1

Introduction to PL/SQL

About PL/SQL

PL/SQL:

- Stands for “Procedural Language extension to SQL”
- Is Oracle Corporation’s standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL

PL/SQL Block Structure

- `DECLARE` (optional)
  - Variables, cursors, user-defined exceptions
- `BEGIN` (mandatory)
  - SQL statements
  - PL/SQL statements
- `EXCEPTION` (optional)
  - Actions to perform when except
- `END`; (mandatory)

Block Types

Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

Enabling Output of a PL/SQL Block

1. To enable output in SQL Developer, execute the following command before running the PL/SQL block:

```
SET SERVEROUTPUT ON
```

2. Use a predefined Oracle package and its procedure in the anonymous block:
   - DBMS_OUTPUT.PUT_LINE

```
DBMS_OUTPUT.PUT_LINE(' The First Name of the
Employee is ' || v_fname);
…
```

2

Declaring PL/SQL Variables

Requirements for Variable Names

A variable name:

- Must start with a letter
- Can include letters or numbers
- Can include special characters (such as $, _, and #)
- Must contain no more than 30 characters
- Must not include reserved words

ORACLE

2 -5

Requirements for Variable Names

The rules for naming a variable are listed in the slide.

Declaring and Initializing PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]  [:= |
     DEFAULT expr];
```

Examples

.

```
DECLARE
  v_hiredat       DATE;
  e               NUMBER(2) NOT NULL := 10;
  v_deptno        VARCHAR2(13) :=
  v_locatio       'Atlanta';
  n               CONSTANT NUMBER := 1400;
  c_comm
```

Declaring and Initializing PL/SQL Variables

1

```
DECLARE
  v_myName VARCHAR2(20);  BEGIN
   DBMS_OUTPUT.PUT_LINE('My name is: '||
 v_myName);
   v_myName := 'John';
 DBMS_OUTPUT.PUT_LINE('My name is: '||
v_myName);  END;
 /
```

2

```
DECLARE
 v_myName VARCHAR2(20):= 'John';  BEGIN
  v_myName := 'Steven';
 DBMS_OUTPUT.PUT_LINE('My name is: '||
v_myName);  END;
 /
```

ORACLE

Delimiters in String Literals

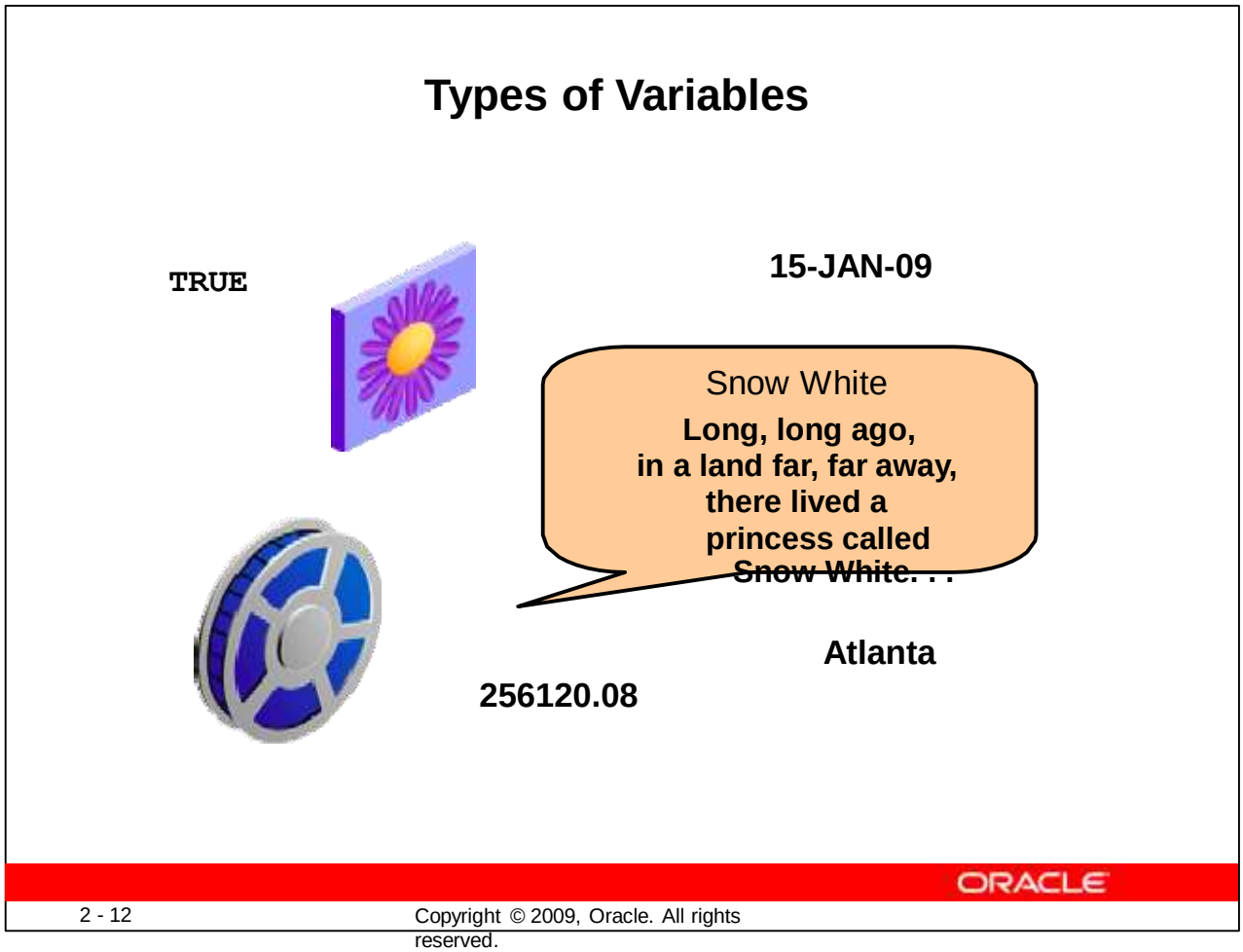
```
DECLARE
    v_event VARCHAR2(15);  BEGIN
       v_event := q'!Father's day!';
  DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :  '||
   v_event );
  v_event := q'[Mother's day]';
  DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :  '||
   v_event );
END;
/
```

Resulting output

```
anonymous block completed
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
```

Types of Variables

- PL/SQL variables:
  - Scalar
  - Reference
  - Large object (`LOB`)
  - Composite
- Non-PL/SQL variables: Bind variables



Types of Variables (continued)

The slide illustrates the following data types:

- `TRUE` represents a Boolean value.
- 15-JAN-09 represents a `DATE`.
- The image represents a `BLOB`.
- The text in the callout can represent a `VARCHAR2` data type or a `CLOB`.
- 256120.08 represents a `NUMBER` data type with precision and scale.
- The film reel represents a `BFILE`.
- The city name *Atlanta* represents a `VARCHAR2` data type.

Guidelines for Declaring and Initializing PL/SQL Variables

- Follow consistent naming conventions.
- Use meaningful identifiers for variables.
- Initialize variables that are designated as `NOT NULL` and `CONSTANT`.
- Initialize variables with the assignment operator (`:=`) or `DEFAULT` keyword:

```
v_myName VARCHAR2(20):='John';
```

```
v_myName VARCHAR2(20) DEFAULT 'John';
```

- Declare one identifier per line for better readability and code maintenance.

ORACLE

2 - 13

Guidelines for Declaring PL/SQL Variables

- Avoid using column names as identifiers.

```
DECLARE
  employee_id  NUMBER(6);  BEGIN
    SELECT employee_id  INTO  employee_id
    FROM  employees
  WHERE   last_name = 'Kochhar';  END;
  /
```

- Use the `NOT NULL` constraint when the variable must hold a value.

Naming Conventions of PL/SQL Structures Used in This Course

PL/SQL Structure	Convention	Example
Variable	`v_variable_name`	`v_rate`
Constant	`c_constant_name`	`c_rate`
Subprogram parameter	`p_parameter_name`	`p_id`
Bind (host) variable	`b_bind_name`	`b_salary`
Cursor	`cur_cursor_name`	`cur_emp`
Record	`rec_record_name`	`rec_emp`
Type	`type_name_type`	`ename_table_type`
Exception	`e_exception_name`	`e_products_invalid`
File handle	`f_file_handle_name`	`f_file`

Base Scalar Data Types

- CHAR [(maximum_length)]
- VARCHAR2 (maximum_length)
- NUMBER [(precision, scale)]
- BINARY_INTEGER
- PLS_INTEGER
- BOOLEAN
- BINARY_FLOAT
- BINARY_DOUBLE

ORACLE

Base Scalar Data Types

Data Type	Description
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER values. In Oracle Database 11*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.
BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, and NULL
BINARY_FLOAT	Represents floating-point number in IEEE 754 format. It requires 5 bytes to store the value.
BINARY_DOUBLE	Represents floating-point number in IEEE 754 format. It requires 9 bytes to store the value.

Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(*precision*, *scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 through 38. The scale *s* can range from –84 through 127.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647

`%TYPE` Attribute

- Is used to declare a variable according to:
  - A database column definition
  - Another declared variable
- Is prefixed with:
  - The database table and column name
  - The name of the declared variable

ORACLE

Declaring Variables with the `%TYPE` Attribute

Syntax

```
identifier     table.column_name%TYPE;
```

Examples

```
...
  v_emp_lname       employees.last_name%TYPE
  e                 ;
...
```

```
...
  v_balance         NUMBER(7,2);
  v_min_balance     v_balance%TYPE :=
  e                 1000;
...
```

Declaring Boolean Variables

- Only the `TRUE`, `FALSE`, and `NULL` values can be assigned to a Boolean variable.
- Conditional expressions use the logical operators `AND` and `OR`, and the unary operator `NOT` to check the variable values.
- The variables always yield `TRUE`, `FALSE`, or `NULL`.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

ORACLE

Bind Variables

Bind variables are:

- Created in the environment
- Also called *host* variables
- Created with the `VARIABLE` keyword*
- Used in SQL statements and PL/SQL blocks
- Accessed even after the PL/SQL block is executed
- Referenced with a preceding colon

Values can be output using the `PRINT` command.

* Required when using SQL*Plus and SQL Developer

ORACLE

Referencing Bind Variables

Example:

```
VARIABLE b_emp_salary NUMBER  BEGIN
   SELECT salary    INTO :b_emp_salary
   FROM   employees WHERE employee_id = 178;  END;
/
PRINT b_emp_salary
SELECT first_name, last_name
FROM employees
WHERE salary=:b_emp_salary;
```

Output

Results | Script Output | Explain | Autotrace

```
anonymous block completed
b_emp_salary
----
7000

FIRST_NAME           LAST_NAME
-------------------- -------------------------
Oliver               Tuvault
Sarath               Sewall
Kimberely            Grant

3 rows selected
```

ORACLE

2 - 31

3

Writing Executable Statements

Commenting Code

- Prefix single-line comments with two hyphens (--).
- Place a block comment between the symbols /* and */. Example:

```
DECLARE
...
v_annual_sal NUMBER (9,2);  BEGIN
/* Compute the annual salary based on the  monthly
   salary input from the user */
v_annual_sal := monthly_sal * 12;
--The following line displays the annual salary
DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

ORACLE

3 -7

Commenting Code

You should comment code to document each phase and to assist debugging. In PL/SQL code:

- A single-line comment is commonly prefixed with two hyphens (--)
- You can also enclose a comment between the symbols /* and */

Note: For multiline comments, you can either precede each comment line with two hyphens, or use the block comment format.

Comments are strictly informational and do not enforce any conditions or behavior on the logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance.

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row functions
- Not available in procedural statements:
  - `DECODE`
  - Group functions

SQL Functions in PL/SQL: Examples

- Get the length of a string:

```
v_desc_size INTEGER(5);
v_prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
v_desc_size:= LENGTH(v_prod_description);
```

- Get the number of months an employee has

```
v_tenure:= MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

Using Sequences in PL/SQL Expressions

Starting in 11*g*:

```
DECLARE
  v_new_id NUMBER;  BEGIN
    v_new_id := my_seq.NEXTVAL;
  END;
  /
```

Before

```
DECLARE
  v_new_id NUMBER;  BEGIN
  SELECT my_seq.NEXTVAL INTO v_new_id FROM Dual;  END;
  /
```

Data Type Conversion

- Converts data to comparable data types
- Is of two types:
  - Implicit conversion
  - Explicit conversion
- Functions:
  - `TO_CHAR`
  - `TO_DATE`
  - `TO_NUMBER`

Data Type Conversion

1

```
-- implicit data type conversion
v_date_of_joining DATE:= '02-Feb-2000';
```

2

```
-- error in data type conversion
v_date_of_joining DATE:= 'February
02,2000';
```

3

```
-- explicit data type conversion
v_date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```

Nested Blocks: Example

```
DECLARE
 v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
   DECLARE
   v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
   DBMS_OUTPUT.PUT_LINE(v_outer_variable);
   END;
 DBMS_OUTPUT.PUT_LINE(v_outer_variable);  END;
```

```
anonymous block completed
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE
```

Using a Qualifier with Nested Blocks

```
BEGIN <<outer>>
DECLARE
 v_father_name VARCHAR2(20):='Patrick';  v_date_of_birth
DATE:='20-Apr-1972';  BEGIN
   DECLARE
    v_child_name VARCHAR2(20):='Mike';  v_date_of_birth
   DATE:='12-Dec-2002';
   BEGIN
    DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                        ||outer.v_date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
  END;
 END;
 END outer;
```

Challenge: Determining Variable Scope

```
<<outer>>
DECLARE
v_sal NUMBER(7,2) := 60000;
v_comm NUMBER(7,2) := v_sal * 0.20;
v_message  VARCHAR2(255) := ' eligible for commission';
 BEGIN
DECLARE
v_sal          NUMBER(7,2) :=
               50000;
  v_com        NUMBER(7,2) := 0;
  v_total_comp NUMBER(7,2) := v_sal +
  v_comm;
  BEGIN
      v_message := 'CLERK not'||v_message;
        outer.v_comm := v_sal * 0.30;
      END;

                v_message :=
        'SALESMAN'||v_message;  END;
      END outer;
      /
```

1

2

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

Same as in SQL

- Exponential operator (**)

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -, ||	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion

Operators in PL/SQL: Examples

- Increment the counter for a loop.

```
loop_count := loop_count + 1;
```

- Set the value of a Boolean flag.

```
good_sal := sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value.

```
valid  := (empno IS NOT NULL);
```


Operators in PL/SQL (continued)

When you are working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- In conditional control statements, if the condition yields `NULL`, its associated sequence of statements is not executed.

4

Interacting with Oracle Database Server: SQL Statements in PL/SQL Programs

SELECT Statements in PL/SQL

Retrieve data from the database with a SELECT statement. Syntax:

```
SELECT   select_list
INTO     {variable_name[, variable_name]...
        | record_name}
FROM     table
[WHERE   condition];
```

SELECT Statements in PL/SQL

- The INTO clause is required.

```
DECLARE
 v_fname VARCHAR2(25);  BEGIN
  SELECT first_name INTO v_fname
  FROM employees WHERE employee_id=200;
 DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
 END;
 /
```

```
anonymous block completed
 First Name is : Jennifer
```

Retrieving Data in PL/SQL: Example

Retrieve `hire_date` and `salary` for the specified employee.

```
DECLARE
 v_emp_hiredate       employees.hire_date%TYPE;
v_emp_salary    employees.salary%TYPE;  BEGIN
   SELECT  hire_date, salary
   INTO    v_emp_hiredate, v_emp_salary  FROM    employees
   WHERE   employee_id = 100;
   DBMS_OUTPUT.PUT_LINE ('Hire date is :'|| v_emp_hiredate);
  DBMS_OUTPUT.PUT_LINE ('Salary is :'|| v_emp_ salary);
 END;
 /
```

```
anonymous block completed
 Hire date is : 17-JUN-87
 Salary  is : 24000
```

Retrieving Data in PL/SQL

Return the sum of salaries for all the employees in the specified department.

Example:

```
DECLARE
   v_sum_sal    NUMBER(10,2);
   v_deptno     NUMBER NOT NULL := 60;  BEGIN
     SELECT     SUM(salary)     -- group function  INTO
  v_sum_sal     FROM employees
     WHERE department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE ('The sum of salary is ' ||
v_sum_sal);  END;
```

```
anonymous block completed
The sum of salary is 28800
```

Naming Ambiguities

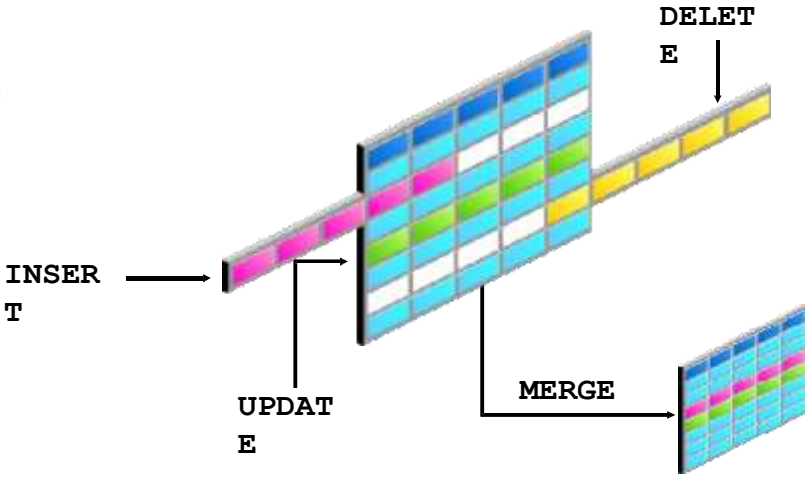
```
DECLARE
  hire_date      employees.hire_date%TYPE;
  sysdate        hire_date%TYPE;
  employee_i     employees.employee_id%TYPE :=
  d              176;
BEGIN         hire_date,         sysdate
  SELECT      hire_date,         sysdate
  INTO  FROM  employees
  WHERE       employee_id              =
END;          employee_id;
/
```

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

Using PL/SQL to Manipulate Data

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



Inserting Data: Example

Add new employee information to the `EMPLOYEES` table.

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',CURRENT_DATE, 'AD_ASST', 4000);
END;
/
```

Updating Data: Example

Increase the salary of all employees

```
DECLARE
  sal_increas    employees.salary%TYPE :=
e  BEGIN         800;
  UPDATE        employees
  SET           salary = salary +
  sal_increase
  WHERE         job_id = 'ST_CLERK';  END;
  /
```

```
anonymous block completed
FIRST_NAME           SALARY
-------------------- ----------------------
Julia                4000
Irene                3500
James                3200
Steven               3000
```

...

```
Curtis               3900
Randall              3400
Peter                3300

20 rows selected
```

ORACLE

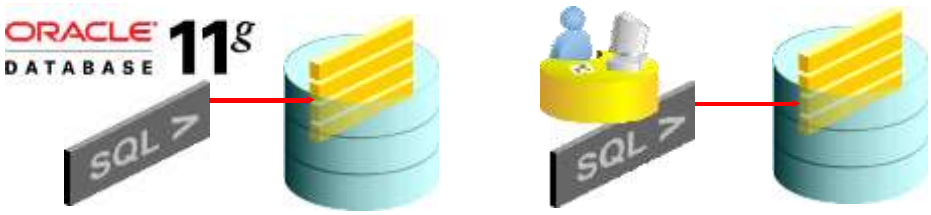
Deleting Data: Example

Delete rows that belong to department 10 from the `employees` table.

```
DECLARE
  deptno   employees.department_id%TYPE :=
BEGIN      10;
  DELETE FROM     employees
  WHERE           department_id
  = deptno;  END;
  /
```

SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle Server. It is used to handle the result set of a `SELECT` statement.
- There are two types of cursors: implicit and explicit.
  - **Implicit:** Created and managed internally by the Oracle Server to process SQL statements
  - **Explicit:** Declared explicitly by the programmer



Implicit cursor **Explicit cursor**

SQL Cursor Attributes for Implicit Cursors

Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%FOUND	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement affected at least one row
SQL%NOTFOUND	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement did not affect even one row
SQL%ROWCOUNT	An integer value that represents the number of rows affected by the most recent SQL statement

SQL Cursor Attributes for Implicit Cursors

Delete rows that have the specified employee ID from the `employees` table. Print the number of rows deleted.

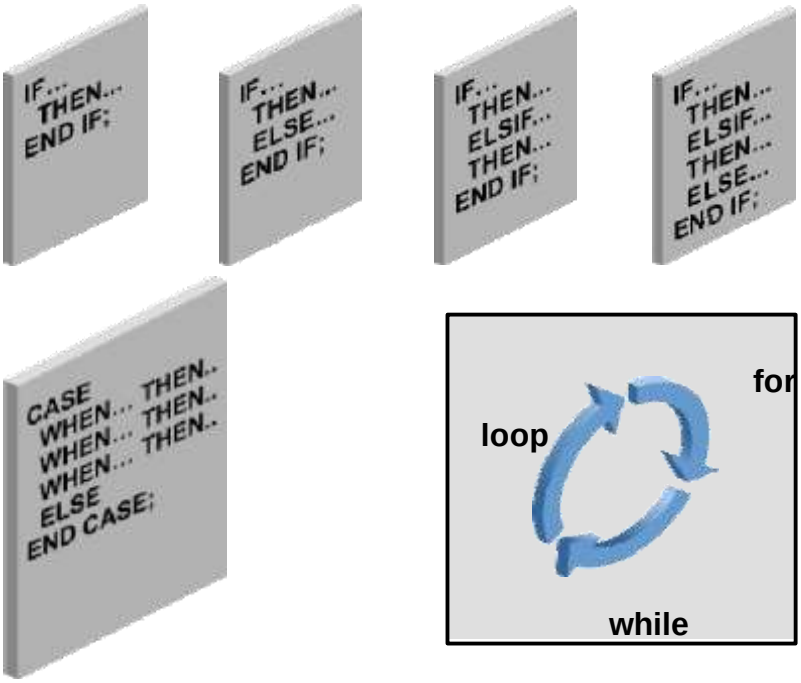
```
DECLARE
  v_rows_deleted VARCHAR2(30)
  v_empno employees.employee_id%TYPE := 176;  BEGIN
    DELETE FROM     employees
    WHERE employee_id = v_empno;  v_rows_deleted :=
  (SQL%ROWCOUNT ||
                        ' row deleted.');
    DBMS_OUTPUT.PUT_LINE (v_rows_deleted);

  END;
```

5

Writing Control Structures

Controlling Flow of Execution



IF Statement

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]  [ELSE
  statements;]  END IF;
```

Simple IF Statement

```
DECLARE
  v_myage number:=31;  BEGIN
    IF v_myage < 11  THEN
      DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;  END;
  /
```

```
anonymous block completed
```

IF THEN ELSE Statement

```
DECLARE
v_myage   number:=31;  BEGIN
IF v_myage     < 11  THEN
DBMS_OUTPUT.PUT_LINE(' I am a child ');  ELSE
DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;  END;
/
```

```
anonymous block completed
 I am not a child
```

IF ELSIF ELSE Clause

```
DECLARE
  v_myage number:=31;  BEGIN
  IF v_myage    < 11 THEN  DBMS_OUTPUT.PUT_LINE(' I am a
      child ');
    ELSIF v_myage     < 20 THEN  DBMS_OUTPUT.PUT_LINE(' I am
      young ');
 ELSIF v_myage  < 30 THEN
      DBMS_OUTPUT.PUT_LINE(' I am in my twenties');  ELSIF
    v_myage< 40 THEN
      DBMS_OUTPUT.PUT_LINE(' I am in my thirties');  ELSE
      DBMS_OUTPUT.PUT_LINE(' I am always young ');  END IF;
     END;
     /
```

```
anonymous block completed
 I am in my thirties
```

NULL Value in IF Statement

```
DECLARE
  v_myage number;  BEGIN
  IF v_myage   < 11 THEN  DBMS_OUTPUT.PUT_LINE(' I am a
    child ');
ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');  END IF;
  END;
  /
```

```
anonymous block completed
 I am not a child
```

CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses expressions. The value returned by these expressions is used to select one of several alternatives.

```
CASE selector
   WHEN expression1 THEN result1  WHEN expression2 THEN
   result2
   ...
   WHEN expressionN THEN resultN  [ELSE resultN+1]
 END;
```

CASE Expressions: Example

```
SET VERIFY OFF  DECLARE

   v_gradeCHAR(1) := UPPER('&grade');  v_appraisal
   VARCHAR2(20);
BEGIN
   v_appraisal := CASE v_grade  WHEN 'A' THEN
         'Excellent'
   WHEN 'B' THEN 'Very Good'  WHEN 'C' THEN 'Good'
         ELSE 'No such grade'  END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
                          Appraisal ' || v_appraisal);
   END;
   /
```

Searched CASE Expressions

```
DECLARE
   v_grade    CHAR(1) := UPPER('&grade');  v_appraisal
   VARCHAR2(20);
BEGIN
    v_appraisal := CASE
         WHEN v_grade   = 'A' THEN 'Excellent'
         WHEN v_grade   IN ('B','C') THEN 'Good'  ELSE
         'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade|| '
                  Appraisal ' || v_appraisal);
END;
/
```

Handling Nulls

When you are working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- If the condition yields `NULL` in conditional control statements, its associated sequence of statements is not executed.

Consider the following example:

```
x := 5;
y := NULL;
          ...
IF x != y THEN …..
END IF;
OUTPUT:: NULL
```

Logic Tables

Build a simple Boolean condition with a comparison operator

AND	*TRUE*	*FALSE*	*NULL*	OR	*TRUE*	*FALSE*	*NULL*	NOT	
TRUE	TRUE	FALSE	NULL	*TRUE*	TRUE	TRUE	TRUE	*TRUE*	FALSE
FALSE	FALSE	FALSE	FALSE	*FALSE*	TRUE	FALSE	NULL	*FALSE*	TRUE
NULL	NULL	FALSE	NULL	*NULL*	TRUE	NULL	NULL	*NULL*	NULL

Boolean Expressions or Logical Expression?

What is the value of `flag` in each case?

```
flag := reorder_flag AND available_flag;
```

REORDER_FLAG	AVAILABLE_FLAG	FLAG
TRUE	TRUE	? (1)
TRUE	FALSE	? (2)
NULL	TRUE	? (3)
NULL	FALSE	? (4)

Iterative Control: `LOOP` Statements

- Loops repeat a statement (or a sequence of statements) multiple times.
- There are three loop types:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop

Basic Loops

Syntax

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];  END LOOP;
```

Basic Loop: Example

```
declare
  num number:=1;
begin
 loop
  DBMS_OUTPUT.PUT_LINE(num);
  num:= num+1;
 exit when num>=11;
end loop;
end;
```

WHILE Loops

Syntax

```
WHILE condition LOOP  statement1;  statement2;
                          . . .
END LOOP;
```

Use the `WHILE` loop to repeat statements while a condition is `TRUE`.

WHILE Loops: Example

```
declare
   num number:=1;
begin
 while num<=10 loop
   DBMS_OUTPUT.PUT_LINE(num);
   num:= num+1;
  end loop;
end;
```

FOR Loops

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP  statement1;
    statement2;
    . . .
  END LOOP;
```

FOR Loops: Example

```
declare
   num number;
begin
 FOR num IN 1..10 LOOP
   DBMS_OUTPUT.PUT_LINE(num);
  end loop;
end;
```

```
declare
  num number;
begin
 FOR num IN REVERSE 1..10 LOOP
  DBMS_OUTPUT.PUT_LINE(num);
 end loop;
end;
```

`FOR` Loop Rules

- Reference the counter only within the loop; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be `NULL`.

ORACLE

Suggested Use of Loops

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the `WHILE` loop if the condition must be evaluated at the start of each iteration.
- Use a `FOR` loop if the number of iterations is known.

ORACLE

Suggested Use of Loops

A basic loop allows the execution of its statement at least once, even if the condition is already met upon entering the loop. Without the `EXIT` statement, the loop would be infinite.

You can use the `WHILE` loop to repeat a sequence of statements until the controlling condition is no longer `TRUE`. The condition is evaluated at the start of each iteration. The loop terminates when the condition is `FALSE`. If the condition is `FALSE` at the start of the loop, no further iterations are performed.

`FOR` loops have a control statement before the `LOOP` keyword to determine the number of iterations that the PL/SQL performs. Use a `FOR` loop if the number of iterations is predetermined.

Nested Loops and Labels

- You can nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the `EXIT` statement that references the label.

ORACLE

Nested Loops and Labels

You can nest the `FOR`, `WHILE`, and basic loops within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception is raised. However, you can label loops and exit the outer loop with the `EXIT` statement.

Label names follow the same rules as the other identifiers. A label is placed before a statement, either on the same line or on a separate line. White space is insignificant in all PL/SQL parsing except inside literals. Label basic loops by placing the label before the word `LOOP` within label delimiters (<<*label*>>). In `FOR` and `WHILE` loops, place the label before `FOR` or `WHILE`.

If the loop is labeled, the label name can be included (optionally) after the `END LOOP` statement for clarity.

Nested Loops and Labels: Example

```
...  BEGIN
  <<Outer_loop>>  LOOP
    v_counter := v_counter+1;  EXIT WHEN v_counter>10;
      <<Inner_loop>>  LOOP
        ...
        EXIT Outer_loop WHEN total_done = 'YES';
        -- Leave both loops
        EXIT WHEN inner_done = 'YES';
        -- Leave inner loop only
        ...
      END LOOP Inner_loop;
      ...
  END LOOP Outer_loop;  END;
  /
```

PL/SQL `CONTINUE` Statement

- Definition
  - Adds the functionality to begin the next loop iteration
  - Provides programmers with the ability to transfer control to the next iteration of a loop
  - Uses parallel structure and semantics to the `EXIT` statement
- Benefits
  - Eases the programming process
  - May provide a small performance improvement over the previous programming workarounds to simulate the `CONTINUE` statement

ORACLE

PL/SQL CONTINUE Statement: Example 1

```
DECLARE
  v_total SIMPLE_INTEGER := 0;
BEGIN
    FOR i IN 1..10 LOOP
1   v_total := v_total + i;
    dbms_output.put_line
     ('Total is: '|| v_total);
    CONTINUE WHEN i > 5;
     v_total := v_total + i;
2   dbms_output.put_line
     ('Out of Loop Total is:
      '|| v_total);
  END LOOP;
END;
  /
```

Results | Script Output | Expl

```
anonymous block completed
Total is: 1
Out of Loop Total is:
      2
Total is: 4
Out of Loop Total is:
      6
Total is: 9
Out of Loop Total is:
      12
Total is: 16
Out of Loop Total is:
      20
Total is: 25
Out of Loop Total is:
      30
Total is: 36
Total is: 43
Total is: 51
Total is: 60
Total is: 70
```

PL/SQL CONTINUE Statement: Example 2

```
DECLARE
 v_total NUMBER := 0;  BEGIN
  <<BeforeTopLoop>>
  FOR i IN 1..10 LOOP
    v_total := v_total + 1;  dbms_output.put_line
     ('Total is: ' || v_total);  FOR j IN 1..10 LOOP
       CONTINUE BeforeTopLoop WHEN i + j > 5;  v_total := v_total +
     1;




   END LOOP;
 END LOOP;
END two_loop;
```

Results | Script Output | Exp

```
anonymous block completed
Total is: 1
Total is: 6
Total is: 10
Total is: 13
Total is: 15
Total is: 16
Total is: 17
Total is: 18
Total is: 19
Total is: 20
```

ORACLE

6

Working with Composite Data Types

Creating a PL/SQL Record

Syntax:

1
```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
```

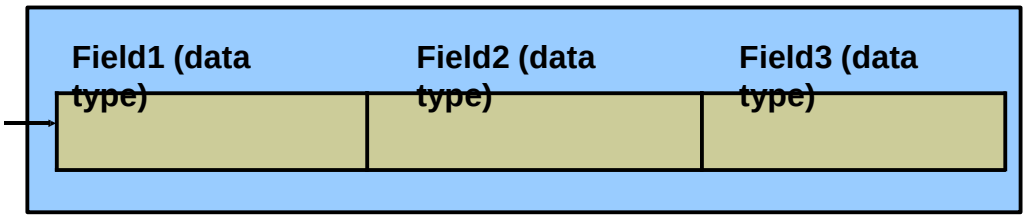
2
```
identifier    type_name;
```

field_declaration

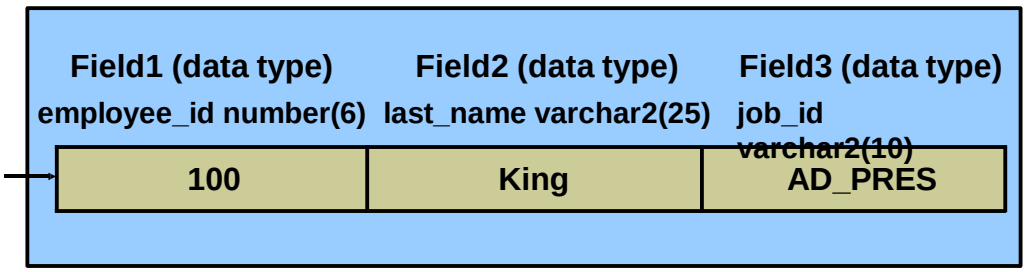
```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

PL/SQL Record Structure

Field declarations:



Example



6 -9

PL/SQL Record Structure

Fields in a record are accessed with the name of the record. To reference or initialize an individual field, use the dot notation:

```
record_name.field_name
```

For example, you reference the `job_id` field in the `emp_record` record as follows:

```
emp_record.job_id
```

You can then assign a value to the record field:

```
emp_record.job_id := 'ST_CLERK';
```

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram. They cease to exist when you exit the block or subprogram.

%ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE
   identifier reference%ROWTYPE;
```

Creating a PL/SQL Record: Example

```
DECLARE
  TYPE t_rec IS RECORD
    (v_sal number(8),
     v_minsal number(8) default 1000,  v_hire_date
     employees.hire_date%type,  v_rec1 employees%rowtype);
  v_myrec t_rec;  BEGIN
    v_myrec.v_sal := v_myrec.v_minsal + 500;
  v_myrec.v_hire_date := sysdate;
    SELECT * INTO v_myrec.v_rec1
      FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name ||' '||
  to_char(v_myrec.v_hire_date) ||' '||
  to_char(v_myrec.v_sal));
    END;
```

```
anonymous block completed
King 16-FEB-09 1500
```

ORACLE

Advantages of Using the `%ROWTYPE` Attribute

- The number and data types of the underlying database columns need not be known—and, in fact, might change at run time.
- The `%ROWTYPE` attribute is useful when you want to retrieve a row with:
  - The `SELECT*` statement
  - Row-level `INSERT` and `UPDATE` statements

ORACLE

Another %ROWTYPE Attribute Example

```
DECLARE
  v_employee_number number:= 124;  v_emp_rec   employees%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM employees  WHERE employee_id =
 v_employee_number;
 INSERT INTO retired_emps(empno, ename, job, mgr,
                 hiredate, leavedate, sal, comm, deptno)  VALUES
   (v_emp_rec.employee_id, v_emp_rec.last_name,
                        v_emp_rec.job_id, v_emp_rec.manager_id,
           v_emp_rec.hire_date, SYSDATE,  v_emp_rec.salary,
             v_emp_rec.commission_pct,  v_emp_rec.department_id);
              END;
/
```

```
SELECT * FROM retired_emps;
/
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-JUN-09	5800	(null)	50

Inserting a Record by Using %ROWTYPE

```
...
DECLARE
  v_employee_number number:= 124;  v_emp_rec
  retired_emps%ROWTYPE;
BEGIN
 SELECT employee_id, last_name, job_id, manager_id,
 hire_date, hire_date, salary, commission_pct,  department_id
 INTO v_emp_rec FROM employees  WHERE employee_id =
 v_employee_number;  INSERT INTO retired_emps VALUES
 v_emp_rec;
END;
/
SELECT * FRO
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

Updating a Row in a Table by Using a Record

```
SET VERIFY OFF  DECLARE
  v_employee_number number:= 124;  v_emp_rec
  retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM retired_emps;
 v_emp_rec.leavedate:=CURRENT_DATE;
 UPDATE retired_emps SET ROW = v_emp_rec WHERE
  empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

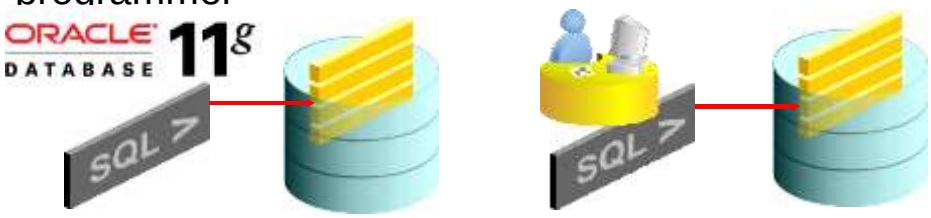
7

Using Explicit Cursors

Cursors

Every SQL statement that is executed by the Oracle Server has an associated individual cursor:

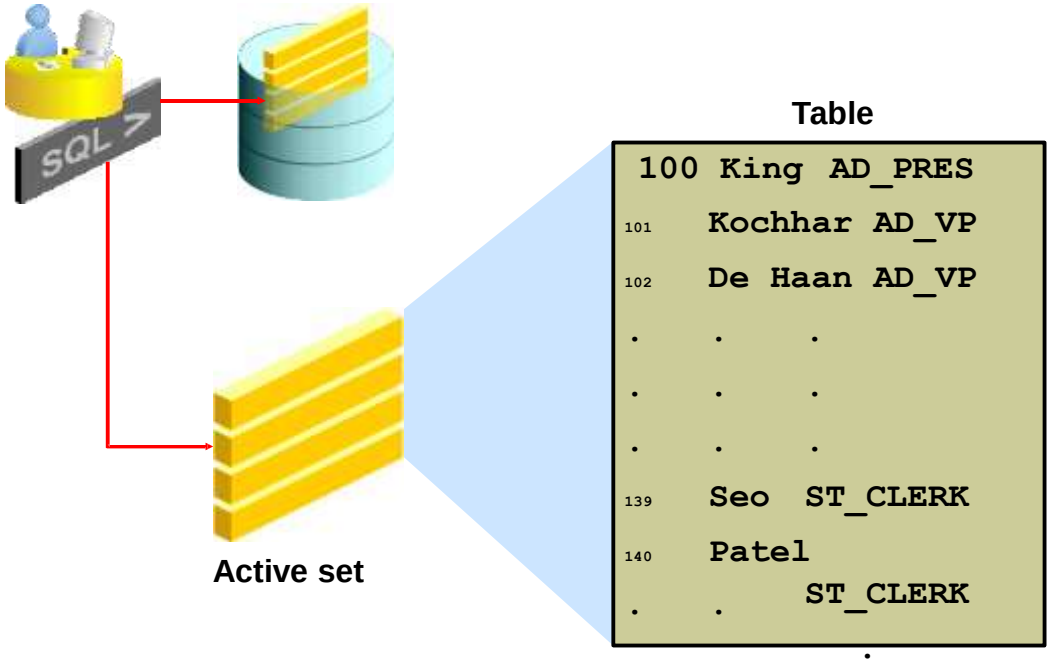
- Implicit cursors: declared and managed by PL/SQL for all DML and PL/SQL `SELECT` statements
- Explicit cursors: declared and managed by the programmer



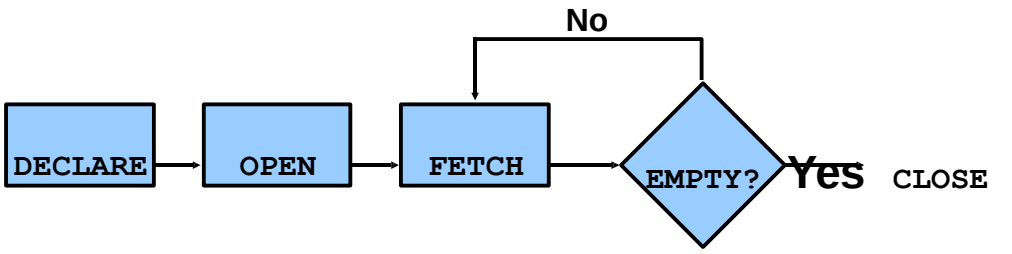
Implicit cursor

Explicit cursor

Explicit Cursor Operations



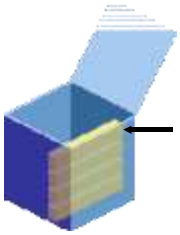
Controlling Explicit Cursors



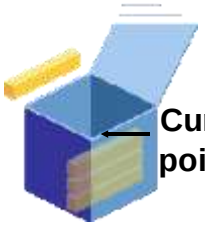
- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to FETCH if rows are found.
- Release the active set.

Controlling Explicit Cursors

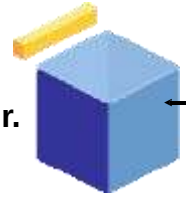
1 Open the cursor.



2 Fetch a row.



3 Close the cursor.



Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples

```
DECLARE
  CURSOR c_emp_cursor IS
  SELECT employee_id, last_name FROM employees  WHERE
  department_id =30;
```

```
DECLARE
  v_locid NUMBER:= 1700;  CURSOR c_dept_cursor IS
  SELECT * FROM departments  WHERE location_id =
  v_locid;
...
```

Opening the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees  WHERE
   department_id =30;
...
BEGIN
  OPEN c_emp_cursor;
```

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees  WHERE
   department_id =30;
  v_empno employees.employee_id%TYPE;
  v_lname employees.last_name%TYPE;  BEGIN
    OPEN c_emp_cursor;
    FETCH c_emp_cursor INTO v_empno, v_lname;
  DBMS_OUTPUT.PUT_LINE( v_empno ||'     '||v_lname);
  END;
  /
```

```
anonymous block completed
114  Raphaely
```

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees  WHERE
   department_id =30;
  v_empno employees.employee_id%TYPE;  v_lname
  employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;  LOOP
    FETCH c_emp_cursor INTO v_empno, v_lname;  EXIT WHEN
    c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno ||'   '||v_lname);
  END LOOP;
  END;
  /
```

Closing the Cursor

```
...
  LOOP
    FETCH c_emp_cursor INTO empno, lname;  EXIT WHEN
    c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno ||' '||v_lname);  END
  LOOP;
   CLOSE c_emp_cursor;
  END;
  /
```

Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL record.

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees  WHERE
     department_id =30;
   v_emp_record     c_emp_cursor%ROWTYPE;  BEGIN
     OPEN c_emp_cursor;  LOOP
       FETCH c_emp_cursor INTO v_emp_record;  EXIT WHEN
    c_emp_cursor%NOTFOUND;
       DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                            ||'
                         '||v_emp_record.last_name);
     END LOOP;
  CLOSE c_emp_cursor;  END;
```

Cursor FOR Loops

```
FOR record_name IN cursor_name LOOP
  statement1;  statement2;
  . . .
END LOOP;
```

- The cursor `FOR` loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

ORACLE

Cursor FOR Loops

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees  WHERE
   department_id =30;
BEGIN
   FOR emp_record IN c_emp_cursor  LOOP
    DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
     ||' ' ||emp_record.last_name);  END LOOP;
 END;
 /
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

ORACLE

Explicit Cursor Attributes

Use explicit cursor attributes to obtain status information about a cursor.

Attribute	Type	Description
`%ISOPEN`	Boolean	Evaluates to `TRUE` if the cursor is open
`%NOTFOUND`	Boolean	Evaluates to `TRUE` if the most recent fetch does not return a row
`%FOUND`	Boolean	Evaluates to `TRUE` if the most recent fetch returns a row; complement of `%NOTFOUND`
`%ROWCOUNT`	Number	Evaluates to the total number of rows returned so far

%ISOPEN Attribute

- You can fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

```
 IF NOT c_emp_cursor%ISOPEN THEN  OPEN c_emp_cursor;
END IF;  LOOP
FETCH c_emp_cursor...
```

%ROWCOUNT and %NOTFOUND:

```
DECLARE
  CURSOR c_emp_cursor IS SELECT employee_id,  last_name FROM
    employees;
v_emp_record    c_emp_cursor%ROWTYPE;
BEGIN
OPEN c_emp_cursor;  LOOP
 FETCH c_emp_cursor INTO v_emp_record;
 EXIT WHEN c_emp_cursor%ROWCOUNT > 10 OR
                    c_emp_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                                  ||'
              '||v_emp_record.last_name);
                   END LOOP;
                   CLOSE c_emp_cursor;
END ; /
```

```
anonymous block completed
174 Abel
166 Ande
130 Atkinson
105 Austin
204 Baer
116 Baida
167 Banda
172 Bates
192 Bell
151 Bernstein
```

ORACLE

Cursor FOR Loops Using Subqueries

```
BEGIN
  FOR emp_record IN (SELECT employee_id, last_name  FROM
   employees WHERE department_id =30)
 LOOP
  DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
   ||' '||emp_record.last_name);  END LOOP;
 END;
 /
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

ORACLE

Cursors with Parameters

Syntax:

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

- Pass parameter values to a cursor when the cursor is opened and the query is executed.
- Open an explicit cursor several times with a different active set each time.

```
OPEN   cursor_name(parameter_value,.....) ;
```


cursor_name	Is a PL/SQL identifier for the declared cursor
parameter_name	Is the name of a parameter
datatype	Is the scalar data type of the parameter
select_statement	Is a `SELECT` statement without the `INTO` clause

The parameter notation does not offer greater functionality; it simply allows you to specify input values easily and clearly. This is particularly useful when the same cursor is referenced repeatedly.

Cursors with Parameters

```
DECLARE
  CURSOR  c_emp_cursor (deptno NUMBER) IS  SELECT
     employee_id, last_name
  FROM    employees
  WHERE   department_id = deptno;
   ...  BEGIN
     OPEN c_emp_cursor (10);
     ...
     CLOSE c_emp_cursor;  OPEN c_emp_cursor (20);
     ...
```

FOR UPDATE Clause

Syntax:

```
SELECT ...
FROM              ...
FOR UPDATE [OF column reference][NOWAIT | WAIT n];
```

- Use explicit locking to deny access to other sessions for the duration of a transaction.
- Lock the rows *before* the update or delete.

ORACLE

FOR UPDATE Clause

If there are multiple sessions for a single database, there is the possibility that the rows of a particular table were updated after you opened your cursor. You see the updated data only when you reopen the cursor. Therefore, it is better to have locks on the rows before you update or delete rows. You can lock the rows with the `FOR UPDATE` clause in the cursor query.

In the syntax:

column_reference	Is a column in the table against which the query is performed (A list of columns may also be used.)
`NOWAIT`	Returns an Oracle Server error if the rows are locked by another session

The `FOR UPDATE` clause is the last clause in a `SELECT` statement, even after `ORDER BY` (if it exists). When you want to query multiple tables, you can use the `FOR UPDATE` clause to confine row locking to particular tables. `FOR UPDATE OF col_name(s)` locks rows only in tables that contain `col_name(s)`.

WHERE CURRENT OF Clause

```
WHERE CURRENT OF cursor ;
```

- Use cursors to update or delete the current row.
- Include the `FOR UPDATE` clause in the cursor query to first lock the rows.
- Use the `WHERE CURRENT OF` clause to reference the current row from an explicit cursor.

```
UPDATE employees
   SET    salary = ...
   WHERE CURRENT OF c_emp_cursor;
```

ORACLE

WHERE CURRENT OF Clause

The `WHERE CURRENT OF` clause is used in conjunction with the `FOR UPDATE` clause to refer to the current row in an explicit cursor. The `WHERE CURRENT OF` clause is used in the `UPDATE` or `DELETE` statement, whereas the `FOR UPDATE` clause is specified in the cursor declaration.

You can use the combination for updating and deleting the current row from the corresponding database table. This enables you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the row ID. You must include the `FOR UPDATE` clause in the cursor query so that the rows are locked on `OPEN`.

In the syntax:

cursor Is the name of a declared cursor (The cursor must have been declared with the `FOR UPDATE` clause.)

Summary

In this lesson, you should have learned to:

- Distinguish cursor types:
  - Implicit cursors are used for all DML statements and single-row queries.
  - Explicit cursors are used for queries of zero, one, or more rows.
- Create and handle explicit cursors
- Use simple loops and cursor `FOR` loops to handle multiple rows in the cursors
- Evaluate cursor status by using cursor attributes
- Use the `FOR UPDATE` and `WHERE CURRENT OF` clauses to update or delete the current fetched row

ORACLE

Summary

The Oracle Server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return multiple rows, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor `FOR` loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor `FOR` loops do this implicitly. If you are updating or deleting rows, lock the rows by using a `FOR UPDATE` clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a `WHERE CURRENT OF` clause in conjunction with the `FOR UPDATE` clause to reference the current row fetched by the cursor.

8

Handling Exceptions

What Is an Exception?

```
DECLARE
  v_lname VARCHAR2(15);  BEGIN
    SELECT last_name INTO v_lname  FROM employees
    WHERE first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is :'
  ||v_lname);

END;
```

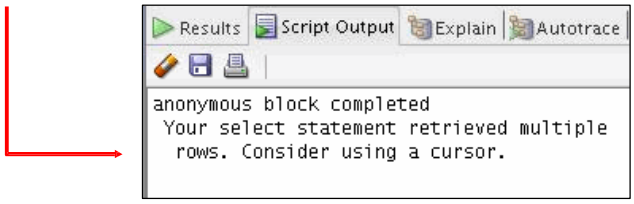
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
Error starting at line 3 in command:
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname FROM employees WHERE
  first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
END;
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

ORACLE

Handling the Exception: An Example

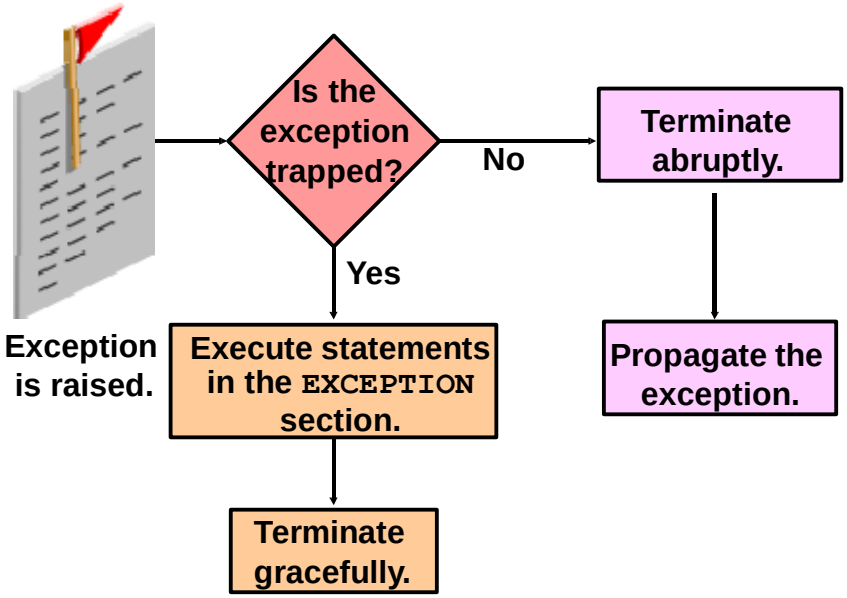
```
DECLARE
  v_lname VARCHAR2(15);  BEGIN
    SELECT last_name INTO v_lname  FROM employees
    WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is :' ||v_lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement retrieved
   multiple rows. Consider using a cursor.');
END;
/
```



Understanding Exceptions with PL/SQL

- An exception is a PL/SQL error that is raised during program execution.
- An exception can be raised:
  - Implicitly by the Oracle Server
  - Explicitly by the program
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

Handling Exceptions



Syntax to Trap Exceptions

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;  statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;  statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;  statement2;
    . . .]
```

Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - `NO_DATA_FOUND`
  - `TOO_MANY_ROWS`
  - `INVALID_CURSOR`
  - `ZERO_DIVIDE`
  - `DUP_VAL_ON_INDEX`

Non-Predefined Error Trapping: Example

To trap Oracle Server error `01400` ("cannot insert `NULL`").

```
DECLARE
 PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);
 e_insert_excep EXCEPTION;
BEGIN
  INSERT INTO departments
  (department_id, department_name) VALUES (280,
EXCEPTION
 WHEN e_insert_excep
 THEN DBMS_OUTPUT.PUT_LINE('INSERT OPERATION
   FAILED');  DBMS_OUTPUT.PUT_LINE(SQLERRM);
   END;
   /
```

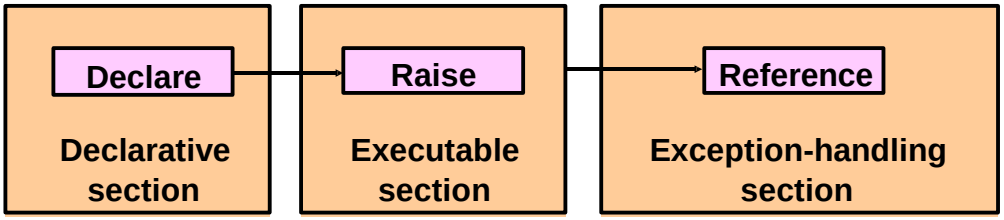
1 2 3

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
anonymous block completed
INSERT OPERATION FAILED
ORA-01400: cannot insert NULL into ("ORA41"."DEPARTMENTS"."DEPARTMENT_NAME")
```

ORACLE

Trapping User-Defined Exceptions



Name the exception.

Explicitly raise the exception by using the `RAISE` statement.

Handle the raised exception.

Trapping User-Defined Exceptions

```
DECLARE
v_deptno NUMBER := 500;
v_name VARCHAR2(20) := 'Testing';
e_invalid_department EXCEPTION;
BEGIN
UPDATE departments
SET department_name = v_name  WHERE
department_id = v_deptno;
IF SQL%NOTFOUND THEN
RAISE e_invalid_department;  END IF;
COMMIT;  EXCEPTION
WHEN e_invalid_department THEN

  DBMS_OUTPUT.PUT_LINE('No such department
id.');  END;
  /
```

1

2

3

Results | Script Output | Explain

```
anonymous block completed
No such department id.
```

Propagating Exceptions in a Subblock

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  . . .
  e_no_rows       exception;
  e_integrity     exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP  BEGIN
   SELECT ...
   UPDATE ...
   IF SQL%NOTFOUND THEN
       RAISE e_no_rows;  END IF;
    END;  END LOOP;
  EXCEPTION
    WHEN e_integrity THEN ...  WHEN e_no_rows
  THEN ...
  END;
  /
```

RAISE_APPLICATION_ERROR Procedure

```
raise_application_error (error_number,
             message[, {TRUE | FALSE}]);
```

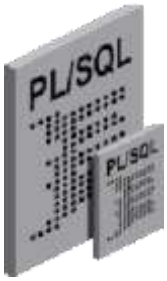
- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

9

Introducing Stored Procedures and Functions

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without the `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



ORACLE

Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	If functions, must return values
Cannot take parameters	Can take parameters

Block Structure for Anonymous PL/SQL Blocks

`DECLARE` **(optional)**
Declares PL/SQL objects to be used within this block

`BEGIN` **(mandatory)**
Defines the executable statements

`EXCEPTION` **(optional)**
Defines the actions that take place if an error or exception arises

`END;` **(mandatory)**

ORACLE

ORACLE

Block Structure for PL/SQL Subprograms

<header>

`IS | AS`

→ Subprogram specification

Declaration section

`BEGIN`

Executable section

`EXCEPTION`

(optional) Exception section

→ Subprogram body

`END;`

What Is a Procedure?

- A procedure is a type of subprogram that performs an action.
- A procedure can be stored in the database, as a schema object, for repeated execution.

ORACLE

9-10

ORACLE

Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments), and be invoked. Generally speaking, you use a procedure to perform an action. A procedure has a header, a declaration section, an executable section, and an optional exception-handling section.

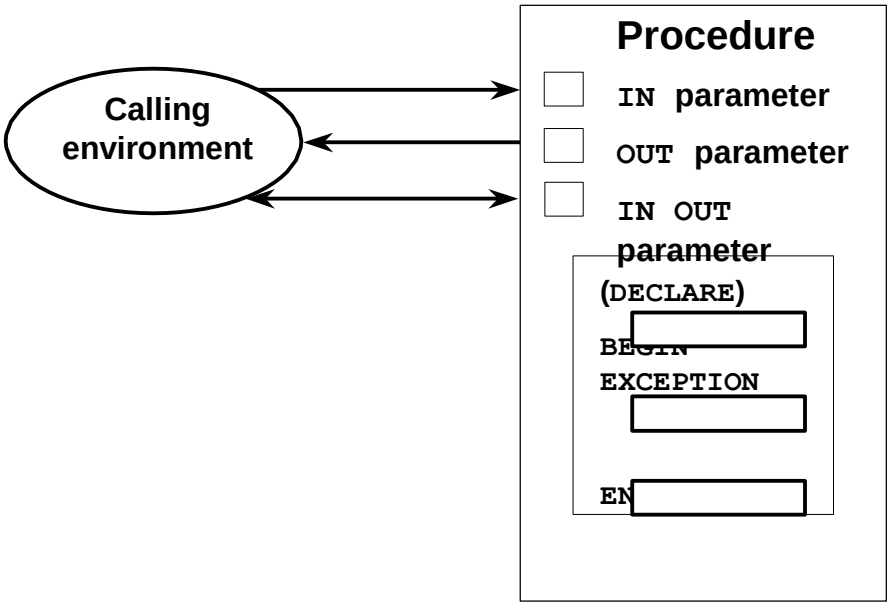
A procedure can be compiled and stored in the database as a schema object.

Procedures promote reusability and maintainability. When validated, they can be used in any number of applications. If the requirements change, only the procedure needs to be updated.

Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(argument1 [mode1] datatype1,  argument2 [mode2]
   datatype2,
  . . .)]
IS|AS
procedure_body;
```

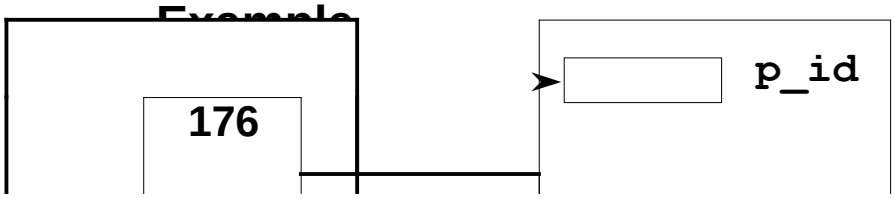
Procedural Parameter Modes



Type of Parameter	Description
IN (default)	Passes a constant value from the calling environment into the procedure
OUT	Passes a value from the procedure to the calling environment
IN OUT	Passes a value from the calling environment into the procedure and a possibly different value from the procedure back to the calling environment using the same parameter

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

IN Parameters: Example




```
CREATE OR REPLACE PROCEDURE raise_salary  (p_id IN
  employees.employee_id%TYPE)
IS
BEGIN
UPDATE employees
SET    salary = salary * 1.10  WHERE employee_id = p_id;
END raise_salary;
/
```

Procedure created.

ORACLE

ORACLE

IN Parameters: Example

The example in the slide shows a procedure with one `IN` parameter. Running this statement in *i*SQL*Plus creates the `RAISE_SALARY` procedure. When invoked, `RAISE_SALARY` accepts the parameter for the employee ID and updates the employee's record with a salary increase of 10 percent.

To invoke a procedure in *i*SQL*Plus, use the `EXECUTE` command.

```
EXECUTE raise_salary (176)
```

To invoke a procedure from another procedure, use a direct call. At the location of calling the new procedure, enter the procedure name and actual parameters.

```
raise_salary (176);
```

`IN` parameters are passed as constants from the calling environment into the procedure. Attempts to change the value of an `IN` parameter result in an error.

Creating a Procedure

```
...
CREATE TABLE dept AS SELECT * FROM departments;
CREATE PROCEDURE add_dept IS  v_dept_id
 dept.department_id%TYPE;
 v_dept_name dept.department_name%TYPE;  BEGIN
  v_dept_id:=280;  v_dept_name:='ST-Curriculum';
  INSERT INTO dept(department_id,department_name)
  VALUES(v_dept_id,v_dept_name);
 DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT
 ||' row ');  END;
```

Using OUT Parameters: Example

```
CREATE OR REPLACE PROCEDURE query_emp
(id  IN   employees.employee_id%TYPE,
name OUT employees.last_name%TYPE,  salary OUT
employees.salary%TYPE) IS
BEGIN
SELECT   last_name, salary INTO name, salary
FROM employees
WHERE    employee_id = id;  END query_emp;
```

```
DECLARE
emp_name employees.last_name%TYPE;
emp_sal  employees.salary%TYPE;
BEGIN
query_emp(171, emp_name, emp_sal); ...
END;
```

ORACLE

Using IN OUT Parameters: Example

Calling environment

phone_no (before the call)

'8006330575'

phone_no (after the call)

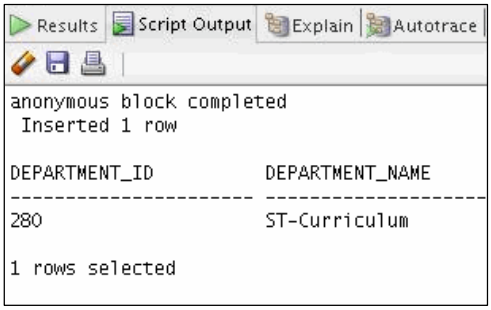
'(800)633-0575'

```
CREATE OR REPLACE PROCEDURE format_phone  (phone_no
  IN OUT VARCHAR2) IS
BEGIN
phone_no := '(' || SUBSTR(phone_no,1,3) ||  ')' ||
  SUBSTR(phone_no,4,3)      ||          '-'      ||
  SUBSTR(phone_no,7);
END format_phone;
/
```

ORACLE

Invoking a Procedure

```
...  BEGIN
 add_dept;  END;
 /
 SELECT department_id, department_name FROM dept
WHERE department_id=280;
```



Syntax for Passing Parameters

- **Positional:**
  - **Lists the actual parameters in the same order as the formal parameters**
- **Named:**
  - **Lists the actual parameters in arbitrary order and uses the association operator (=>) to associate a named formal parameter with its actual parameter**
- **Combination:**
  - **Lists some of the actual parameters as positional and some as named**

ORACLE

Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(
name IN departments.department_name%TYPE,  loc IN
departments.location_id%TYPE) IS
BEGIN
INSERT INTO departments(department_id,
department_name, location_id)
VALUES (departments_seq.NEXTVAL, name, loc);  END
add_dept;
/
```

- Passing by positional

```
EXECUTE add_dept ('TRAINING', 2500)
```

- Passing by named

```
EXECUTE add_dept (loc=>2400, name=>'EDUCATION')
```

ORACLE

Using the DEFAULT Option for Parameters

- Defines default values for

```
CREATE OR REPLACE PROCEDURE add_dept(
name departments.department_name%TYPE :='Unknown',
loc  departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
INSERT INTO departments (...)
VALUES (departments_seq.NEXTVAL, name, loc);  END
add_dept;
```

- Provides flexibility by combining the positional and named parameter-passing

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', loc => 1200)
EXECUTE add_dept (loc => 1200)
```

ORACLE

Invoking Procedures

You can invoke procedures by:

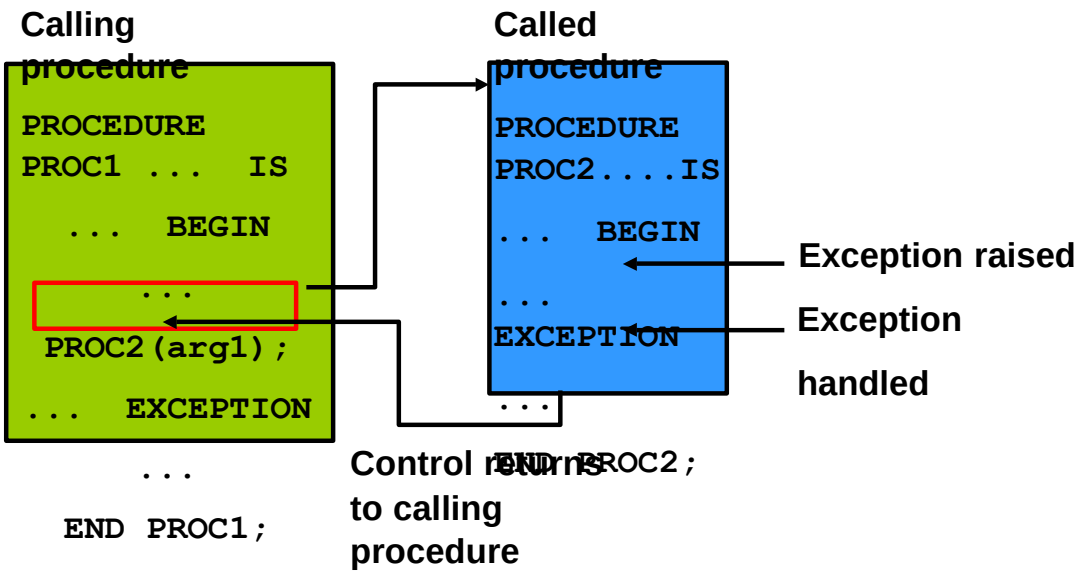
- **Using anonymous blocks**
- **Using another procedure,**

as in the following example:

```
CREATE OR REPLACE PROCEDURE process_employees  IS
CURSOR emp_cursor IS  SELECT employee_id  FROM
     employees;
BEGIN
FOR emp_rec IN emp_cursor  LOOP
       raise_salary(emp_rec.employee_id, 10);
                END LOOP;  COMMIT;
              END process_employees;
                        /
```


ORACLE

Handled Exceptions

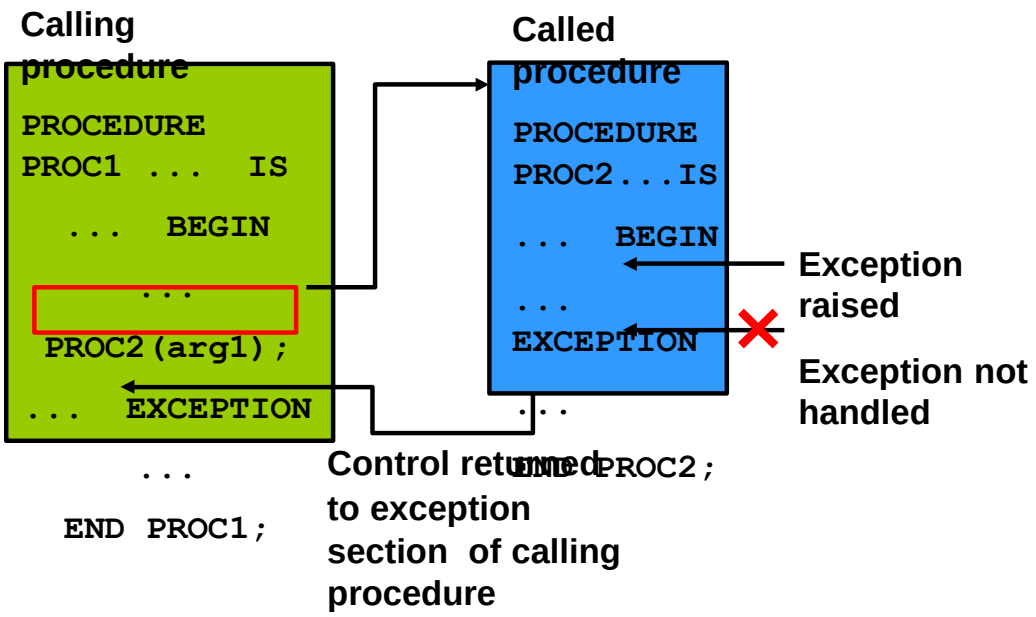


Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
name VARCHAR2, mgr NUMBER, loc NUMBER) IS  BEGIN
INSERT INTO DEPARTMENTS (department_id,
department_name, manager_id, location_id)
VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);  EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Err: adding dept: '||name);  END;
```

```
CREATE PROCEDURE create_departments IS  BEGIN
add_department('Media', 100, 1800);
add_department('Editing', 99, 1800);
add_department('Advertising', 101, 1800);  END;
```

Exceptions Not Handled



Exceptions Not Handled: Example

```
SET SERVEROUTPUT ON
CREATE PROCEDURE add_department_noex(
name VARCHAR2, mgr NUMBER, loc NUMBER) IS  BEGIN
INSERT INTO DEPARTMENTS (department_id,  department_name,
manager_id, location_id)
VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);  END;
```

```
CREATE PROCEDURE create_departments_noex IS  BEGIN
add_department_noex('Media', 100, 1800);
add_department_noex('Editing', 99, 1800);
add_department_noex('Advertising', 101, 1800);  END;
```

Removing Procedures

You can remove a procedure that is stored in the database.

- Syntax:

```
DROP PROCEDURE procedure_name
```

- Example

```
DROP PROCEDURE raise_salary;
```

ORACLE

Overview of Stored Functions

A function:

- **Is a named PL/SQL block that returns a value**
- **Can be stored in the database as a schema object for repeated execution**
- **Is called as part of an expression or is used to provide a parameter value**

ORACLE

Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
  [(argument1 [mode1] datatype1,  argument2 [mode2]
   datatype2,
  . . .)]
RETURN datatype
IS|AS
function_body;
```

Stored Function: Example

- **Create the function:**

```
CREATE OR REPLACE FUNCTION get_sal
(id employees.employee_id%TYPE) RETURN NUMBER IS
sal employees.salary%TYPE := 0;
BEGIN
SELECT salary  INTO    sal
FROM employees
WHERE    employee_id = id;  RETURN sal;
END get_sal;
/
```

- **Invoke the function as an expression or as a parameter value:**

```
EXECUTE dbms_output.put_line(get_sal(100))
```

ORACLE

Ways to Execute Functions

- Invoke as part of a PL/SQL expression
  - Using a host variable to obtain the result:

```
VARIABLE salary NUMBER
EXECUTE :salary := get_sal(100)
```

  - Using a local variable to obtain the result:

```
DECLARE sal employees.salary%type;  BEGIN
sal := get_sal(100); ...
END;
```

- Use as a parameter to another

```
EXECUTE dbms_output.put_line(get_sal(100))
```

- Use in a SQL statement (subject to

```
SELECT job_id, get_sal(employee_id) FROM employees;
```

Advantages of User-Defined Functions in SQL Statements

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

ORACLE

Function in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(value IN NUMBER)
 RETURN NUMBER IS
BEGIN
RETURN (value * 0.08);  END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
 FROM employees
WHERE department_id = 100;
```

```
Function created.
```

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

```
6 rows selected.
```

ORACLE

Locations to Call User-Defined Functions

User-defined functions act like built-in single-row functions and can be used in:

- The `SELECT` list or clause of a query
- Conditional expressions of the `WHERE` and `HAVING` clauses
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Be stored in the database
  - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
  - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
  - Parameters must be specified with positional notation
  - You must own the function or have the `EXECUTE` privilege

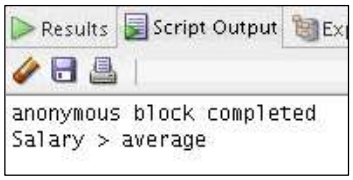
ORACLE

Creating a Function

```
CREATE FUNCTION check_sal RETURN Boolean IS
v_dept_id employees.department_id%TYPE;  v_empno
     employees.employee_id%TYPE;  v_sal
     employees.salary%TYPE;  v_avg_sal
 employees.salary%TYPE;
BEGIN
v_empno:=205;
 SELECT salary,department_id INTO v_sal,v_dept_id FROM
employees
  WHERE employee_id= v_empno;
 SELECT avg(salary) INTO v_avg_sal FROM employees WHERE
department_id=v_dept_id;
 IF v_sal > v_avg_sal THEN  RETURN TRUE;
ELSE
  RETURN FALSE;  END IF;  EXCEPTION
  WHEN NO_DATA_FOUND THEN  RETURN NULL;
END;
```

Invoking a Function

```
BEGIN
 IF (check_sal IS NULL) THEN  DBMS_OUTPUT.PUT_LINE('The
 function returned
  NULL due to exception');  ELSIF (check_sal) THEN
  DBMS_OUTPUT.PUT_LINE('Salary > average');
  ELSE
  DBMS_OUTPUT.PUT_LINE('Salary < average');  END IF;
 END;
 /
```



Passing a Parameter to the Function

```
DROP FUNCTION check_sal;
CREATE FUNCTION check_sal(p_empno employees.employee_id%TYPE)
RETURN Boolean IS
 v_dept_id employees.department_id%TYPE;  v_sal
     employees.salary%TYPE;  v_avg_sal employees.salary%TYPE;
BEGIN
 SELECT salary,department_id INTO v_sal,v_dept_id FROM
   employees  WHERE employee_id=p_empno;
 SELECT avg(salary) INTO v_avg_sal FROM employees  WHERE
   department_id=v_dept_id;
 IF v_sal > v_avg_sal THEN  RETURN TRUE;
ELSE
  RETURN FALSE;  END IF;  EXCEPTION
    ...
```

Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
 IF (check_sal(205) IS NULL) THEN  DBMS_OUTPUT.PUT_LINE('The
 function returned
  NULL due to exception');  ELSIF (check_sal(205)) THEN
  DBMS_OUTPUT.PUT_LINE('Salary > average');  ELSE
  DBMS_OUTPUT.PUT_LINE('Salary < average');  END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');  IF
 (check_sal(70) IS NULL) THEN  DBMS_OUTPUT.PUT_LINE('The
 function returned
  NULL due to exception');  ELSIF (check_sal(70)) THEN
 ...  END IF;  END;
 /
```

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(sal NUMBER)
   RETURN NUMBER IS
BEGIN
INSERT INTO employees(employee_id, last_name,
email, hire_date, job_id, salary)  VALUES(1,
   'Frost', 'jfrost@company.com',
SYSDATE, 'SA_MAN', sal);  RETURN (sal + 100);
END;
```

```
UPDATE employees
SET salary = dml_call_sql(2000)  WHERE employee_id
= 170;
```

```
UPDATE employees SET salary = dml_call_sql(2000)
                 *  ERROR at line 1:
        ORA-04091: table PLSQL.EMPLOYEES is mutating,
            trigger/function may not see it
      ORA-06512: at "PLSQL.DML_CALL_SQL", line 4
```

Removing Functions

Removing a stored function:

- **You can drop a stored function by using the following syntax:**

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

- **All the privileges that are granted on a function are revoked when the function is dropped.**
- **The `CREATE OR REPLACE` syntax is equivalent to dropping a function and re-creating it. Privileges granted on the function remain the same when this syntax is used.**

ORACLE

Procedures Versus Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain `RETURN` clause in the header	Must contain a `RETURN` clause in the header
Can return values (if any) in output parameters	Must return a single value
Can contain a `RETURN` statement without a value	Must contain at least one `RETURN` statement

ORACLE