### **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 |

### **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;

---

### **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

1. 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;

1. 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;

### **Introduction to REF CURSORs**

Using REF CURSORs 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 CURSORs 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;

### **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"](https://docs.oracle.com/database/121/TDDDG/tdddg_dml.htm#BCGBGBIE))
* DDL statement (described in ["About Data Definition Language (DDL) Statements"](https://docs.oracle.com/database/121/TDDDG/tdddg_objects.htm#CIHGAJDJ))
* 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.
   1. Run all BEFORE *row* triggers that apply to the statement.
   2. Lock and change row, and perform integrity constraint checking. (The lock is not released until the transaction is committed.)
   3. 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;