Stored procedures and functions

Oracle Academy Study Materials

PL/SQL Subprograms

- A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters.
- You can declare and define a subprogram within either a PL/SQL block or another subprogram.
- A subprogram consists of a specification and a body.
- A subprogram can be a procedure or a function.
- Typically, you use a procedure to perform an action and a function to compute and return a value.

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	Subprograms called functions must return values
Cannot take parameters	Can take parameters

Stored Procedures and Functions

What Are Procedures?

- Are a type of subprogram that perform an action
- Can be stored in the database as a schema object
- Promote reusability and maintainability

Creating Procedures with the SQL `CREATE OR REPLACE` Statement

- Use the `CREATE` clause to create a stand-alone procedure that is stored in the Oracle database.
- Use the `OR REPLACE` option to overwrite an existing procedure.

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; ...]
BEGIN
      -- actions;
END [procedure_name];
```

PL/SQL block

Anonymous blocks vs. procedures

Anonymous blocks

```
DECLARE    (Optional)
   Variables, cursors, etc.;
BEGIN      (Mandatory)
   SQL and PL/SQL statements;
EXCEPTION (Optional)
   WHEN exception-handling actions;
END; (Mandatory)
```

Subprograms (procedures)

```
CREATE [OR REPLACE] PROCEDURE name [parameters] IS|AS (Mandatory)
  Variables, cursors, etc.; (Optional)
BEGIN      (Mandatory)
   SQL and PL/SQL statements;
EXCEPTION (Optional)
   WHEN exception-handling actions;
END [name]; (Mandatory)
```

Procedure: Example

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

Parameters and Parameter Modes

- Are declared after the subprogram name in the PL/SQL header
- Pass or communicate data between the caller and the subprogram
- Are used like local variables but are dependent on their parameter-passing mode:
  - An `IN` parameter mode (the default) provides values for a subprogram to process
  - An `OUT` parameter mode returns a value to the caller
  - An `IN OUT` parameter mode supplies an input value, which may be returned (output) as a modified value

Formal and Actual Parameters

- Formal parameters: Local variables declared in the parameter list of a subprogram specification
- Actual parameters (or arguments): Literal values, variables, and expressions used in the parameter list of the calling subprogram

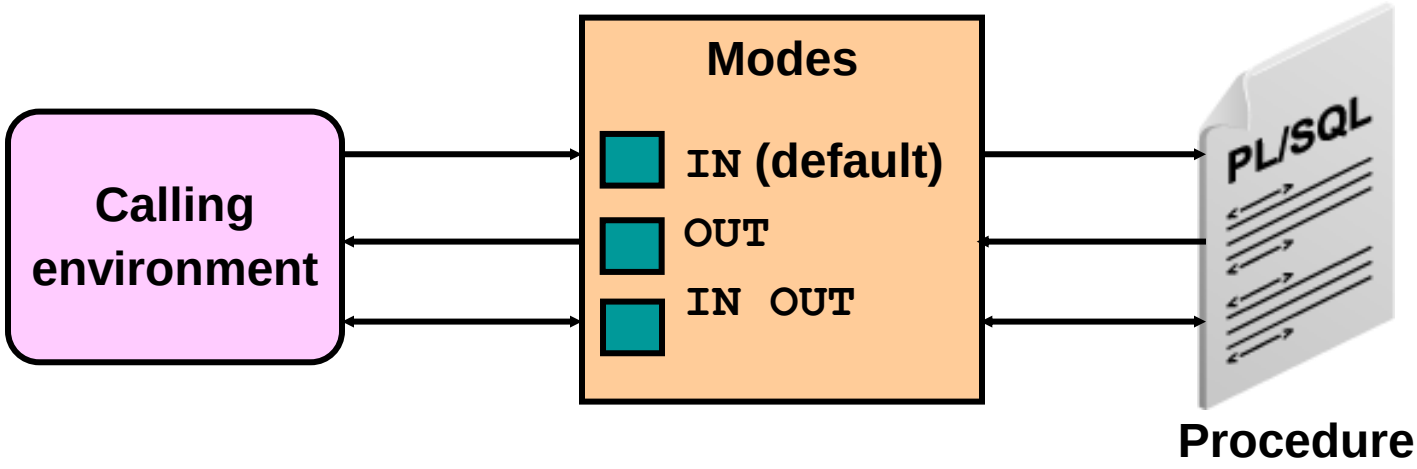
```
-- Procedure definition, Formal_parameters
CREATE PROCEDURE raise_sal(p_id NUMBER, p_sal NUMBER) IS
BEGIN
. . .
END raise_sal;
```

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

Procedural Parameter Modes

- Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.
- The `IN` mode is the default if no mode is specified.

```
CREATE PROCEDURE procedure(param [mode] datatype)
...
```



Comparing the Parameter Modes

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

Using IN Parameters: Example

```
CREATE OR REPLACE PROCEDURE raise_salary
  (id      IN employees.employee_id%TYPE,
   percent IN NUMBER)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * (1 + percent/100)
  WHERE  employee_id = id;
END raise_salary;
/
```

```
EXECUTE raise_salary(176,10)
```

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;
```

Using IN OUT Parameters: Example

Calling environment

phone_no (before the call)	phone_no (after the call)
'8006330575'	'(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;
/
```

Available Notations for Passing Actual Parameters

- When calling a subprogram, you can write the actual parameters using the following notations:
- 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
- Mixed:
  - Lists some of the actual parameters as positional and some as named

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 notation:

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

Passing by named notation:

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

Calling Procedures

You can call procedures using anonymous blocks, another procedure, or packages.

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

Inserted 1 row
PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME
280	ST-Curriculum

Calling Procedures

You can call procedures using anonymous blocks, another procedure, or packages.

```
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.
```

Note: You cannot invoke a procedure from inside a SQL statement such as SELECT.

Handled Exceptions

Calling procedure

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

Called procedure

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

Exception raised

Exception handled

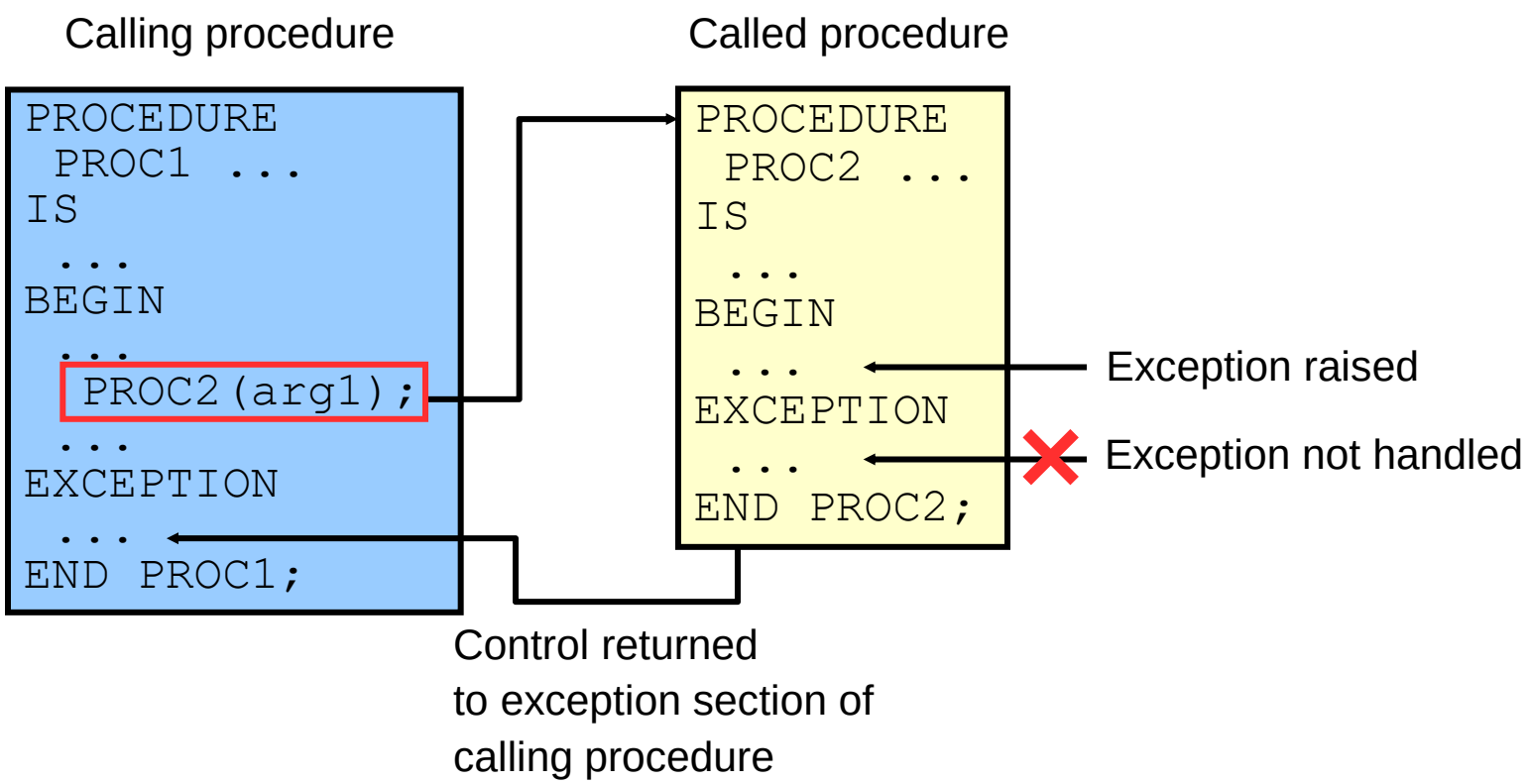
Control returns to calling procedure

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;
```

Exceptions Not Handled



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;
```

Function: Syntax

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

Function: Example

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

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

Invoking the Function

```
-- Invoke the function as an expression or as
-- a parameter value.

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

```
-- 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;/
```

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

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

Invoking the Function

```
-- Invoke the function as an expression or as
-- a parameter value.

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

```
-- 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;/
```

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

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

Stored Procedures and Functions

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

```
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;
```

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

```
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
```

Stored Procedures and 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

```
SELECT employee_id, tax(salary)
FROM   employees
WHERE  tax(salary) > (SELECT MAX(tax(salary))
                      FROM employees
                      WHERE department_id = 30)
ORDER BY tax(salary) DESC;
```

Stored Procedures and Functions

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_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', p_sal);
  RETURN (p_sal + 100);
END;
```

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

```
FUNCTION dml_call_sql(p_sal Compiled.

Error starting at line 1 in command:
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170
Error report:
SQL Error: ORA-04091: table ORA62.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA62.DML_CALL_SQL", line 4
04091. 00000 -  "table %s.%s is mutating, trigger/function may not see it"
```

Triggers

Oracle Academy Study Materials

Creating Triggers – Trigger Definition

A trigger:

- 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 (implicitly) executes a trigger when specified conditions occur
- Is associated with a table, view, schema, or database

Creating Triggers – Trigger Event Types

You can write triggers that fire whenever one of the following operations occurs in the database:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation such as SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.

You can use triggers for:

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

Creating Triggers – Available Trigger Types

- Simple DML triggers
  - `BEFORE`
  - `AFTER`
  - `INSTEAD OF`
- Compound triggers
- Non-DML triggers
  - DDL event triggers
  - Database event triggers
- A trigger event type determines which DML statement causes the trigger to execute. The possible events are:
  - `INSERT`
  - `UPDATE [OF column]`
  - `DELETE`
- A trigger body determines what action is performed and is a PL/SQL block or a `CALL` to a procedure

Creating DML Triggers

```
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
```

Creating Triggers - Trigger Timing

- `BEFORE`: Execute the trigger body before the triggering DML event on a table and is frequently used in the following situations:
  - To determine whether the triggering statement should be allowed to complete (This eliminates unnecessary processing and enables a rollback in cases where an exception is raised in the triggering action.)
  - To derive column values before completing an INSERT or UPDATE statement
  - To initialize global variables or flags, and to validate complex business rules
- `AFTER`: Execute the trigger body after the triggering DML event on a table and is frequently used in the following situations:
  - To complete the triggering statement before executing the triggering action
  - To perform different actions on the same triggering statement if a BEFORE trigger is already present
- `INSTEAD OF`: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

Creating Triggers - Trigger Timing

- If multiple triggers are defined for the same object, then the order of firing triggers is arbitrary.
  - If the order in which they fire is important, then you can control the firing order using the FOLLOWS clause.
  - If it is practical, you should consider replacing the set of individual triggers for a particular timing point with a single compound trigger that explicitly codes the actions in the order you intend.

```
CREATE OR REPLACE TRIGGER
trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
FOLLOWS trigger_follows_test_trg_2
BEGIN

DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_1
- Executed');
END;
```

Creating Triggers – Types of DML Triggers

- The trigger type determines whether the body executes for each row or only once for the triggering statement.
- A statement trigger:
  - Executes once for the triggering event
  - Is the default type of trigger
  - Fires once even if no rows are affected at all
- A row trigger:
  - Executes once for each row affected by the triggering event
  - Is not executed if the triggering event does not affect any rows
  - Is indicated by specifying the `FOR EACH ROW` clause

Trigger-Firing Sequence

```
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

BEFORE statement trigger

BEFORE row trigger

AFTER row trigger

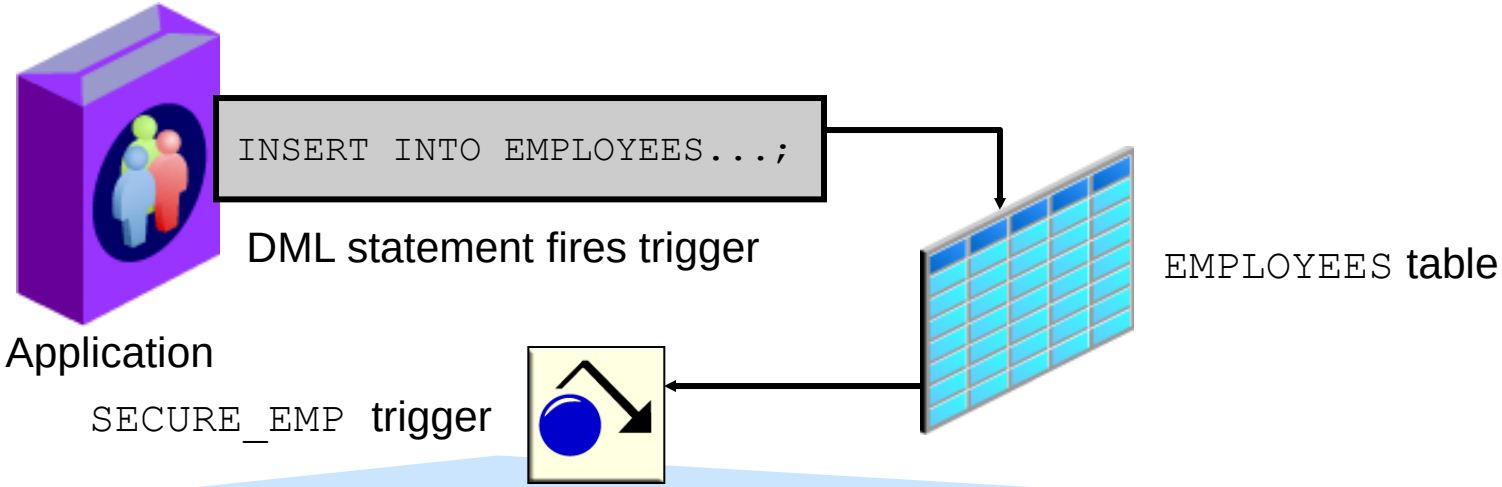
BEFORE row trigger

AFTER row trigger

AFTER statement trigger

Creating Triggers

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);
```

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

```
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'
```