PROCEDURES, FUNCTIONS & TRIGGERS

PROCEDURES

- A procedure is a module performing one or more actions; it does not need to return any values.
- The syntax for creating a procedure is as follows:

```
CREATE OR REPLACE PROCEDURE name
    [(parameter[, parameter, ...])]
AS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

PROCEDURES

- A procedure may have 0 to many parameters.
- Every procedure has two parts:
  1. The header portion, which comes before AS (sometimes you will see IS—they are interchangeable), keyword (this contains the procedure name and the parameter list),
  2. The body, which is everything after the IS keyword.
- The word REPLACE is optional.
- When the word REPLACE is not used in the header of the procedure, in order to change the code in the procedure, it must be dropped first and then re-created.

Example

```
-- ch11_01a.sql
CREATE OR REPLACE PROCEDURE Discount
AS
    CURSOR c_group_discount
    IS
     SELECT distinct s.course_no,
    c.description
     FROM section s, enrollment e, course c
     WHERE s.section_id = e.section_id
          AND c.course_no = s.course_no
    GROUP BY s.course_no, c.description,
          e.section_id, s.section_id
    HAVING COUNT(*) >=8;
BEGIN
```

Example

```
FOR r_group_discount IN c_group_discount
    LOOP
     UPDATE course
          SET cost = cost * .95
     WHERE course_no =
    r_group_discount.course_no;
     DBMS_OUTPUT.PUT_LINE
          ('A 5% discount has been given to'||
          r_group_discount.course_no||' '||
          r_group_discount.description
          );
    END LOOP;
END;
```

Example

- In order to execute a procedure in SQL*Plus use the following syntax:

EXECUTE Procedure_name

```
SQL> EXECUTE Discount
```

PARAMETERS

- Parameters are the means to pass values to and from the calling environment to the server.
- These are the values that will be processed or returned via the execution of the procedure.
- There are three types of parameters:
- IN, OUT, and IN OUT.
- Modes specify whether the parameter passed is read in or a receptacle for what comes out.

Types of Parameters

Mode	Description	Usage
IN	Passes a value into the program	Read only value Constants, literals, expressions Cannot be changed within program Default mode
OUT	Passes a value back from the program	Write only value Cannot assign default values Has to be a variable Value assigned only if the program is successful
IN OUT	Passes values in and also send values back	Has to be a variable Value will be read and then written

FORMAL AND ACTUAL PARAMETERS

- *Formal parameters* are the names specified within parentheses as part of the header of a module.
- *Actual parameters* are the values—expressions specified within parentheses as a parameter list—when a call is made to the module.
- The formal parameter and the related actual parameter must be of the same or compatible data types.

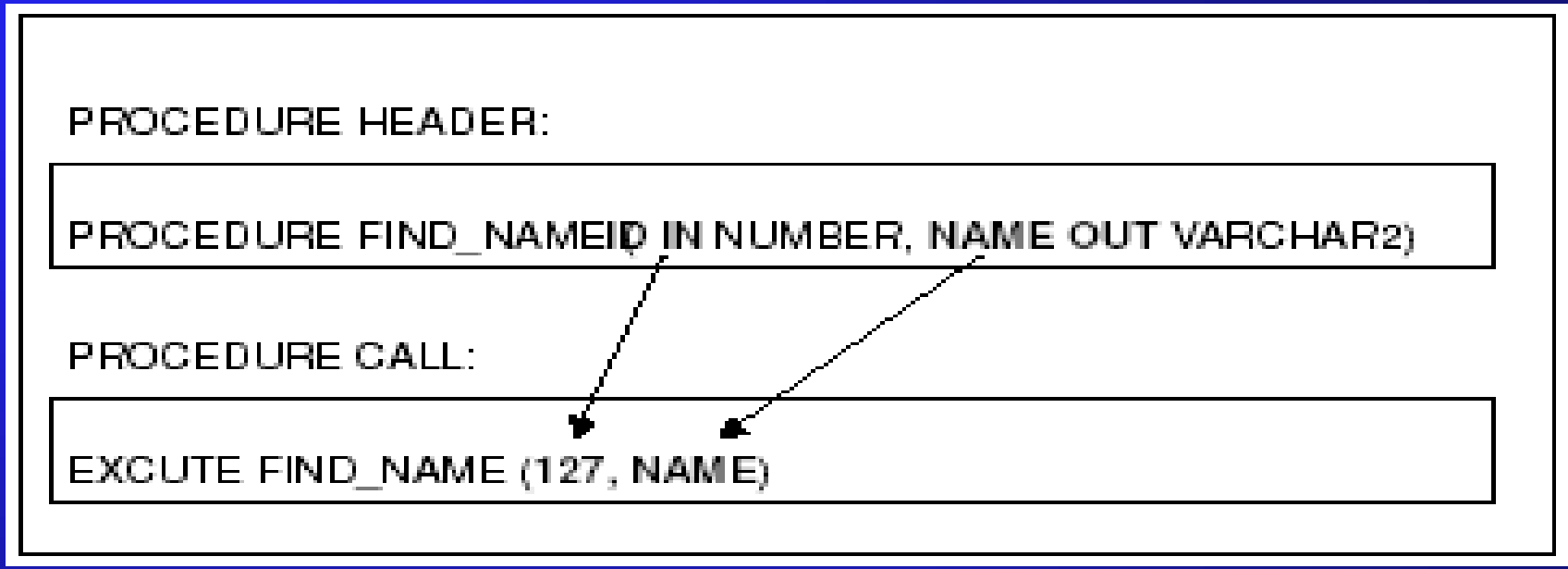
MATCHING ACTUAL AND FORMAL PARAMETERS

- Two methods can be used to match actual and formal parameters: positional notation and named notation.
- *Positional notation* is simply association by position: The order of the parameters used when executing the procedure matches the order in the procedure's header exactly.
- *Named notation* is explicit association using the symbol =>

```
Syntax: formal_parameter_name =>
    argument_value
```

- In named notation, the order does not matter.
- If you mix notation, list positional notation before named notation.

MATCHING ACTUAL AND FORMAL PARAMETERS



FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL block that *returns* a single value.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: It must be used in some context. You can think of it as a sentence fragment.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

FUNCTIONS

- The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
        (parameter list)
        RETURN datatype
IS
BEGIN
        <body>
        RETURN (return_value);
END;
```

FUNCTIONS

- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.
- The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).
- A function may have IN, OUT, or IN OUT parameters. but you rarely see anything except IN parameters.

Example

```
CREATE OR REPLACE FUNCTION show_description
        (i_course_no number)
RETURN varchar2
AS
        v_description varchar2(50);
BEGIN
        SELECT description
                INTO v_description
                FROM course
                WHERE course_no = i_course_no;
        RETURN v_description;
EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
                RETURN('The Course is not in the database');
        WHEN OTHERS
        THEN
                RETURN('Error in running show_description');
END;
```

Making Use Of Functions

- **In a anonymous block**

```
SET SERVEROUTPUT ON
DECLARE
    v_description VARCHAR2(50);
BEGIN
    v_description := show_description(&sv_cnumber);
    DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

- **In a SQL statement**

```
SELECT course_no, show_description(course_no)
    FROM course;
```

TRIGGERS

A database trigger is a stored PL/SQL program unit associated with a specific database table. ORACLE executes (fires) a database trigger automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table. Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.

TRIGGERS

Database triggers can be used to perform any of the following:

- Audit data modification
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicate tables

TRIGGERS

- You can associate up to 12 database triggers with a given table. A database trigger has three parts: a **triggering event**, an **optional trigger constraint**, and a **trigger action**.

- When an event occurs, a database trigger is fired, and an predefined PL/SQL block will perform the necessary action.

TRIGGERS

SYNTAX:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER} triggering_event ON table_name
[FOR EACH ROW]
[WHEN condition]
DECLARE
Declaration statements
BEGIN
Executable statements
EXCEPTION
Exception-handling statements
END;
```

TRIGGERS

The trigger_name references the name of the trigger.

BEFORE or AFTER specify when the trigger is fired (before or after the triggering event).

The triggering_event references a DML statement issued against the table (e.g., INSERT, DELETE, UPDATE).

The table_name is the name of the table associated with the trigger.

The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row.

A WHEN clause specifies the condition for a trigger to be fired.

Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.

TYPES OF TRIGGERS

Triggers may be called BEFORE or AFTER the following events:

INSERT, UPDATE and DELETE.

The before/after options can be used to specify when the trigger body should be fired with respect to the triggering statement. If the user indicates a BEFORE option, then Oracle fires the trigger before executing the triggering statement. On the other hand, if an AFTER is used, Oracle fires the trigger after executing the triggering statement.

TYPES OF TRIGGERS

- A trigger may be a ROW or STATEMENT type. If the statement FOR EACH ROW is present in the CREATE TRIGGER clause of a trigger, the trigger is a row trigger. A row trigger is fired for each row affected by an triggering statement.
- A statement trigger, however, is fired only once for the triggering statement, regardless of the number of rows affected by the triggering statement

TYPES OF TRIGGERS

Example: statement trigger

```
CREATE OR REPLACE TRIGGER mytrig1 BEFORE DELETE OR INSERT
      OR UPDATE ON employee
BEGIN
IF (TO_CHAR(SYSDATE, 'day') IN ('sat', 'sun')) OR
      (TO_CHAR(SYSDATE,'hh:mi') NOT BETWEEN '08:30' AND '18:30')
      THEN      RAISE_APPLICATION_ERROR(-20500, 'table is secured');
END IF;
END;
/
```

The above example shows a trigger that limits the DML actions to the employee table to weekdays from 8.30am to 6.30pm. If a user tries to insert/update/delete a row in the EMPLOYEE table, a warning message will be prompted.

Example: ROW Trigger

```
CREATE OR REPLACE TRIGGER mytrig2
AFTER DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
BEGIN
IF DELETING THEN
INSERT INTO xemployee (emp_ssn, emp_last_name,emp_first_name, deldate)
VALUES (:old.emp_ssn, :old.emp_last_name,:old.emp_first_name, sysdate);
ELSIF INSERTING THEN
 INSERT INTO nemployee (emp_ssn, emp_last_name,emp_first_name, adddate)
VALUES (:new.emp_ssn, :new.emp_last_name,:new.emp_first_name, sysdate);
 ELSIF UPDATING('emp_salary') THEN
INSERT INTO cemployee (emp_ssn, oldsalary, newsalary, up_date)
VALUES (:old.emp_ssn,:old.emp_salary, :new.emp_salary, sysdate);     ELSE
INSERT INTO uemployee (emp_ssn, emp_address, up_date)
VALUES (:old.emp_ssn, :new.emp_address, sysdate);
END IF;
END;
/
```

TYPES OF TRIGGERS

Example: ROW Trigger

- The previous trigger is used to keep track of all the transactions performed on the employee table. If any employee is deleted, a new row containing the details of this employee is stored in a table called xemployee. Similarly, if a new employee is inserted, a new row is created in another table called nemployee, and so on.
- Note that we can specify the old and new values of an updated row by prefixing the column names with the :OLD and :NEW qualifiers.

TYPES OF TRIGGERS

```
SQL>  DELETE FROM  employee WHERE
      emp_last_name = 'Joshi';
1 row deleted.
SQL> SELECT * FROM xemployee;

EMP_SSN   EMP_LAST_NAME   EMP_FIRST_NAME DELDATE
---------- ---------------------- ------------------------- -----------------
999333333 Joshi                  Dinesh                    02-MAY-03
```

ENABLING, DISABLING, DROPPING TRIGGERS

SQL>ALTER TRIGGER trigger_name DISABLE;

SQL>ALTER TABLE table_name DISABLE ALL TRIGGERS;

To enable a trigger, which is disabled, we can use the following syntax:

SQL>ALTER TABLE table_name ENABLE trigger_name;

All triggers can be enabled for a specific table by using the following command

SQL> ALTER TABLE table_name ENABLE ALL TRIGGERS;

SQL> DROP TRIGGER trigger_name

END