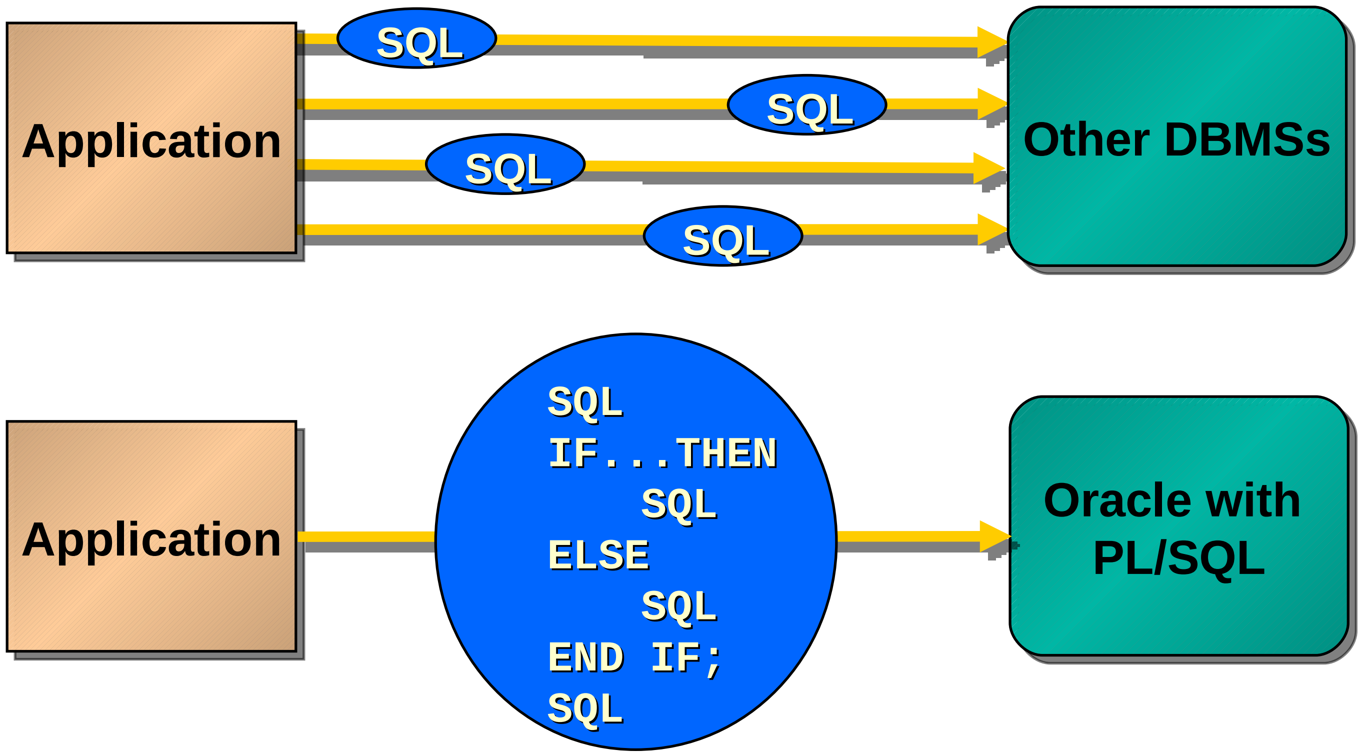
Procedural Language for SQL

PL/SQL

What is PL/SQL

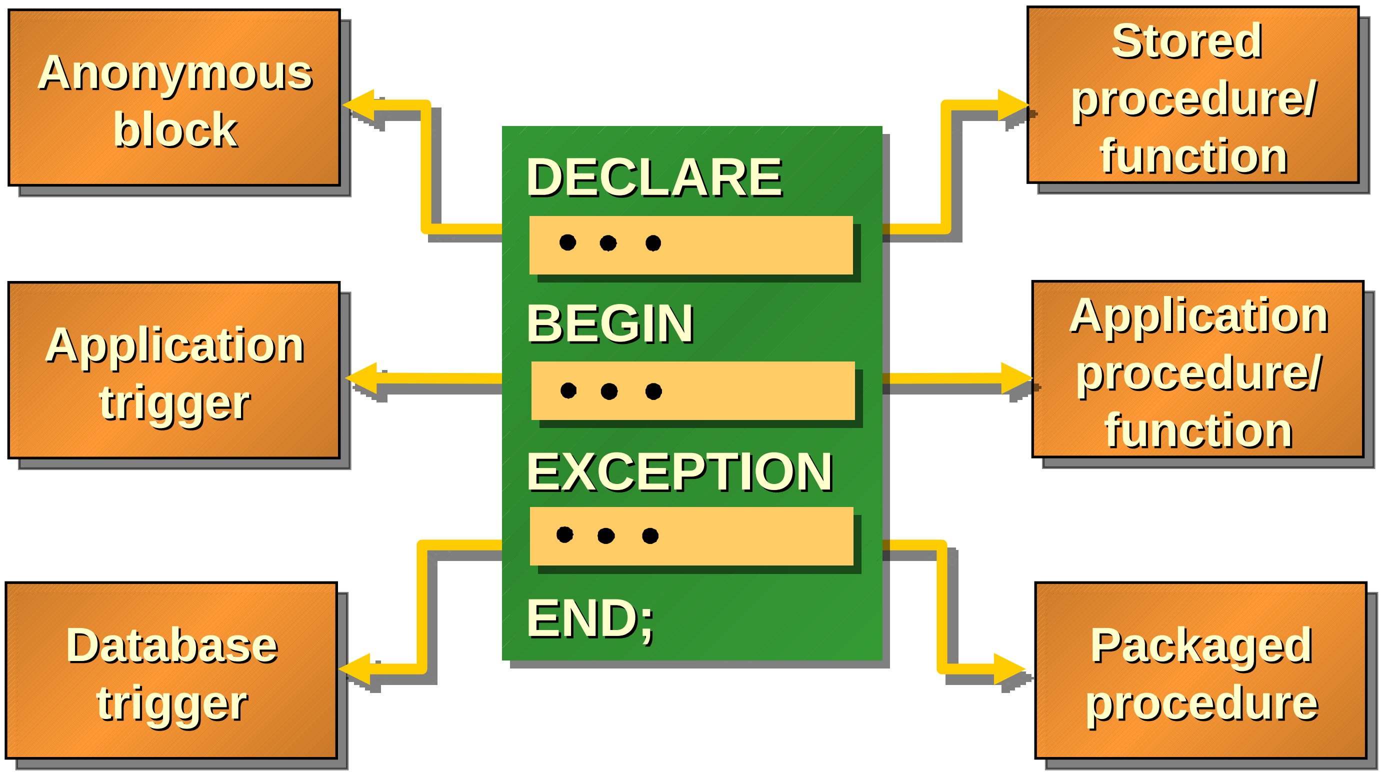
- PL/SQL is an extension to SQL with design features of programming languages.
- Data manipulation and query statements of SQL are included within procedural units of code.



```
SQL
IF...THEN
    SQL
ELSE
    SQL
END IF;
SQL
```

Benefits of PL/SQL

Modularize program development

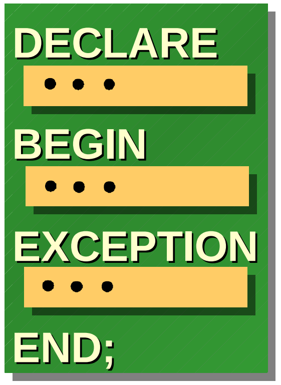


Benefits of PL/SQL

- **You can declare identifiers.**
- **You can program with procedural language control structures.**
- **It can handle errors.**

Anatomy of a PL/SQL Block

- DECLARE – Optional
  - Variables, constants, cursors, user-defined exceptions
- BEGIN – Mandatory
  - SQL statements
  - PL/SQL control statements
- EXCEPTION – Optional
  - Actions to perform when errors occur
- END; – Mandatory



```
DECLARE
   v_variable  VARCHAR2(5)
BEGIN
   SELECT   column_name
   INTO  v_variable
   FROM  table_name
END;
```

Declaring PL/SQL Variables

Syntax

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

Examples

```
Declare
   v_hiredate     DATE;
   v_deptno       NUMBER(2) NOT NULL := 10;
   v_location     VARCHAR2(13) := 'Atlanta';
   c_ comm        CONSTANT NUMBER := 1400;
```

Assigning

```
identifier := expr;

v_hiredate := '31-DEC-1998';
```

Base Scalar Datatypes

- VARCHAR2(*maximum_length*)
- NUMBER [(*precision, scale*)]
- DATE
- CHAR [(*maximum_length*)]
- LONG
- LONG RAW
- BOOLEAN
- BINARY_INTEGER

```
v_job          VARCHAR2(9);
v_count        BINARY_INTEGER := 0;
v_total_sal    NUMBER(9,2) := 0;
v_orderdate    DATE := SYSDATE + 7;
c_tax_rate     CONSTANT NUMBER(3,2) := 8.25;
v_valid        BOOLEAN NOT NULL := TRUE;
```

The %TYPE Attribute

- Declare a variable according to
  - A database column definition.
  - Another previously declared variable.
- Prefix %TYPE with
  - The database table and column.
  - The previously declared variable name.

```
...
  v_ename                  emp.ename%TYPE;
  v_balance                NUMBER(7,2);
  v_min_balance            v_balance%TYPE := 10;
...
```

Commenting Code

- Prefix single-line comments with two dashes (--).
- Place multi-line comments between the symbols /* and */.

Example

```
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the

     monthly salary input from the user */
  v_sal := v_sal * 12;
END; -- This is the end of the transaction
```

SQL Functions in PL/SQL

- **Most of SQL functions are valid in PL/SQL:**
  - **Single-row number, Single-row character, Datatype conversion, Date**

```
v_mailing_address := v_name||CHR(10)||
                             v_address||CHR(10)||
v_state||CHR(10)||v_zip;

v_ename  := LOWER(v_ename);
```

- **Group functions not available**
  - **The following example is an error**

```
v_total      := SUM(number_table);
```

Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example

```
:return_code := 0;
IF credit_check_ok(acct_no) THEN
   :return_code := 1;
END IF;
```

In SQL*Plus you can display the value of the bind variable using the PRINT command.

```
SQL>  PRINT return_code

RETURN_CODE
-----------
          1
```

Interacting with the Server

SQL Statements in PL/SQL

- **Extract a row of data from the database by using the SELECT command.**
- **Make changes to rows in the database by using DML commands.**
- **Control a transaction with the COMMIT, ROLLBACK, or SAVEPOINT command.**
- **Determine DML outcome with implicit cursors.**
- **PL/SQL does not support**
  - **data definition language (DDL), such as CREATE TABLE, ALTER TABLE, or DROP TABLE.**
  - **data control language (DCL), such as GRANT or REVOKE.**

SELECT Statements in PL/SQL

Retrieve data from the database with SELECT.

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

Example

```
DECLARE
  v_deptno NUMBER(2);
  v_loc VARCHAR2(15);
BEGIN
  SELECT   deptno, loc
    INTO   v_deptno, v_loc
  FROM dept
  WHERE dname = 'SALES';
...
END;
```

Retrieving Data in PL/SQL

Retrieve the order date and the ship date for the specified order.

Example

```
DECLARE
  v_orderdate   ord.orderdate%TYPE;
  v_shipdate    ord.shipdate%TYPE;
BEGIN
  SELECT   orderdate, shipdate
    INTO   v_orderdate, v_shipdate
  FROM     ord
  WHERE    id = 157;
   ...
END;
```

Retrieving Data in PL/SQL

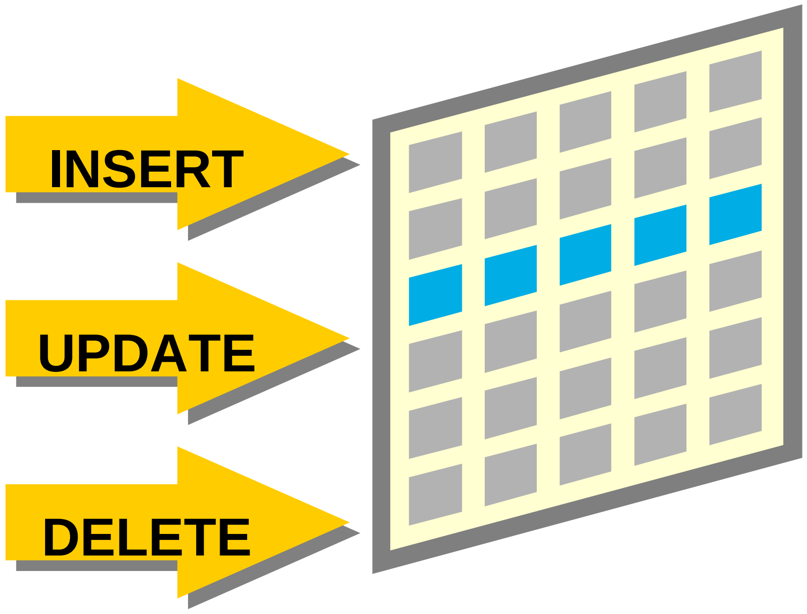
Return the sum of the salaries for all employees in the specified department.

Example

```
DECLARE
  v_sum_sal   emp.sal%TYPE;
  v_deptno    NUMBER NOT NULL := 10;
BEGIN
  SELECT SUM(sal)  -- group function
    INTO v_sum_sal
  FROM   emp
  WHERE  deptno = v_deptno;
END;
```

Manipulating Data Using PL/SQL

- Make changes to database tables by using DML commands:
  - INSERT
  - UPDATE
  - DELETE



Inserting Data

Add new employee information to the emp table.

Example

```
DECLARE
  v_empno      emp.empno%TYPE;
BEGIN
  SELECT   empno_sequence.NEXTVAL
    INTO   v_empno
    FROM   dual;
  INSERT INTO emp(empno, ename, job, deptno)
    VALUES(v_empno, 'HARDING', 'CLERK', 10);
END;
```

Updating Data

Increase the salary of all employees in the emp table who are Analysts.

Example

```
DECLARE
  v_sal_increase   emp.sal%TYPE := 2000;
BEGIN
  UPDATE    emp
    SET     sal = sal + v_sal_increase
    WHERE   job = 'ANALYST';
END;
```

Deleting Data

Delete rows that have belong to department 10 from the emp table.

Example

```
DECLARE
  v_emp   deptno.emp%TYPE := 10;
BEGIN
  DELETE FROM emp
    WHERE deptno = v_deptno;
END;
```

Controlling Transactions

Determine the transaction processing for the following PL/SQL block.

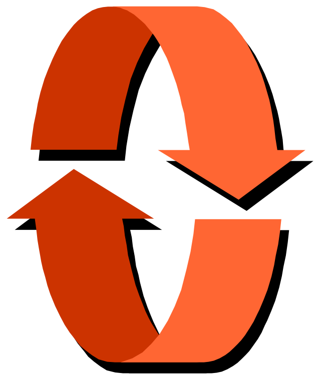
```
BEGIN
  INSERT INTO temp(num_col1, num_col2, char_col)
    VALUES (1, 1, 'ROW 1');
  SAVEPOINT a;
  INSERT INTO temp(num_col1, num_col2, char_col)
    VALUES (2, 1, 'ROW 2');
  SAVEPOINT b;
  INSERT INTO temp(num_col1, num_col2, char_col)
    VALUES (3, 3, 'ROW 3');
  SAVEPOINT c;
  ROLLBACK TO SAVEPOINT b;
  COMMIT;
END;
```

Writing Control Structures

Controlling PL/SQL Flow of Execution

You can change the logical flow of statements using conditional IF statements and loop control structures.

- Conditional IF statements:
  - IF-THEN
  - IF-THEN-ELSE
  - IF-THEN-ELSIF



IF Statements

Syntax

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

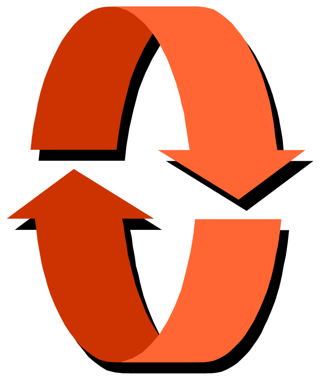
```
IF v_ename = 'OSBORNE' THEN
  v_mgr := 22;
END IF;
. . .
IF v_ename = 'MILLER' THEN
  v_job := 'SALESMAN';
  v_deptno := 35;
  v_new_comm := sal * 0.20;
END IF;
```

```
...
IF v_shipdate - v_orderdate < 5 THEN
  v_ship_flag := 'Acceptable';
ELSE
  v_ship_flag := 'Unacceptable';
END IF;
...
```

```
. . .
IF v_start > 100 THEN
  RETURN (2 * v_start);
ELSIF v_start >= 50 THEN
  RETURN (.5 * v_start);
ELSE
  RETURN (.1 * v_start);
END IF;
. . .
```

Iterative Control: LOOP Statements

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



Basic Loop

Syntax

```
LOOP                                    -- delimiter
  statement1;                           -- statements
  . . .                                 /* EXIT statement, condition
  EXIT [WHEN condition];                is a Boolean variable or expression*/
END LOOP;                               -- delimiter
```

```
. . .
  v_ordid      item.ordid%TYPE := 101;
  v_counter    NUMBER(2) := 1;
BEGIN
. . .
  LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
. . .
```

FOR Loop

```
FOR index in [REVERSE] lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

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

```
-- Insert the first 10 new line items for order number 101.
. . .
  v_ordid   item.ordid%TYPE := 101;
BEGIN
. . .
  FOR i IN 1..10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, i);
  END LOOP;
. . .
```

WHILE Loop

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

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

Condition is evaluated at the beginning of each iteration.

```
ACCEPT p_price PROMPT ‘Enter the price of the item: ‘
ACCEPT p_itemtot PROMPT ‘Enter the maximum total forpurchase of item: ‘
DECLARE
...
v_qty           NUMBER(8) := 1;
v_running_total  NUMBER(7,2) := 0;
BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
  v_qty := v_qty + 1;
  v_running_total := v_qty * p_price;
  END LOOP;
...
```

Nested Loops and Labels

- Nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the EXIT statement referencing the label.

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

Working with Composite Datatypes

PL/SQL Records

- **Must contain one or more components of any scalar, RECORD, or PL/SQL TABLE datatype-called fields.**
- **Are similar in structure to records in a 3GL.**

Syntax

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

Where *field_declaration* stands for

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

Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

Example

```
...
  TYPE emp_record_type IS RECORD
    (ename      VARCHAR2(10),
     job        VARCHAR2(9),
     sal        NUMBER(7,2));
  emp_record    emp_record_type;
...
```

The %ROWTYPE Attribute

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

Examples

Declare a variable to store the same information about a department as it is stored in the DEPT table.

```
dept_record    dept%ROWTYPE;
```

Declare a variable to store the same information about a employee as it is stored in the EMP table.

```
emp_record     emp%ROWTYPE;
```

Writing Cursors

About Cursors

Every executed SQL statement has an individual cursor associated with it:

- **A cursor is a private SQL work area.**
- **Implicit cursors:**
  - **Declared for all DML and PL/SQL SELECT statements.**
- **Explicit cursors:**
  - **Declared and named by the programmer.**
  - **Useful for managing queries that return one or more rows of data**

SQL Implicit Cursor Attributes

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

SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value).
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows.
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.

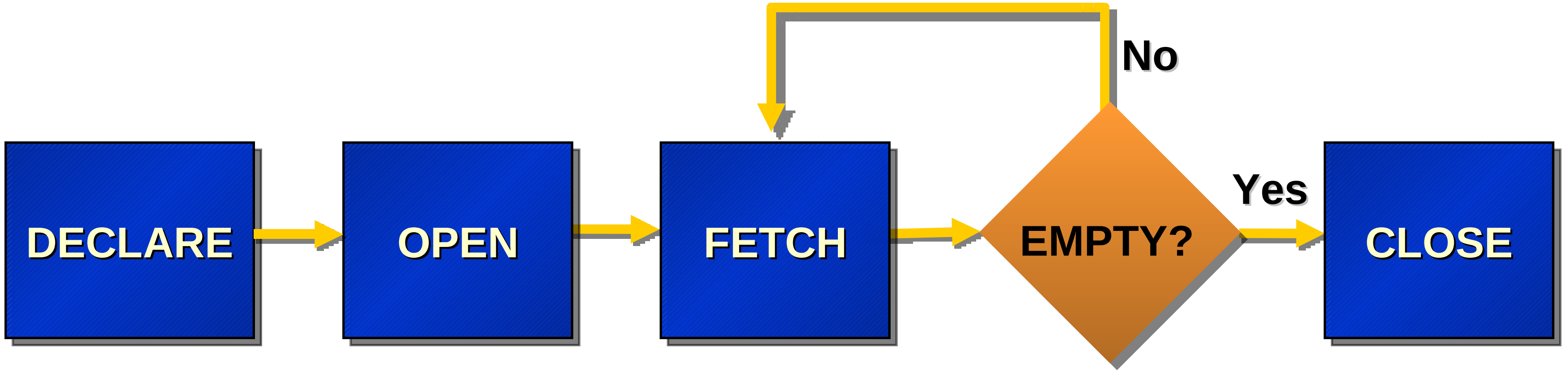
SQL Cursor Attributes

Delete rows that have the specified order number from the ITEM table. Print the number of rows deleted.

Example

```
   VARIABLE rows_deleted VARCHAR2(20)
DECLARE
  v_ordid  NUMBER := 605;
BEGIN
  DELETE FROM item
  WHERE  ordid = v_ordid;
  :rows_deleted :=(SQL%ROWCOUNT ||' rows deleted.');
END;
   /
   PRINT rows_deleted
```

Controlling Explicit Cursors



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

- TESTS:
%FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN

Controlling Explicit Cursors

- **Declare a Cursor**

```
CURSOR cursor_name IS
     select_statement;
```

- **Open a Cursor**

```
OPEN cursor_name;
```

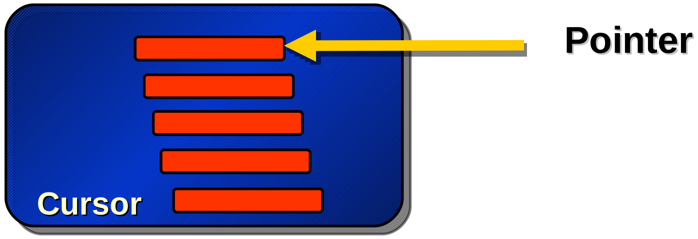
- **Fetch data from a Cursor**

```
FETCH cursor_name INTO
 [variable1, variable2, ...] |
  record_name];
```

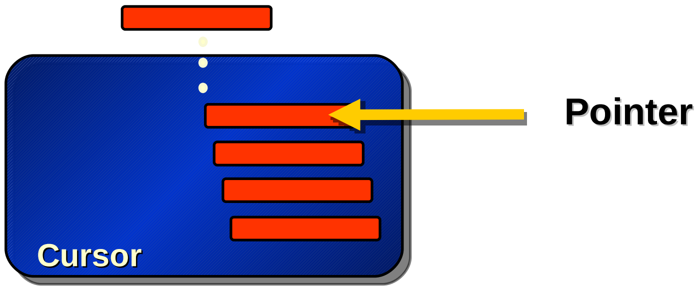
- **Close a Cursor**

```
CLOSE cursor_name;
```

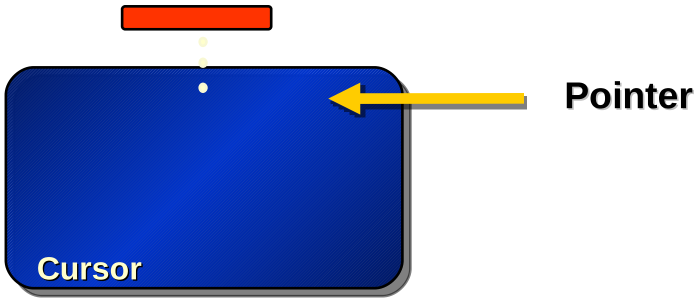
Open the cursor.



Fetch a Row from the cursor.



Continue until empty.



Explicit Cursor Attributes

Obtain status information about a cursor.

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

Explicit Cursor - example

```
DECLARE
   CURSOR EMP_CUR IS
     SELECT empno, ename, sal FROM emp;
   CURSOR c1 IS
     SELECT empno, ename, job, sal FROM   emp
      WHERE  sal > 2000;
BEGIN
  IF NOT emp_cur%ISOPEN THEN  -- or run OPEN EMP_CUR;
    OPEN emp_cur;
  END IF;
  LOOP
    fetch emp_cur into v_empno, v_ename, v_sal;
    EXIT when emp_cur%NOTFOUND;
    IF ename_cur%ROWCOUNT > 20 THEN
     ...
    IF (v_sal > 1000) then
      DBMS_OUTPUT.put_line(v_empno || ' ' || v_ename || ' ' || v_sal);
    ELSE
      DBMS_OUTPUT.put_line(v_ename || ' sal is less then 1000');
    END IF;
  END LOOP;
  close emp_cur;
  DBMS_OUTPUT.put_line('Execution Complete');
END;
```

Cursors and Records

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

Example

```
...
  CURSOR emp_cursor IS
    SELECT   empno, sal, hiredate, rowid
    FROM     emp
    WHERE deptno = 20;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
. . .
  FETCH emp_cursor INTO emp_record;
```

Cursor FOR Loops

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

- **Shortcut to process explicit cursors.**
- **Implicit open, fetch, and close occur.**

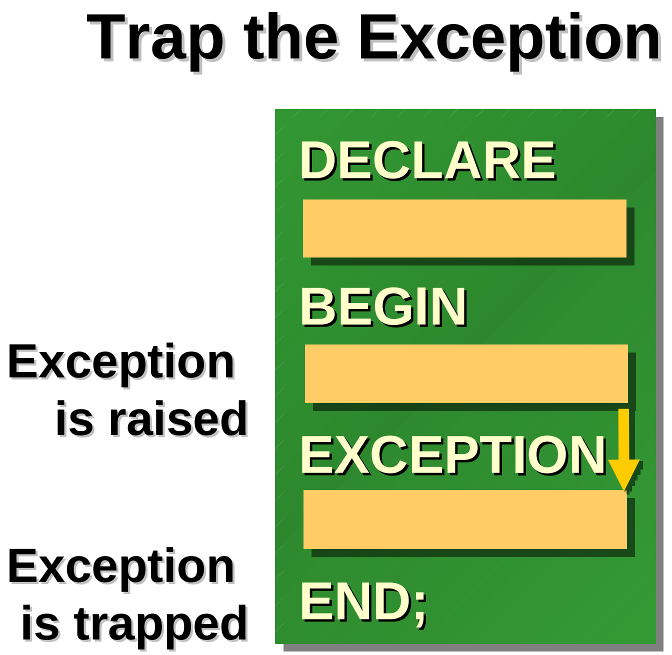
```
DECLARE
 cursor c1 is
      select sal from emp
      where job = 'MANAGER';

BEGIN
    total_val := 0;
    FOR employee_rec in c1
    LOOP
      total_val := total_val + employee_rec.sal;
    END LOOP;
END;
```

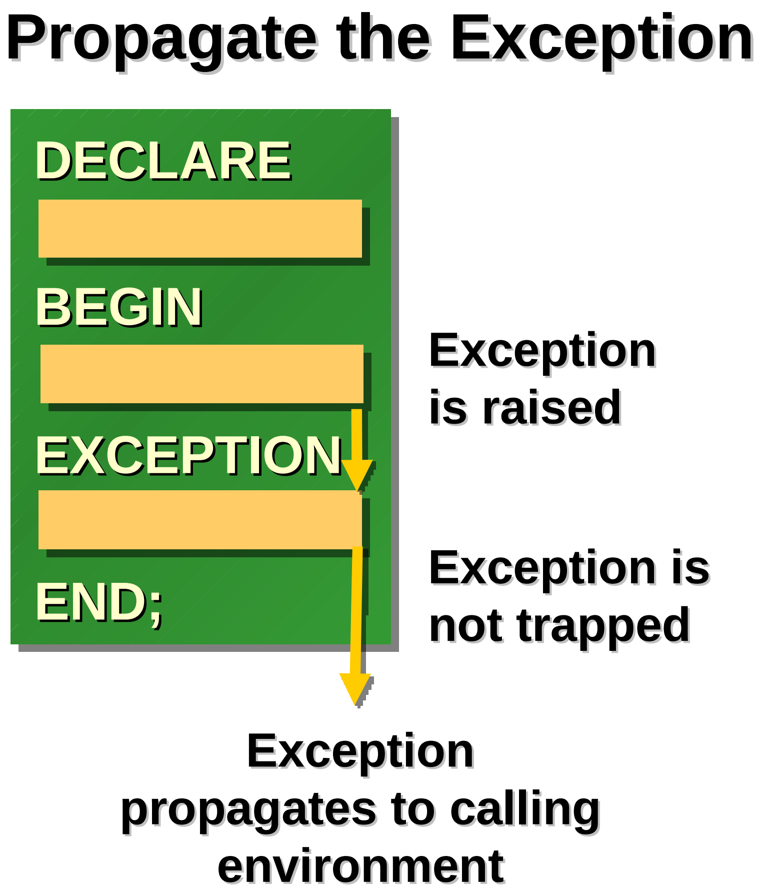
Handling Exceptions

Handling Exceptions with PL/SQL

- What is an exception?
  - Identifier in PL/SQL that is raised during execution.
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

Handling Exceptions





Trapping Exceptions

Syntax

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

Declaring Exception

```
DECLARE exception_name EXCEPTION;
```

- **Exception Types**
  - **Predefined – Implicitly raised**
    - NO_DATA_FOUND
    - TOO_MANY_ROWS
    - INVALID_CURSOR
    - ZERO_DIVIDE
    - . . .
  - **User-defined – Explicitly raised**

Predefined Exception

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1; statement2;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_prodid)||' is invalid.');
  WHEN TOO_MANY_ROWS THEN
    statement1;
    DBMS_OUTPUT.PUT_LINE('Invalid Data');
  WHEN OTHERS THEN
    statement1; statement2; statement3;
    DBMS_OUTPUT.PUT_LINE('Other error');
END;
```

User-Defined Exception

- **Each exception has an error code (default is 1) and a error message (default is "User-defined exception"), unless using the EXCEPTION_INIT pragma**

exception_name

```
[DECLARE]
  e_products_remainingEXCEPTION;
  PRAGMA EXCEPTION_INIT (e_products_remaining, -22292);
. . .
BEGIN
  . . .
  RAISE e_products_remaining;
  ...
EXCEPTION
  WHEN e_products_remaining THEN
    DBMS_OUTPUT.PUT_LINE ('Product code specified is not valid.');
. . .
END;
```

error_number

- ***error_number*** **is a negative integer in the range -20000 .. -20999**

Functions for Trapping Exceptions

- **SQLCODE**
  - **Returns the numeric value for the error code.**
- **SQLERRM**
  - **Returns the message associated with the error number.**

```
...
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
   ROLLBACK;
   v_error_code := SQLCODE;
   v_error_message := SQLERRM;
   INSERT INTO errors VALUES(v_error_code, v_error_message);
END;
```

Creating Procedures

Overview of Procedures

- **A procedure is a named PL/SQL block that performs an action.**
- **A procedure can be stored in the database, as a database object, for repeated execution.**

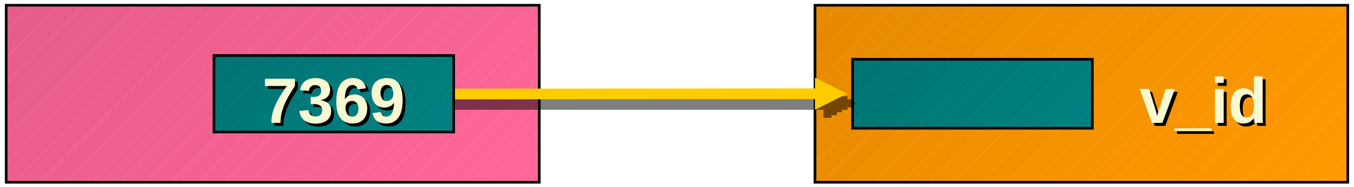
Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 (argument1 [mode] datatype1,
  argument2 [mode] datatype2,
  . . .
IS [AS]
PL/SQL Block;
```

- ***mode*: has one of the following values: IN, OUT, IN OUT**
- **A stored procedure can be removed as follow:**

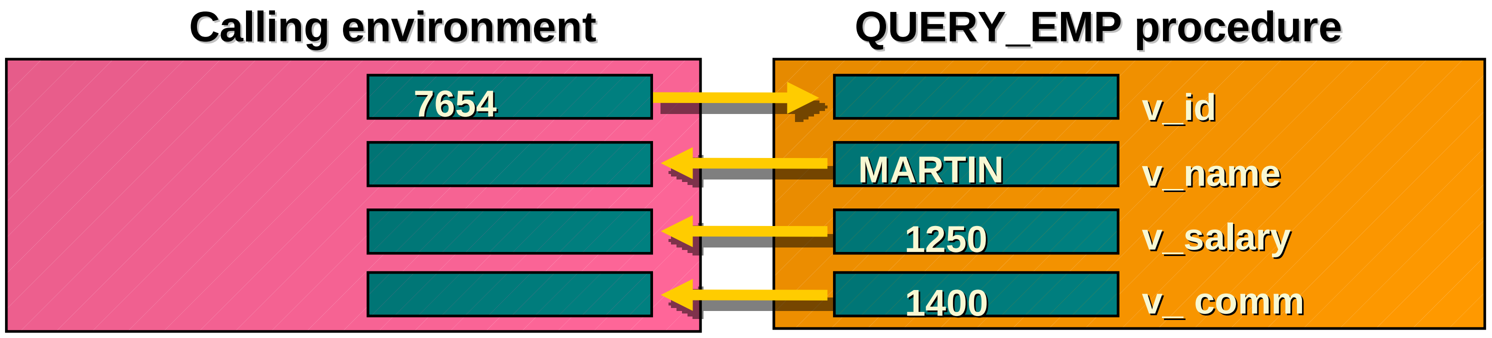
```
DROP PROCEDURE procedure_name
```

IN Parameters: Example



```
SQL> CREATE OR REPLACE PROCEDURE raise_salary
  2  (v_id in emp.empno%TYPE)
  3  IS
  4  BEGIN
  5 UPDATE  emp
  6 SET     sal = sal * 1.10
  7 WHERE   empno = v_id;
  8  END raise_salary;
  9  /
Procedure created.

SQL> EXECUTE raise_salary (7369)
PL/SQL procedure successfully completed.
```

OUT Parameters: Example



```
SQL> CREATE OR REPLACE PROCEDURE query_emp
  1  (v_id      IN   emp.empno%TYPE,
  2   v_name    OUT  emp.ename%TYPE,
  3   v_salary OUT  emp.sal%TYPE,
  4   v_comm    OUT  emp.comm%TYPE)
  5  IS
  6  BEGIN
  7    SELECT    ename, sal, comm
  8    INTO      v_name, v_salary, v_comm
  9    FROM      emp
 10    WHERE     empno = v_id;
 11  END query_emp;
 12  /
```

OUT Parameters and SQL*Plus

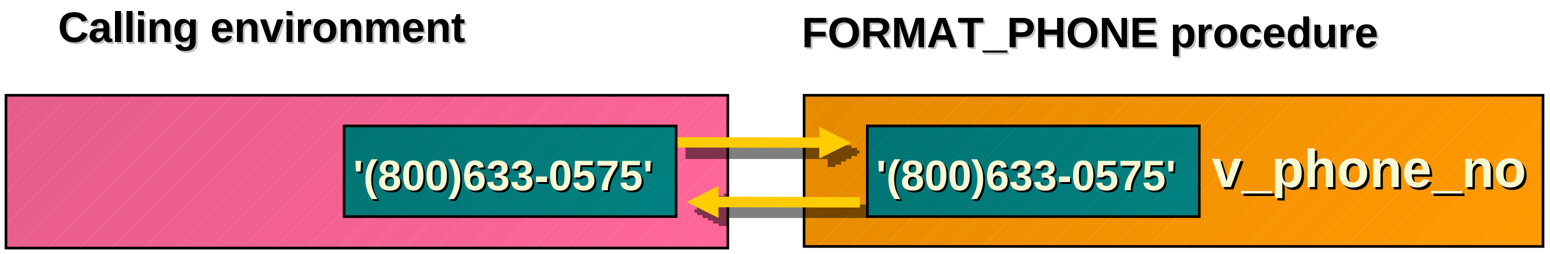
```
SQL> START emp_query.sql
Procedure created.
```

```
SQL> VARIABLE g_name        varchar2(15)
SQL> VARIABLE g_salary      number
SQL> VARIABLE g_comm        number
```

```
SQL> EXECUTE query_emp (7654, :g_name, :g_salary,
  2  :g_comm)
PL/SQL procedure successfully completed.
```

```
SQL> PRINT g_name
G_NAME
---------------
MARTIN
```

IN OUT Parameters



```
SQL> CREATE OR REPLACE PROCEDURE format_phone
  2  (v_phone_no IN OUT VARCHAR2)
  3  IS
  4  BEGIN
  5   v_phone_no := '(' || SUBSTR(v_phone_no,1,3) ||
  6                 ')' || SUBSTR(v_phone_no,4,3) ||
  7                 '-' || SUBSTR(v_phone_no,7);
  8  END format_phone;
  9 /
```

Invoking FORMAT_PHONE from SQL*Plus

```
SQL>VARIABLE g_phone_no varchar2(15)

SQL> BEGIN :g_phone_no := '8006330575'; END;
  2  /
PL/SQL procedure successfully completed.

SQL> EXECUTE format_phone (:g_phone_no)
PL/SQL procedure successfully completed.

SQL> PRINT g_phone_no
```

```
G_PHONE_NO
---------------
(800)633-0575
```

Invoking a Procedure from

- **From an Anonymous PL/SQL Block**

```
DECLARE
  v_id NUMBER := 7900;
BEGIN
  raise_salary(v_id);    --invoke procedure
COMMIT;
...
END;
```

- **From a Stored procedure**

```
SQL> CREATE OR REPLACE PROCEDURE process_emps
  2  IS
  3    CURSOR emp_cursor IS
  4    SELECT empno
  5    FROM   emp;
  6  BEGIN
  7    FOR emp_rec IN emp_cursor LOOP
  8       raise_salary(emp_rec.empno);  --invoke procedure
  9    END LOOP;
 10  COMMIT;
 11  END process_emps;
 12 /
```

Creating Functions

Overview of Stored Functions

- **A function is a named PL/SQL block that returns a value.**
- **A function can be stored in the database, as a database object, for repeated execution.**

```
CREATE [OR REPLACE] FUNCTION function_name
 (argument1 [mode] datatype1,
  argument2 [mode] datatype2,
  . . .
RETURN datatype
IS|AS
PL/SQL Block;
```

- ***mode*: has only the value: IN**

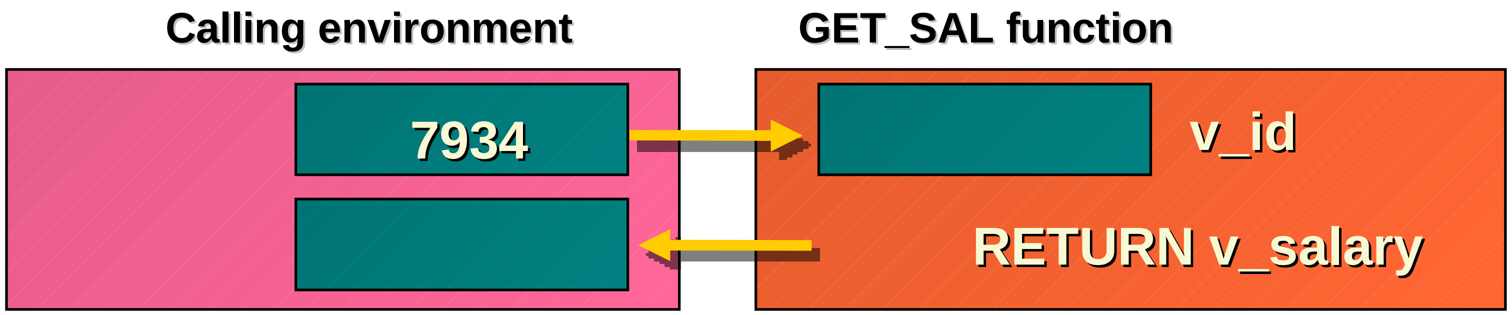
A stored function can be removed as follow:

```
DROP FUNCTION function_name
```

Creating a Stored Function Using SQL*Plus: Example

```
SQL> CREATE OR REPLACE FUNCTION get_sal
  2    (v_id   IN    emp.empno%TYPE)
  3   RETURN NUMBER
  4   IS
  5     v_salary  emp.sal%TYPE :=0;
  6   BEGIN
  7     SELECT sal
  8     INTO   v_salary
  9     FROM   emp
 10     WHERE  empno = v_id;
 11     RETURN (v_salary);
 12 END get_sal;
 13 /
```

Executing Functions in SQL*Plus: Example



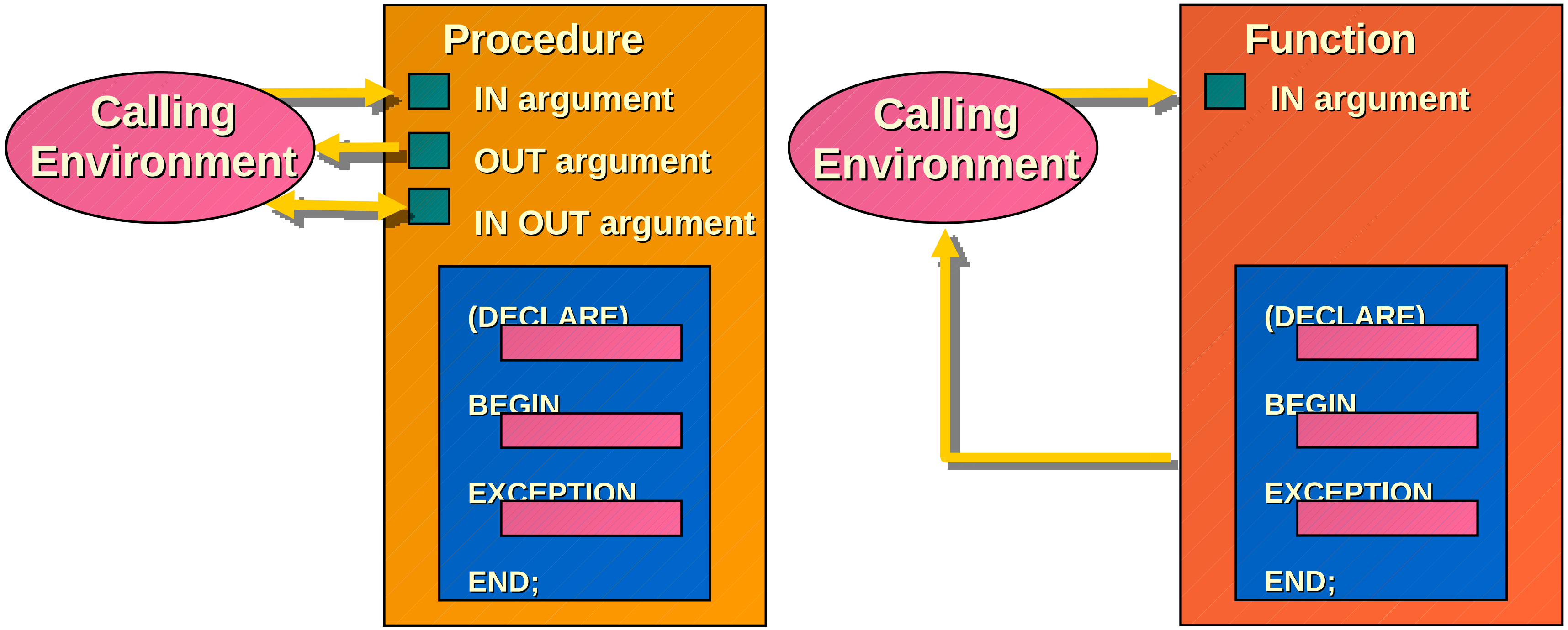
```
SQL> START get_salary.sql
Procedure created.
```

```
SQL> VARIABLE g_salary number
```

```
SQL> EXECUTE :g_salary := get_sal(7934)
PL/SQL procedure successfully completed.
```

```
SQL> PRINT g_salary
    G_SALARY
------------------
      1300
```

Procedure or Function?



Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No RETURN datatype	Must contain a RETURN datatype
Can return one or more values	Must return a value

Creating Packages

Overview of Packages

- **Group logically related PL/SQL types, items, and subprograms**
- **Advantages**
  - **Modularity**
  - **Information hiding**
  - **Added functionality**
- **Consist of two parts:**
  - **Specification**
    - Lists all the objects that are publicly available
  - **Body**
    - Code needed to implement procedures, functions, and cursors listed in the specification, as well as any private objects

Creating the Package Specification

```
CREATE [OR REPLACE] PACKAGE package_name
IS | AS
    public type and item declarations
    subprogram specifications
END package_name;
```

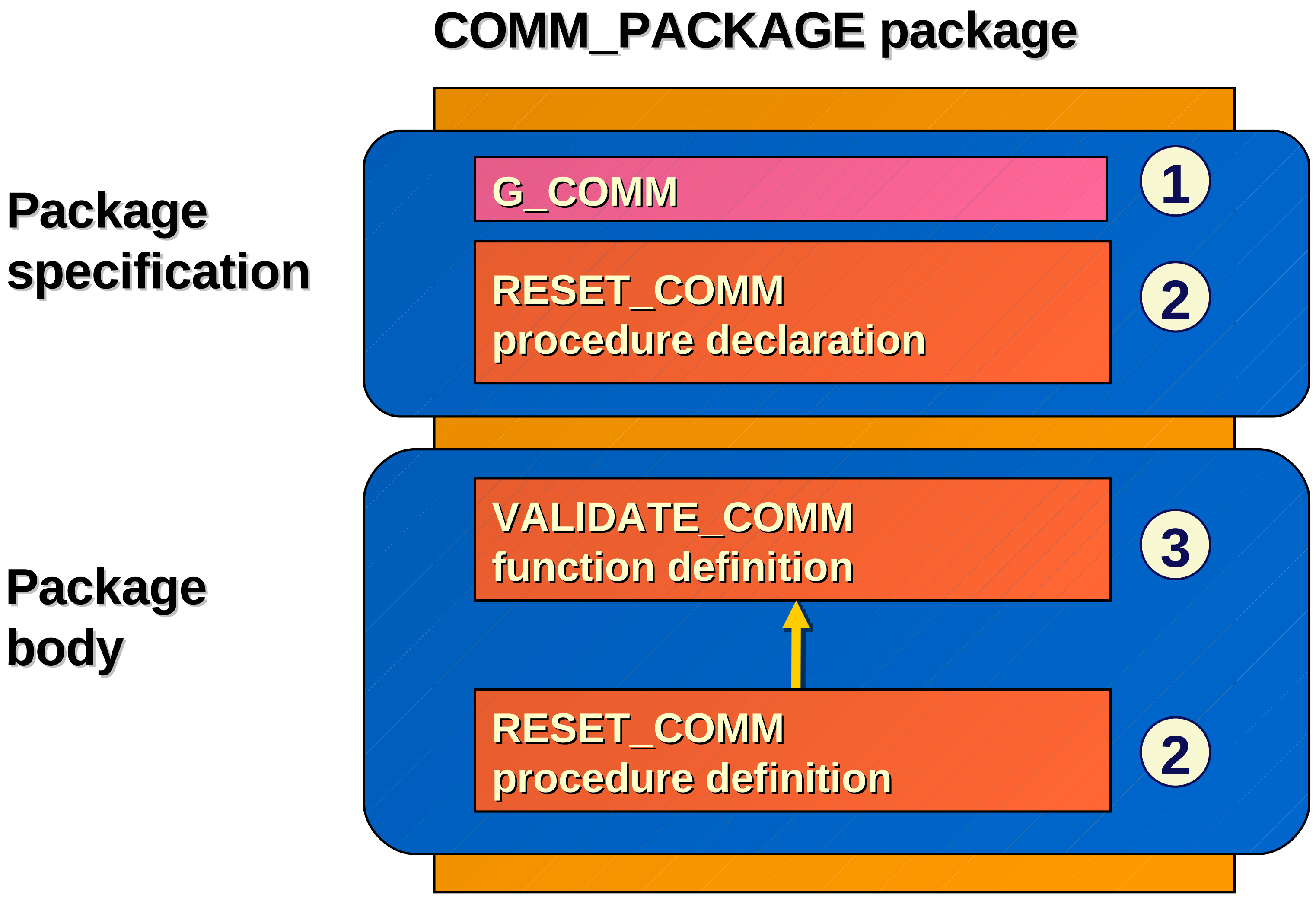
Creating the Package Body

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS | AS
    private type and item declarations
    subprogram bodies
END package_name;
```

Example

```
CREATE OR REPLACE PACKAGE time_pkg IS
   FUNCTION  GetTimestamp  RETURN DATE;
   PROCEDURE ResetTimestamp;
END time_pkg;

CREATE OR REPLACE PACKAGE BODY time_pkg IS
   StartTimeStamp   DATE := SYSDATE; -- package data.
   FUNCTION GetTimestamp RETURN DATE IS
   BEGIN
      RETURN StartTimeStamp;
   END GetTimestamp;
   PROCEDURE ResetTimestamp IS
   BEGIN
      StartTimeStamp := SYSDATE;
   END ResetTimestamp;
END time_pkg;
```

Public and Private Constructs


COMM_PACKAGE package
Package specification
G_COMM
1
RESET_COMM procedure declaration
2
Package body
VALIDATE_COMM function definition
3
RESET_COMM procedure definition
2

Creating a Package Body: Example

```
SQL>CREATE OR REPLACE PACKAGE BODY comm_package IS
    FUNCTION validate_comm
     (v_comm   IN NUMBER) RETURN  BOOLEAN
    IS
     v_max_comm NUMBER;
    BEGIN
     SELECT MAX(comm)
     INTO   v_max_comm
     FROM   emp;
     IF v_comm > v_max_comm THEN RETURN(FALSE);
     ELSE RETURN(TRUE);
      END IF;
    END validate_comm;
    END comm_package;
    /
```

Creating a Package Body: Example

```
SQL> PROCEDURE reset_comm
  2 (v_comm IN NUMBER)
  3 IS
  4  v_valid     BOOLEAN;
  5 BEGIN
  6  v_valid := validate_comm(v_comm);
  7 IF v_valid = TRUE THEN
  8  g_comm := v_comm;
  9 ELSE
 10    RAISE_APPLICATION_ERROR
 11    (-20210,'Invalid commission');
 12 END IF;
 13 END reset_comm;
 14 END comm_package;
 15 /
```

Invoking Package Constructs

The elements declared in the specification are referenced from the calling application via dot notation:

package_name.package_element

For example,

DBMS_OUTPUT.PUT_LINE('This is parameter data');

Example 1: Invoke a package procedure from SQL*Plus.

```
SQL>  EXECUTE comm_package.reset_comm(1500);
```

Example 2: Invoke a package procedure in a different schema.

```
SQL>  EXECUTE scott.comm_package.reset_comm(1500);
```

Example 3: Invoke a package procedure in a remote database.

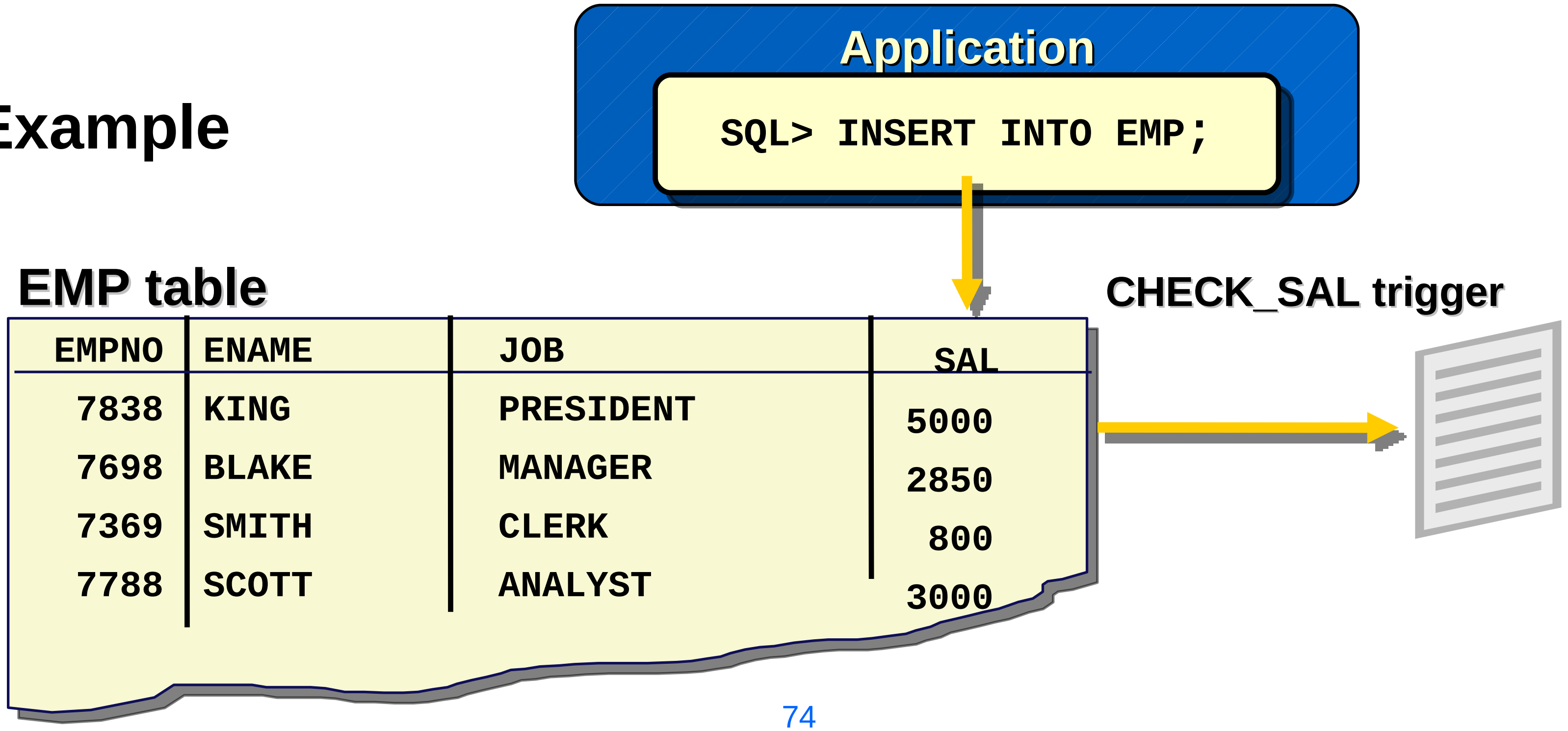
```
SQL>  EXECUTE comm_package.reset_comm@ny (1500);
```

Creating Database Triggers

Overview of Triggers

- A trigger is a PL/SQL block that executes implicitly whenever a particular event takes place.
- A trigger can be either a database trigger or an application trigger.

Example



```
SQL> INSERT INTO EMP;
```

EMPNO	ENAME	JOB	SAL
7838	KING	PRESIDENT	5000
7698	BLAKE	MANAGER	2850
7369	SMITH	CLERK	800
7788	SCOTT	ANALYST	3000

Creating Triggers

- **Trigger timing:**
  - **BEFORE: The code in the trigger body will execute before the triggering DML event.**
  - **AFTER: The code in the trigger body will execute after the triggering DML event.**
- **Triggering event: INSERT or UPDATE or DELETE**
- **Table name: On table**
- **Trigger type:**
  - **Statement: The trigger body executes once for the triggering event. This is the default.**
  - **Row: The trigger body executes once for each row affected by the triggering**
- **Trigger body:**

```
[DECLARE]
BEGIN
[EXCEPTIONS]
END
```

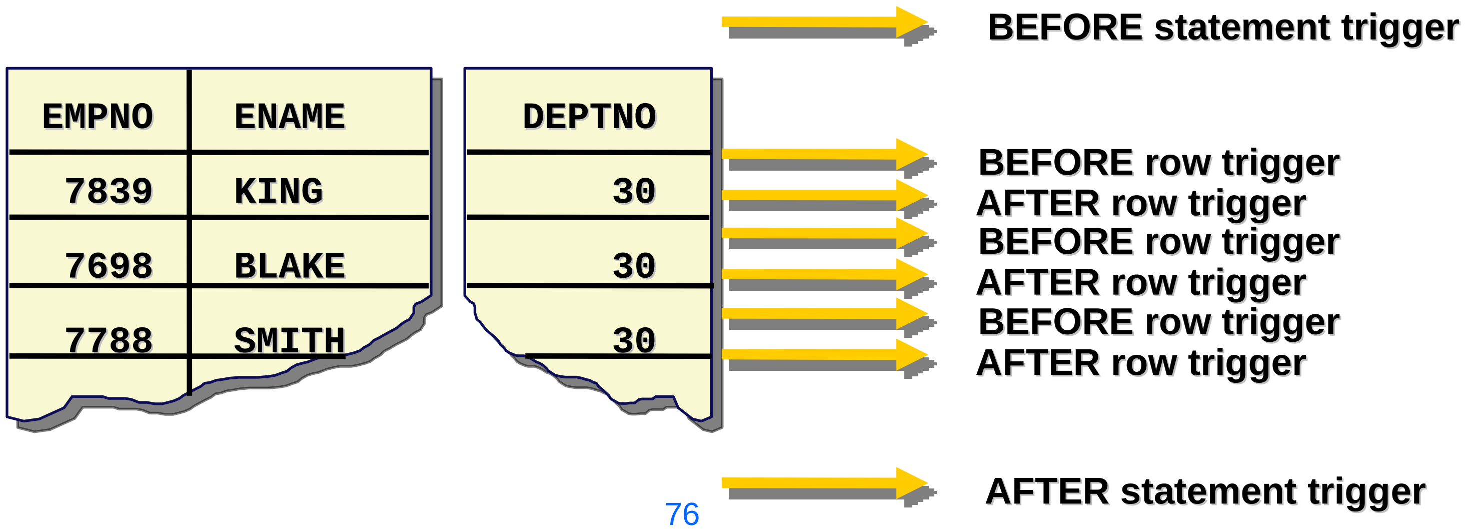
Statement and Row Triggers

Example 1

```
SQL> INSERT INTO dept (deptno, dname, loc)
  2  VALUES (50, 'EDUCATION', 'NEW YORK');
```

Example 2

```
SQL> UPDATE emp
  2  SET sal = sal * 1.1
  3  WHERE deptno = 30;
```



Syntax for Creating Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing event_1 [OR event_2 OR event_3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
PL/SQL block;
```

timing **is** `BEFORE or AFTER`

trigger-name **is the name of the Trigger Object**

event_i **is either INSERT, DELETE or UPDATE.**

It is possible to combine these, for example:

create or replace trigger FIRE_AFTER_ALL after insert or update or delete on tab1

Before Statement Trigger: Example

```
SQL> CREATE OR REPLACE TRIGGER secure_emp
     BEFORE INSERT ON emp
     BEGIN
      IF (TO_CHAR (sysdate,'DY') IN ('SAT','SUN'))
        OR (TO_CHAR(sysdate,'HH24')NOT BETWEEN
       '08' AND '18'
        THEN RAISE_APPLICATION_ERROR (-20500,
       'You may only insert into EMP during normal
        hours.');
      END IF;
     END;
     /
```

Example

```
SQL> INSERT INTO emp (empno, ename, deptno)
  2  VALUES          (7777, 'BAUWENS', 40);
INSERT INTO emp (empno, ename, deptno)
            *
ERROR at line 1:
ORA-20500: You may only insert into EMP during
normal hours.
ORA-06512: at "SCOTT.SECURE_EMP", line 4
ORA-04088: error during execution of trigger
'SCOTT.SECURE_EMP'
```

Using Conditional Predicates

```
SQL>CREATE OR REPLACE TRIGGER secure_emp
  2 BEFORE INSERT OR UPDATE OR DELETE ON emp
  3 BEGIN
  4  IF (TO_CHAR (sysdate,'DY') IN ('SAT','SUN')) OR
  5  (TO_CHAR (sysdate, 'HH24') NOT BETWEEN '08' AND '18') THEN
  6 IF DELETING THEN
  7   RAISE_APPLICATION_ERROR (-20502,
  8   'You may only delete from EMP during normal hours.');
  9      ELSIF INSERTING THEN
 10       RAISE_APPLICATION_ERROR (-20500,
 11       'You may only insert into EMP during normal hours.');
 12      ELSIF UPDATING ('SAL') THEN
 13           RAISE_APPLICATION_ERROR (-20503,
 14           'You may only update SAL during normal hours.');
 15      ELSE
 16        RAISE_APPLICATION_ERROR (-20504,
 17        'You may only update EMP during normal hours.');
 18     END IF;
 19  END IF;
 20 END;
 21 /
```

After statement Trigger: Example

```
CREATE OR REPLACE TRIGGER emp_log_t
   AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
   dmltype  CHAR(1);
BEGIN
   IF INSERTING THEN
      dmltype := 'I';
      INSERT INTO emp_log (who, operation, timestamp)
         VALUES (USER, dmltype, SYSDATE);
   ELSIF UPDATING  THEN
      dmltype := 'U';
      INSERT INTO emp_log (who, operation, timestamp)
         VALUES (USER, dmltype, SYSDATE);
   ELSIF DELETING  THEN
      dmltype := 'D';
      INSERT INTO emp_log (who, operation, timestamp)
         VALUES (USER, dmltype, SYSDATE);
   END IF;
END;
```

After Row Trigger: Example

```
CREATE OR REPLACE TRIGGER emp_log_t
   AFTER INSERT OR UPDATE OR DELETE ON emp
   FOR EACH ROW
DECLARE
   dmltype  CHAR(1);
BEGIN
   IF INSERTING THEN
      dmltype := 'I';
      INSERT INTO emp_log (who, operation, timestamp)
         VALUES (USER, dmltype, SYSDATE);
   ELSIF UPDATING  THEN
      dmltype := 'U';
      INSERT INTO emp_log (who, operation, timestamp)
         VALUES (USER, dmltype, SYSDATE);
   ELSIF DELETING  THEN
      dmltype := 'D';
      INSERT INTO emp_log (who, operation, timestamp)
         VALUES (USER, dmltype, SYSDATE);
   END IF;
END;
```

Special Variables :old and :new

- **<u>FOR EACH ROW clause:</u>**
  - **A Statement level trigger fires only once for the triggering event. No access to the column.**
  - **A Row-Level trigger fires for each affected row. Can access the original and new column.**

- **<u>Old and New Column Values:</u>**

Row level triggers have access to both the copies of the column values. The old and the new values. These are referenced through the special variables:

- **:old.*column-name***
- **:new.*column-name***

After Row Trigger: Example

```
CREATE OR REPLACE TRIGGER emp_log_t
   AFTER INSERT OR UPDATE OR DELETE ON emp
   FOR EACH ROW
DECLARE
   dmltype  CHAR(1);
BEGIN
   IF INSERTING THEN
      dmltype := 'I';
      INSERT INTO emp_log (who, operation, timestamp, emp_no)
         VALUES (USER, dmltype, SYSDATE, :new.empno);
   ELSIF UPDATING  THEN
      dmltype := 'U';
      INSERT INTO emp_log (who, operation, timestamp, emp_no)
         VALUES (USER, dmltype, SYSDATE, :new.empno);
   ELSIF DELETING  THEN
      dmltype := 'D';
      INSERT INTO emp_log (who, operation, timestamp, emp_no)
         VALUES (USER, dmltype, SYSDATE, :old.empno);
   END IF;
END;
```

Using Old and New Qualifiers

```
SQL>CREATE OR REPLACE TRIGGER audit_emp_values
  2 AFTER DELETE OR INSERT OR UPDATE ON emp
  3 FOR EACH ROW
  4 BEGIN
  5   INSERT INTO audit_emp_values (user_name,
  6    timestamp, id, old_last_name, new_last_name,
  7    old_title, new_title, old_salary, new_salary)
  8   VALUES (USER, SYSDATE, :old.empno, :old.ename,
  9    :new.ename, :old.job, :new.job,
 10    :old.sal, :new.sal);
 11 END;
 12 /
```

User Audit_Emp_Values Table

USER_NAME	TIMESTAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME
EGRAVINA	12-NOV-97	7950	NULL	HUTTON
NGREENBE	10-DEC-97	7844	MAGEE	TURNER

Continuation

OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
NULL	ANALYST	NULL	3500
CLERK	SALESMAN	1100	1100

Restricting a Row Trigger

```
SQL>CREATE OR REPLACE TRIGGER derive_commission_pct
  2 BEFORE INSERT OR UPDATE OF sal ON emp
  3 FOR EACH ROW
  4 WHEN (new.job = 'SALESMAN')
  5 BEGIN
  6  IF INSERTING THEN  :new.comm := 0;
  7   ELSE          /* UPDATE of salary */
  8    IF :old.comm IS NULL THEN
  9      :new.comm :=0;
 10    ELSE
 11      :new.comm := :old.comm * (:new.sal/:old.sal);
 12    END IF;
 13  END IF;
 14 END;
 15 /
```

Managing Triggers

Disable or Re-enable a database trigger

```
ALTER TRIGGER trigger_name  DISABLE | ENABLE
```

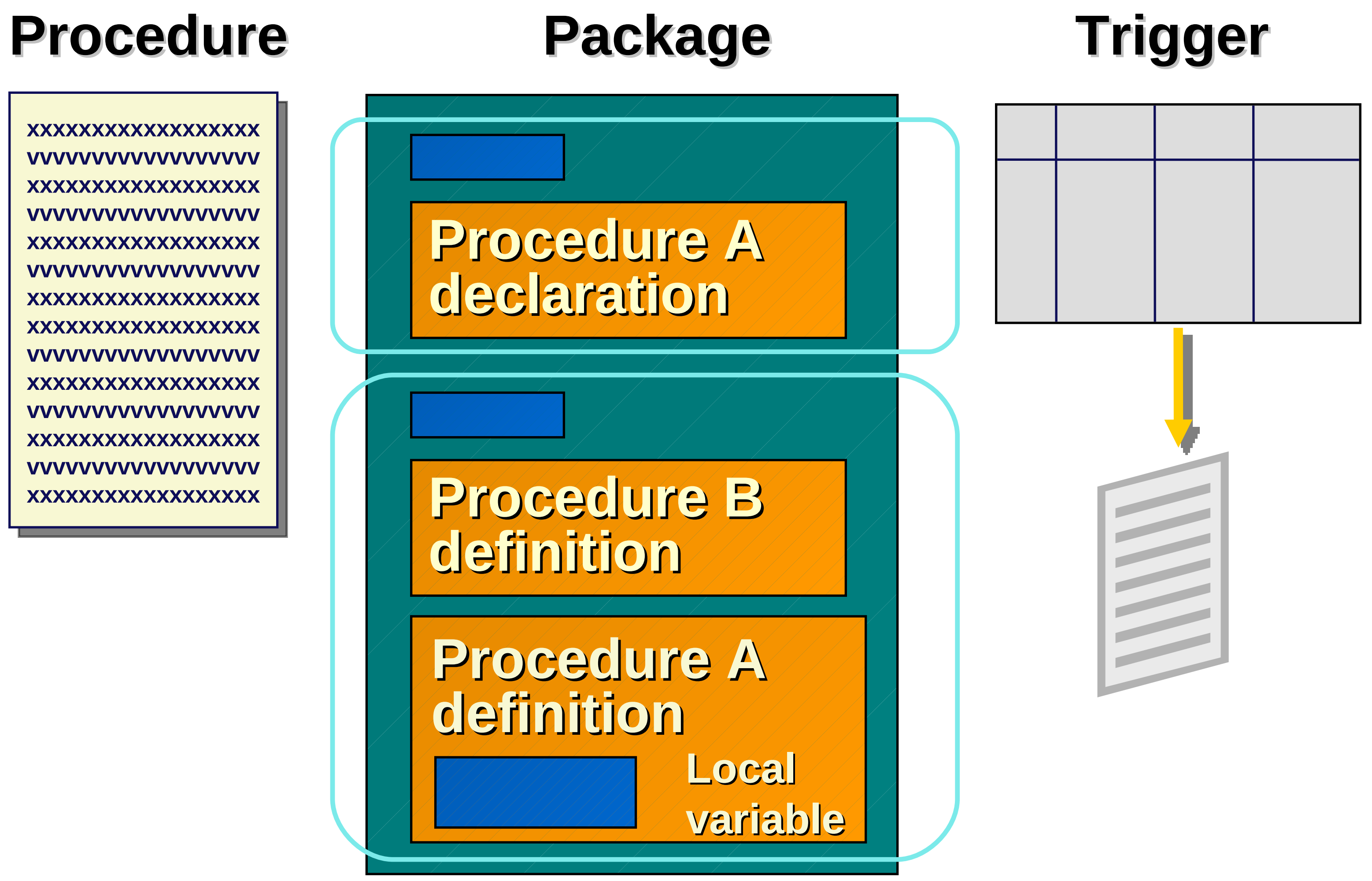
Disable or Re-enable all triggers for a table

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

Recompile a trigger for a table

```
ALTER TRIGGER trigger_name COMPILE
```

Summary


Procedure
Package
Trigger
Procedure A declaration
Procedure B definition
Procedure A definition
Local variable