1. Specifying Subprogram Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes are `IN` (the default), `OUT`, and `IN` `OUT`.

Any parameter mode can be used with any subprogram. Avoid using the `OUT` and `IN` `OUT` modes with functions.

Using the IN Mode

An `IN` parameter lets you pass values to the subprogram being called. Inside the subprogram, an `IN` parameter acts like a constant. It cannot be assigned a value.

You can pass a constant, literal, initialized variable, or expression as an IN parameter.

`IN` parameters can be initialized to default values, which are used if those parameters are omitted from the subprogram call.

Using the OUT Mode

An `OUT` parameter returns a value to the caller of a subprogram. Inside the subprogram, an `OUT` parameter acts like a variable. You can change its value, and reference the value after assigning it:

```
DECLARE
  emp_num       NUMBER(6) := 120;
  bonus         NUMBER(6) := 50;
  emp_last_name VARCHAR2(25);
  PROCEDURE raise_salary (emp_id IN NUMBER, amount IN NUMBER,
                         emp_name OUT VARCHAR2) IS
    BEGIN
      UPDATE employees SET salary = salary + amount WHERE employee_id = emp_id;
      SELECT last_name INTO emp_name FROM employees WHERE employee_id = emp_id;
  END raise_salary;
BEGIN
  raise_salary(emp_num, bonus, emp_last_name);
  DBMS_OUTPUT.PUT_LINE('Salary has been updated for: ' || emp_last_name);
END;
```

Using the IN OUT Mode

An `IN` `OUT` parameter passes initial values to a subprogram and returns updated values to the caller. It can be assigned a value and its value can be read. Typically, an `IN OUT` parameter is a string buffer or numeric accumulator, that is read inside the subprogram and then updated.

The actual parameter that corresponds to an `IN OUT` formal parameter must be a variable; it cannot be a constant or an expression.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. If you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Passes values to a subprogram	Returns values to the caller	Passes initial values to a subprogram and returns updated values to the caller
Formal parameter acts like a constant	Formal parameter acts like an uninitialized variable	Formal parameter acts like an initialized variable
Formal parameter cannot be assigned a value	Formal parameter must be assigned a value	Formal parameter should be assigned a value
Actual parameter can be a constant, initialized variable, literal, or expression	Actual parameter must be a variable	Actual parameter must be a variable
Actual parameter is passed by reference (a pointer to the value is passed in)	Actual parameter is passed by value (a copy of the value is passed out) unless `NOCOPY` is specified	Actual parameter is passed by value (a copy of the value is passed in and out) unless `NOCOPY` is specified

2. Using Default Values for Subprogram Parameters

```
DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6);
  merit   NUMBER(4);
  PROCEDURE raise_salary (emp_id IN NUMBER, amount IN NUMBER DEFAULT 100,
                          extra IN NUMBER DEFAULT 50) IS
    BEGIN
      UPDATE employees SET salary = salary + amount + extra
        WHERE employee_id = emp_id;
  END raise_salary;
BEGIN
  raise_salary(120); -- same as raise_salary(120, 100, 50)
  raise_salary(emp_num, extra => 25); -- same as raise_salary(120, 100, 25)
```

```
END;
```

---

3. Package

Package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

A package always has a specification, which declares the public items that can be referenced from outside the package.

If the public items include cursors or subprograms, then the package must also have a **body**. The body must define queries for public cursors and code for public subprograms. The body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package.

```
CREATE PACKAGE emp_actions AS

   /* Declare externally visible types, cursor, exception. */

   TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);

   invalid_salary EXCEPTION;

    /* Declare externally callable subprograms. */

   PROCEDURE raise_salary (emp_id INT, grade INT);

END emp_actions;
```

```
CREATE PACKAGE BODY emp_actions AS

   number_hired INT;  -- visible only in this package

   PROCEDURE raise_salary (emp_id INT, grade INT)  IS

      ………

   END raise_salary;

BEGIN  -- initialization part starts here

   ……….

END emp_actions;
```

4. Cursor FOR LOOP

The cursor `FOR` `LOOP` statement implicitly declares its loop index as a record variable of the row type that a specified cursor returns, and then opens a cursor.

- SQL Cursor FOR LOOP
- Explicit Cursor FOR LOOP

a. SQL Cursor FOR LOOP

```
BEGIN
FOR item IN
        (SELECT last_name, job_id FROM employees WHERE job_id LIKE '%CLERK%')
LOOP
        DBMS_OUTPUT.PUT_LINE
            ('Name = ' || item.last_name || ', Job = ' || item.job_id);
END LOOP;
END;
```

b. Explicit Cursor FOR LOOP

```
DECLARE
        CURSOR c1 IS    SELECT last_name, job_id FROM employees
                        WHERE job_id LIKE '%CLERK%' AND manager_id > 120;
BEGIN
        FOR item IN c1
        LOOP
            DBMS_OUTPUT.PUT_LINE
            ('Name = ' || item.last_name || ', Job = ' || item.job_id);
        END LOOP;
END;
```

PL/SQL cursor with parameters

```
DECLARE
```

```
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS

    SELECT * FROM employees WHERE job_id = job AND salary > max_wage;

BEGIN

  FOR person IN c1('CLERK', 3000)

  LOOP

     -- process data record

    DBMS_OUTPUT.PUT_LINE('Name = ' || person.last_name || ', salary = ' ||

                         person.salary || ', Job Id = ' || person.job_id );

  END LOOP;

END;
```

5. Introduction to REF CURSORs

Using `REF CURSOR`s is one of the most powerful, flexible, and scalable ways to return query results from an Oracle Database to a client application.

A `REF CURSOR` is a PL/SQL data type whose value is the memory address of a query work area on the database. In essence, a `REF CURSOR` is a pointer or a handle to a result set on the database. `REF CURSOR`s are represented through the `OracleRefCursor` ODP.NET class.

A `ref cursor` being a pointer to an `open cursor` used to send an open cursor as an out argument to the client app to loop through the record. If you want to loop through then,

```
declare
    ref_cur sys_refcursor;
    v_name all_tables.table_name%TYPE;
BEGIN
    OPEN ref_cur FOR SELECT table_name FROM all_tables WHERE ROWNUM < 5;
    LOOP
        FETCH  ref_cur INTO  v_name;
        exit when ref_cur%notfound;
        dbms_output.put_line(v_name);
    END LOOP;
    CLOSE ref_cur;
END;
```

You can not use `for` loop just as you do against an `implicit/explicit cursors`

```
declare
    ref_cur sys_refcursor;
BEGIN
    OPEN ref_cur FOR SELECT table_name FROM all_tables WHERE ROWNUM < 5;
    for i in ref_cur loop
        dbms_output.put_line(i.table_name);
    end loop;
END;
/
```

Return refcursor from a function:

```
create or replace function emp_list return sys_refcursor is
        rc   sys_refcursor;
  begin
        open rc for select * from emp;
        return rc;
  end;
--
create or replace procedure list_emps is
        e sys_refcursor;
        r emp%rowtype;
  begin
        e := emp_list;
        loop
            fetch e into r;
            exit when e%notfound;
            dbms_output.put_line(r.empno||','||r.hiredate);
        end loop;
        close e;
    end;
```

6. Database Triggers

A **trigger** is a PL/SQL unit that is stored in the database and (if it is in the enabled state) automatically executes ("fires") in response to a specified event.

A trigger has this structure:

```
TRIGGER trigger_name

  triggering_event
  [ trigger_restriction ]

BEGIN

  triggered_action;

END;
```

The `trigger_name` must be unique for triggers in the schema. A trigger can have the same name as another kind of object in the schema (for example, a table); however, Oracle recommends using a naming convention that avoids confusion.

If the trigger is in the **enabled** state, the `triggering_event` causes the database to execute the `triggered_action` if the `trigger_restriction` is either `TRUE` or omitted. The `triggering_event` is associated with either a table, a view, a schema, or the database, and it is one of these:

- DML statement (described in "About Data Manipulation Language (DML) Statements")
- DDL statement (described in "About Data Definition Language (DDL) Statements")
- Database operation (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`)

If the trigger is in the **disabled** state, the `triggering_event` does not cause the database to execute the `triggered_action`, even if the `trigger_restriction` is `TRUE` or omitted.

By default, a trigger is created in the enabled state. You can disable an enabled trigger, and enable a disabled trigger.

Unlike a subprogram, a trigger cannot be invoked directly. A trigger is invoked only by its triggering event, which can be caused by any user or application. You might be unaware that a trigger is executing unless it causes an error that is not handled properly.

A **simple trigger** can fire at exactly one of these **timing points**:

- Before the triggering event executes (statement-level `BEFORE` trigger)
- After the triggering event executes (statement-level `AFTER` trigger)
- Before each row that the event affects (row-level `BEFORE` trigger)
- After each row that the event affects (row-level `AFTER` trigger)

A **compound trigger** can fire at multiple timing points.

The Execution Model for Triggers and Integrity Constraint Checking

Oracle uses the following execution model to maintain the proper firing sequence of multiple triggers and constraint checking:

1. Run all `BEFORE` `statement` triggers that apply to the statement.
2. Loop for each row affected by the SQL statement.
    a. Run all `BEFORE` `row` triggers that apply to the statement.

    b. Lock and change row, and perform integrity constraint checking. (The lock is not released until the transaction is committed.)
    c. Run all `AFTER` *`row`* triggers that apply to the statement.
3. Complete deferred integrity constraint checking.
4. Run all `AFTER` *`statement`* triggers that apply to the statement.

Creating Triggers

To create triggers, use either the SQL Developer tool Create Trigger or the DDL statement `CREATE TRIGGER`. This section shows how to use both of these ways to create triggers.

About OLD and NEW Pseudorecords

When a row-level trigger fires, the PL/SQL runtime system creates and populates the two pseudorecords `OLD` and `NEW`.

For the row that the trigger is processing:

- For an `INSERT` trigger, `OLD` contains no values, and `NEW` contains the new values.
- For an `UPDATE` trigger, `OLD` contains the old values, and `NEW` contains the new values.
- For a `DELETE` trigger, `OLD` contains the old values, and `NEW` contains no values.

To reference a pseudorecord, put a colon before its name—:`OLD` or :`NEW`

Conditional Predicates

Conditional Predicate	**TRUE if and only if:**
`INSERTING`	An `INSERT` statement fired the trigger.
`UPDATING`	An `UPDATE` statement fired the trigger.
`UPDATING ('`*`column`*`')`	An `UPDATE` statement that affected the specified column fired the trigger.
`DELETING`	A `DELETE` statement fired the trigger.

This example creates a DML trigger that uses conditional predicates to determine which of its four possible triggering statements fired it.

```
CREATE OR REPLACE TRIGGER t

  BEFORE
```

```
    INSERT OR
    UPDATE OF salary, department_id OR
    DELETE
  ON employees
BEGIN
  CASE
    WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting');
    WHEN UPDATING('salary') THEN
      DBMS_OUTPUT.PUT_LINE('Updating salary');
    WHEN UPDATING('department_id') THEN
      DBMS_OUTPUT.PUT_LINE('Updating department ID');
    WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
  END CASE;
END;
/
```

Example: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted

```
CREATE OR REPLACE
TRIGGER NEW_EVALUATION_TRIGGER
BEFORE INSERT ON EVALUATIONS
FOR EACH ROW
```

```
BEGIN

  :NEW.evaluation_id := evaluations_sequence.NEXTVAL

END;
```

Dropping Triggers

```
DROP TRIGGER EVAL_CHANGE_TRIGGER;
```