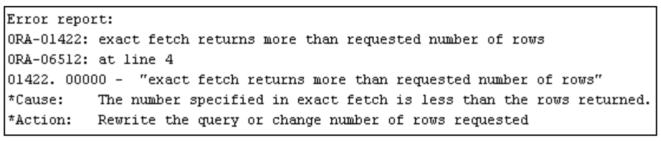
HANDLING EXCEPTIONS

ORACLE

Example of 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;
```



```
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

Example of 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);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement
  retrieved multiple rows. Consider using a
  cursor.');
END;
/
```

```
anonymous block completed
 Your select statement retrieved multiple
  rows. Consider using a cursor.
```

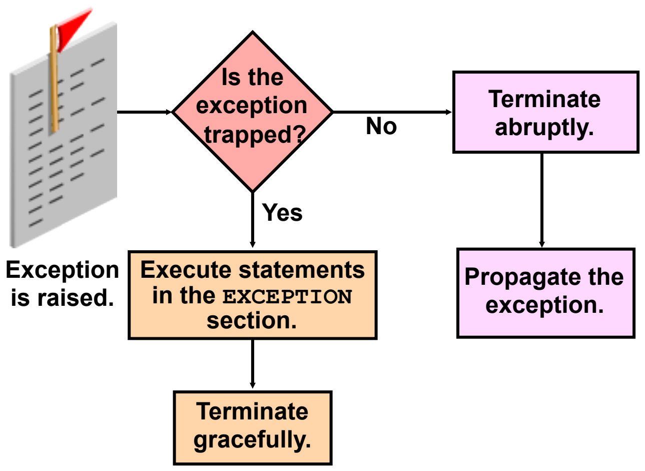
ORACLE

Handling Exceptions with PL/SQL

- Une exception est une erreur de PL/SQL qui est déclenchée pendant l'exécution du programme.
- Une exception peut être levée :
  - implicitement par le serveur Oracle
  - explicitement par le programme
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

ORACLE

Handling Exceptions



II-148

Exception Types

- Predefined Oracle server
- Non-predefined Oracle server

} Implicitly raised

- User-defined

Explicitly raised

II-149

Trapping Exceptions

Syntax:

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

ORACLE

Guidelines for Trapping Exceptions

- **The `EXCEPTION` keyword starts the exception-handling section.**
- **Several exception handlers are allowed.**
- **Only one handler is processed before leaving the block.**
- **`WHEN OTHERS` is the last clause.**

ORACLE

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`

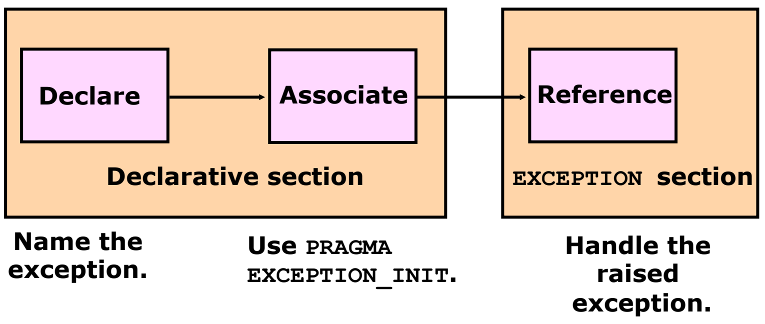
ORACLE

Trapping Predefined Oracle Server Errors

```
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 NO_DATA_FOUND THEN
     DBMS_OUTPUT.PUT_LINE (' Your select statement
     retrieved no rows.');
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement
     retrieved multiple rows. Consider using a
     cursor.');
END;
/
```

ORACLE

Trapping Non-Predefined Oracle Server Errors



ORACLE

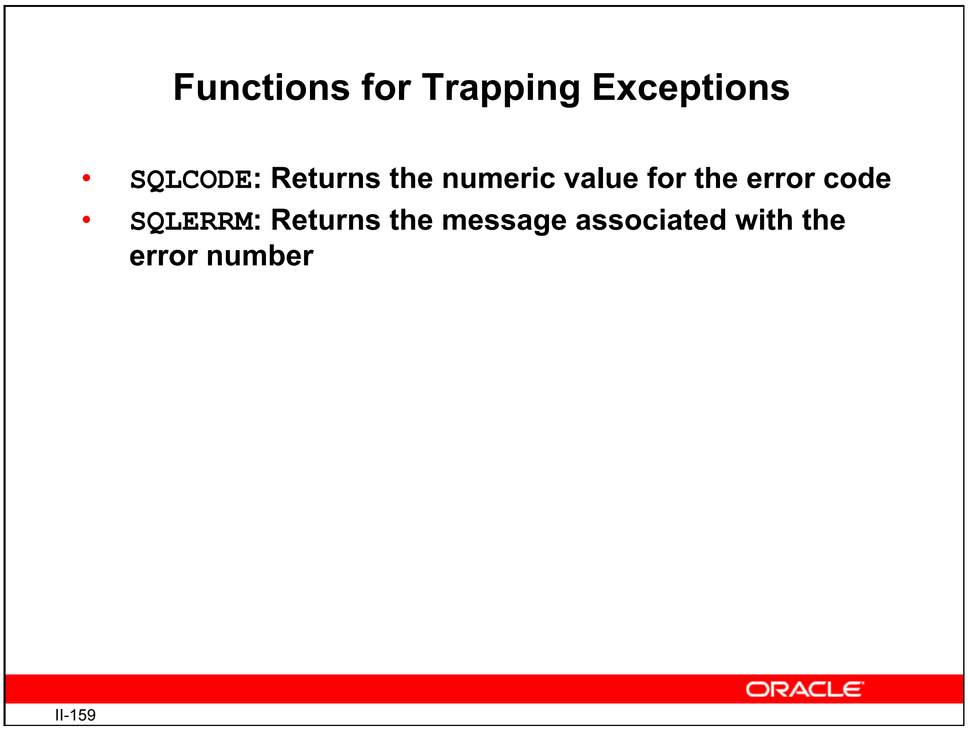
Non-Predefined Error

To trap Oracle server error number `–01400` (“cannot insert `NULL`”):

```
DECLARE
 e_insert_excep EXCEPTION;                          1
 PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);          2
BEGIN
 INSERT INTO departments
 (department_id, department_name) VALUES (280, NULL);
EXCEPTION                                           3
 WHEN e_insert_excep THEN
   DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
   DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

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

ORACLE

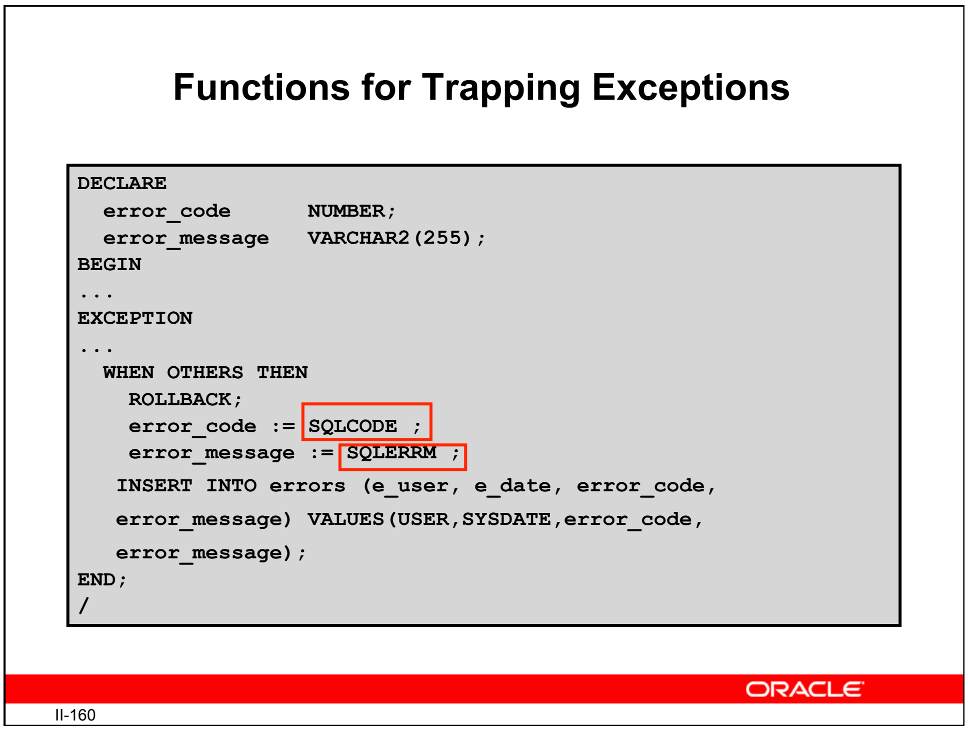


Functions for Trapping Exceptions

- `SQLCODE`: Returns the numeric value for the error code
- `SQLERRM`: Returns the message associated with the error number

ORACLE

II-159



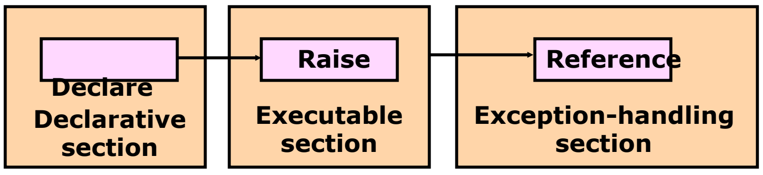
Functions for Trapping Exceptions

```
DECLARE
  error_code      NUMBER;
  error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    error_code  := SQLCODE ;
    error_message := SQLERRM ;
   INSERT INTO errors (e_user, e_date, error_code,
   error_message) VALUES(USER,SYSDATE,error_code,
   error_message);
END;
/
```

ORACLE

II-160

Trapping User-Defined Exceptions



Name the exception.

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

Handle the raised exception.

ORACLE

Trapping User-Defined Exceptions

```
DECLARE
  v_deptno NUMBER := 500;
  v_name VARCHAR2(20) := 'Testing';
  e_invalid_department EXCEPTION;          1
BEGIN
  UPDATE departments
  SET department_name = v_name
  WHERE department_id = v_deptno;
  IF SQL % NOTFOUND THEN
    RAISE e_invalid_department;            2
  END IF;
  COMMIT;                                  3
EXCEPTION
WHEN e_invalid_department THEN
  DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```

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

ORACLE

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;
/
```

ORACLE

`RAISE_APPLICATION_ERROR` Procedure

Syntax:

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

- **Vous pouvez utiliser cette procédure pour émettre des messages d'erreur définis par l'utilisateur de sous-programmes stockées.**
- **Vous pouvez signaler des erreurs de votre application et éviter le déclenchement d’exceptions non gérées.**

ORACLE

RAISE_APPLICATION_ERROR Procedure

- Used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

ORACLE

II-165

RAISE_APPLICATION_ERROR Procedure

Executable section:

```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

Exception section:

```
...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
     RAISE_APPLICATION_ERROR (-20201,
        'Manager is not a valid employee.');
END;
/
```

ORACLE

II-166

Exercices

a. In the declarative section, declare two variables: `v_ename` of type `employees.last_name` and `v_emp_sal` of type `employees.salary`. Initialize the latter to 6000.

b. In the executable section, retrieve the last names of employees whose salaries are equal to the value in `v_emp_sal`.
Note: Do not use explicit cursors.
If the salary entered returns only one row, insert into the `messages` table the employee's name and the salary amount.

c. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "No employee with a salary of <*salary*>."

d. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the `messages` table the message "More than one employee with a salary of <*salary*>."

e. Handle any other exception with an appropriate exception handler and insert into the `messages` table the message "Some other error occurred."

ORACLE

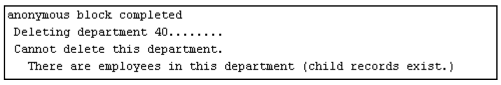
Exercices

2. Use the Oracle server error `ORA-02292 (integrity constraint violated – child record found).`

   a. In the declarative section, declare an exception `e_childrecord_exists`. Associate the declared exception with the standard Oracle server error `–02292`.

   b. In the executable section, display "Deleting department 40…." Include a `DELETE` statement to delete the department with `department_id` 40.

   c. Include an exception section to handle the `e_childrecord_exists` exception and display the appropriate message. Sample output is as follows:



```
anonymous block completed
 Deleting department 40........
 Cannot delete this department.
   There are employees in this department (child records exist.)
```

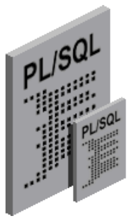
3. Rewrite the block to remove all departments who have no employee

ORACLE

CREATING STORED PROCEDURES AND FUNCTIONS

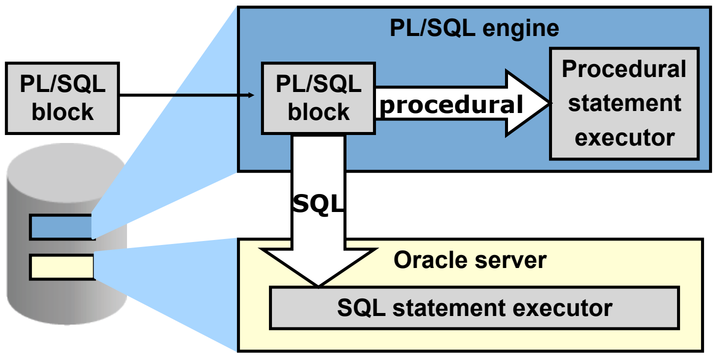
Procedures and Functions

- Sont des block PL/SQL nommés
- Ont une structure semblable à celle des blocs anonymes :
  - Optional declarative section (without the `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



PL/SQL Execution Environment

The PL/SQL run-time architecture:

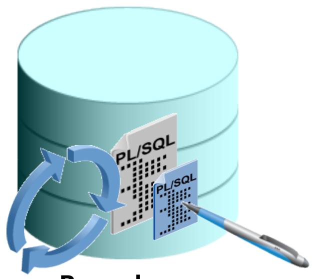


ORACLE

II-174

What Are Procedures?

- Sont un type de sous-programme qui exécutent une action
- Peuvent être stockés dans la base de données comme un objet de schéma
- Promeuvent la réutilisation et la maintenabilité

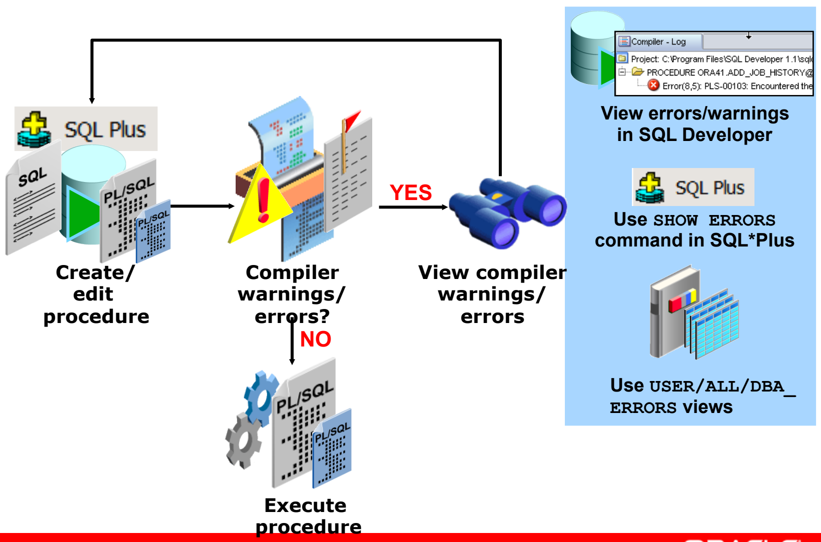


Procedures

ORACLE

II-175

Creating Procedures: Overview


SQL Plus
SQL
PL/SQL
Create/ edit procedure
Compiler warnings/ errors?
YES
NO
View compiler warnings/ errors
Compiler - Log
View errors/warnings in SQL Developer
SQL Plus
Use SHOW ERRORS command in SQL*Plus
Use USER/ALL/DBA_ ERRORS views
PL/SQL
Execute procedure

```
Project: C:\Program Files\SQL Developer 1.1\sql
PROCEDURE ORA41.ADD_JOB_HISTORY@
Error(8,5): PLS-00103: Encountered the
```

ORACLE

Procedure: Syntax

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

ORACLE

Procedure: Example

```
...
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;
```

Formal and Actual Parameters

- **Paramètres formels : les variables locales déclarées dans la liste de paramètres d'une spécification de sous-programme**
- **Véritables paramètres (ou arguments): valeurs littérales, variables et expressions utilisées dans la liste des paramètres de l'appel de sous-programme**

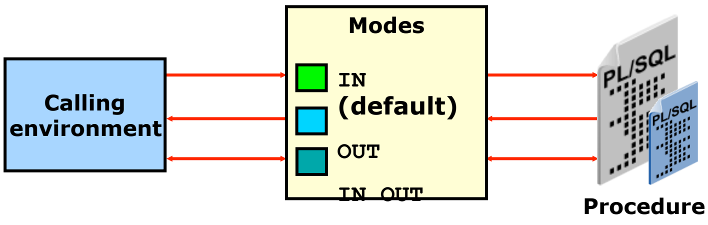
```
-- Procedure definition, Formal_parameters
CREATE PROCEDURE raise_sal(p_id NUMBER, p_sal NUMBER) IS
BEGIN
. . .
END raise_sal;
```

```
-- Procedure calling, Actual parameters (arguments)
v_emp_id := 100;
raise_sal(v_emp_id, 2000)
```

Procedural Parameter Modes

- Les modes des paramètre sont précisés dans la déclaration des paramètres formels, après le nom du paramètre et avant son type de données.
- Lee mode `IN` est la valeur par défaut si aucun mode n'est spécifié.

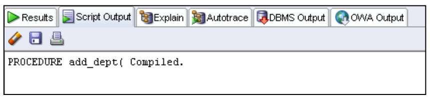
```
CREATE PROCEDURE proc_name(param_name [mode] datatype)
...
```



ORACLE

Passing Actual Parameters: Creating the `add_dept` Procedure

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

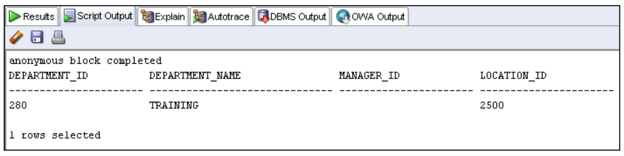


```
PROCEDURE add_dept( Compiled.
```

ORACLE

Passing Actual Parameters: Examples

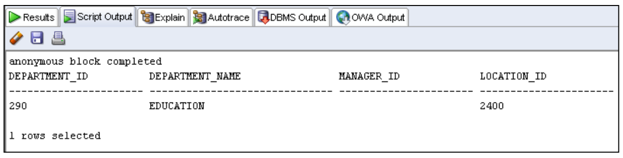
```
-- Passing parameters using the positional notation.
EXECUTE add_dept ('TRAINING', 2500)
```



```
anonymous block completed
DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- ----------------------
280                    TRAINING                                              2500

1 rows selected
```

```
-- Passing parameters using the named notation.
EXECUTE add_dept (p_loc=>2400, p_name=>'EDUCATION')
```



```
anonymous block completed
DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- ----------------------
290                    EDUCATION                                             2400

1 rows selected
```

Invoking the Procedure

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

```
anonymous block completed
 Inserted 1 row

DEPARTMENT_ID          DEPARTMENT_NAME
---------------------- ------------------------------
280                    ST-Curriculum

1 rows selected
```

Calling Procedures

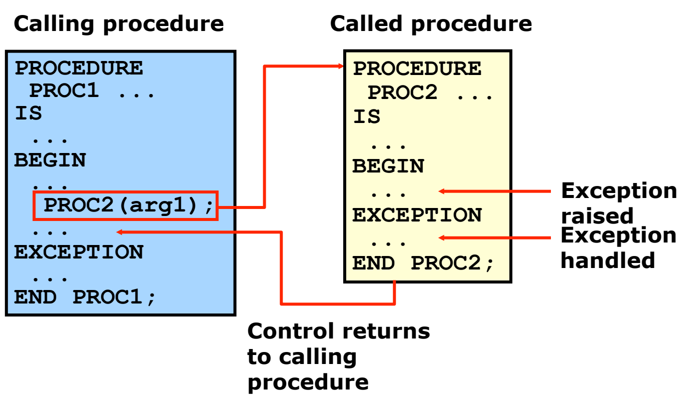
Vous pouvez appeler des procédures à l'aide de blocs anonymes, une autre procédure ou package.

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

```
PROCEDURE process_employees Compiled.
```

ORACLE

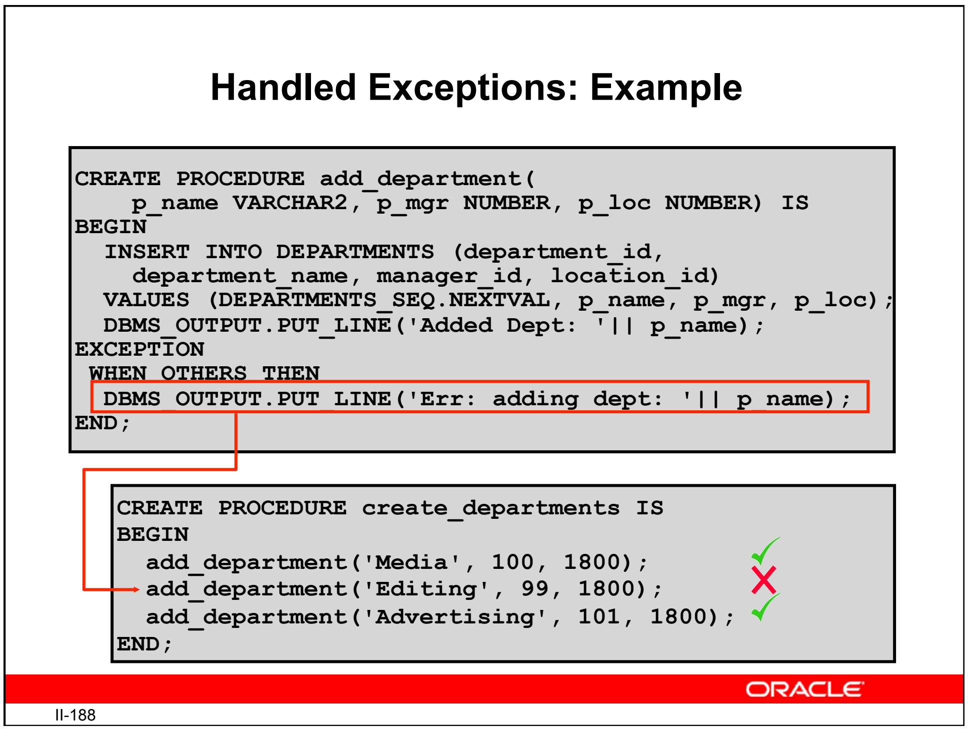
Handled Exceptions



```
PROCEDURE
 PROC1 ...
IS
 ...
BEGIN
 ...
  PROC2(arg1);
 ...
EXCEPTION
 ...
END PROC1;
```

```
PROCEDURE
 PROC2 ...
IS
 ...
BEGIN
 ...
EXCEPTION
 ...
END PROC2;
```

ORACLE

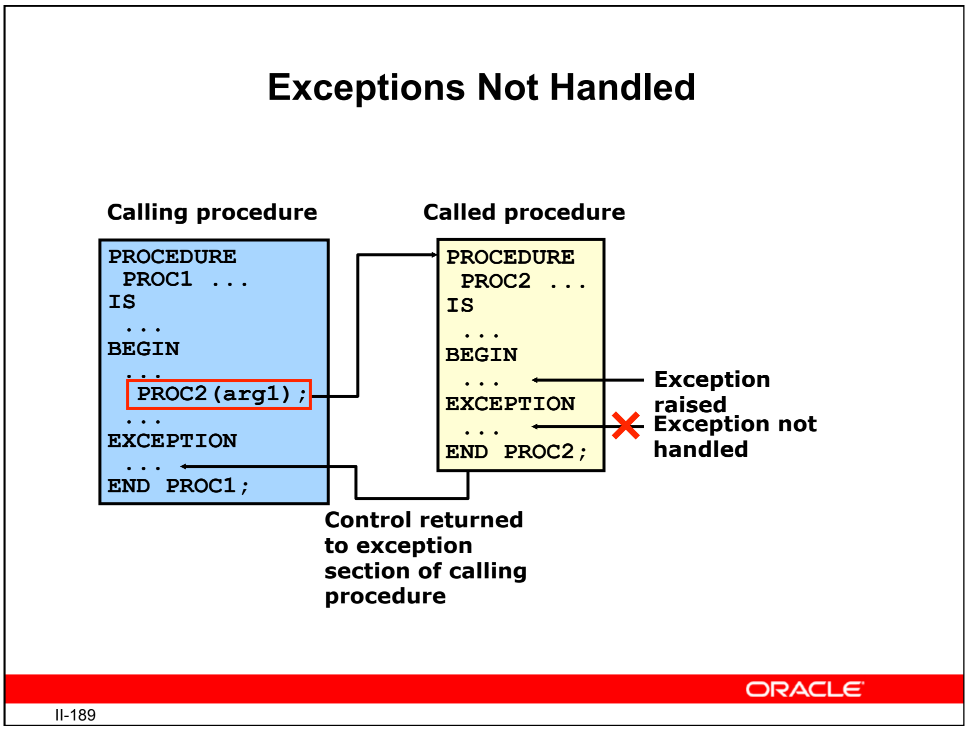


Handled Exceptions: Example

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

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

ORACLE



Exceptions Not Handled

Calling procedure

```
PROCEDURE
 PROC1 ...
IS
 ...
BEGIN
 ...
  PROC2(arg1);
 ...
EXCEPTION
 ...
END PROC1;
```

Called procedure

```
PROCEDURE
 PROC2 ...
IS
 ...
BEGIN
 ...
EXCEPTION
 ...
END PROC2;
```

Exception raised

Exception not handled

Control returned to exception section of calling procedure

ORACLE

Exceptions Not Handled: Example

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

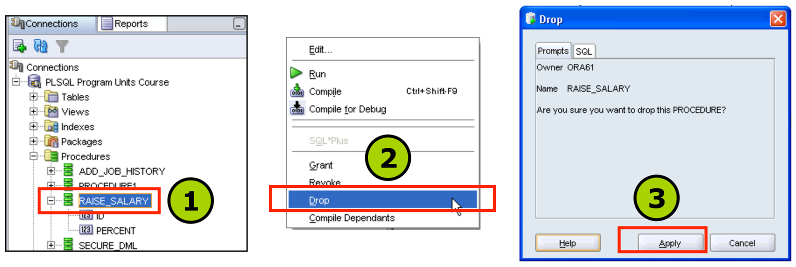
ORACLE

Removing Procedures: Using the `DROP` SQL Statement or SQL Developer

- **Using the `DROP` statement:**

```
DROP PROCEDURE raise_salary;
```

- **Using SQL Developer:**



Owner ORA61

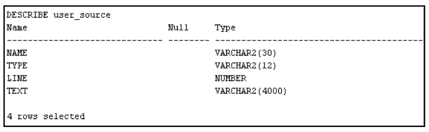
Name RAISE_SALARY

Are you sure you want to drop this PROCEDURE?

ORACLE

Viewing Procedure Information Using the Data Dictionary Views

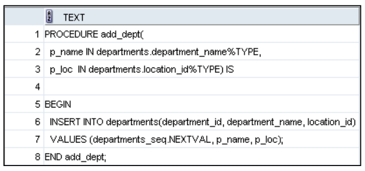
```
DESCRIBE user_source
```



```
DESCRIBE user_source
Name                           Null     Type
------------------------------ -------- ----------------------------------------
NAME                                    VARCHAR2(30)
TYPE                                    VARCHAR2(12)
LINE                                    NUMBER
TEXT                                    VARCHAR2(4000)

4 rows selected
```

```
SELECT text
FROM   user_source
WHERE  name = 'ADD_DEPT' AND type = 'PROCEDURE'
ORDER BY line;
```



```
  TEXT
1 PROCEDURE add_dept(
2   p_name IN departments.department_name%TYPE,
3   p_loc  IN departments.location_id%TYPE) IS
4
5 BEGIN
6   INSERT INTO departments(department_id, department_name, location_id)
7   VALUES (departments_seq.NEXTVAL, p_name, p_loc);
8 END add_dept;
```

ORACLE

Exercice

a. **Create a procedure called `CHECK_SALARY` as follows:**

i. The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.

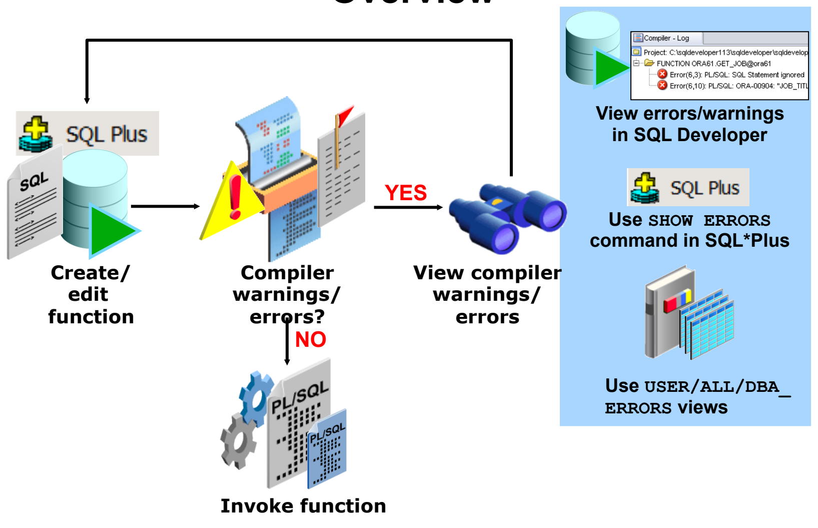
ii. The procedure uses the job ID to determine the minimum and maximum salary for the specified job.

iii. If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>".

b. **Create a procedure called `PROCESS_CHECK_SALARY` to check salary for all employees .**

ORACLE

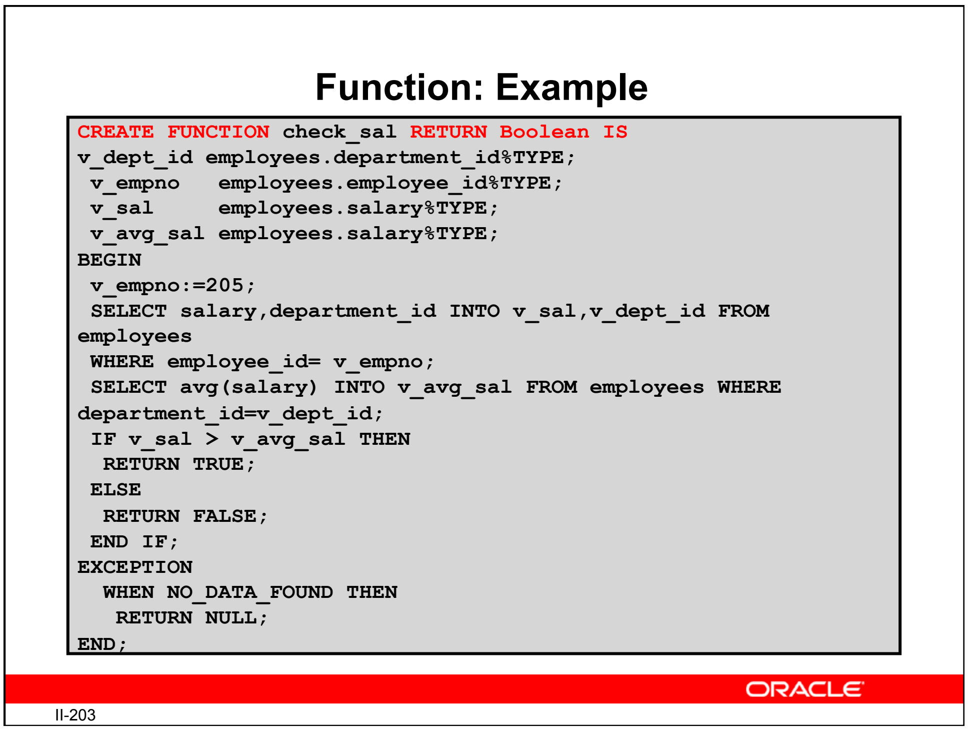
Creating and Running Functions: Overview


SQL Plus
SQL
Create/ edit function
Compiler warnings/ errors?
YES
NO
View compiler warnings/ errors
Invoke function
PL/SQL
View errors/warnings in SQL Developer
SQL Plus
Use SHOW ERRORS command in SQL*Plus
Use USER/ALL/DBA_ ERRORS views
Compiler - Log

```
Project: C:\sqldeveloper113\sqldeveloper\sqldevelop
FUNCTION ORA61.GET_JOB@ora61
Error(6,3): PL/SQL: SQL Statement ignored
Error(6,10): PL/SQL: ORA-00904: "JOB_TITL
```

Function: Syntax

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

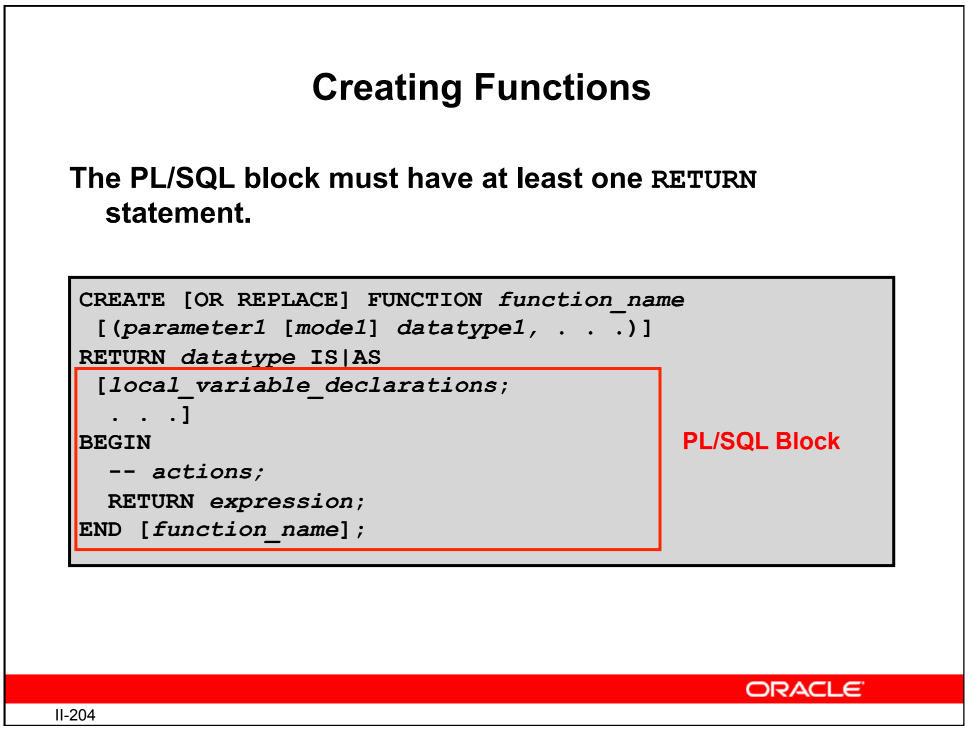


Function: Example

```
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;
```

ORACLE

II-203



Creating Functions

The PL/SQL block must have at least one `RETURN` statement.

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, . . .)]
RETURN datatype IS|AS
 [local_variable_declarations;
  . . .]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

ORACLE

II-204

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;
/
```

Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables

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

```
anonymous block completed
b_salary
-----
24000
```

```
-- As a PL/SQL expression, get the results using a local
-- variable

DECLARE
  sal employees.salary%type;
BEGIN
  sal := get_sal(100);
  DBMS_OUTPUT.PUT_LINE('The salary is: '|| sal);
END;/
```

```
anonymous block completed
The salary is: 24000
```

ORACLE

Using Different Methods for Executing Functions

```
-- Use as a parameter to another subprogram

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

```
anonymous block completed
24000
```

```
-- Use in a SQL statement (subject to restrictions)

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

```
JOB_ID     GET_SAL(EMPLOYEE_ID)
---------- ----------------------
SH_CLERK   2600
SH_CLERK   2600
AD_ASST    4400
MK_MAN     13000
```

. . .

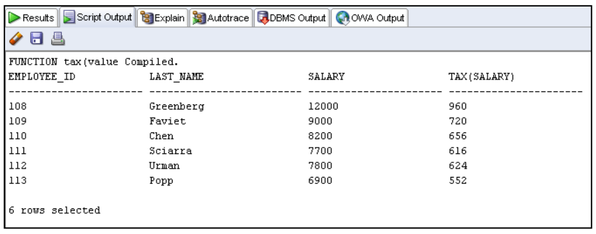
```
SH_CLERK   3100
SH_CLERK   3000

107 rows selected
```

ORACLE

Using a Function in a SQL Expression: Example

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



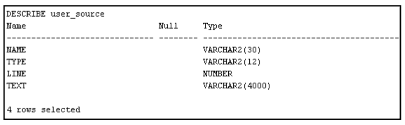
```
FUNCTION tax(value Compiled.
EMPLOYEE_ID            LAST_NAME                 SALARY                 TAX(SALARY)
---------------------- ------------------------- ---------------------- ----------------------
108                    Greenberg                 12000                  960
109         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

6 rows selected
```

ORACLE

Viewing Functions Using Data Dictionary Views

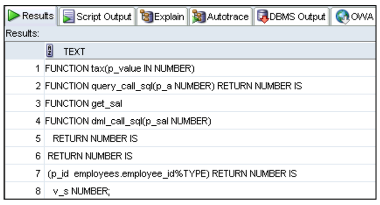
```
DESCRIBE USER_SOURCE
```



```
DESCRIBE user_source
Name                           Null     Type
------------------------------ -------- ----------------------------------------
NAME                                    VARCHAR2(30)
TYPE                                    VARCHAR2(12)
LINE                                    NUMBER
TEXT                                    VARCHAR2(4000)

4 rows selected
```

```
SELECT  text
FROM    user_source
WHERE   type = 'FUNCTION'
ORDER   BY line;
```



```
Results:
     TEXT
  1 FUNCTION tax(p_value IN NUMBER)
  2 FUNCTION query_call_sql(p_a NUMBER) RETURN NUMBER IS
  3 FUNCTION get_sal
  4 FUNCTION dml_call_sql(p_sal NUMBER)
  5   RETURN NUMBER IS
  6  RETURN NUMBER IS
  7  (p_id  employees.employee_id%TYPE) RETURN NUMBER IS
  8   v_s NUMBER;
```

. . .

ORACLE

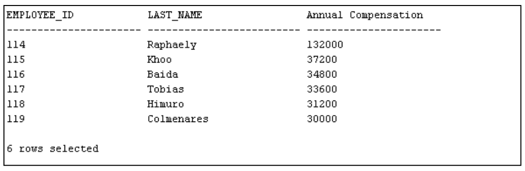
Exercice

2. Create a function called `GET_ANNUAL_COMP` to return the annual salary computed from an employee's monthly salary and commission passed as parameters.

   a. Create the `GET_ANNUAL_COMP` function, which accepts parameter values for the monthly salary and commission. Either or both values passed can be `NULL`, but the function should still return a non-`NULL` annual salary. Use the following basic formula to calculate the annual salary:

   ```
   (salary*12) + (commission_pct*salary*12)
   ```

   b. Use the function in a `SELECT` statement against the `EMPLOYEES` table for employees in department 30.



```
EMPLOYEE_ID            LAST_NAME                 Annual Compensation
---------------------- ------------------------- ----------------------
114                    Raphaely                  132000
115                    Khoo                      37200
116                    Baida                     34800
117                    Tobias                    33600
118                    Himuro                    31200
119                    Colmenares                30000

6 rows selected
```

ORACLE

II-212

WORKING WITH PACKAGES

ORACLE

II-215

Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department
    (p_deptno departments.department_id%TYPE,
     p_name departments.department_name%TYPE :='unknown',
     p_loc  departments.location_id%TYPE := 1700);

PROCEDURE add_department
   (p_name departments.department_name%TYPE := 'unknown',
    p_loc  departments.location_id%TYPE := 1700);
END dept_pkg;
/
```

ORACLE

Overloading Procedures Example: Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
PROCEDURE add_department -- First procedure's declaration
  (p_deptno departments.department_id%TYPE,
   p_name   departments.department_name%TYPE := 'unknown',
   p_loc    departments.location_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES  (p_deptno, p_name, p_loc);
  END add_department;

PROCEDURE add_department -- Second procedure's declaration
  (p_name   departments.department_name%TYPE := 'unknown',
   p_loc    departments.location_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
  END add_department;
 END dept_pkg; /
```

ORACLE

Examples of Some Oracle-Supplied Packages

Here is an abbreviated list of some Oracle-supplied packages:

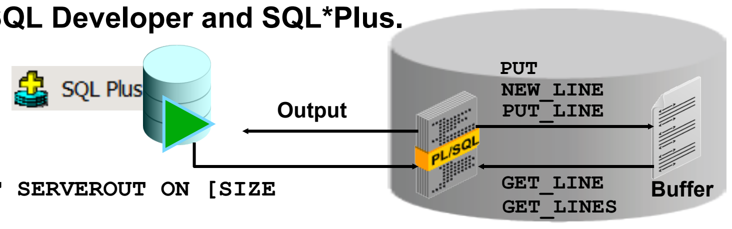
- `DBMS_OUTPUT`
- `UTL_FILE`
- `UTL_MAIL`
- `DBMS_ALERT`
- `DBMS_LOCK`
- `DBMS_SESSION`
- `HTP`
- `DBMS_SCHEDULER`

ORACLE

How the `DBMS_OUTPUT` Package Works

The `DBMS_OUTPUT` package enables you to send messages from stored subprograms and triggers.

- `PUT` and `PUT_LINE` place text in the buffer.
- `GET_LINE` and `GET_LINES` read the buffer.
- Messages are not sent until the sending subprogram or trigger completes.
- Use `SET SERVEROUTPUT ON` to display messages in SQL Developer and SQL*Plus.



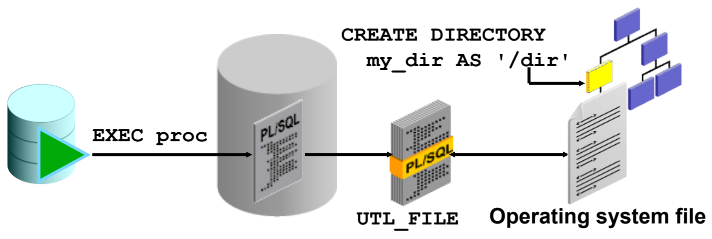
```
SET SERVEROUT ON [SIZE
n]
EXEC proc
```

ORACLE

Using the UTL_FILE Package to Interact with Operating System Files

The UTL_FILE package extends PL/SQL programs to read and write operating system text files:

- Provides a restricted version of operating system stream file I/O for text files
- Can access files in operating system directories defined by a CREATE DIRECTORY statement



Using UTL_FILE: Example

```
CREATE OR REPLACE PROCEDURE sal_status(
  p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
  f_file UTL_FILE.FILE_TYPE;
  CURSOR cur_emp IS
    SELECT last_name, salary, department_id
    FROM employees ORDER BY department_id;
  v_newdeptno employees.department_id%TYPE;
  v_olddeptno employees.department_id%TYPE := 0;
BEGIN
  f_file:= UTL_FILE.FOPEN (p_dir, p_filename, 'W');
  UTL_FILE.PUT_LINE(f_file,
   'REPORT: GENERATED ON ' || SYSDATE);
  UTL_FILE.NEW_LINE (f_file);
. . .
```

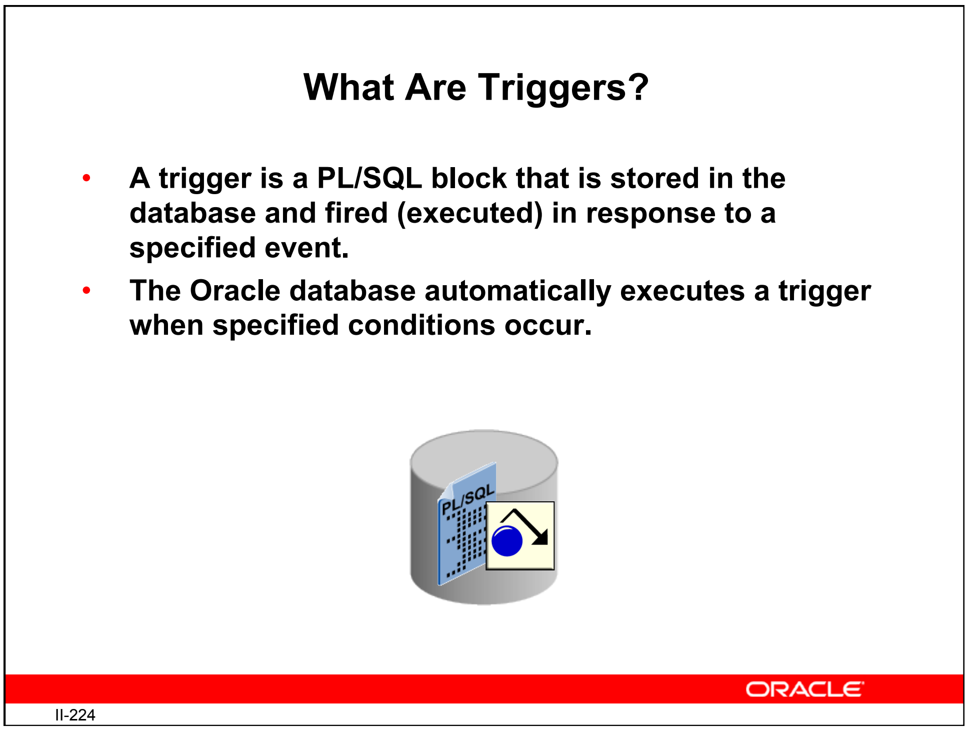
Using UTL_FILE: Example

```
. . .
FOR emp_rec IN cur_emp LOOP
    IF emp_rec.department_id <> v_olddeptno THEN
      UTL_FILE.PUT_LINE (f_file,
       'DEPARTMENT: ' || emp_rec.department_id);
     UTL_FILE.NEW_LINE (f_file);
    END IF;
    UTL_FILE.PUT_LINE (f_file,
       '  EMPLOYEE: ' || emp_rec.last_name ||
           ' earns: ' || emp_rec.salary);
    v_olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (f_file);
  END LOOP;
  UTL_FILE.PUT_LINE(f_file,'*** END OF REPORT ***');
  UTL_FILE.FCLOSE (f_file);
EXCEPTION
 WHEN UTL_FILE.INVALID_FILEHANDLE THEN
  RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
 WHEN UTL_FILE.WRITE_ERROR THEN
  RAISE_APPLICATION_ERROR (-20002, 'Unable to write to file');
END sal_status;/
```

ORACLE

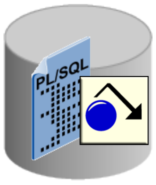
CREATING TRIGGERS

ORACLE



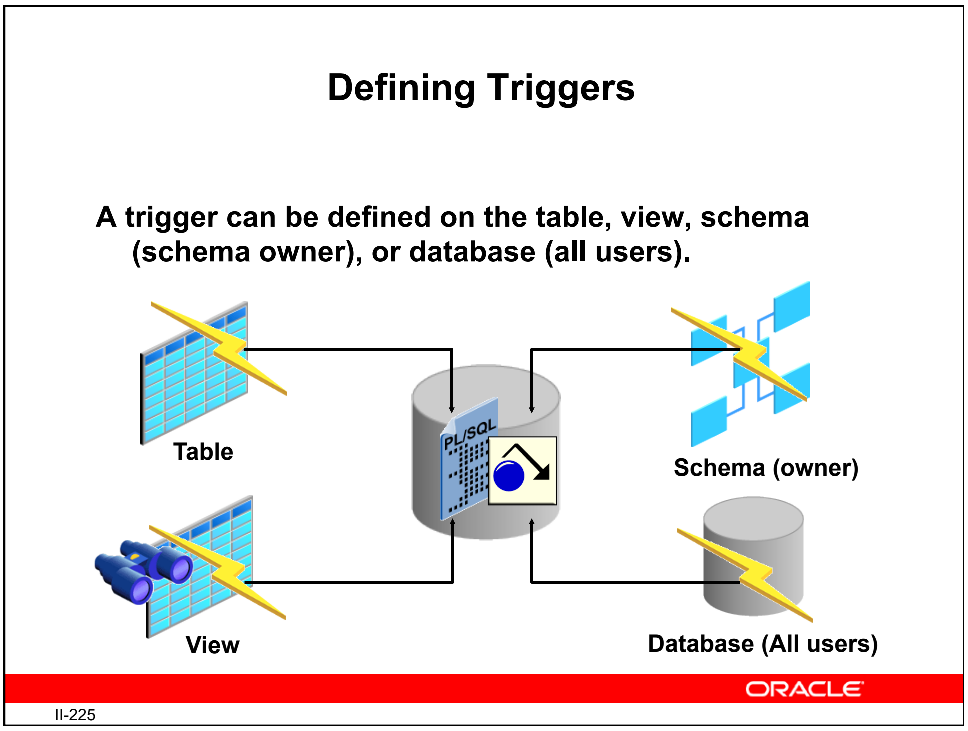
What Are Triggers?

- A trigger is a PL/SQL block that is stored in the database and fired (executed) in response to a specified event.
- The Oracle database automatically executes a trigger when specified conditions occur.



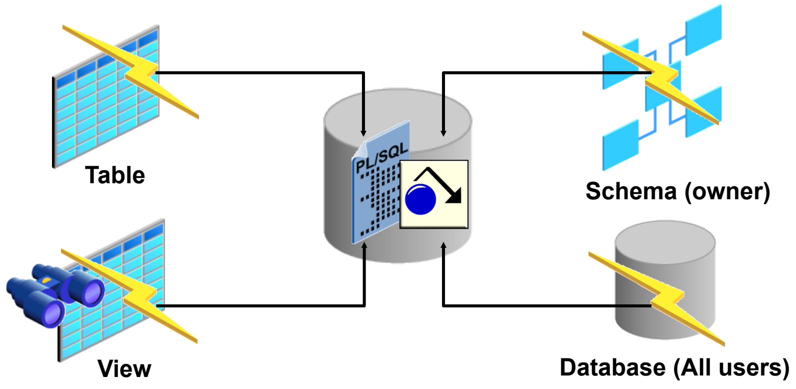
ORACLE

II-224



Defining Triggers

A trigger can be defined on the table, view, schema (schema owner), or database (all users).



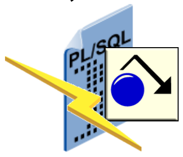
ORACLE

II-225

Trigger Event Types

Vous pouvez écrire des triggers qui se déclenchent lorsqu'une des opérations suivantes se produit dans la base de données :

- Une manipulation de la base de données (DML) (`DELETE`, `INSERT`, or `UPDATE`).
- Une requête de définition de base de données (DDL) (`CREATE`, `ALTER`, or `DROP`).
- Une opération de base de données tels que `SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`.

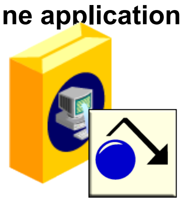


ORACLE

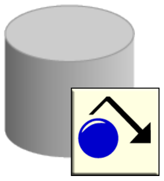
II-226

Application and Database Triggers

- Database trigger
  - Se déclenchent chaque fois qu'un événement DML, DLL ou système se produit sur une base de données ou un schéma
- Application trigger:
  - Se déclenchant si un événement se produit dans une application particulière



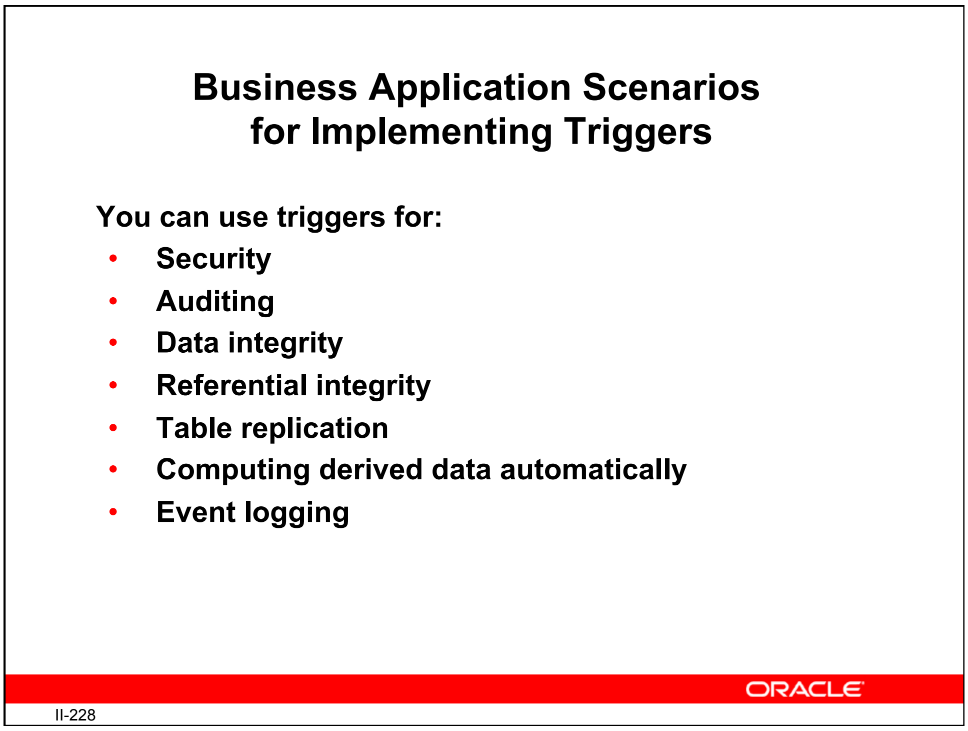
Application Trigger



Database Trigger

ORACLE

II-227



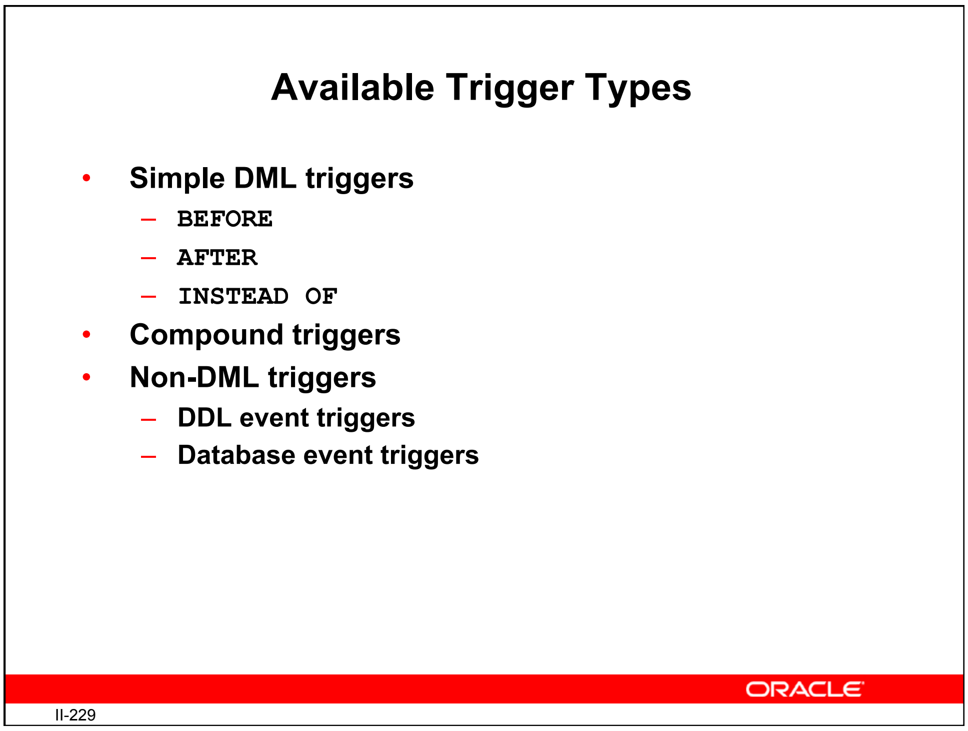
Business Application Scenarios for Implementing Triggers

You can use triggers for:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

ORACLE

II-228



Available Trigger Types

- Simple DML triggers
  - `BEFORE`
  - `AFTER`
  - `INSTEAD OF`
- Compound triggers
- Non-DML triggers
  - DDL event triggers
  - Database event triggers

ORACLE

II-229

Trigger Event Types and Body

- Le type de déclencheur détermine quelle instruction DML provoque l'exécution du trigger. Les événements possibles sont :
  - `INSERT`
  - `UPDATE [OF column]`
  - `DELETE`
- Le corps de déclencheur détermine quelle action est exécutée et est un bloc PL/SQL ou un appel d'une procédure

ORACLE

II-230

Creating DML Triggers Using the `CREATE TRIGGER` Statement

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing -- when to fire the trigger
event1 [OR event2 OR event3]
ON object_name
[REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW -- default is statement level trigger
WHEN (condition)]]
DECLARE]
BEGIN
... trigger_body -- executable statements
[EXCEPTION . . .]
END [trigger_name];
```

```
timing =  BEFORE | AFTER | INSTEAD OF
```

```
event = INSERT | DELETE | UPDATE | UPDATE OF column_list
```

ORACLE

II-231

Specifying the Trigger Firing (Timing)

Vous pouvez spécifier le moment de déclenchement quant à l'éxécution de l'action avant ou après l'instruction de déclenchement :

- `BEFORE`: Exécute le corps du déclencheur avant l'événement de déclencheur DML sur une table.
- `AFTER`: Exécuter le corps de déclencheur après l'événement de déclencheur DML sur une table.
- `INSTEAD OF`: Exécuter le corps de déclencheur au lieu de l'instruction de déclenchement. Ceci est utilisé pour les vues qui ne sont pas modifiables.

ORACLE

II-232

Statement-Level Triggers Versus Row-Level Triggers

Statement-Level Triggers	Row-Level Triggers
Is the default when creating a trigger	Use the `FOR EACH ROW` clause when creating a trigger.
Fires once for the triggering event	Fires once for each row affected by the triggering event
Fires once even if no rows are affected	Does not fire if the triggering event does not affect any rows

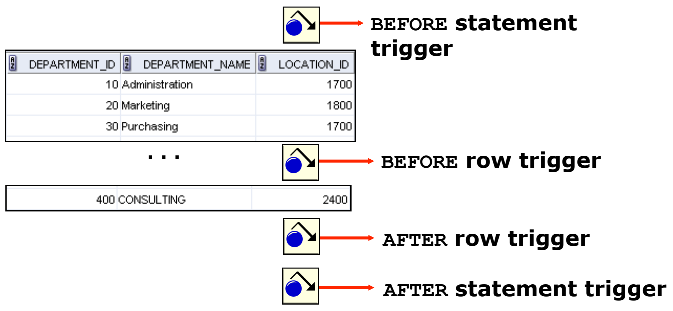
ORACLE

II-233

Trigger-Firing Sequence: Single-Row Manipulation

Use the following firing sequence for a trigger on a table when a single row is manipulated:

```
INSERT INTO departments
    (department_id,department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```

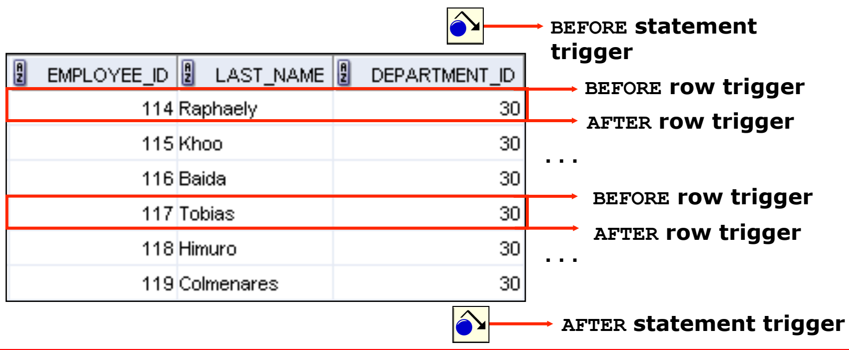


DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
400	CONSULTING	2400

Trigger-Firing Sequence: Multirow Manipulation

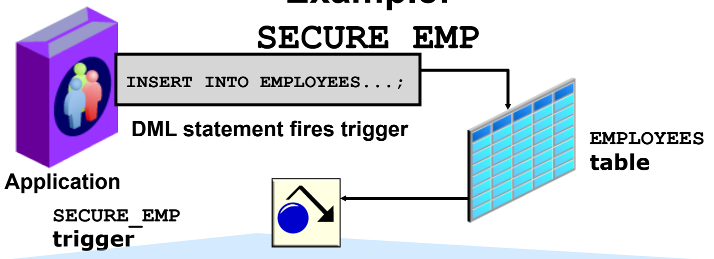
Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30;
```



EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

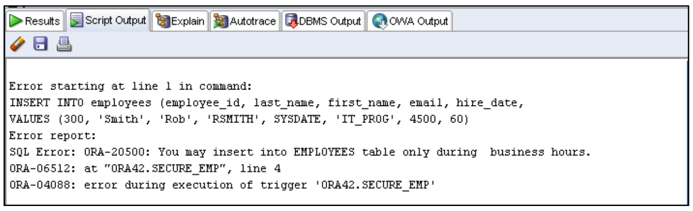
Creating a DML Statement Trigger Example: SECURE_EMP



```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
  BEGIN
   IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
      (TO_CHAR(SYSDATE,'HH24:MI')
                       NOT BETWEEN '08:00' AND
'18:00') THEN
   RAISE_APPLICATION_ERROR(-20500, 'You may insert'
     ||' into EMPLOYEES table only during '
     ||' normal business hours.');
   END IF;
  END;
```

Testing Trigger SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
             first_name, email, hire_date,
                      job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
            'IT_PROG', 4500, 60);
```



```
Error starting at line 1 in command:
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date,
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60)
Error report:
SQL Error: ORA-20500: You may insert into EMPLOYEES table only during  business hours.
ORA-06512: at "ORA42.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'ORA42.SECURE_EMP'
```

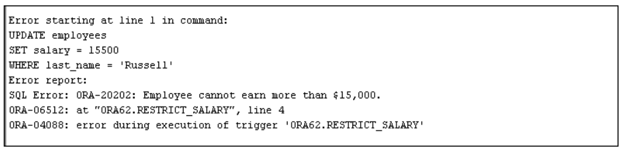
Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees
  BEGIN
    IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
       (TO_CHAR(SYSDATE,'HH24')
        NOT BETWEEN '08' AND '18') THEN
      IF DELETING THEN RAISE_APPLICATION_ERROR(
        -20502,'You may delete from EMPLOYEES table'||
        'only during normal business hours.');
      ELSIF INSERTING  THEN RAISE_APPLICATION_ERROR(
        -20500,'You may insert into EMPLOYEES table'||
        'only during normal business hours.');
      ELSIF UPDATING  ('SALARY') THEN
        RAISE_APPLICATION_ERROR(-20503, 'You may '||
        'update SALARY only normal during business hours.');
      ELSE RAISE_APPLICATION_ERROR(-20504,'You may'||
        ' update EMPLOYEES table only during'||
        ' normal business hours.');
      END IF;
    END IF;
  END;
```

Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
     AND :NEW.salary > 15000 THEN
    RAISE_APPLICATION_ERROR (-20202,
      'Employee cannot earn more than $15,000.');
  END IF;
END;/
```

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```



```
Error starting at line 1 in command:
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell'
Error report:
SQL Error: ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "ORA62.RESTRICT_SALARY", line 4
ORA-04088: error during execution of trigger 'ORA62.RESTRICT_SALARY'
```

Using OLD and NEW Qualifiers

- When a row-level trigger fires, the PL/SQL run-time engine creates and populates two data structures:
  - `OLD`: Stores the original values of the record processed by the trigger
  - `NEW`: Contains the new values
- `NEW` and `OLD` have the same structure as a record declared using the `%ROWTYPE` on the table to which the trigger is attached.

Data Operations	Old Value	New Value
`INSERT`	`NULL`	Inserted value
`UPDATE`	Value before update	Value after update
`DELETE`	Value before delete	`NULL`

ORACLE

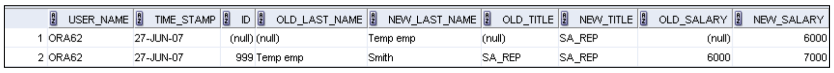
Using OLD and NEW Qualifiers: Example

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO audit_emp(user_name, time_stamp, id,
    old_last_name, new_last_name, old_title,
    new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
    :OLD.last_name, :NEW.last_name, :OLD.job_id,
    :NEW.job_id, :OLD.salary, :NEW.salary);
END;
/
```

ORACLE

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP

```
INSERT INTO employees (employee_id, last_name, job_id,
salary, email, hire_date)
VALUES (999, 'Temp emp', 'SA_REP', 6000, 'TEMPEMP',
TRUNC(SYSDATE));
/
UPDATE employees
 SET salary = 7000, last_name = 'Smith'
 WHERE employee_id = 999;
/
SELECT *
FROM  audit_emp;
```



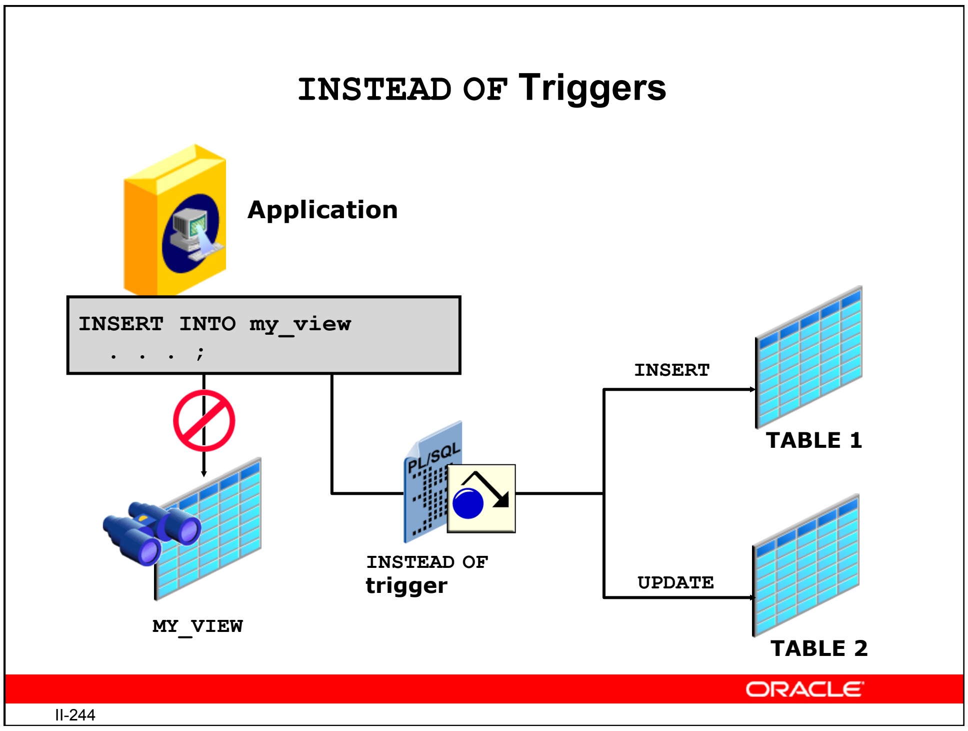
	USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
1	ORA62	27-JUN-07	(null)	(null)	Temp emp	(null)	SA_REP	(null)	6000
2	ORA62	27-JUN-07	999	Temp emp	Smith	SA_REP	SA_REP	6000	7000

II-242

Using the WHEN Clause to Fire a Row Trigger Based on a Condition

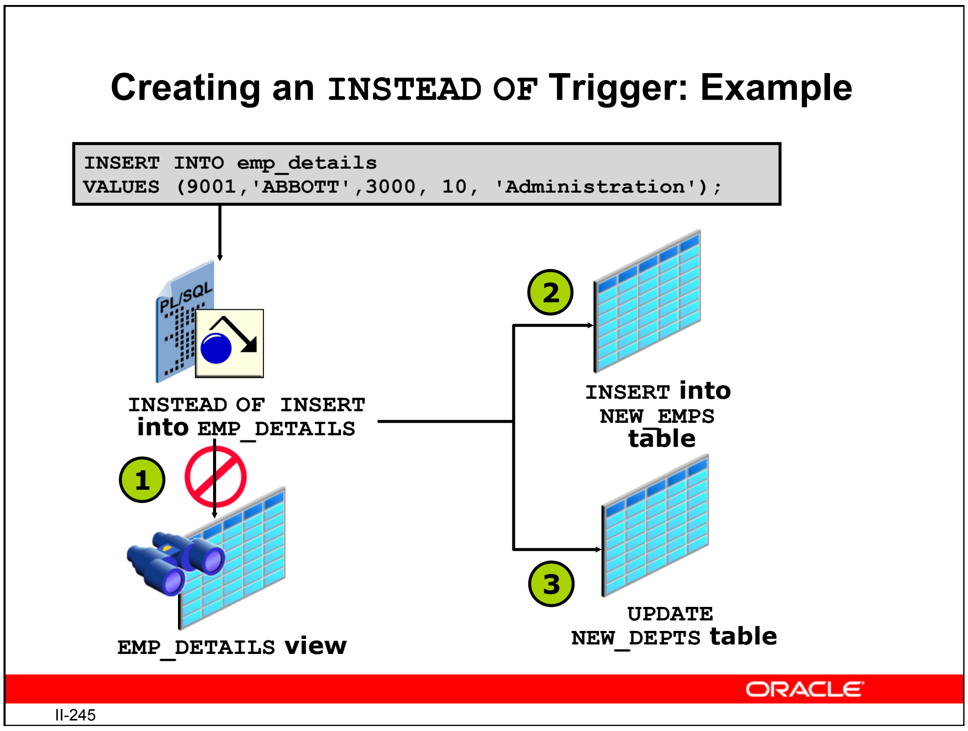
```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
 IF INSERTING THEN
   :NEW.commission_pct := 0;
 ELSIF :OLD.commission_pct IS NULL THEN
   :NEW.commission_pct := 0;
 ELSE
   :NEW.commission_pct := :OLD.commission_pct+0.05;
 END IF;
END;
/
```

II-243



INSTEAD OF Triggers

```
INSERT INTO my_view
  . . . ;
```



Creating an INSTEAD OF Trigger: Example

```
INSERT INTO emp_details
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```

INSTEAD OF INSERT into EMP_DETAILS
1
EMP_DETAILS view
2
INSERT into NEW_EMPS table
3
UPDATE NEW_DEPTS table
ORACLE

Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```
CREATE TABLE new_emps AS
 SELECT employee_id,last_name,salary,department_id
     FROM employees;

CREATE TABLE new_depts AS
 SELECT d.department_id,d.department_name,
        sum(e.salary) dept_sal
     FROM employees e, departments d
 WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
 SELECT e.employee_id, e.last_name, e.salary,
        e.department_id, d.department_name
 FROM employees e, departments d
 WHERE e.department_id = d.department_id
GROUP BY d.department_id,d.department_name;
```

ORACLE

Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO new_emps
    VALUES (:NEW.employee_id, :NEW.last_name,
               :NEW.salary, :NEW.department_id);
    UPDATE new_depts
      SET dept_sal = dept_sal + :NEW.salary
      WHERE department_id = :NEW.department_id;
  ELSIF DELETING THE
      DELETE FROM new_emps
             WHERE employee_id = :OLD.employee_id;
                    UPDATE new_depts
             SET dept_sal = dept_sal - :OLD.salary
           WHERE department_id = :OLD.department_id;
```

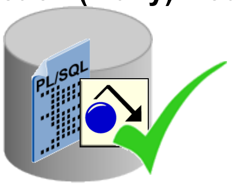
ORACLE

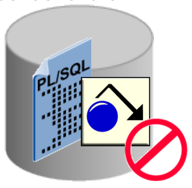
```
ELSIF UPDATING ('salary') THEN
    UPDATE new_emps
      SET salary = :NEW.salary
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal +
                     (:NEW.salary- :OLD.salary)
      WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('department_id') THEN
    UPDATE new_emps
      SET department_id = :NEW.department_id
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal - :OLD.salary
      WHERE department_id = :OLD.department_id;
    UPDATE new_depts
      SET dept_sal = dept_sal + :NEW.salary
      WHERE department_id = :NEW.department_id;
  END IF;
END;
/
```

The Status of a Trigger

Un déclencheur est défini dans un des deux modes distincts :

- **Enabled: The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).**
- **Disabled: The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.**





Managing Triggers Using the ALTER and DROP SQL Statements

```
-- Disable or reenable a database trigger:

ALTER TRIGGER trigger_name DISABLE | ENABLE;
```

```
-- Disable or reenable all triggers for a table:

ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS;
```

```
-- Recompile a trigger for a table:

ALTER TRIGGER trigger_name COMPILE;
```

```
-- Remove a trigger from the database:

DROP TRIGGER trigger_name;
```

ORACLE

II-251

Viewing Trigger Information

You can view the following trigger information:

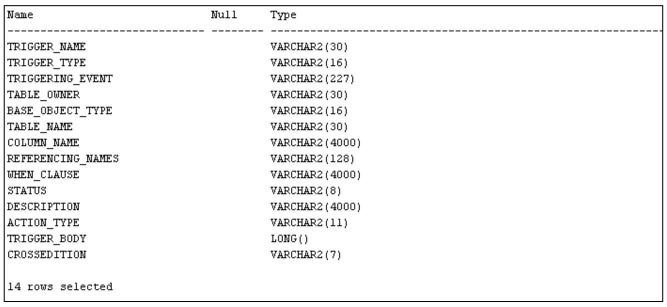
Data Dictionary View	Description
`USER_OBJECTS`	Displays object information
`USER/ALL/DBA_TRIGGERS`	Displays trigger information
`USER_ERRORS`	Displays PL/SQL syntax errors for a trigger

ORACLE

II-252

Using `USER_TRIGGERS`

```
DESCRIBE user_triggers
```



```
Name                           Null     Type
------------------------------ -------- ------------------------------------------------------------------
TRIGGER_NAME                            VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGERING_EVENT                        VARCHAR2(227)
TABLE_OWNER                             VARCHAR2(30)
BASE_OBJECT_TYPE                        VARCHAR2(16)
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
REFERENCING_NAMES                       VARCHAR2(128)
WHEN_CLAUSE                             VARCHAR2(4000)
STATUS                                  VARCHAR2(8)
DESCRIPTION                             VARCHAR2(4000)
ACTION_TYPE                             VARCHAR2(11)
TRIGGER_BODY                            LONG()
CROSSEDITION                            VARCHAR2(7)

14 rows selected
```

```
SELECT trigger_type, trigger_body
FROM user_triggers
WHERE trigger_name = 'SECURE_EMP';
```

ORACLE

II-253

Exercices

1. Create a trigger called `CHECK_SALARY_TRG` on the `EMPLOYEES` table that fires before an `INSERT` or `UPDATE` operation on each row:
   i. The trigger must call the CHECK_SALARY procedure to carry out the business logic.
   ii. The trigger should pass the new job ID and salary to the procedure parameters.
2. Update the `CHECK_SALARY_TRG` trigger to fire only when the job ID or salary values have actually changed.
   a. Implement the business rule using a `WHEN` clause to check whether the `JOB_ID` or `SALARY` values have changed.

   Note: Make sure that the condition handles the `NULL` in the `OLD.column_name` values if an `INSERT` operation is performed; otherwise, an an `INSERT` operation will fail.

ORACLE

II-257

Exercices

3. You are asked to prevent employees from being deleted during business hours.

   Write a statement trigger called `DELETE_EMP_TRG` on the `EMPLOYEES` table to prevent rows from being deleted during weekday business hours, which are from 9:00 AM through 6:00 PM.

ORACLE