aa.zip='36608';


SELECT * FROM client;
```

```
   NAME          ADDRESS
---------------------------------------------------------------------
   Walsh       C##DBUSER.ADDRESS_OBJ('1 A St.','Mobile','AL','36608')
```

```
SELECT c.name, DEREF(c.address).city  FROM client c;
```

```
  NAME     ADDRESS.CITY
-------------- --------------------
  Walsh         Mobile
```

Alternatively, SELECT c.name, c.address.city FROM client c;

Outline

- Success of the relational model
- Limitations of the Relational Model
- Active RDBMS Extensions
- Object-Relational DBMS
  - UDTs
  - ☞ **UDFs**
  - Inheritance
  - Polymorphism, Overloading, Overriding
  - Collection Types

User-Defined Functions (UDFs)

- Every RDBMS comes with a set of built-in functions, e.g., `MIN(), MAX(), AVG(),` etc.
- **User-Defined Functions (UDFs)** allow users to extend these by explicitly defining their own functions
- Every UDF consists of
  - name
  - input and output arguments
  - implementation

User-Defined Functions (UDFs)

- UDFs are stored in the ORDBMS and hidden from the applications
- UDFs can be overloaded
- Two types in implementing UDFs
  - sourced functions
  - external functions

UDF with Sourced Function

- UDF which is based on an existing, built-in function

```
CREATE DISTINCT TYPE MONETARY AS DECIMAL(8,2);
```

```
CREATE FUNCTION avg(monetary)
RETURNS MONETARY
SOURCE AVG(DECIMAL(8,2));


SELECT dnr, avg(salary)
FROM employee
GROUP BY dnr;
```

```
CREATE TABLE employee
( ssn SMALLINT NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress  address,
  salary monetary,
  …
  empfingerprint fingerprint,
  photograph image);
```

UDF with External function

- **External functions** are written in an external host language
  - Python, C, Java, etc.
- Can return a single value (scalar) or table of values

Interface of UDF

- ORDBMS can include the signature or **interface** of a method in the definitions of data types and tables - **Information hiding**

```
CREATE TYPE type-name (
   <list of component attributes and their types>
   <declaration of functions (methods)>
);
```

Example: Interface of UDF

```
CREATE TYPE employeetype
( ssn SMALLINT NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress INTERNATIONAL_ADDRESS,
  …
  empfingerprint FINGERPRINT,
  photograph IMAGE,
  FUNCTION age(employeetype) RETURNS integer);

CREATE TABLE employee OF TYPE employeetype
PRIMARY KEY (SSN);

SELECT ssn, fname, lname, photograph
FROM employee
WHERE age = 60
```

Example with Oracle

```
CREATE TYPE demo_typ2 AS OBJECT
( a1 NUMBER,
  MEMBER FUNCTION get_square RETURN NUMBER);

CREATE TYPE BODY demo_typ2 IS
  MEMBER FUNCTION get_square
  RETURN NUMBER
  IS
  x NUMBER;
  BEGIN
    SELECT c.col.a1*c.col.a1
    INTO x
    FROM demo_tab2 c;
    RETURN (x);
  END;
END; /
```

```
CREATE TABLE demo_tab2 (
 col  demo_typ2
);

INSERT INTO demo_tab2
VALUES (demo_typ2(2));

SELECT t.col.get_square()
FROM demo_tab2 t;
```

```
 T.COL.GET_SQUARE()
------------------------------
             4
```

Outline

- Success of the relational model
- Limitations of the Relational Model
- Active RDBMS Extensions
- Object-Relational RDBMS
  - UDTs
  - UDFs
  - ☞ **Inheritance**
  - Polymorphism, Overloading, Overriding
  - Collection Types

Inheritance

- ORDBMS extends an RDBMS by providing explicit **support for inheritance**, both at
  - the level of a data type
  - the level of a typed table

Inheritance at Data Type Level

- Child data type inherits all the properties of a parent data type and can then further specialize it by adding specific characteristics

```
CREATE ROW TYPE address AS
( streetAddress CHAR(20) NOT NULL,
  zipCode CHAR(8),
  city CHAR(15) NOT NULL
);

CREATE ROW TYPE international_address AS
( country CHAR(25) NOT NULL
) UNDER address;
```

Inheritance at Data Type Level (cont.)

```
CREATE TABLE employee
( ssn SMALLINT NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress international_address,
  salary monetary,
  …
  empfingerprint fingerprint,
  photograph image
);

SELECT fname, lname, empaddress
FROM employee
WHERE empaddress.country = ‘USA'
AND empaddress.city LIKE ‘Fort%'
```

Inheritance at Table Type Level

- We can also apply the concept of inheritance to table type.

```
CREATE TYPE employeetype
( ssn SMALLINT NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress INTERNATIONAL_ADDRESS
  …
  empfingerprint FINGERPRINT,
  photograph IMAGE)

CREATE TYPE engineertype AS
  (degree CHAR(10) NOT NULL,
  license CHAR(20) NOT NULL) UNDER employeetype

CREATE TYPE managertype AS
  (startdate DATE,
   title CHAR(20)) UNDER employeetype
```

Inheritance at Table Type Level (cont.)

- Create tables using various table types.

```
CREATE TABLE employee OF TYPE employeetype
PRIMARY KEY (SSN);
CREATE TABLE engineer OF TYPE engineertype
UNDER employee;
CREATE TABLE manager OF TYPE managertype UNDER employee;
```

```
SELECT ssn, fname, lname, startdate, title
FROM manager
```

```
SELECT ssn, fname, lname
FROM employee
```

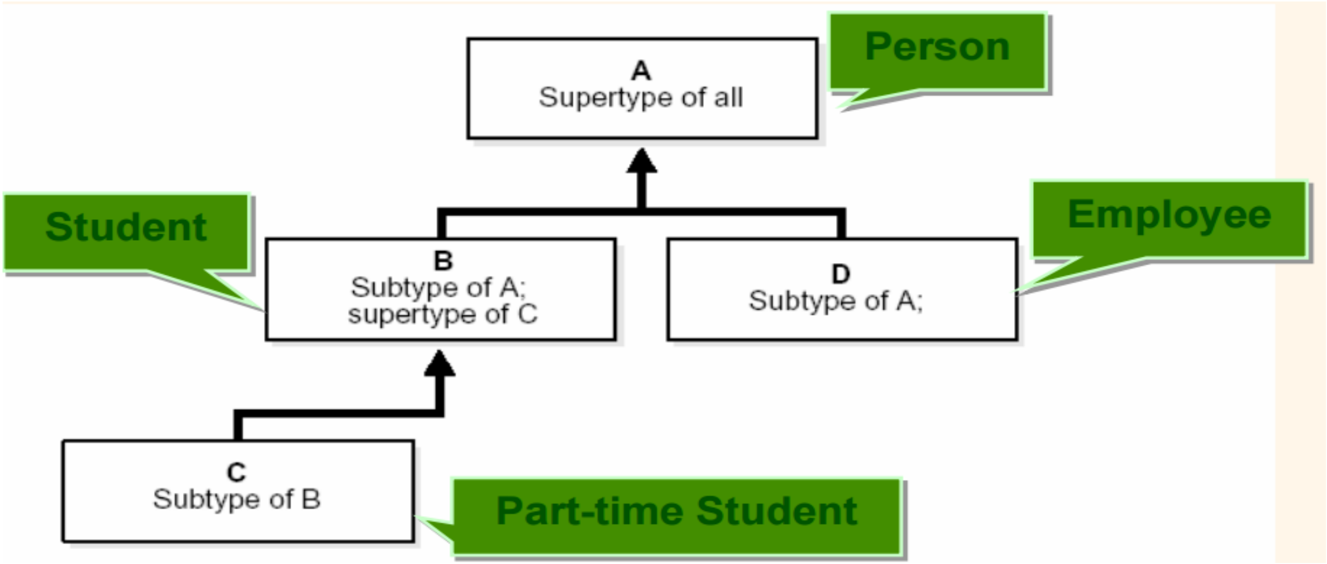
← **Tuples added to a sub table are automatically visible to queries on the super table**

This query retrieves all employees, including the managers and engineers.

```
SELECT ssn, fname, lname
FROM ONLY employee
```

← **Exclude the sub tables**

Example with Oracle



```
CREATE TYPE person_t As OBJECT
(name VARCHAR2(100),  ssn NUMBER) NOT FINAL;
CREATE TYPE employee_t UNDER  person_t
(department_id NUMBER,  salary NUMBER)  NOT FINAL;
CREATE TYPE part_time_emp_t UNDER employee_t
(num_hrs  NUMBER) ;
```

Outline

- Success of the relational model
- Limitations of the Relational Model
- Active RDBMS Extensions
- Object-Relational RDBMS
    - UDTs
    - UDFs
    - Inheritance
    - ☞ **Polymorphism, Overloading, Overriding**
    - Collection Types

Polymorphism

- A subtype inherits both the attribute types and functions of its supertype
- A subtype can also override functions to provide more specialized implementations
- **Polymorphism**: same function call can invoke different implementations

Polymorphism

```
CREATE FUNCTION total_salary(employee e)
RETURNING INT
AS SELECT e.salary


CREATE FUNCTION total_salary(manager m)
RETURNING INT
AS SELECT m.salary + <monthly_bonus>
```

← This query will retrieve **the TOTAL_SALARY of managers.**

```
SELECT total_salary FROM employee
```

← This query will retrieve the TOTAL_SALARY of **all employees, both managers and non-managers.**

Overloading

- **Overloading** functions(or subprograms)
  - You can use the same name for several different subprograms (functions) as long as their formal parameters differ in number, order or datatype family.
  - Overloading is a type of **polymorphism**
- Example: Define the inherited method `Enlarge()` to deal with different types of input parameters.

```
CREATE TYPE Shape_typ AS OBJECT (...,
    MEMBER PROCEDURE Enlarge(x NUMBER),
    ...) NOT FINAL;
CREATE TYPE Circle_typ UNDER Shape_typ (...,
    MEMBER PROCEDURE Enlarge(x CHAR(1))  );
```

Oracle Syntax

Oracle Example – Overloading

```
DECLARE
  TYPE DateTabTyp IS TABLE OF DATE INDEX BY PLS_INTEGER;
  TYPE NumTabTyp IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  hiredate_tab DateTabTyp;                 -- PLS_INTEGER type is a type for
  sal_tab NumTabTyp;                       storing singed integers
  PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
    BEGIN
      FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
    END LOOP;
  END initialize;
  PROCEDURE initialize (tab OUT NumTabTyp, n INTEGER) IS
  BEGIN
    FOR i IN 1..n LOOP
      tab(i) := 0.0;
    END LOOP;
  END initialize;
BEGIN
  initialize(hiredate_tab, 50); -- calls first (DateTabTyp) version
  initialize(sal_tab, 100); -- calls second (NumTabTyp) version
END; /
```

Overriding

- **Overriding** means having two methods with the same method name and parameters (i.e., *method signature*).
  - One of the methods is in the parent data type and the other is in the child data types.
  - Overriding allows a child type to provide a specific implementation of a method that is already provided its parent class.
- Example: Redefine an inherited method `Area()` to make it do something different

```
CREATE TYPE Shape_typ AS OBJECT (...,
   MEMBER PROCEDURE Area(),
   FINAL MEMBER FUNCTION id(x NUMBER)...
) NOT FINAL;
```

Oracle syntax

```
-- Circle_typ  specializes Shape_typ
CREATE TYPE Circle_typ UNDER Shape_typ (...,
   OVERRIDING MEMBER PROCEDURE Area(),      ...);
```

Oracle Example - Overriding

```
CREATE TYPE super_t AS OBJECT
   (n NUMBER, MEMBER FUNCTION func RETURN NUMBER) NOT final;
CREATE TYPE BODY super_t AS
   MEMBER FUNCTION func RETURN NUMBER IS
   BEGIN RETURN 1; END;
END;

CREATE OR REPLACE TYPE sub_t UNDER super_t
  (n2 NUMBER, OVERRIDING MEMBER FUNCTION func RETURN NUMBER) NOT final;

CREATE TYPE BODY sub_t AS
  OVERRIDING MEMBER FUNCTION func RETURN NUMBER IS
  BEGIN RETURN 2; END;
END;

CREATE OR REPLACE TYPE final_t UNDER sub_t
  (n3 NUMBER);


DECLARE
   v  super_t := final_t(100); -- v is an instance of super_t, but a value of sub_t to it.  So,
BEGIN
   DBMS_OUTPUT.PUT_LINE(v.func); -- prints 2
END;
```

Outline

- Success of the relational model
- Limitations of the Relational Model
- Active RDBMS Extensions
- Object-Relational DBMS
  - UDTs
  - UDFs
  - Inheritance
  - Polymorphism, Overloading, Overriding
  - ☞ **Collection Types**

Collection Types

- **Collection types** can be instantiated as a collection of instances of standard data types or UDTs
- **Set**: unordered collection, no duplicates
- **Multiset** or **bag**: unordered collection, duplicates allowed
- **List**: ordered collection, duplicates allowed
- **Array**: ordered and indexed collection, duplicates allowed

Collection Type - SET

```
CREATE TYPE employeetype
  (ssn SMALLINT NOT NULL,
  fname CHAR(25) NOT NULL,
  lname CHAR(25) NOT NULL,
  empAddress international_address,
  …
  empFingerprint fingerprint,
  photograph image,
  telephone SET (CHAR(12)),
  FUNCTION age(employeetype) RETURNS INTEGER);

CREATE TABLE employee OF TYPE employeetype(PRIMARY KEY ssn);

SELECT ssn, fname, lname
FROM employee
WHERE '2123375000' IN (telephone)
```

Collection Type – a SET of References

```
CREATE TYPE departmenttype AS
(dnr CHAR(3) NOT NULL,
 dname CHAR(25) NOT NULL,
 manager REF(EMPLOYEETYPE),
 personnel SET (REF(employeetype));
```

```
-- personnel contains a
set of references to the
employee working in the
department
```

```
CREATE TABLE department OF TYPE departmenttype
(PRIMARY KEY dnr);
```

```
SELECT personnel
FROM department
WHERE dnr = '123‘;
```

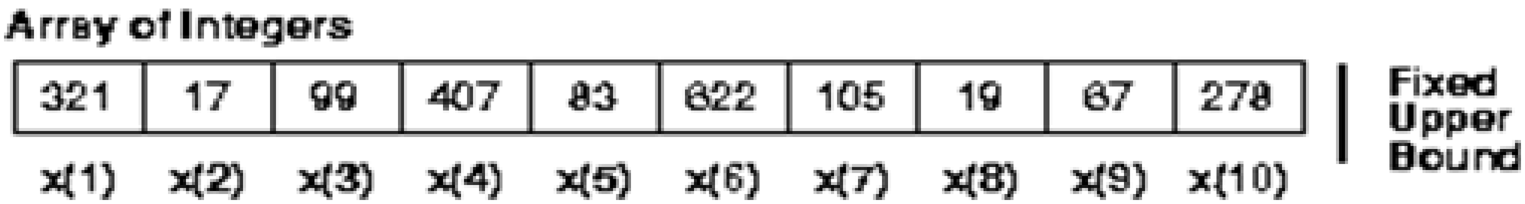
```
SELECT DEREF(personnel).fname, DEREF(personnel).lname
FROM department
WHERE dnr = '123‘;
```

Collection Type in Oracle : VARRAY

- **VARRAY** (short for variable-size arrays) hold an ordered set of data elements.

```
CREATE TYPE ProjectList AS VARRAY(50) OF VARCHAR2(16);
```

  - Each element has an index.
  - VARRAY is used for storing a fixed number of items.
  - VARRAY has fixed upper bound.



Array of Integers

321	17	99	407	83	622	105	19	67	278
x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)	x(9)	x(10)

Fixed Upper Bound

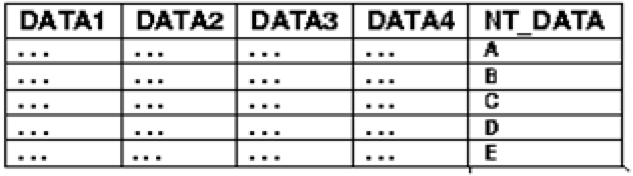
Oracle Example of a VARRAY Data Type

```
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')));

SELECT * FROM dept_phone_list;
```

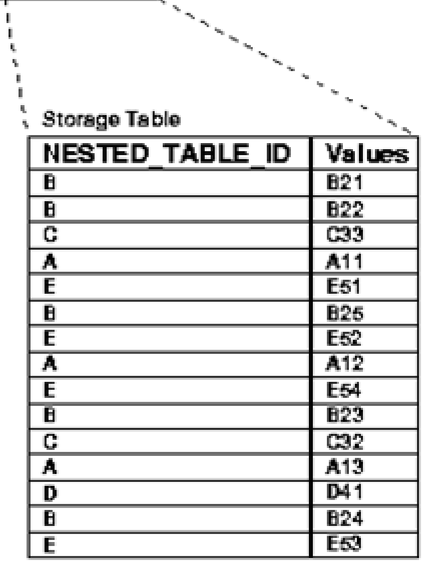
```
 DEPT_NO     PHONE_LIST
--------------- ------------------------------------------------------------------------------------------------------------
100
C##DBUSER.PHONE_VARRAY_TYP(C##DBUSER.PHONE_TYP('01','650','5550123'),C##DBUSER.PHONE_TYP('01','650','
5550148'),C##DBUSER5.PHONE_TYP('01','650','5550192'))
```

Oracle Collection Type: Nested Table

- **Nested tables** hold an unordered set of data elements, all of the same data type.
  - It can have any number of data elements.
  - Elements of a nested table are actually stored in a separate storage table.
  - It is good for mass insert, update or delete.



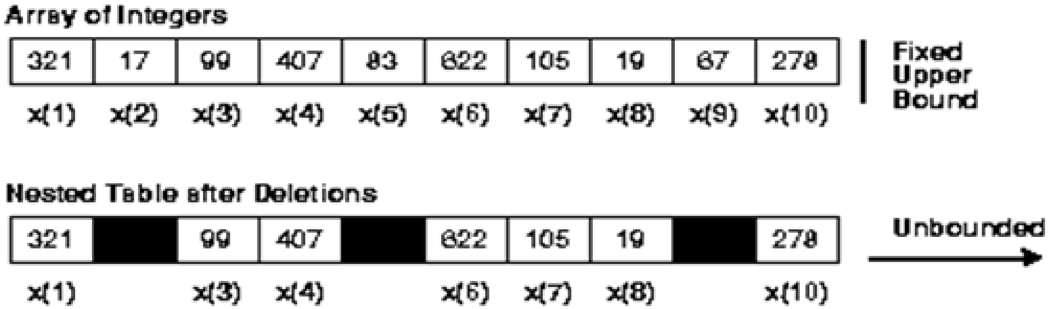
DATA1	DATA2	DATA3	DATA4	NT_DATA
...	...	...	...	A
...	...	...	...	B
...	...	...	...	C
...	...	...	...	D
...	...	...	...	E



Storage Table

NESTED_TABLE_ID	Values
B	B21
B	B22
C	C33
A	A11
E	E51
B	B25
E	E52
A	A12
E	E54
B	B23
C	C32
A	A13
D	D41
B	B24
E	E53

- VARRAY (ordered) vs. Nested table (unordered)



Array of Integers

321	17	99	407	83	622	105	19	67	278
x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)	x(9)	x(10)

Fixed Upper Bound

Nested Table after Deletions

321		99	407		622	105	19		278
x(1)		x(3)	x(4)		x(6)	x(7)	x(8)		x(10)

Unbounded

Example of Oracle Nested Table

```
--Declare the table type used for a nested table.
CREATE TYPE people_type AS TABLE OF person_type;

--Declare a nested table
CREATE TABLE contacts (
    contact  people_type,
    c_date DATE
) NESTED TABLE contact STORE AS people_table_st ;
   --This NESTED TABLE clause is required whenever a database table has a nested table
   column. The clause identifies the nested table and names a system-generated store table, in
   which Oracle stores the nested table data.
```

Reference: https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/05_colls.htm

Oracle Example – Nested Table

```
CREATE TYPE person_typ AS OBJECT ( idno NUMBER, name VARCHAR2(30), phone VARCHAR2(20));
CREATE TYPE people_typ AS TABLE OF person_typ;
CREATE 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 ;

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;
```

Large objects (LOB)

- Large object (LOB) data are often stored in a separate table and tablespace
- Types of LOB data:
  - **BLOB (Binary Large Object):** a variable-length binary string whose interpretation is left to an external application
  - **CLOB (Character Large Object):** variable-length character strings made up of single-byte characters
  - **DBCLOB (Double Byte Character Large Object):** variable-length character strings made up of double-byte characters.
- Many ORDBMSs will also provide customized SQL functions for LOB data
  - Examples are functions to search in image or video data or access text at a specified potions