II

Base de données II
Concepts et programmation procédurale

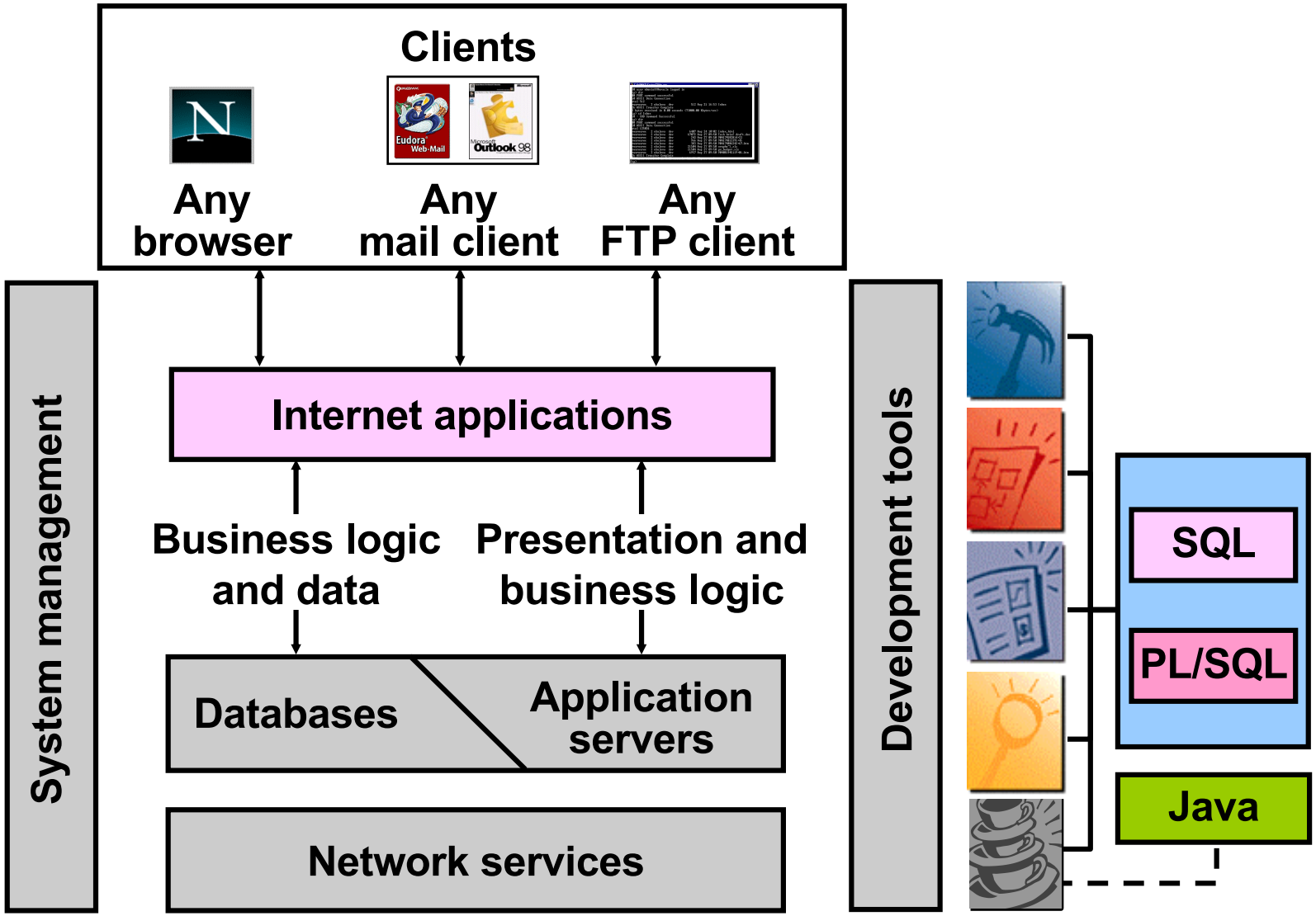
Mise en oeuvre avec PL/SQL

Présenté par Pr. B. El Asri

Objectifs

- Comprendre ce que PL/SQL fournit comme extensions de programmation à SQL
- Écrire du code PL/SQL en interface avec la base de données
- Concevoir des blocks PL/SQL qui s'exécutent efficacement
- Utiliser des constructions PL/SQL pour le traitement conditionnel et de boucle
- Gérer les erreurs d'exécution
- Décrire les fonctions et procédures stockées

Oracle Internet Platform



Généralités
Procedural language for SQL

Oracle : PL/SQL

PosgreSQL : PL/pgSQL

SQL Server : Transact-SQL

DB2 : SQLPL

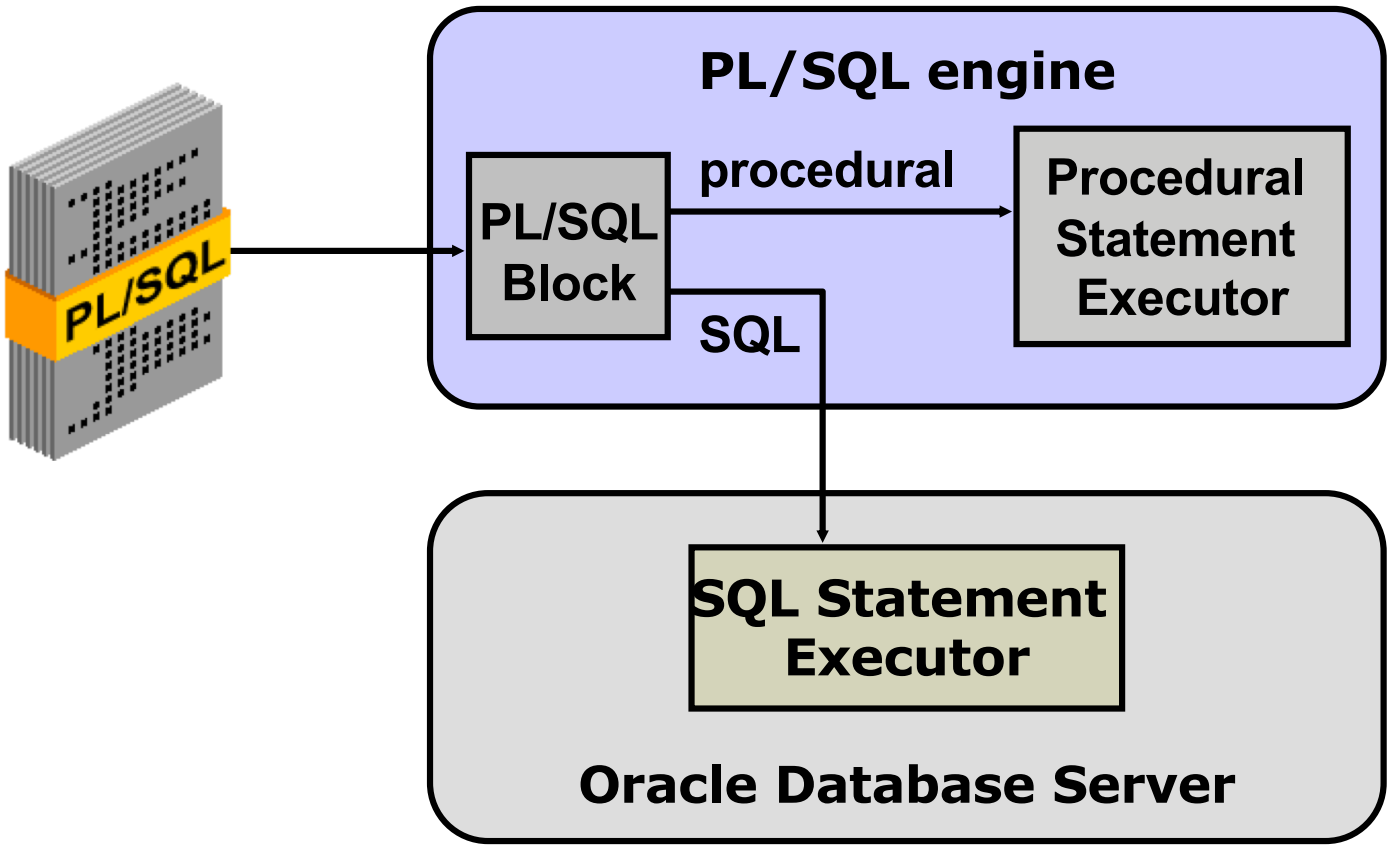
MySQL PL/mySQL depuis 5.0

Généralités

PL/SQL : Langage procédural

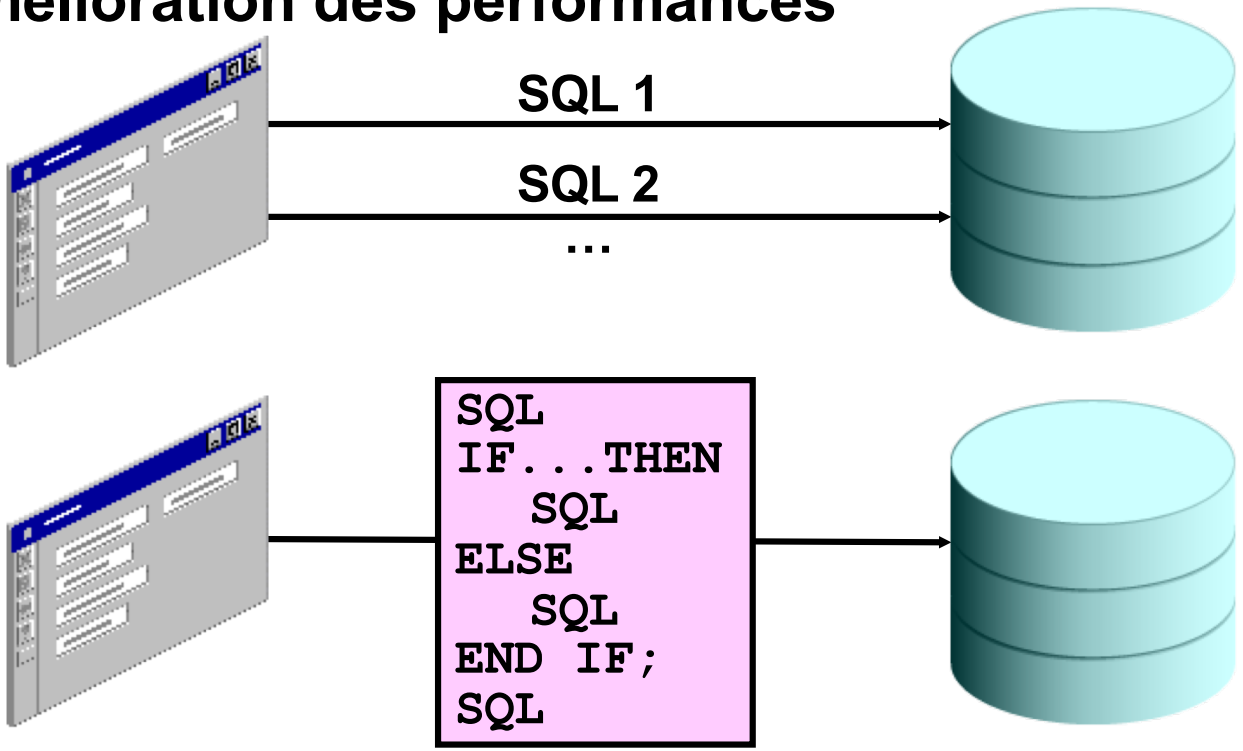
- **Extension de SQL**
- **Déclaration de variables et de constantes**
- **Définition de sous-programmes**
- **Gestion des erreurs à l'exécution (exceptions)**
- **Manipulation de données avec SQL**

L’environnement PL/SQL



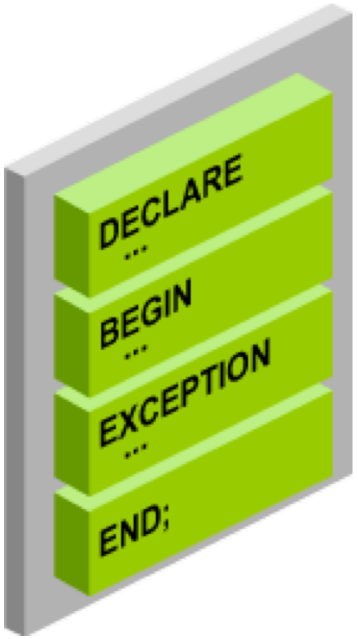
Les avantages de PL/SQL

- Intégration des constructions procédurales avec SQL
- Amélioration des performances



PL/SQL: Structure d'un Block

- **DECLARE (optional)**
  - Variables, cursors, user-defined exceptions
- **BEGIN (mandatory)**
  - SQL statements
  - PL/SQL statements
- **EXCEPTION (optional)**
  - Actions to perform when errors occur
- **END; (mandatory)**



Block Types

Anonymous

```
[DECLARE]

BEGIN
  --statements

[EXCEPTION]

END;
```

Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

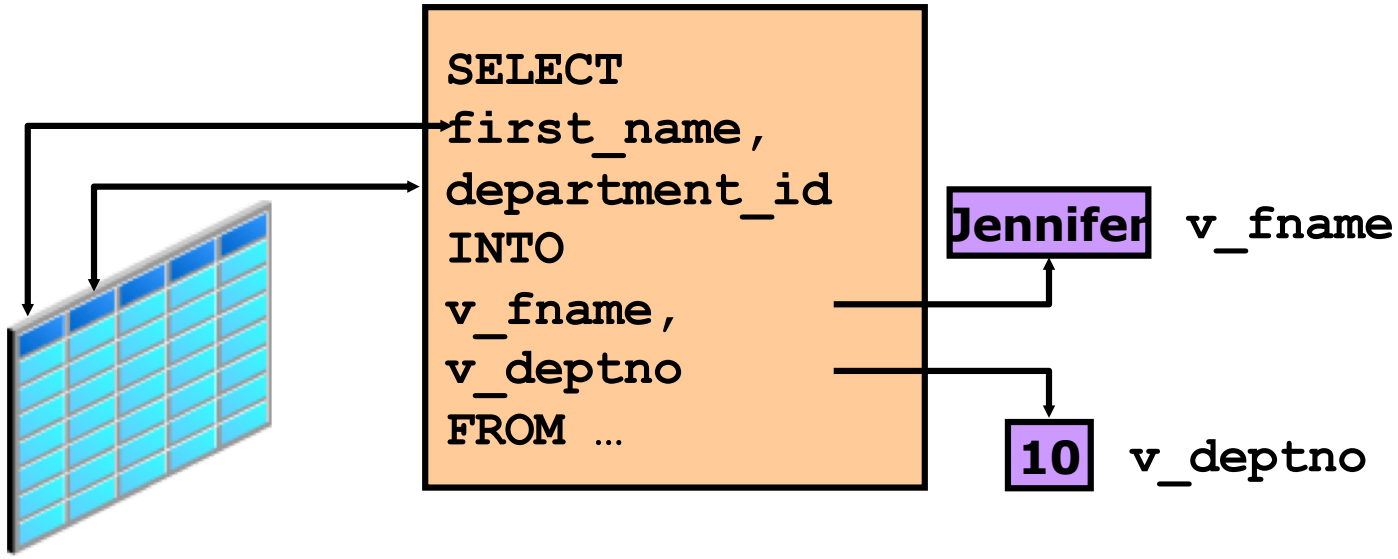
```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

LES VARIABLES PL/SQL

Les Variables

Les Variables peuvent être utilisées pour :

- **le stockage temporaire des données**
- **la manipulation des valeurs stockées**
- **la réutilisabilité**



Les variables

PL/SQL gère deux types de variables

- **Les variables PL/SQL**
  - **Scalar**
  - **Composite**
  - **Reference**
  - **Large object (LOB)**
- **Les variables non PL/SQL**
  - **Les variables champs écrans FORMS**
  - **Les variables de lien (“ bind ” variables -variables SQL).**
  - **Les variables du langage hôte dans les langages PRO.**
  - **Elles sont toujours préfixées de ':' lors de leur utilisation.**
  - **Les variables PL/SQL déclarées dans les packages sont toujours préfixées du nom du package lors de leur utilisation.**

Declarer et Initialiser des Variables PL/SQL

Syntax:

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

Exemples:

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

Declarer et Initialiser des Variables PL/SQL

1

```
DECLARE
  v_myName VARCHAR2(20);
BEGIN
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
 v_myName := 'John';
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

2

```
DECLARE
 v_myName VARCHAR2(20):= 'John';
BEGIN
 v_myName := 'Steven';
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

Comment déclarer et initialiser des Variables PL/SQL

- Suivre les conventions d'affectation de noms.
- Utiliser des identificateurs significatifs pour les variables.
- Initialiser des variables désignés comme non NULL et constante.
- Initialiser des variables avec l'opérateur d'assignation (: =) ou le mot clé DEFAULT :

```
v_myName VARCHAR2(20):='John';
```

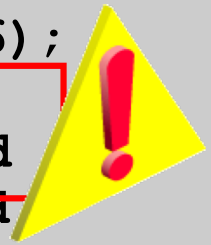
```
v_myName VARCHAR2(20) DEFAULT 'John';
```

- Déclarer chaque identificateur sur une ligne à part pour faciliter la maintenance de code et sa lisibilité.

Comment déclarer et initialiser des Variables PL/SQL

- Évitez d'utiliser des noms de colonne en tant qu'identificateurs.

```
DECLARE
   employee_id  NUMBER(6);
BEGIN
   SELECT     employee_id
   INTO       employee_id
   FROM       employees
   WHERE      last_name = 'Kochhar';
END;
/
```



- Utiliser la contrainte `NOT NULL` lorsque la variable doit recevoir une valeur.

Types basiques

- `CHAR [(maximum_length)]`
- `VARCHAR2 (maximum_length)`
- `NUMBER [(precision, scale)]`
- `BINARY_INTEGER`
- `PLS_INTEGER`
- `BOOLEAN`
- `BINARY_FLOAT`
- `BINARY_DOUBLE`

L’ Attribut %TYPE

- Est utilisé pour déclarer une variable selon :
  - une définition de colonne de base de données
  - le type d’une variable déjà déclaré
- Est préfixé par:
  - le nom de la table et de la colonne
  - le nom de la variable déjà déclaré

L’ Attribut %TYPE

Syntaxe

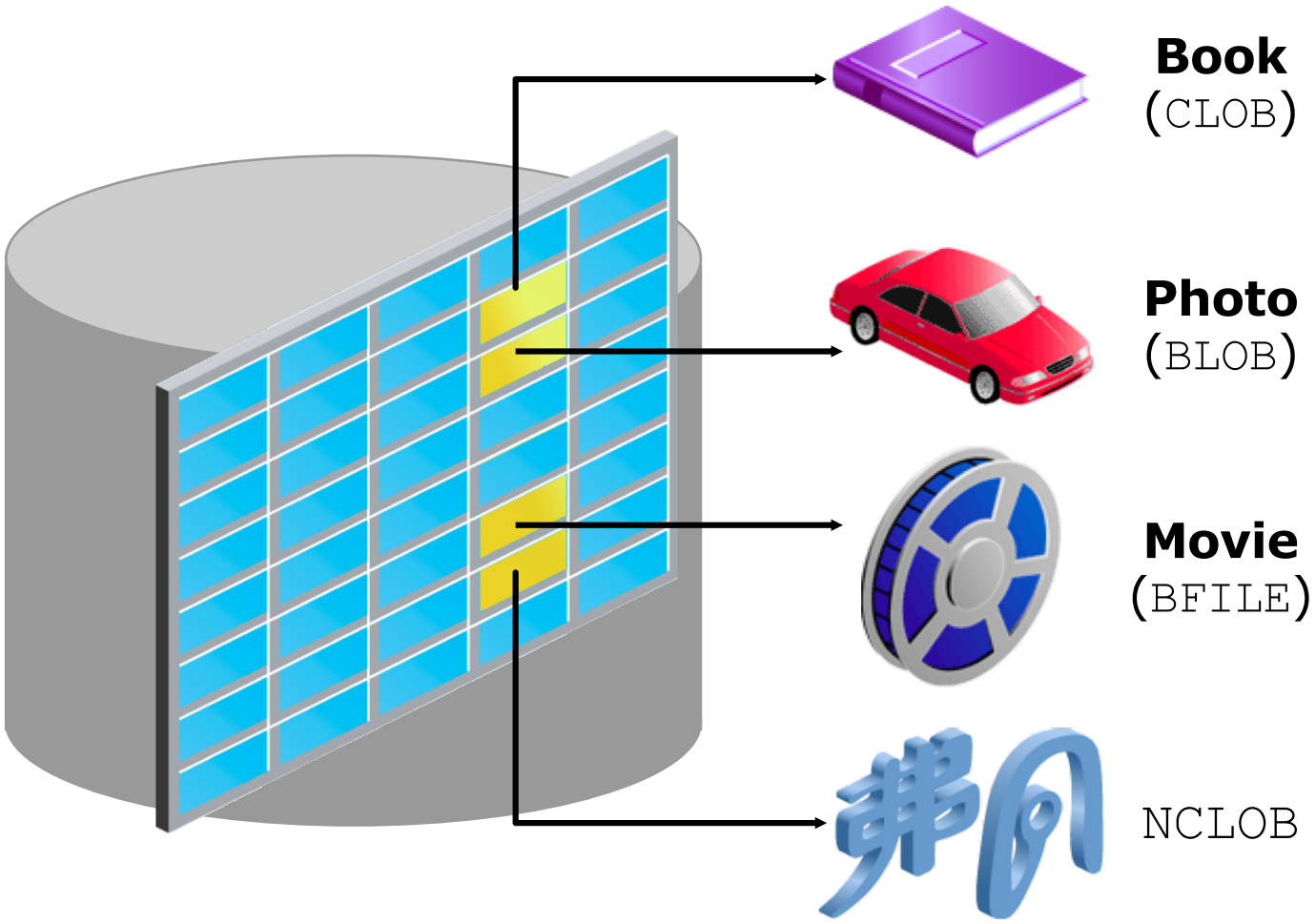
```
identifier      table.column_name%TYPE;
```

Exemples

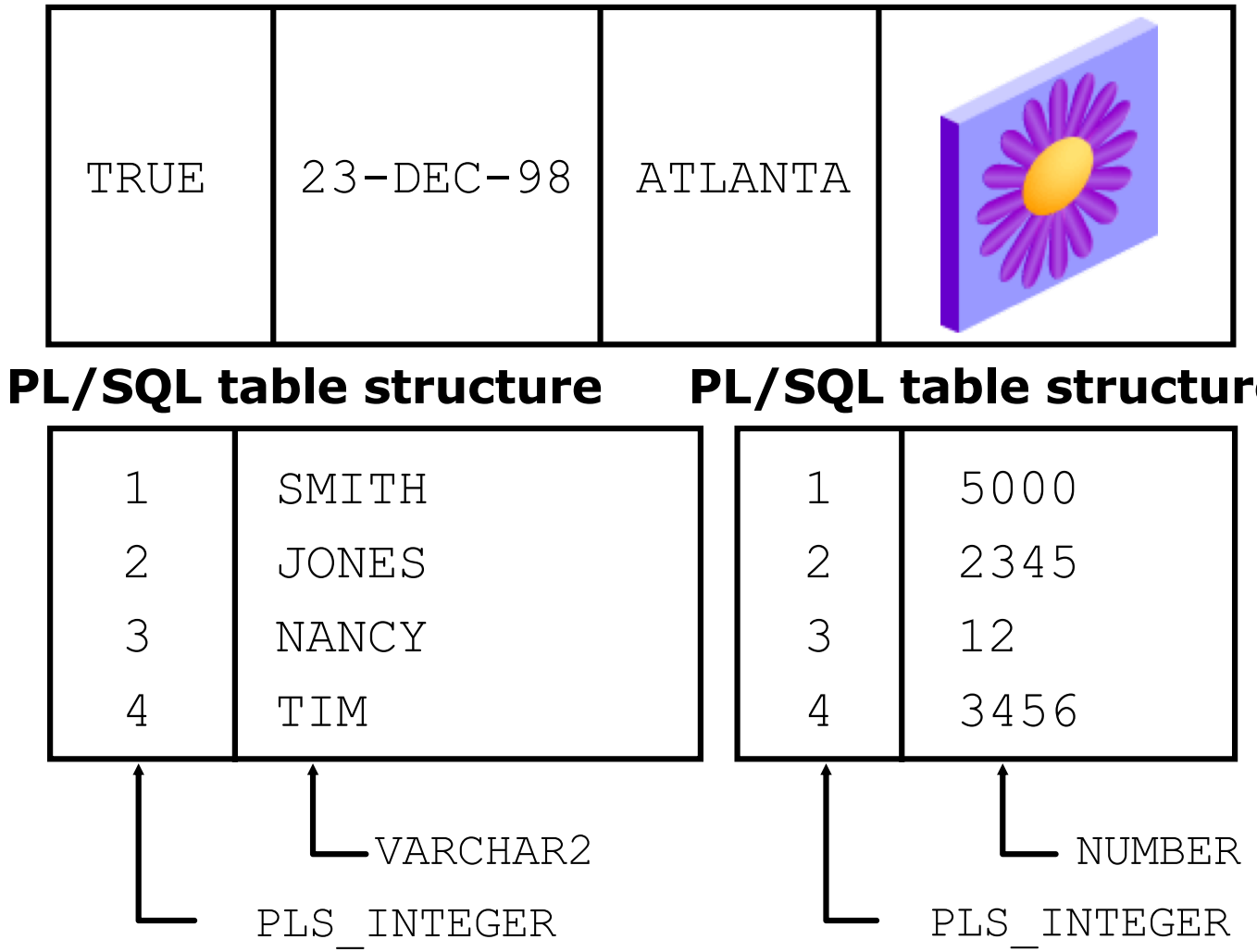
```
...
  emp_lname       employees.last_name%TYPE;
...
```

```
...
  balance         NUMBER(7,2);
  min_balance     balance%TYPE := 1000;
...
```

LOB Data Type Variables



Composite Data Types



Composite Data Types

- Peut contenir plusieurs valeurs
- Peut être de deux types:
  - PL/SQL records
  - PL/SQL collections
    - `INDEX BY` tables or associative arrays
    - Nested table
    - `VARRAY`

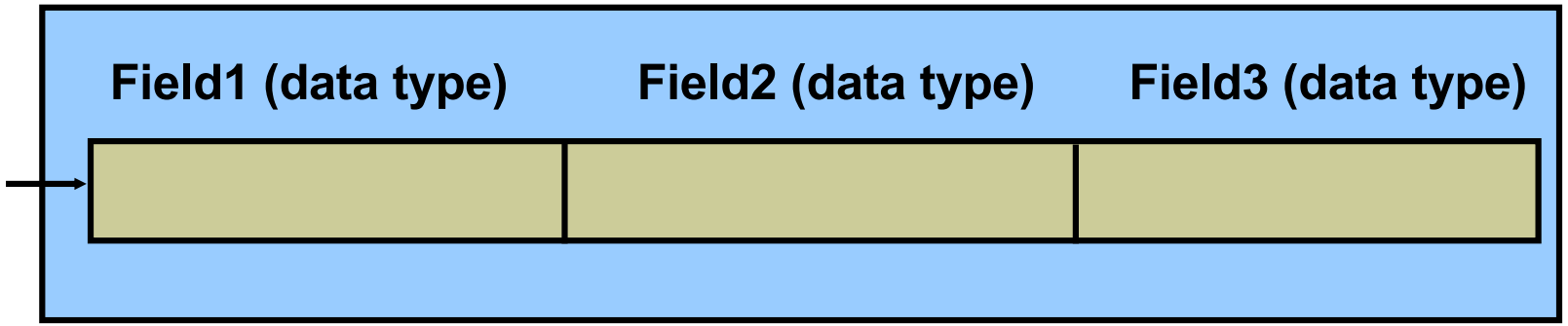
%ROWTYPE Attribute

- Déclare une variable selon une collection de colonnes dans une table de base de données ou une vue.
- Préfixer %ROWTYPE avec la table de base de données ou la vue.
- Les champs dans l'enregistrement prennent les noms et les types de données des colonnes de la table ou la vue.

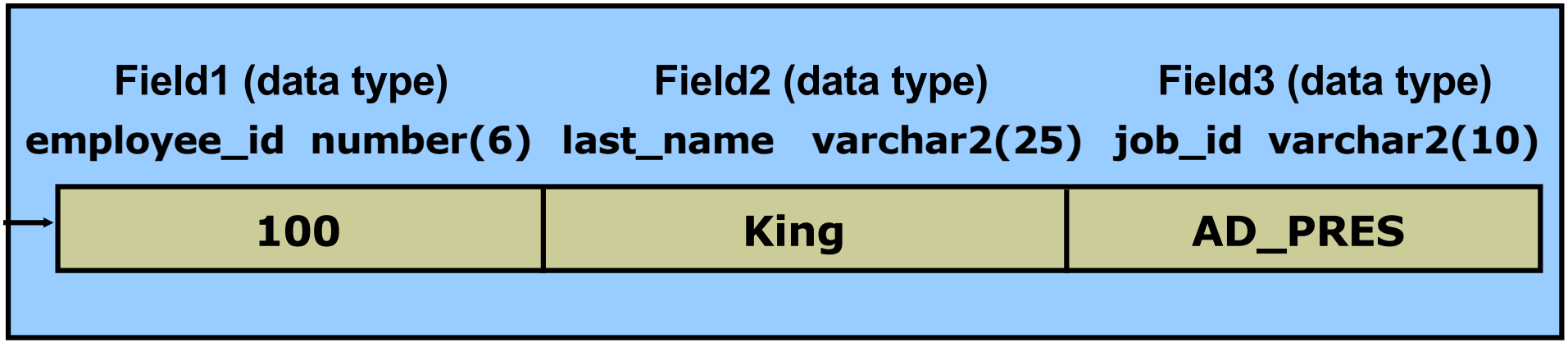
Syntaxe:

```
DECLARE
   identifier  reference%ROWTYPE;
```

PL/SQL Record Structure



Example:



Créer un PL/SQL Record

Syntaxe:

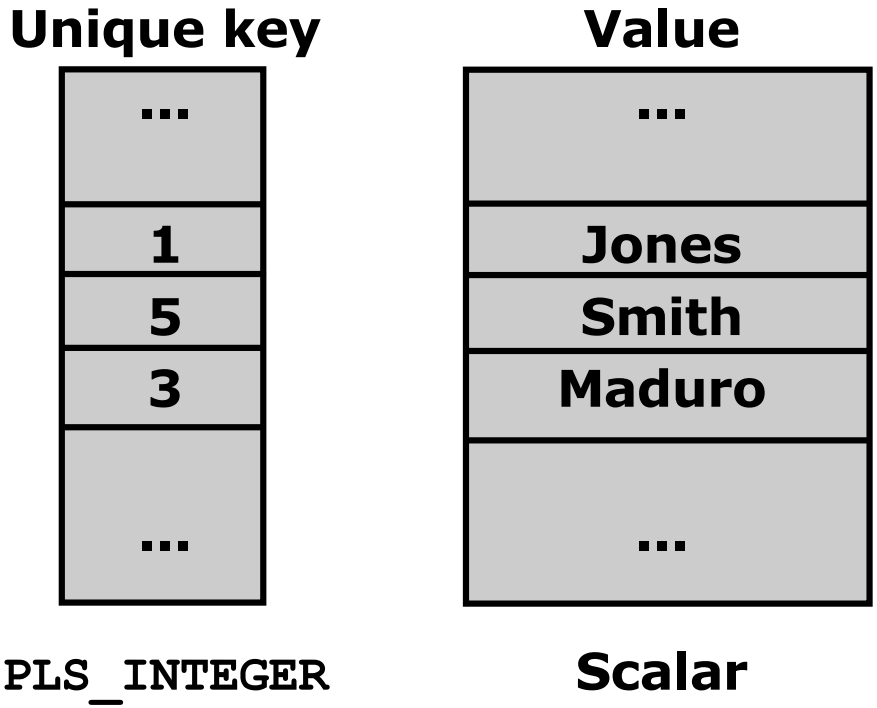
1
```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
```

2
```
identifier    type_name;
```

field_declaration:

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

INDEX BY Table Structure



Création d'un INDEX BY Table

```
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY PLS_INTEGER;
  ename_table        ename_table_type;
  hiredate_table     hiredate_table_type;
BEGIN
  ename_table(1)     := 'CAMERON';
  hiredate_table(8)  := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
    ...
END;
/
```

	ENAME	HIREDT
1	CAMERON	23-FEB-09

ECRIRE DES BLOCKS PL/SQL

Commenter le Code

- Préfixer les commentaires monolignes avec deux traits d'Union (--).
- Placer les commentaires de plusieurs lignes entre les symboles / * et * /.

Exemple:

```
DECLARE
...
v_annual_sal NUMBER (9,2);
BEGIN
/* Compute the annual salary based on the
   monthly salary input from the user */
v_annual_sal := monthly_sal * 12;
--The following line displays the annual salary
DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

Utiliser les Functions SQL dans le code PL/SQL: Examples

- Obtenir la longueur d'une chaîne :

```
v_desc_size INTEGER(5);
v_prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
v_desc_size:= LENGTH(v_prod_description);
```

- Obtenir le nombre de mois, que l'employé a travaillé :

```
v_tenure:= MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

La Conversion de type

- Convertit les données en types de données comparables
- Peut être de deux types:
  - Implicit conversion
  - Explicit conversion
- Des Fonctions:
  - `TO_CHAR`
  - `TO_DATE`
  - `TO_NUMBER`
  - `TO_TIMESTAMP`

Conversion de Type

1
```
date_of_joining DATE:= '02-Feb-2000';
```

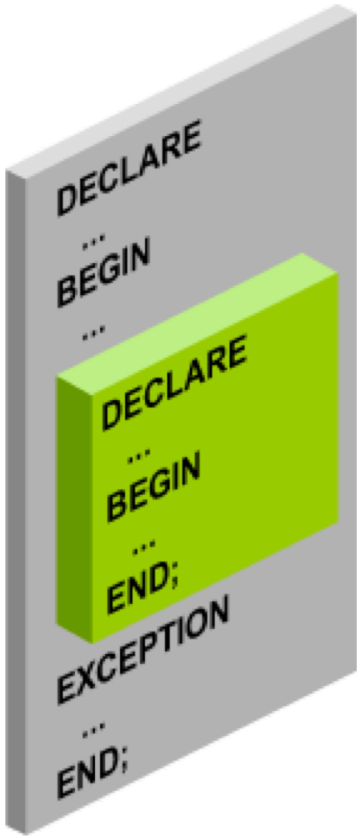
2
```
date_of_joining DATE:= 'February 02,2000';
```

3
```
date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```

Blocks Imbriqués

Les blocs PL/SQL peuvent être imbriqués.

- Une section exécutable (`BEGIN ... END`) peut contenir des blocs imbriqués.
- Une section d'exception peut contenir des blocs imbriqués.

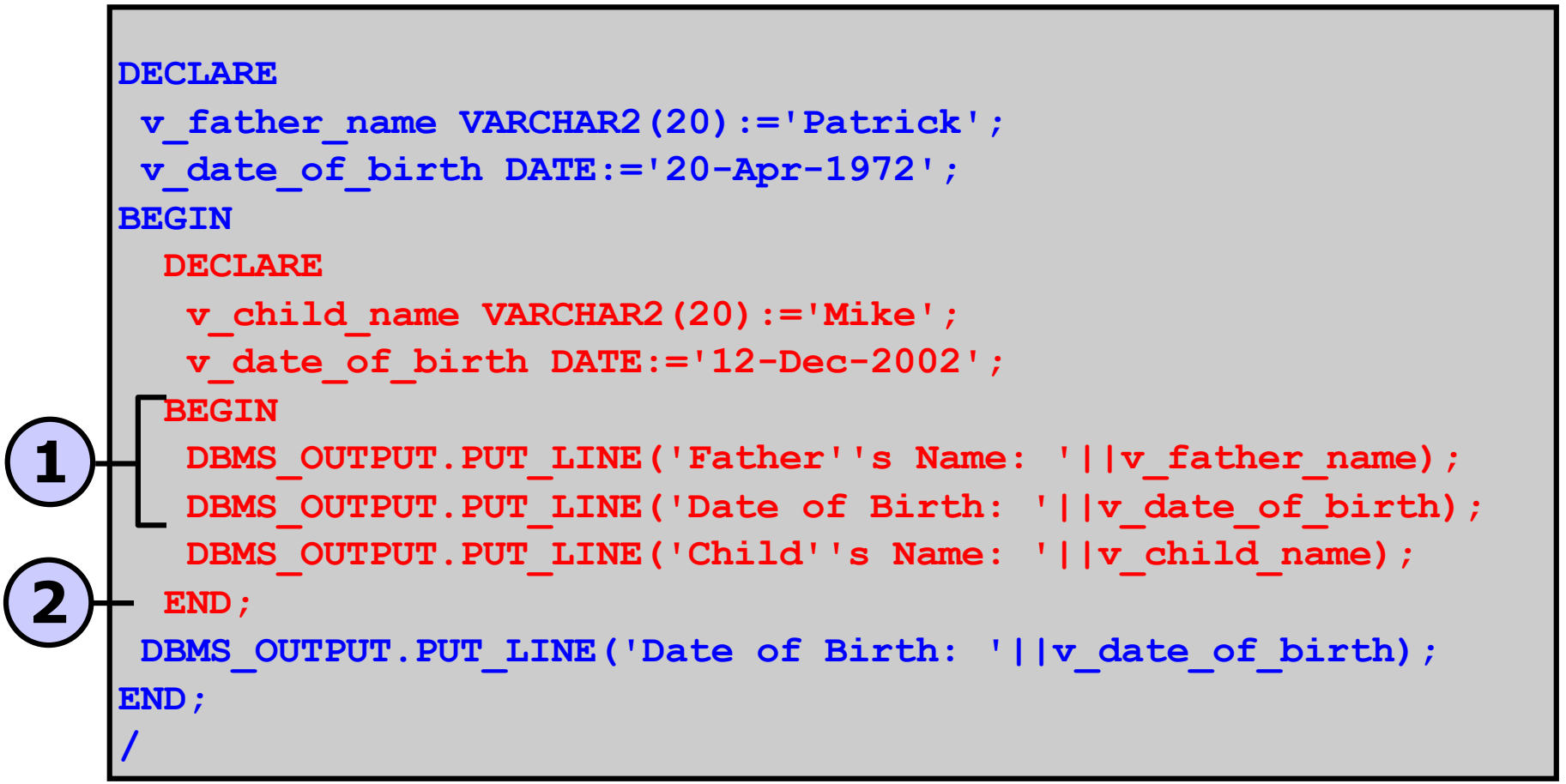


Nested Blocks: Example

```
DECLARE
 v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(v_inner_variable);
   DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

```
anonymous block completed
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE
```

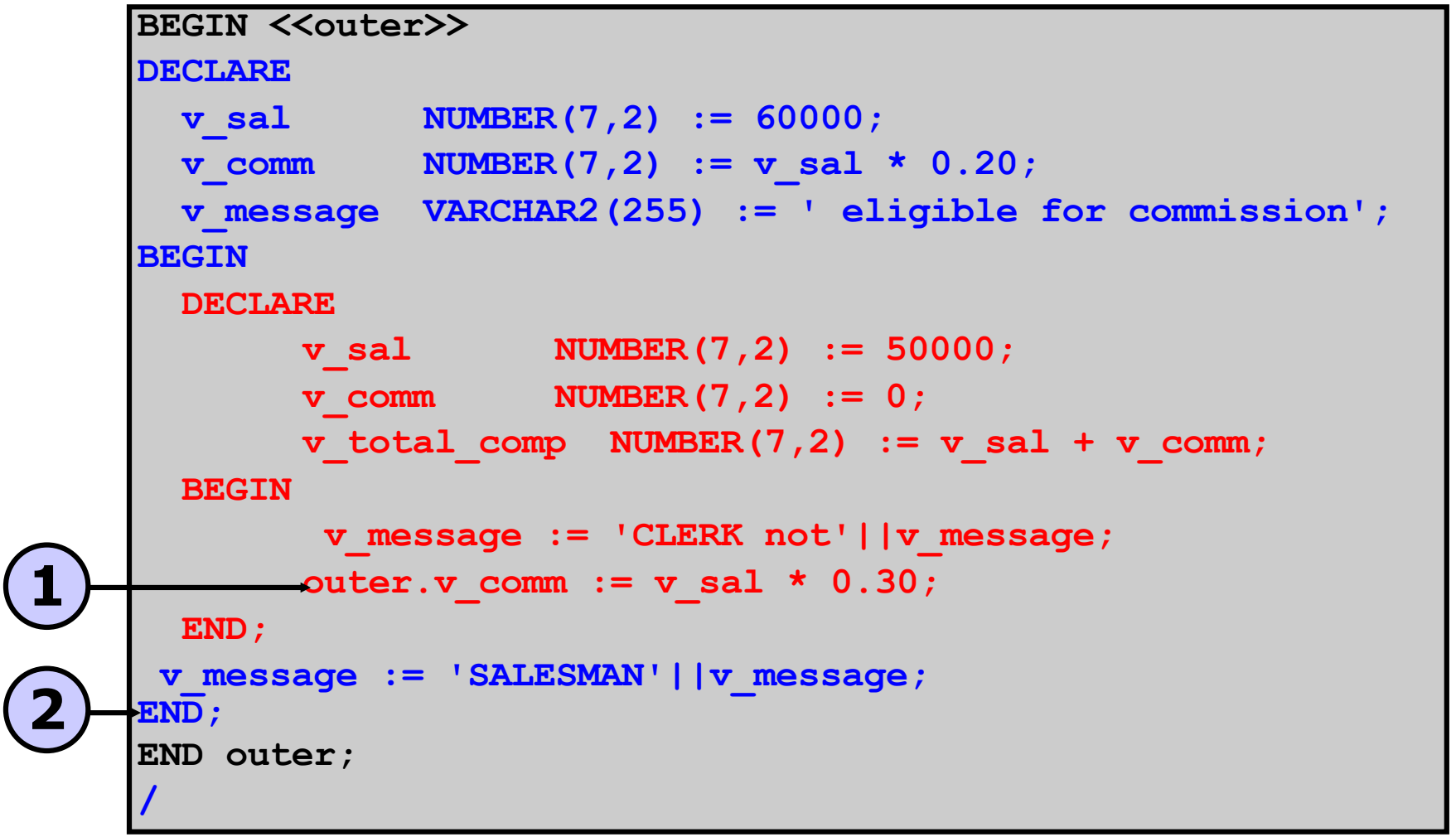
Visibilité et Porté des Variables



Visibilité et Porté des Variables

```
BEGIN <<outer>>
DECLARE
 v_father_name VARCHAR2(20):='Patrick';
 v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
   v_child_name VARCHAR2(20):='Mike';
   v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                        ||outer.v_date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
  END;
END;
END outer;
```

Visibilité et Porté des Variables: Exemple



Exercice

```
 DECLARE
 v_weight     NUMBER(3) := 600;
 v_message    VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   v_weight  NUMBER(3) := 1;
   v_message VARCHAR2(255) := 'Product 11001';
   v_new_locn VARCHAR2(50) := 'Europe';
  BEGIN
   v_weight := v_weight + 1;
   v_new_locn := 'Western '||v_new_locn;
1 ------>
  END;
 v_weight := v_weight + 1;
 v_message := v_message || ' is in stock';
 v_new_locn := 'Western ' || v_new_locn;
2 ------>
END;
/
```

v_weight at position 1 =?

v_new_locn at position 1 = ?

v_weight at position 2 =?

v_message at position 2 =?

v_new_locn at position 2 =?

Interaction avec le serveur de base de données Oracle

SELECT dans PL/SQL

Récupérer des données de la base de données avec une instruction SELECT.

Syntaxe :

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

SELECT dans PL/SQL

- La clause INTO est requise.
- Les requêtes ne doivent retourner qu'une seule ligne.

```
DECLARE
 v_fname VARCHAR2(25);
BEGIN
 SELECT first_name INTO v_fname
 FROM employees WHERE employee_id=200;
 DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
END;
/
```

```
anonymous block completed
 First Name is : Jennifer
```

Récupération de données par PL/SQL: Exemple

Recupérer `hire_date` et `salary` pour un employé.

```
DECLARE
 v_emp_hiredate   employees.hire_date%TYPE;
 v_emp_salary     employees.salary%TYPE;
BEGIN
  SELECT   hire_date, salary
  INTO     v_emp_hiredate, v_emp_salary
  FROM     employees
  WHERE    employee_id = 100;
END;
/
```

Récupération de données par PL/SQL

Retourne la somme des salaires pour tous les employés d'un département spécifié.

Exemple:

```
DECLARE
   v_sum_sal    NUMBER(10,2);
   v_deptno     NUMBER NOT NULL := 60;
BEGIN
  SELECT  SUM(salary)  -- group function
  INTO v_sum_sal  FROM employees
  WHERE     department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE ('The sum of salary is ' || v_sum_sal);
END;
```

```
anonymous block completed
The sum of salary is 28800
```

Récupérer des données : Exemple

Déclarer des variables pour stocker le nom, l'emploi et le salaire d'un nouvel employé.

```
DECLARE
  type t_rec is record
    (v_sal number(8),
     v_minsal number(8) default 1000,
     v_hire_date employees.hire_date%type,
     v_rec1 employees%rowtype);
  v_myrec t_rec;
BEGIN
  v_myrec.v_sal := v_myrec.v_minsal + 500;
  v_myrec.v_hire_date := sysdate;
  SELECT * INTO v_myrec.v_rec1
      FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name ||' '||
  to_char(v_myrec.v_hire_date) ||' '|| to_char(v_myrec.v_sal));
END;
```

```
anonymous block completed
King 16-FEB-09 1500
```

Récupérer des données pour un %ROWTYPE Attribute: Exemple

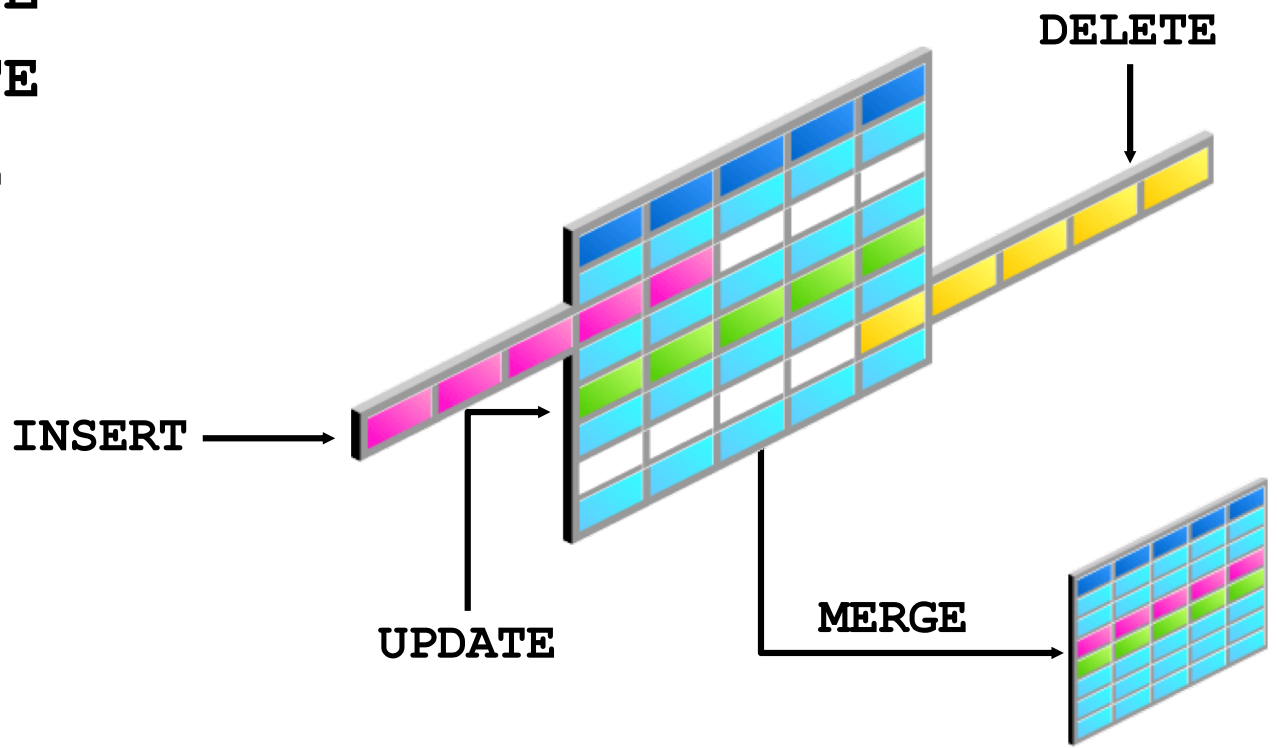
```
DECLARE
  v_employee_number number:= 124;
  v_emp_rec   employees%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps(empno, ename, job, mgr,
                  hiredate, leavedate, sal, comm, deptno)
   VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
           v_emp_rec.job_id, v_emp_rec.manager_id,
           v_emp_rec.hire_date, SYSDATE,
           v_emp_rec.salary,  v_emp_rec.commission_pct,
           v_emp_rec.department_id);
END;
/
```

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-FEB-09	5800	(null)	50

Utiliser PL/SQL pour Manipuler des Données

Apporter des modifications aux tables de base de données à l'aide de commandes DML :

- `INSERT`
- `UPDATE`
- `DELETE`
- `MERGE`



Inserer des données : Exemple

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',CURRENT_DATE, 'AD_ASST', 4000);
END;
/
```

Insérer un enregistrement en utilisant %ROWTYPE

```
...
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT employee_id, last_name, job_id, manager_id,
 hire_date, hire_date, salary, commission_pct,
 department_id INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
SELECT * FROM retired_emps;
```

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

Mettre à jour des données: Exemple

```
DECLARE
  sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE       employees
  SET          salary = salary + sal_increase
  WHERE        job_id = 'ST_CLERK';
END;
/
```

```
anonymous block completed
FIRST_NAME           SALARY
-------------------- ----------------------
Julia                4000
Irene                3500
James                3200
Steven               3000
```

...

```
Curtis               3900
Randall              3400
Peter                3300

20 rows selected
```

Updating a Row in a Table by Using a Record

```
SET VERIFY OFF
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM retired_emps
 where empno = v_employee_number;
 v_emp_rec.leavedate:=CURRENT_DATE;
 UPDATE retired_emps SET ROW = v_emp_rec WHERE
  empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```

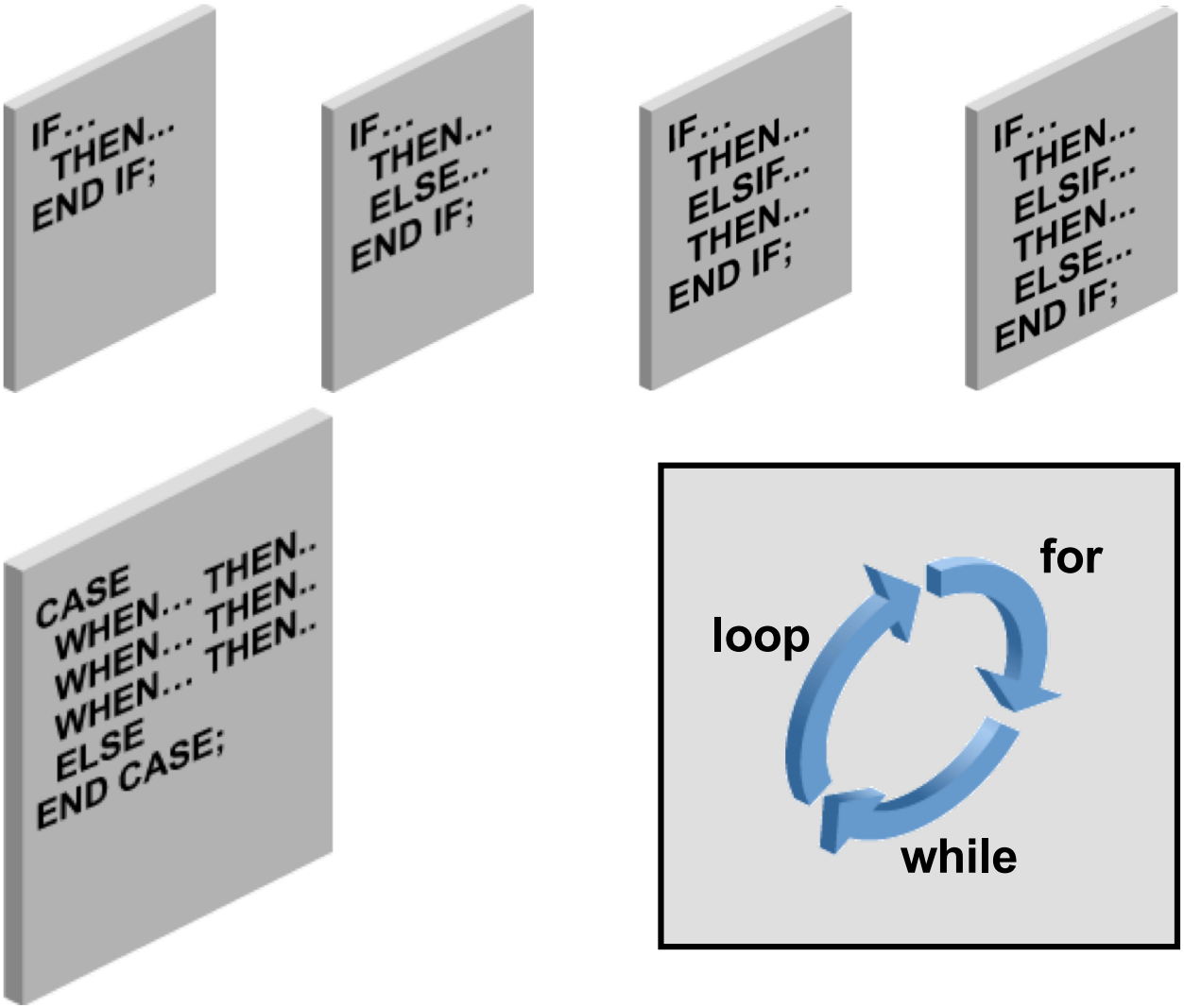
	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-FEB-09	5800	(null)	50

Supprimer des données: Exemple

```
DECLARE
  deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   employees
  WHERE  department_id = deptno;
END;
/
```

STRUCTURES DE CONTRÔLE

Structures de contrôle et boucle



IF Statement

Syntax:

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

Simple IF Statement

```
DECLARE
  v_myage  number:=31;
BEGIN
  IF v_myage  < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;
END;
/
```

```
anonymous block completed
```

IF THEN ELSE Statement

```
DECLARE
v_myage  number:=31;
BEGIN
IF v_myage  < 11
 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
 ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```

```
anonymous block completed
 I am not a child
```

IF ELSIF ELSE Clause

```
DECLARE
  v_myage number:=31;
BEGIN
  IF v_myage  < 11 THEN
       DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSIF v_myage  < 20 THEN
       DBMS_OUTPUT.PUT_LINE(' I am young ');
    ELSIF v_myage  < 30 THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
    ELSIF v_myage  < 40 THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
    ELSE
       DBMS_OUTPUT.PUT_LINE(' I am always young ');
  END IF;
END;
/
```

```
anonymous block completed
 I am in my thirties
```

NULL Value in IF Statement

```
DECLARE
  v_myage  number;
BEGIN
  IF v_myage  < 11 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
/
```

```
anonymous block completed
 I am not a child
```

CASE Expressions

- Une expression CASE sélectionne un résultat et le retourne.
- Pour sélectionner le résultat, l'expression CASE utilise des expressions. La valeur retournée par ces expressions est utilisée pour sélectionner une ou plusieurs alternatives.

```
CASE selector
   WHEN expression1 THEN result1
   WHEN expression2 THEN result2
   ...
   WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
/
```

CASE Expressions: Example

```
SET VERIFY OFF
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   appraisal VARCHAR2(20);
BEGIN
   appraisal := CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade  || '
                       Appraisal ' || appraisal);
END;
/
```

Searched CASE Expressions

```
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   appraisal VARCHAR2(20);
BEGIN
    appraisal := CASE
         WHEN v_grade  = 'A' THEN 'Excellent'
         WHEN v_grade  IN ('B','C') THEN 'Good'
         ELSE 'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade  || '
                  Appraisal ' || appraisal);
END;
/
```

CASE Statement

```
DECLARE
   v_deptid NUMBER;
   v_deptname VARCHAR2(20);
   v_emps NUMBER;
   v_mngid NUMBER:= 108;
BEGIN
  CASE  v_mngid
   WHEN  108 THEN
    SELECT department_id, department_name
     INTO v_deptid, v_deptname FROM departments
     WHERE manager_id=108;
    SELECT count(*) INTO v_emps FROM employees
     WHERE department_id=v_deptid;
   WHEN  200 THEN
     ...
 END CASE;
DBMS_OUTPUT.PUT_LINE ('You are working in the '|| deptname||
' department. There are '||v_emps ||' employees in this
department');
END;
/
```

Exercice 1:

- **Ecrire un programme permettant**
  - **de saisir en entrée le nom d 'un employé**
  - **de mettre à jour le salaire de cet employé en lui ajoutant :**
    - **500 $ s 'il appartient au département vente**
    - **300 $ s 'il appartient au département opérateurs**
    - **400 $ s 'il appartient au département SI**
    - **600 $ s 'il appartient au département Administrateurs**

NB: Faites l'exercice avec

- **IF ..END IF**
- **CASE Expression**
- **CASE Statement**

Exercice 2:

Écrire un programme permettant

de saisir en entrée le nom d ’un employé

de mettre à jour le salaire d ’un employé suivant les conditions suivantes:

Si son année d ’entrée est 1990, augmenter son salaire de 50%

Si son année d ’entrée est 1991, augmenter son salaire de 25%

Si son année d ’entrée est 1992, augmenter son salaire de 10%

Traitements itératifs : LOOP Statements

- Une boucle répète une instruction (ou la séquence d'instructions) plusieurs fois.
- Trois types de boucle:
  - Basic loop
  - FOR loop
  - WHILE loop

Boucle de base

Syntaxe:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

Boucle de base qui permet la répétition d'une séquence d'instructions.

Basic Loops

Example:

```
DECLARE
  v_countryid    locations.country_id%TYPE := 'CA';
  v_loc_id       locations.location_id%TYPE;
  v_counter      NUMBER(2) := 1;
  v_new_city     locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
  END LOOP;
END;
/
```

La boucle WHILE

Syntaxe:

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

La boucle `WHILE` répète les instructions tant que condition est `TRUE`.

WHILE Loops: Example

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
  v_counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

La boucle FOR

- Le nombre d'itérations est connu avant d'entrer dans la boucle.
- Ne pas déclarer le compteur ; Il est déclaré implicitement.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

La boucle FOR : Exemple

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id
    FROM locations
    WHERE country_id = v_countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + i), v_new_city, v_countryid );
  END LOOP;
END;
/
```

Exercice 3

2.. Create a PL/SQL block that inserts an asterisk in the `stars` column for every $1,000 of the employee's salary.

a. In the declarative section of the block,
   - declare a variable `v_empno` of type `emp.employee_id` and initialize it to 176.
   - declare a variable `v_asterisk` of type `emp.stars` and initialize it to `NULL`.
   - create a variable `sal` of type `emp.salary`.

b. In the executable section, write logic to append an asterisk (*) to the string for every $1,000 of the salary amount. For example, if the employee earns $8,000, the string of asterisks should contain eight asterisks. If the employee earns $12,500, the string of asterisks should contain 13 asterisks.

c. Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

Exercice 4

Écrire un programme permettant la mise à jour des salaires de tous les employés suivant les conditions de l ’exercice2.

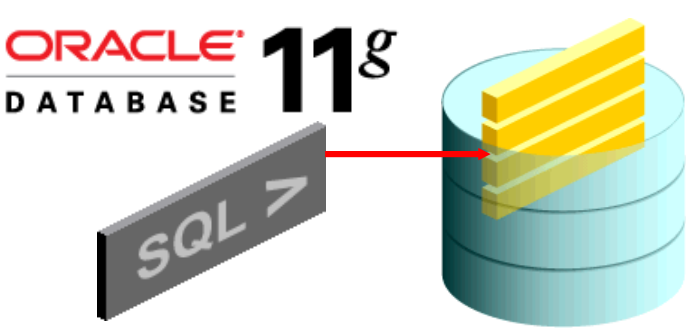
Remarques ..

USING EXPLICIT CURSORS

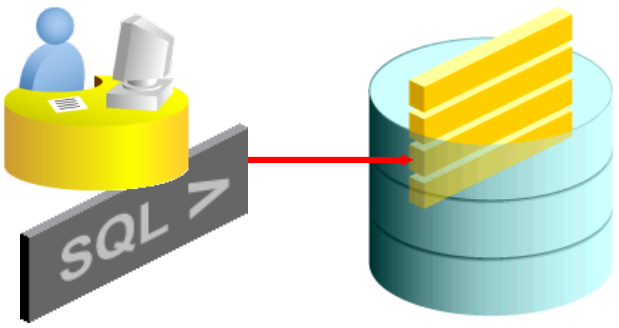
Cursors

Chaque instruction SQL exécutée par le serveur Oracle a un curseur individuel associé qui est soit un:

- **curseur implicite : Déclaré et géré par le PL/SQL pour toutes les instructions DML et PL/SQL SELECT**
- **curseur explicite : déclaré et géré par le programmeur**

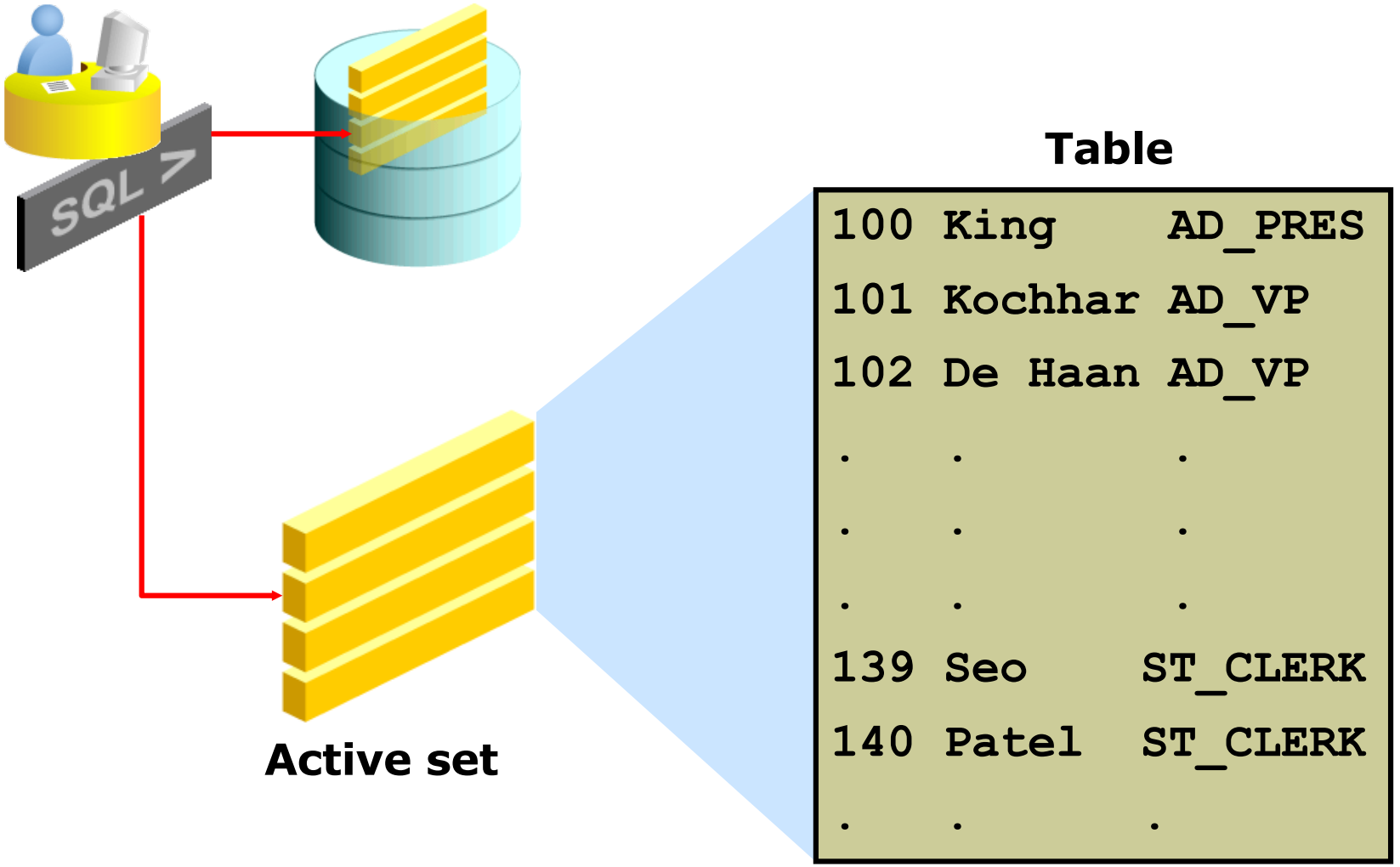


Implicit cursor

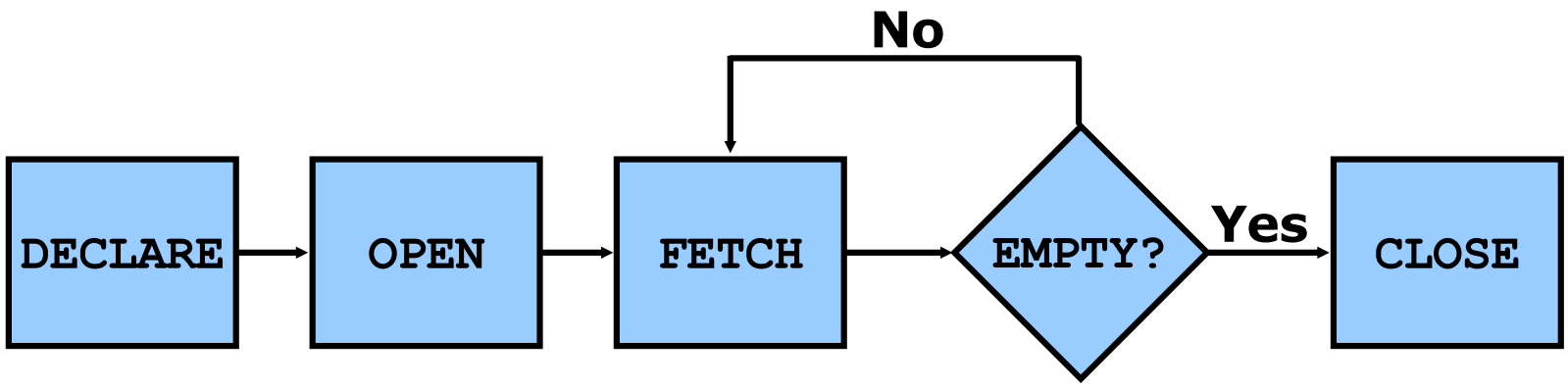


Explicit cursor

Explicit Cursor Operations



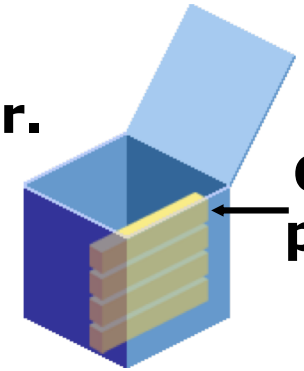
Controlling Explicit Cursors



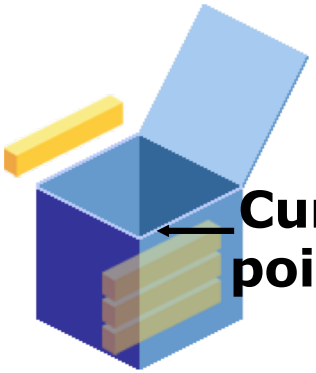
- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to `FETCH` if rows are found.
- Release the active set.

Controlling Explicit Cursors

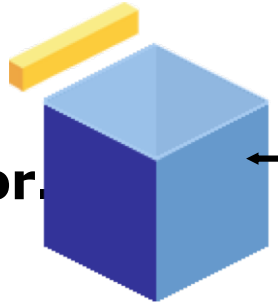
1 Open the cursor.



2 Fetch a row.



3 Close the cursor.



Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples:

```
DECLARE
  CURSOR c_emp_cursor IS
  SELECT employee_id, last_name FROM employees
  WHERE department_id =30;
```

```
DECLARE
  v_locid NUMBER:= 1700;
  CURSOR c_dept_cursor IS
  SELECT * FROM departments
  WHERE location_id = v_locid;
...
```

Opening the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
...
BEGIN
  OPEN c_emp_cursor;
```

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  v_empno employees.employee_id%TYPE;
  v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  FETCH c_emp_cursor INTO v_empno, v_lname;
  DBMS_OUTPUT.PUT_LINE( v_empno ||'   '||v_lname);
END;
/
```

```
anonymous block completed
114  Raphaely
```

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  v_empno employees.employee_id%TYPE;
  v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_empno, v_lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno ||'  '||v_lname);
  END LOOP;
END;
/
```

Closing the Cursor

```
...
  LOOP
    FETCH c_emp_cursor INTO empno, lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno ||' '||v_lname);
  END LOOP;
 CLOSE c_emp_cursor;
END;
/
```

Cursors and Records

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

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE  department_id =30;
   v_emp_record   c_emp_cursor%ROWTYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_emp_record;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                          ||' '||v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

Cursor FOR Loops

Syntax:

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

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

Cursor FOR Loops

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
BEGIN
   FOR emp_record IN c_emp_cursor
    LOOP
     DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
     ||' ' ||emp_record.last_name);
    END LOOP;
END;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

Explicit Cursor Attributes

Use explicit cursor attributes to 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

%ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

Example:

```
IF NOT c_emp_cursor%ISOPEN THEN
    OPEN c_emp_cursor;
END IF;
LOOP
  FETCH c_emp_cursor...
```

%ROWCOUNT and %NOTFOUND: Example

```
DECLARE
  CURSOR c_emp_cursor IS SELECT employee_id,
    last_name FROM employees;
  v_emp_record   c_emp_cursor%ROWTYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
   FETCH c_emp_cursor INTO v_emp_record;
   EXIT WHEN c_emp_cursor%ROWCOUNT > 10 OR
                        c_emp_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                ||' '||v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END ; /
```

```
anonymous block completed
198 OConnell
199 Grant
200 Whalen
201 Hartstein
202 Fay
203 Mavris
204 Baer
205 Higgins
206 Gietz
100 King
```

Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

```
BEGIN
  FOR emp_record IN (SELECT employee_id, last_name
   FROM employees WHERE department_id =30)
  LOOP
   DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
   ||' '||emp_record.last_name);
  END LOOP;
END;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

Cursors with Parameters

Syntax:

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

- **Passer des valeurs de paramètre à un curseur lorsque le curseur est ouvert et que la requête est exécutée.**
- **Ouvrir un curseur explicit plusieurs fois avec un actif différent à chaque fois.**

```
OPEN  cursor_name(parameter_value,.....) ;
```

Cursors with Parameters

```
DECLARE
  CURSOR   c_emp_cursor (deptno NUMBER) IS
   SELECT  employee_id, last_name
   FROM    employees
   WHERE   department_id = deptno;
   ...
BEGIN
  OPEN c_emp_cursor (10);
  ...
  CLOSE c_emp_cursor;
  OPEN c_emp_cursor (20);
  ...
```

```
anonymous block completed
200 Whalen
201 Hartstein
202 Fay
```

FOR UPDATE Clause

Syntax:

```
SELECT ...
FROM           ...
FOR UPDATE [OF column_reference][NOWAIT | WAIT n];
```

- Utilise le verrouillage explicite pour refuser l'accès aux autres sessions pendant toute la durée d'une transaction.
- Verrouille les lignes avant la mise à jour ou la suppression.

WHERE CURRENT OF Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

- Utilise le curseur pour mettre à jour ou supprimer la ligne actuelle.
- Inclure la clause FOR UPDATE dans la requête de curseur pour verrouiller les lignes d'abord.
- Utiliser la clause WHERE CURRENT OF pour faire référence à la ligne courante d'un curseur explicit.

```
UPDATE employees
   SET    salary = ...
   WHERE CURRENT OF c_emp_cursor;
```

Cursors with Subqueries: Example

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.department_id, t1.department_name,
           t2.staff
    FROM   departments t1, (SELECT department_id,
                                   COUNT(*) AS staff
                            FROM employees
                            GROUP BY department_id) t2
    WHERE t1.department_id = t2.department_id
    AND   t2.staff >= 3;
...
```

Exercice

- **Ecrire un programme permettant**
  - **de mettre à jour le salaire de tous les employés en leurs ajoutant :**
    - **500 $ si l'employé appartient au département vente**
    - **300 $ s 'il appartient au département opérateurs**
    - **400 $ s 'il appartient au département SI**
    - **600 $ s 'il appartient au département Administrateurs**

NB: utiliser les curseurs explicites et CASE Expression

Exercices 1

1. Créez un bloc PL/SQL qui effectue les opérations suivantes:
   a. dans la section déclarative, déclarez une variable v_deptno de type numérique et assigner une valeur qui contient l'ID du département.
   b. Déclarer un curseur, c_emp_cursor, qui extrait le last_name, salaire et manager_id des employés travaillant dans le département spécifié dans v_deptno.
   c. dans la section exécutable, utilisez le curseur FOR loop pour opérer sur les données récupérées.
      i. Si le salaire de l'employé est inférieur à 5 000, et si l'ID du manager est 101 ou 124, afficher le message de `<<last_name>> Due for a raise.`
      ii. Dans le cas contraire, afficher le message `<<last_name>> Not due for a raise.`

Exercice 2

a. Dans la section déclarative, déclarez un curseur dept_cursor pour récupérer departement_id et department_name avec departement_id inférieur à 100, ordonnées par departement_id.

b. Déclarer un autre curseur, emp_cursor, qui prend le numéro de département en tant que paramètre et récupère last_name, job_id, hire_date et le salaire des employés dont employee_id est inférieure à 120 et qui travaillent dans ce département.

c. Déclarer des variables pour contenir des valeurs extraites de chaque curseur.

d. Ouvrir dept_cursor, utilisez une boucle simple et extraire les valeurs dans les variables déclarées. Afficher le numéro et le nom du département .

e. Pour chaque département, ouvrez emp_cursor en passant le numéro actuel du département en tant que paramètre. Commencer une autre boucle et extraire les valeurs d'emp_cursor dans des variables et imprimer tous les détails qui provient de la table employees . *Remarque* : Vous pouvez imprimer une ligne après avoir affiché les détails de chaque département. Utilisez des attributs appropriés pour la condition de sortie. En outre, déterminer si un curseur est déjà ouvert avant de l'ouvrir.

a. Fermez tous les curseurs et les boucles et puis terminer la section exécutable.

Exercice 3

Créez un bloc PL/SQL qui détermine les *n* employés les plus rémunérés (n pus grands salaires) et les insère dans une table `top_salaries` .

a. Dans la section déclarative, déclarez une variable *v_num* de type numérique qui contient un nombre n représentant le nombre de salariés de n haut salaires de la table employees. Par exemple, pour afficher les cinq plus grands salaires, entrez 5. Déclarer une autre variable *sal* de type employees.salary. Déclarer un curseur, c_emp_cursor, qui récupère les salaires des employés dans l'ordre décroissant.

b. Dans la section exécutable, ouvrir le curseur et parcourir les n premières lignes du curseur et les insérer dans la table `top_salaries`. Vous utiliserez une boucle simple pour utiliser les attributs `%ROWCOUNT` et `%FOUND` pour vos conditions.

Note: Make sure you add an exit condition to avoid having an infinite loop.

Cursor Variables

- Les variables de curseur agissent comme les pointeurs C. Ils détiennent l'emplacement de la mémoire (adresse) d'un élément au lieu de la valeur de l'élément lui-même.
- Dans PL/SQL, un pointeur est déclaré comme `REF X`, où `REF` est une abréviation de `REFERENCE` et `X` représente une classe d'objets.
- Une variable de curseur dispose du type `REF CURSOR`.
- Un `CURSOR` est statique alors qu'un `REF CURSOR` est dynamique.

Using Cursor Variables

- Vous pouvez utiliser des Ref Cursor pour passer des jeux de résultats de requête entre les blocks PL/SQL.
- PL/SQL peut partager un pointeur vers la zone de travail de requête dans lequel est stocké le jeu de résultats
- Vous pouvez passer la valeur d'une variable de curseur librement d’un champ d'application à un autre.

Defining REF CURSOR Types

Define a REF CURSOR type:

```
Define a REF CURSOR type
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

Example:

```
DECLARE
TYPE DeptCurTyp IS REF CURSOR RETURN
departments%ROWTYPE;
dept_cv DeptCurTyp;
```

Using the OPEN-FOR, FETCH, and CLOSE Statements

- L'instruction OPEN-FOR, associe un curseur à une requête multi ligne, exécute la requête, identifie le jeu de résultats et positionne le curseur vers la première ligne du jeu de résultats.
- L'instruction FETCH retourne une ligne du résultat de la requête multi lignes, assigne les valeurs des éléments de la liste de sélection à des variables ou des champs dans la clause INTO, incrémente le décompte tenu par % ROWCOUNT et avance le curseur à la ligne suivante.
- L'instruction CLOSE désactive une variable de curseur.

Opening REF CURSOR

```
OPEN NOM_CURSEUR FOR REQUETE
 [ USING [ IN | OUT | IN OUT ] ARGUMENT[,... ] ] ;
```

USING: Cette clause permet le paramétrage de la requête SQL dynamique en utilisant une liste des arguments.

IN ARGUMENT : L'argument est passé à la requête SQL dynamique lors de son invocation. Il ne peut pas être modifié à l'intérieur de la requête SQL dynamique.

OUT ARGUMENT : L'argument est ignoré lors de l'invocation de la requête SQL dynamique. À l'intérieur de celle-ci, l'argument se comporte comme une variable PL/SQL n'ayant pas été initialisée, contenant donc la valeur « NULL » et supportant les opérations de lecture et d'écriture. Au terme de la requête SQL dynamique, il retourne à la valeur affectée.

IN OUT ARGUMENT: L'argument combine les deux propriétés « IN» et « OUT ».

Example of Fetching

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv   EmpCurTyp;
   emp_rec  employees%ROWTYPE;
   sql_stmt VARCHAR2(200);
   my_job   VARCHAR2(10) := 'ST_CLERK';
BEGIN
   sql_stmt := 'SELECT * FROM employees
                WHERE job_id = :j';
   OPEN emp_cv FOR sql_stmt USING my_job;
   LOOP
      FETCH emp_cv INTO emp_rec;
      EXIT WHEN emp_cv%NOTFOUND;
      -- process record
   END LOOP;
   CLOSE emp_cv;
END;
/
```

Exercice

Créez le bloc PL/SQL qui permet d'afficher toute les lignes de l'une des tables :

– EMPLOYES_1996,

– EMPLOYES_1997,

– EMPLOYES_1998, …

- EMPLOYES_YYYY

Dynamiquement, suivant l'année passée en argument,

- **vous testez que la table existe et vous affichez tous les enregistrements de la table.**
- **Si la table n'existe pas, vous affichez tous les enregistrements de la table EMPLOYES pour l'année qui a été passée en argument.**

HANDLING EXCEPTIONS

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
```

Example of an Exception

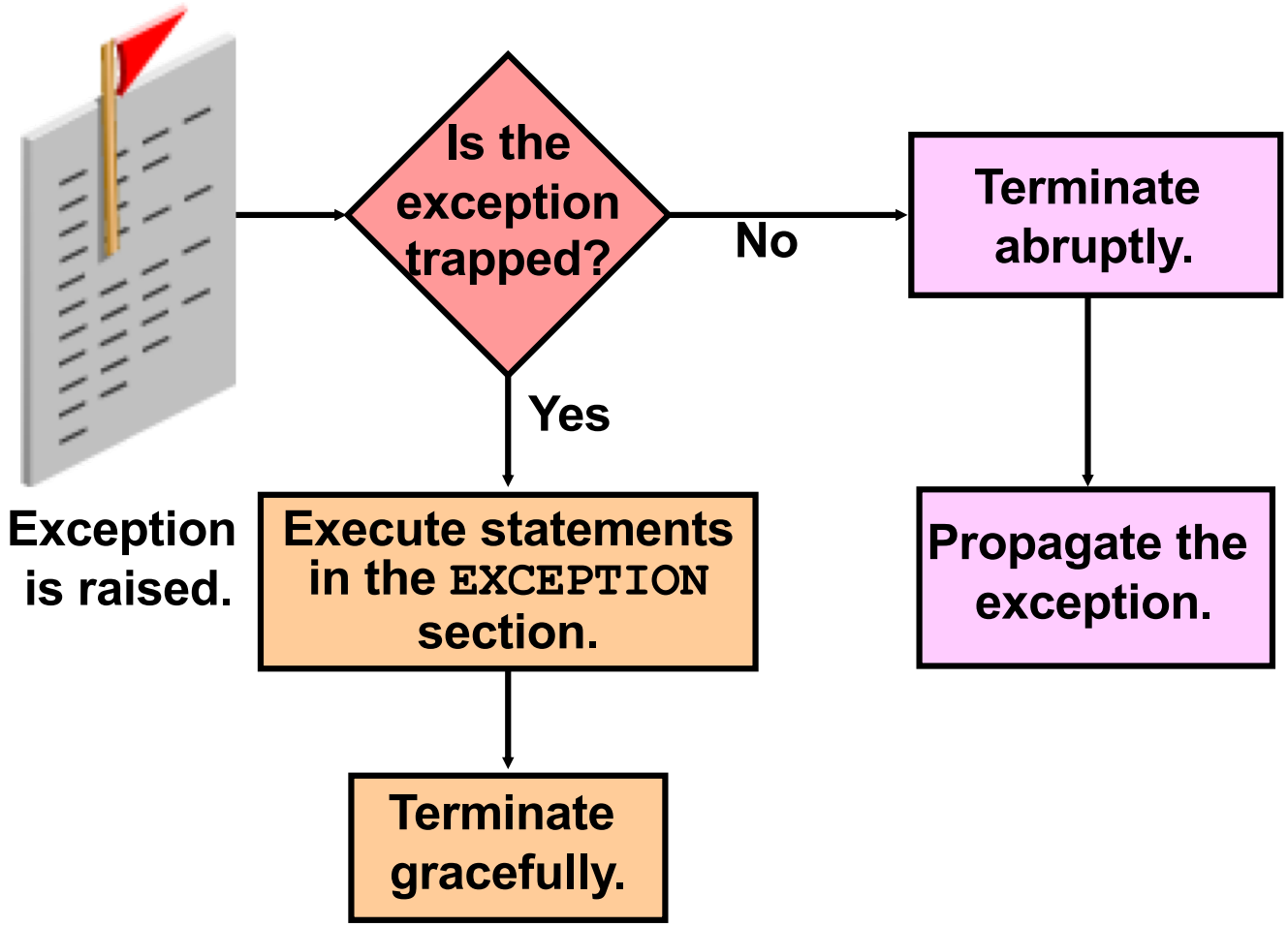
```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees
  WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is :'
                        ||v_lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement
  retrieved multiple rows. Consider using a
  cursor.');
END;
/
```

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

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
- Une exception peut être traitée :
  - par une capture et un traitment par le bloc courant
  - en la propageant à l'environnement appelant

Handling Exceptions



Exception Types

- Predefined Oracle server
- Non-predefined Oracle server

} Implicitly raised

Explicitly raised

- User-defined

Trapping Exceptions

Syntax:

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

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.

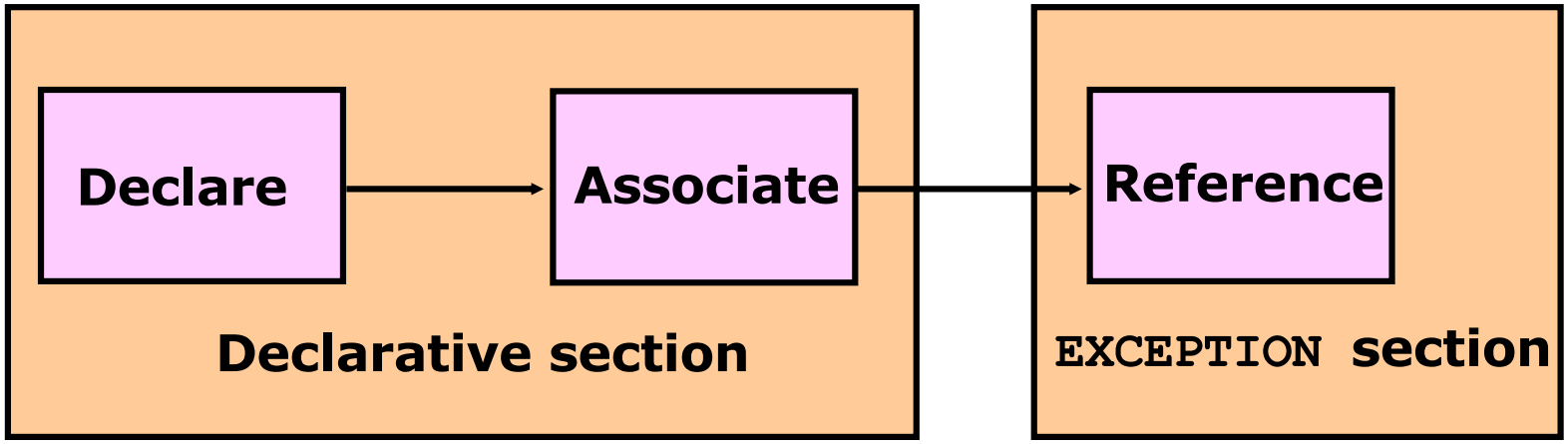
Trapping Predefined Oracle Server Errors

- 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`

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;
/
```

Trapping Non-Predefined Oracle Server Errors



Name the exception.

Use `PRAGMA EXCEPTION_INIT`.

Handle the raised exception.

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")
```

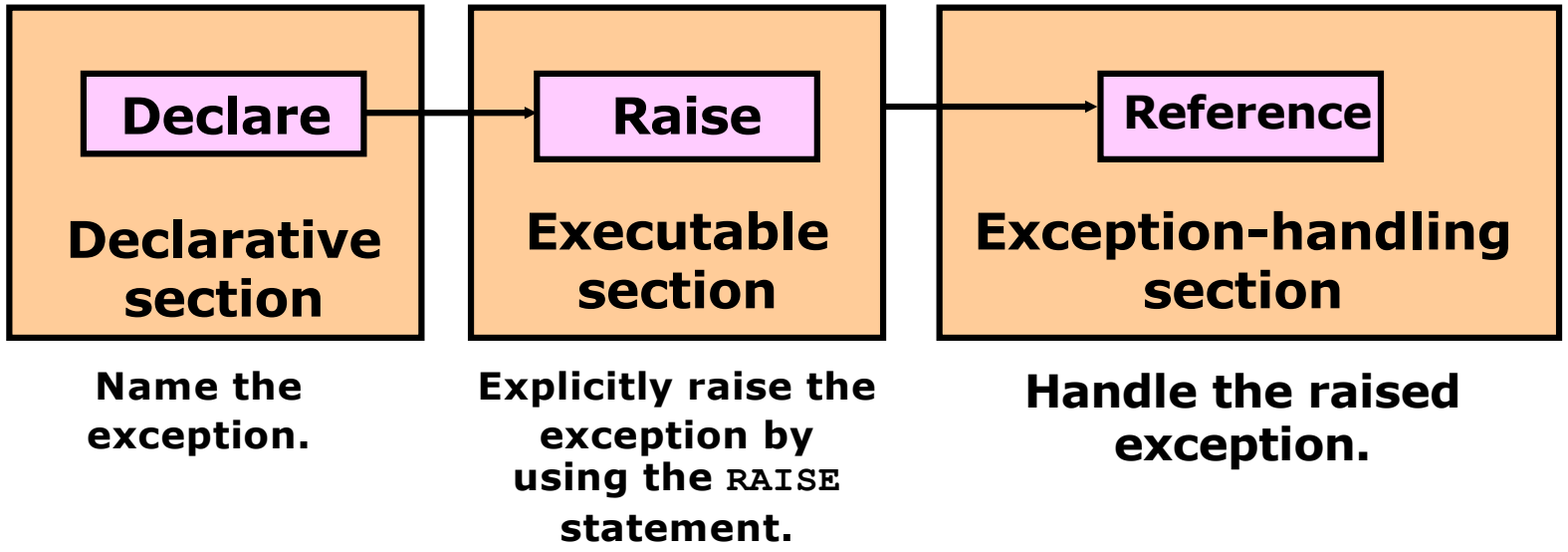
Functions for Trapping Exceptions

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

Functions for Trapping Exceptions

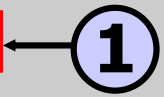
```
DECLARE
  error_code       NUMBER;
  error_message    VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    error_code := SQLCODE ;
    error_message := SQLERRM ;
   INSERT INTO errors (e_user, e_date, error_code,
   error_message) VALUES(USER,SYSDATE,error_code,
   error_message);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

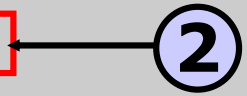
Trapping User-Defined Exceptions



Trapping User-Defined Exceptions

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






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

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;
/
```

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.

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

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;
/
```

Another example

```
        DECLARE
        e_name EXCEPTION;
        PRAGMA EXCEPTION_INIT (e_name, -20999);
BEGIN
  ...
  DELETE FROM employees
  WHERE  last_name = 'Higgins';
  IF SQL%NOTFOUND THEN
   RAISE_APPLICATION_ERROR(-20999,'This is not a
   valid last name');
  END IF;
EXCEPTION
  WHEN e_name THEN
  -- handle the error
  ...
END;
        /
```

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."

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:

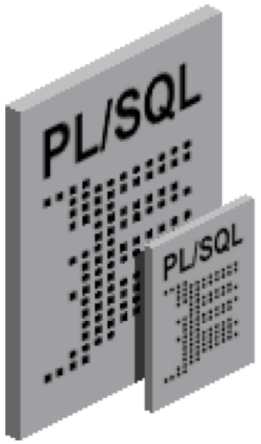
```
anonymous block completed
 Deleting department 40........
 Cannot delete this department.
   There are employees in this department (child records exist.)
```

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

CREATING STORED PROCEDURES AND FUNCTIONS

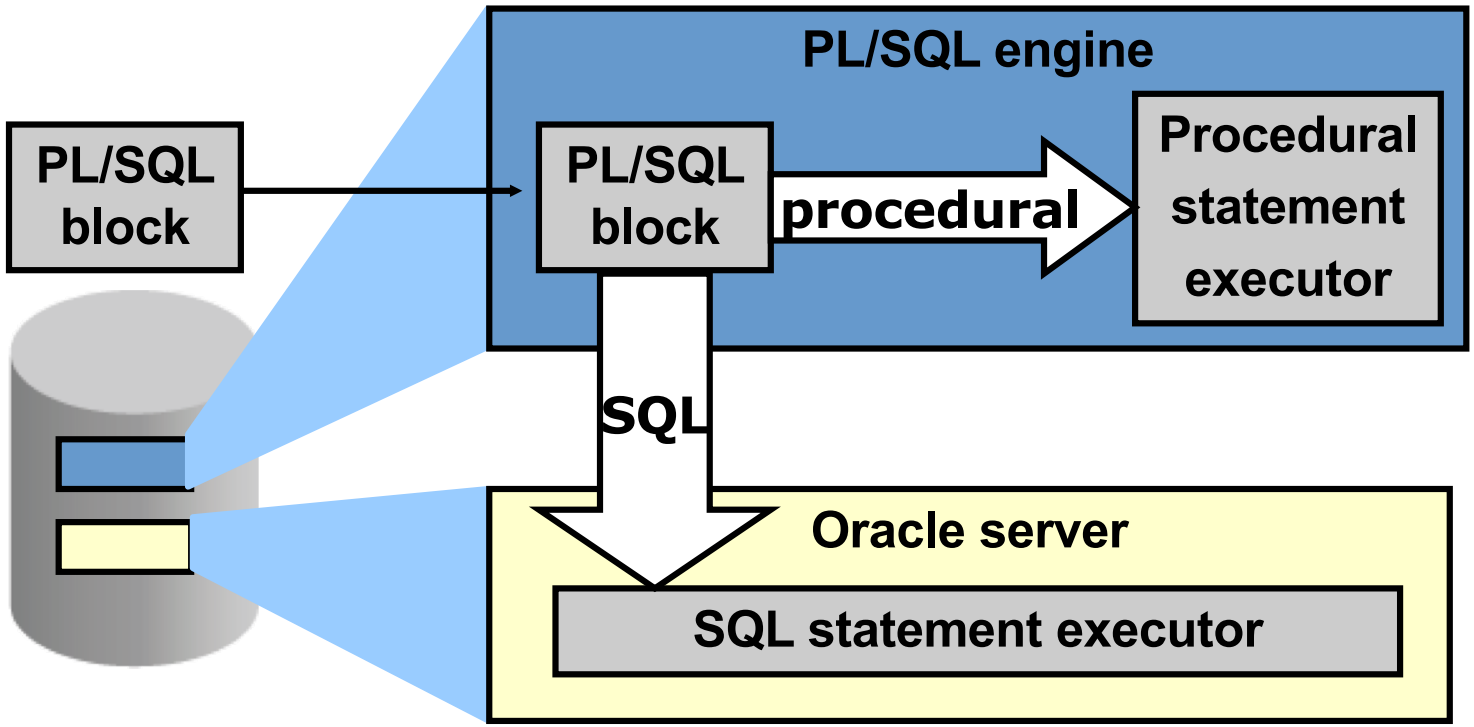
Procedures and Functions

- 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



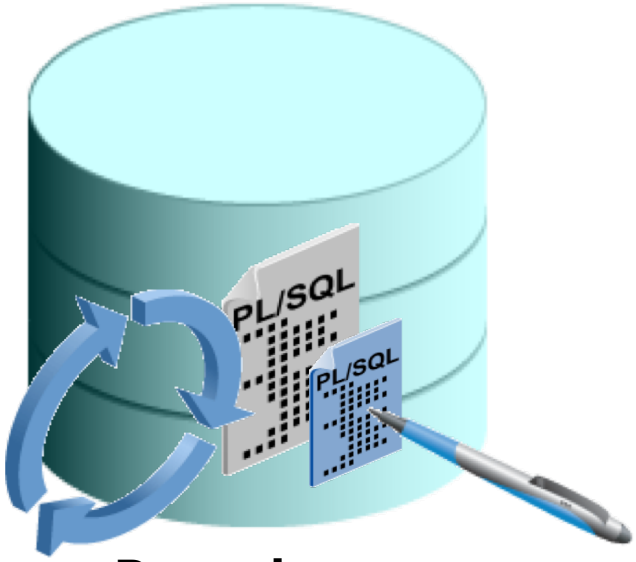
PL/SQL Execution Environment

The PL/SQL run-time architecture:



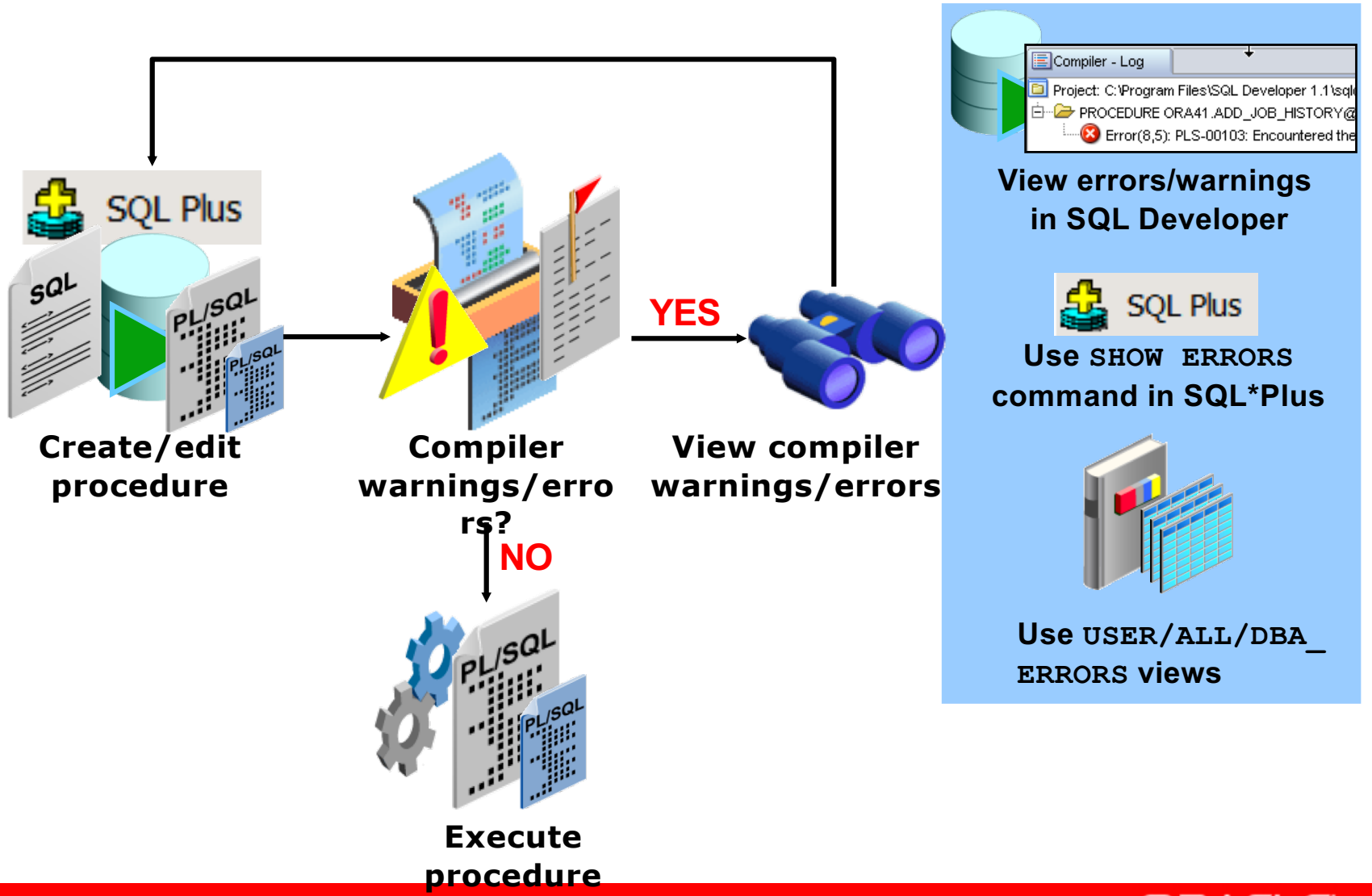
What Are Procedures?

- 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

Creating Procedures: Overview



Procedure: Syntax

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

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**

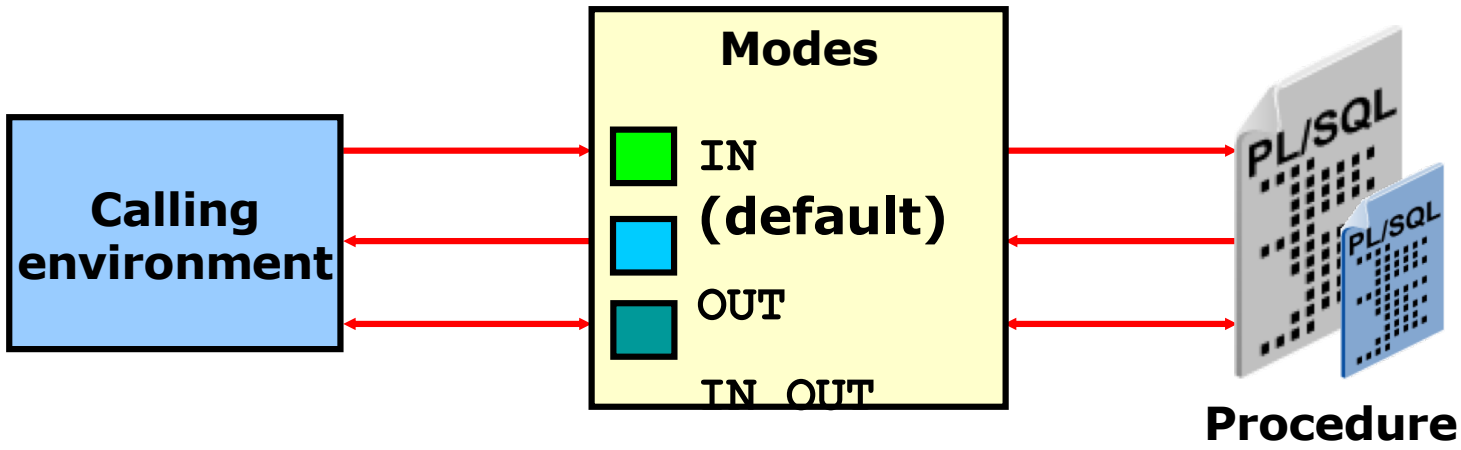
```
-- Procedure definition, Formal_parameters
CREATE PROCEDURE raise_sal(p_id NUMBER, p_sal NUMBER) IS
BEGIN
. . .
END raise_sal;
```

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

Procedural Parameter Modes

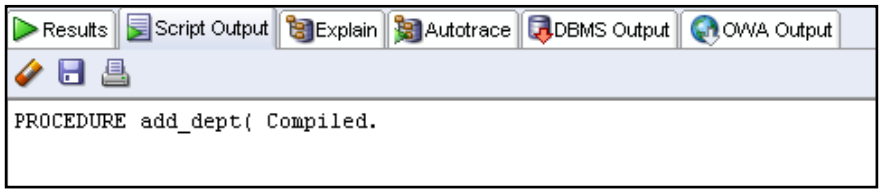
- Les modes des paramètre sont précisés dans la déclaration des paramètres formels, après le nom du paramètre et avant son type de données.
- Lee mode `IN` est la valeur par défaut si aucun mode n'est spécifié.

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



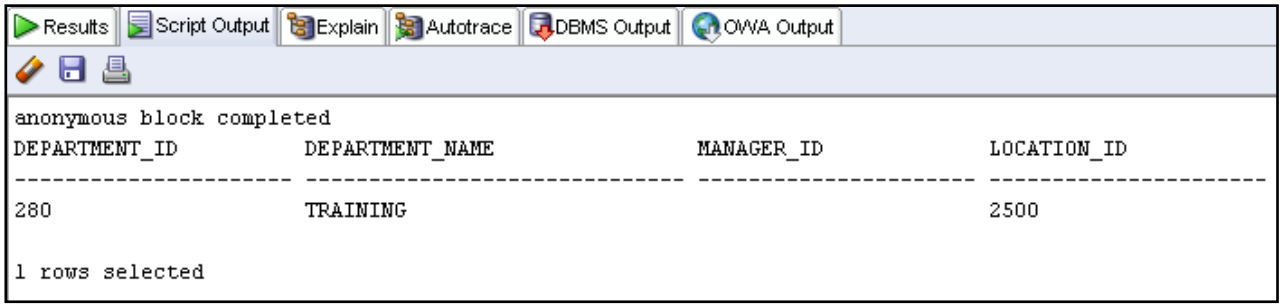
Passing Actual Parameters: Creating the add_dept Procedure

```
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;
/
```

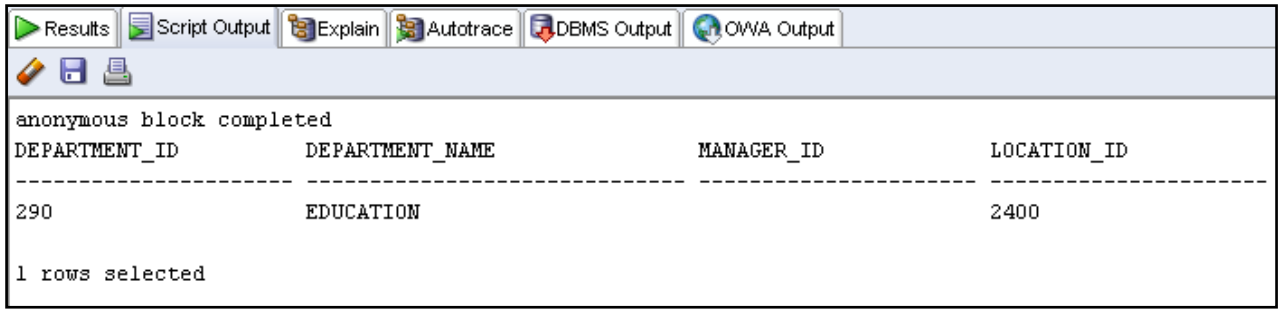


Passing Actual Parameters: Examples

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




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



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
```

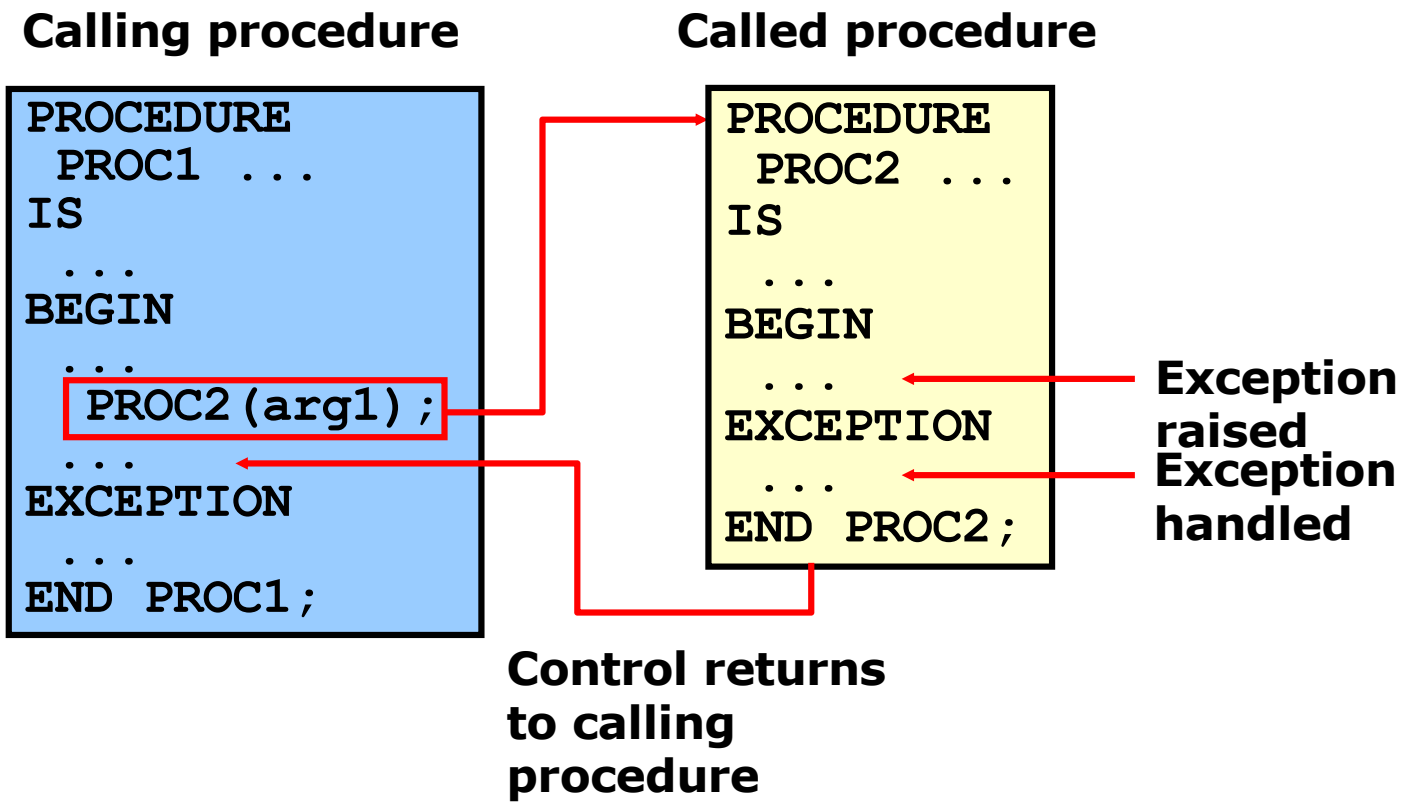
Calling Procedures

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.
```

Handled Exceptions

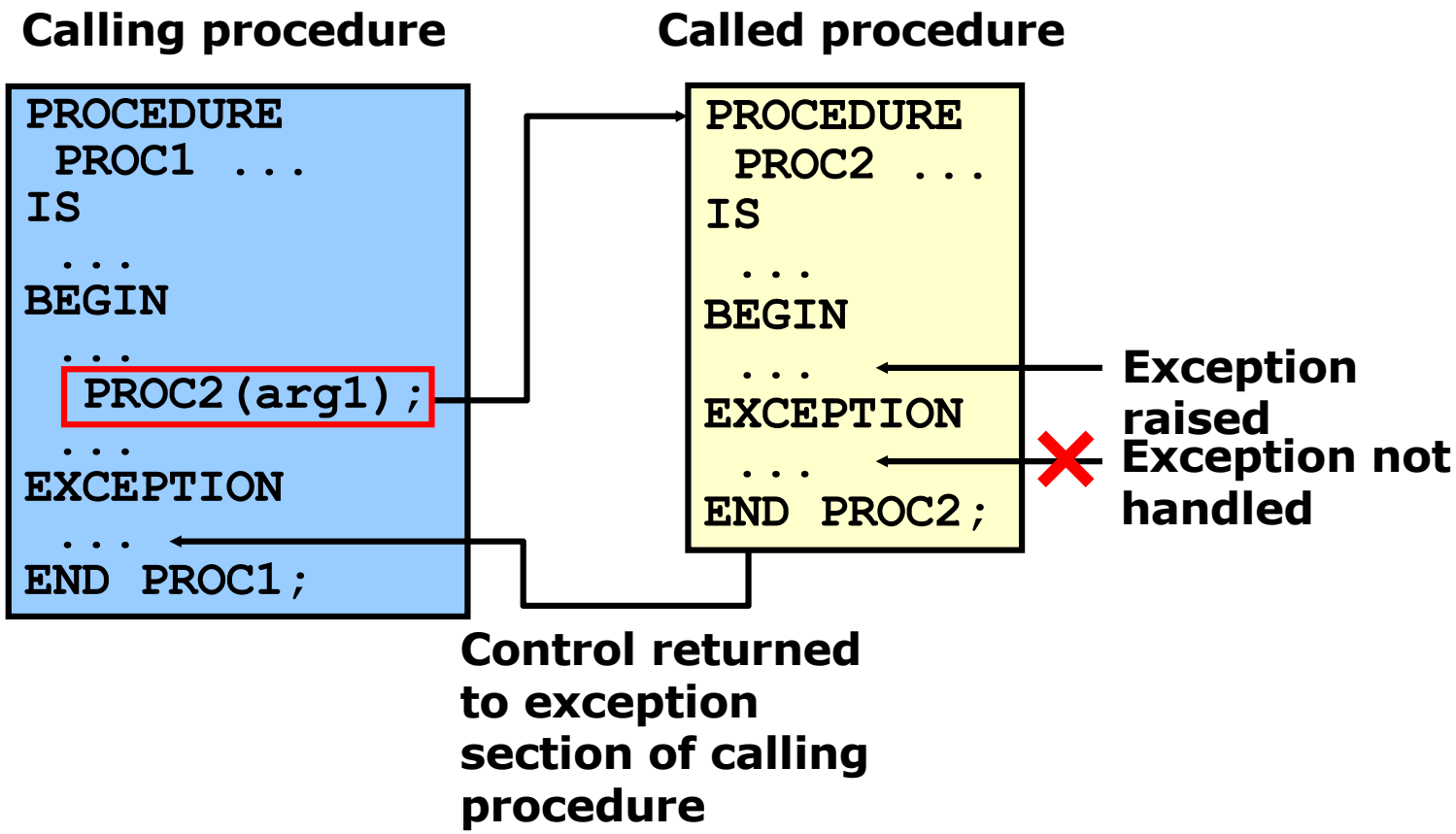


Handled Exceptions: Example

```
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;
```

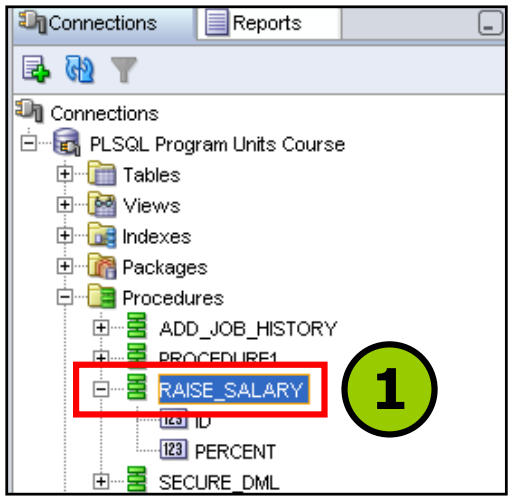
```
CREATE PROCEDURE create_departments_noex IS
BEGIN
  add_department_noex('Media', 100, 1800);
  add_department_noex('Editing', 99, 1800);
  add_department_noex('Advertising', 101, 1800);
END;
```

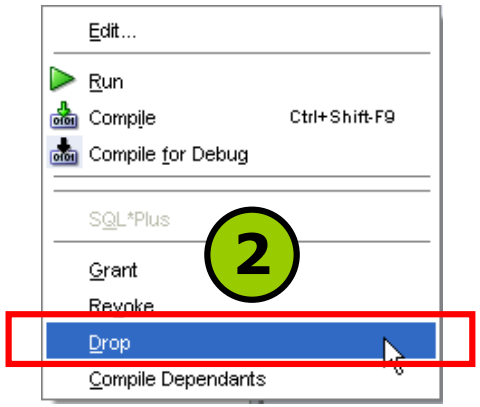
Removing Procedures: Using the DROP SQL Statement or SQL Developer

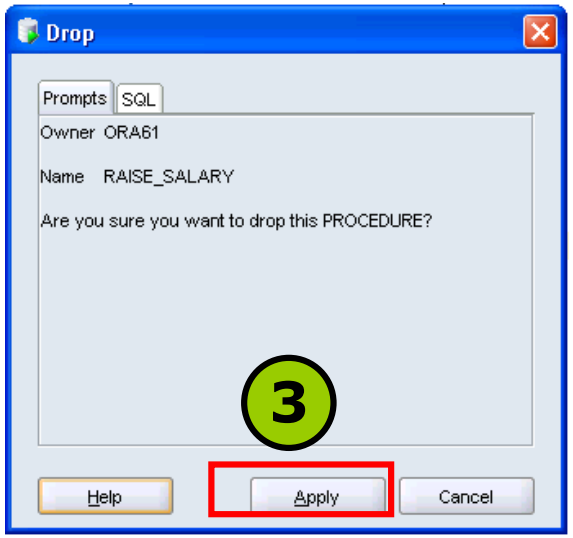
- Using the DROP statement:

```
DROP PROCEDURE raise_salary;
```

- Using SQL Developer:







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;

Exercice

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

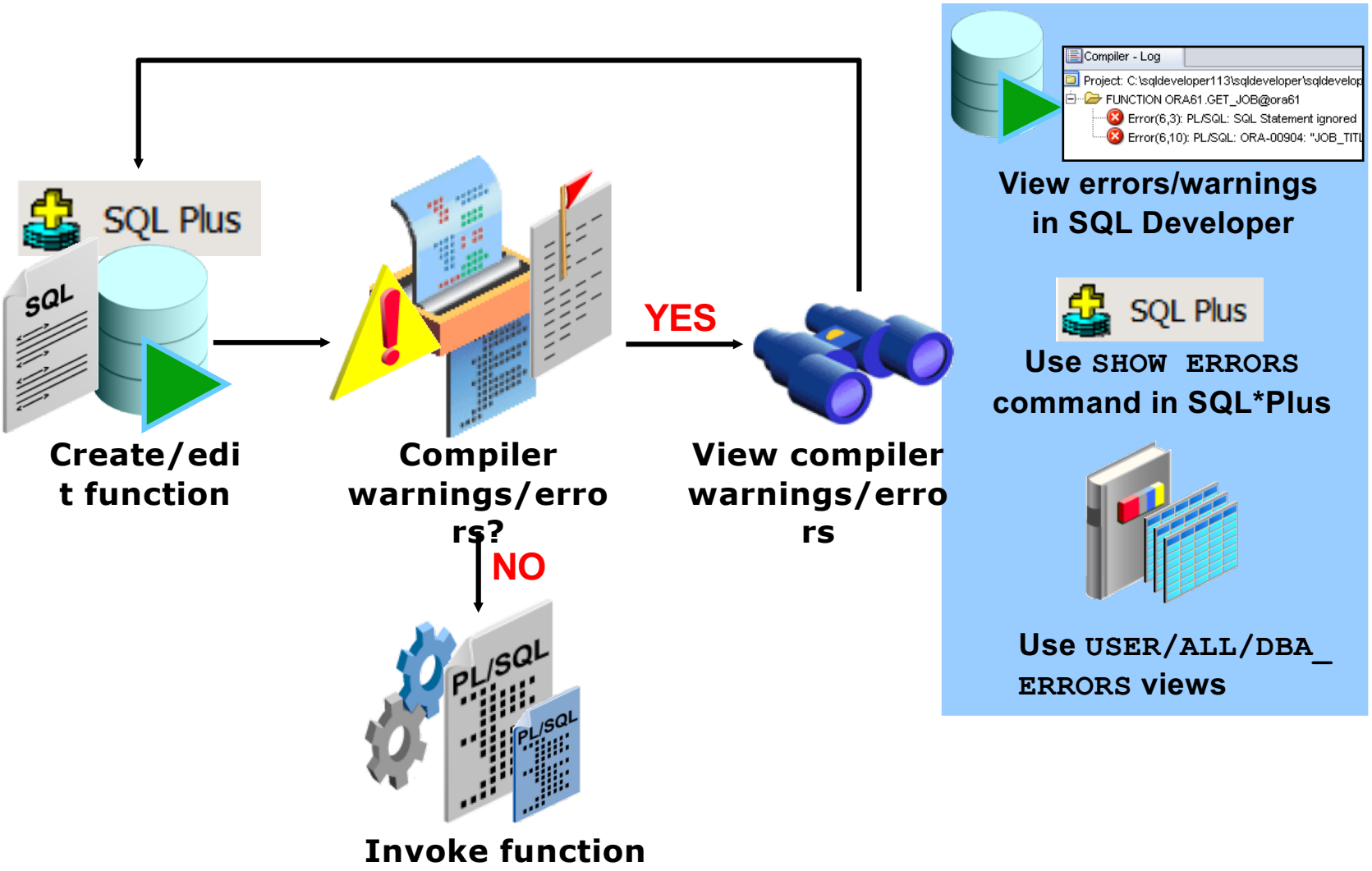
   i. **The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.**

   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 .**

Creating and Running Functions: Overview



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;
```

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

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
```

Using Different Methods for Executing Functions

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

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

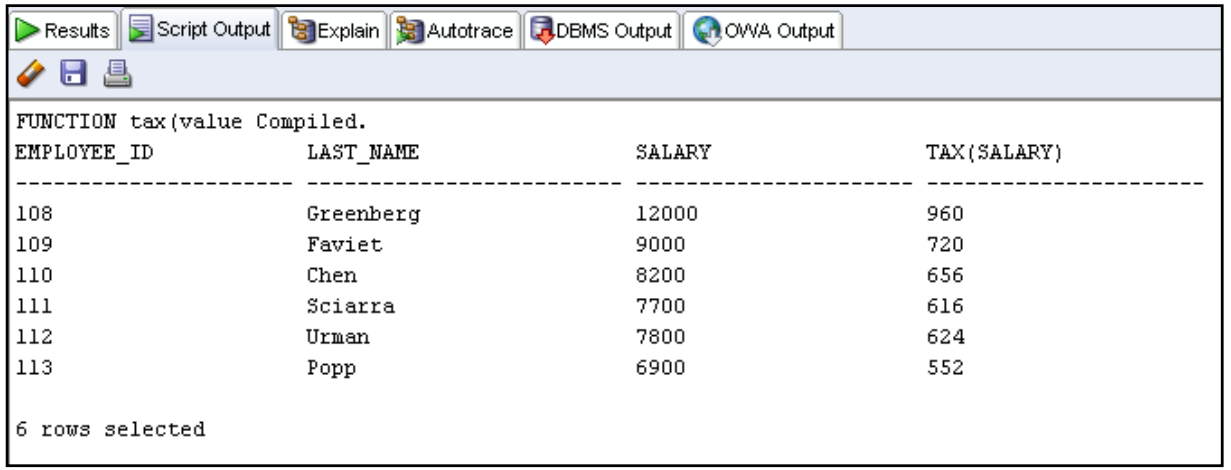
```
anonymous block completed
24000
```

```
-- 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
```

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;
```



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

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;

. . .

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
```

WORKING WITH PACKAGES

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;
/
```

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; /
```

Examples of Some Oracle-Supplied Packages

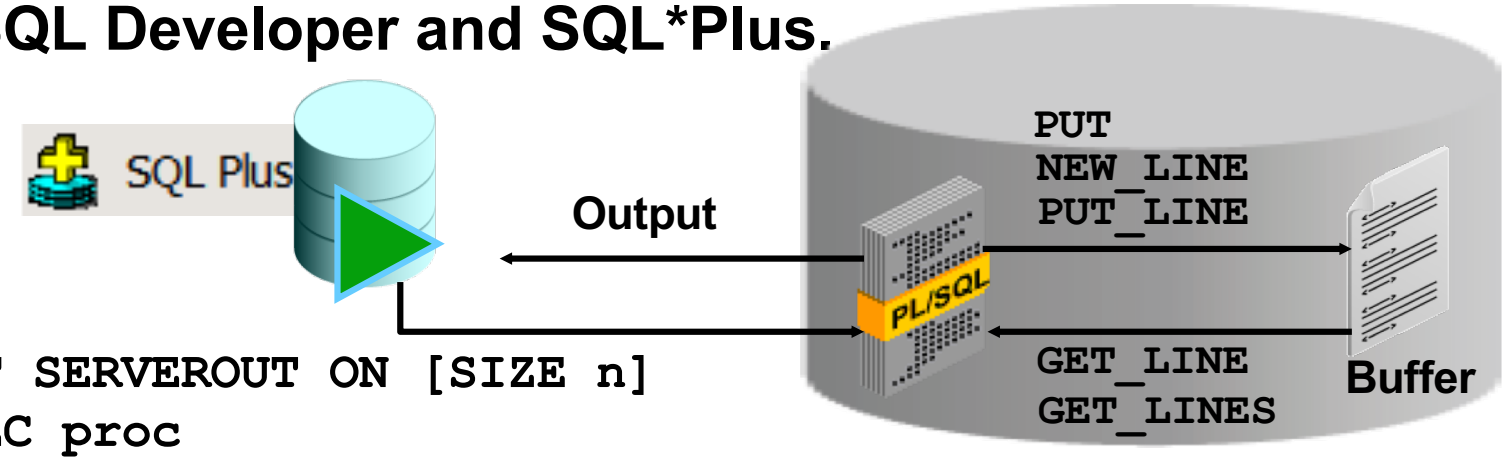
Here is an abbreviated list of some Oracle-supplied packages:

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

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.

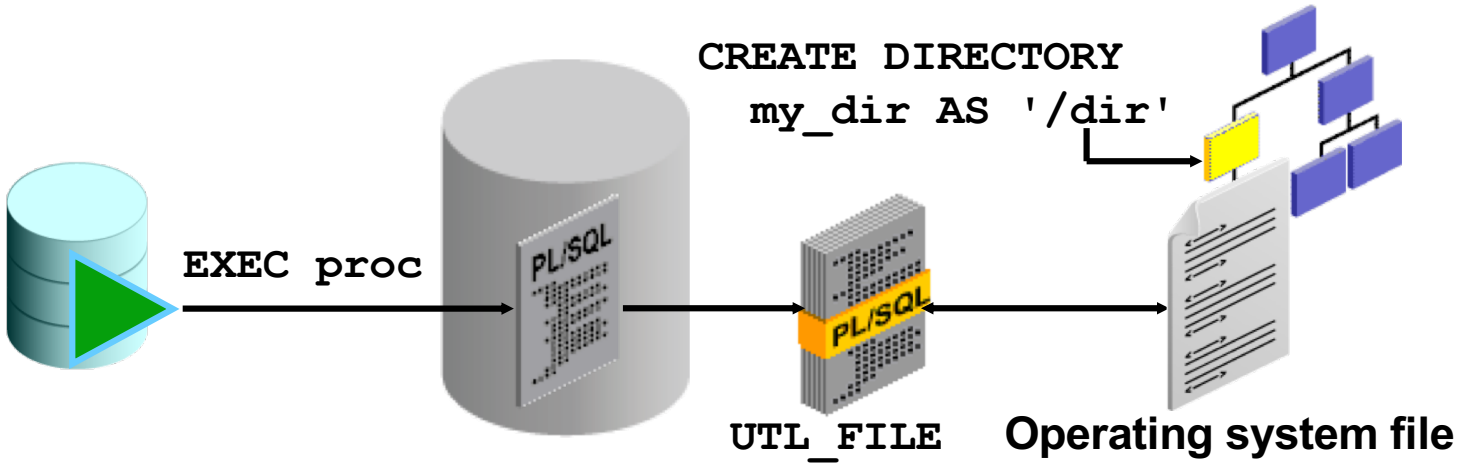



```
SET SERVEROUT ON [SIZE n]
EXEC proc
```

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);
. . .
```

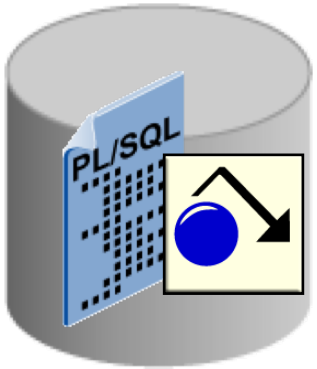
Using UTL_FILE: Example

```
. . .
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;/
```

CREATING TRIGGERS

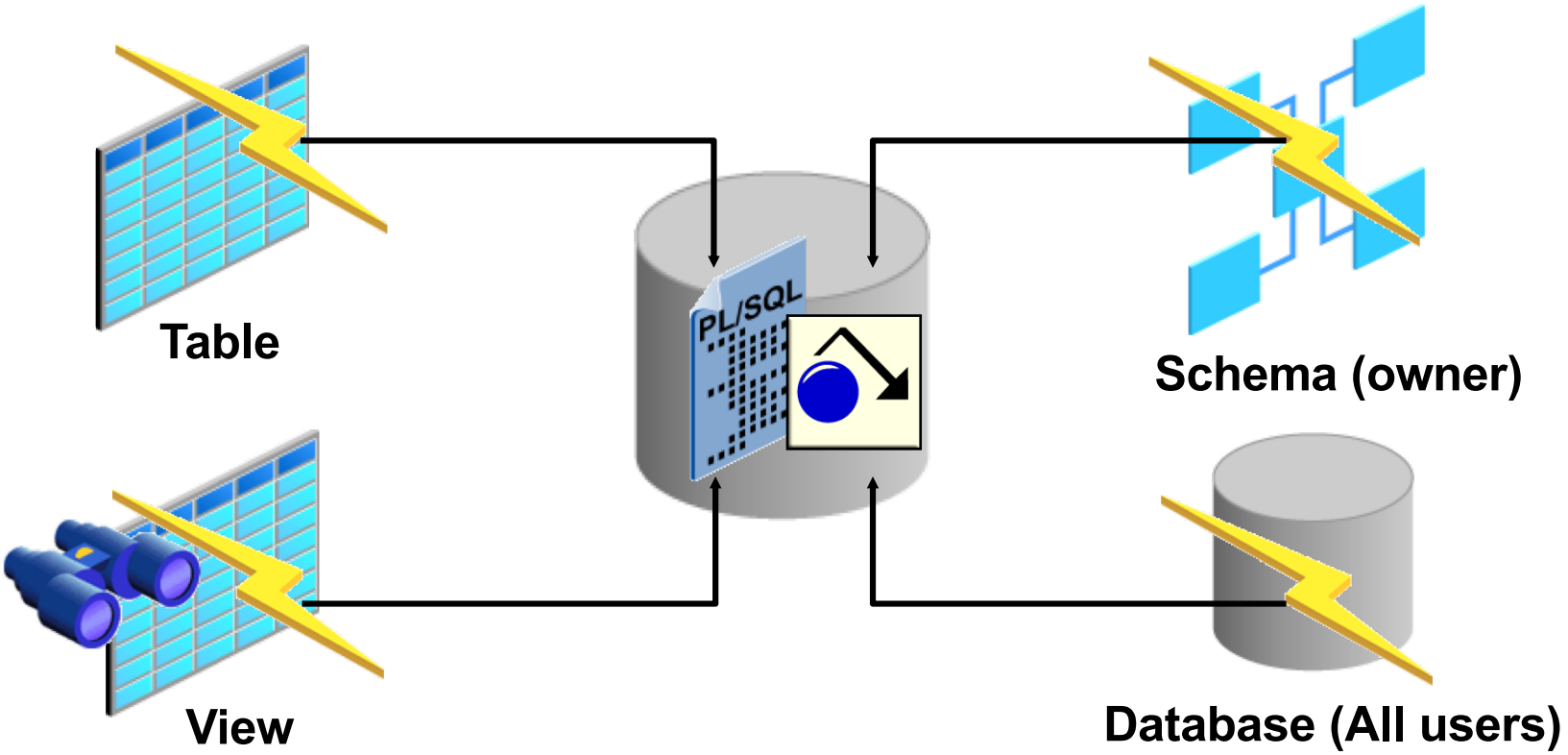
What Are Triggers?

- 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.



Defining Triggers

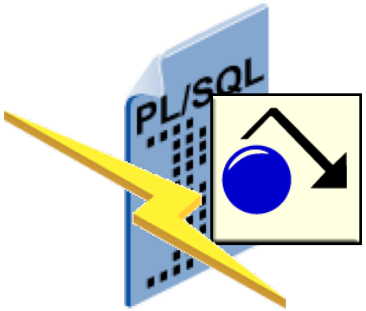
A trigger can be defined on the table, view, schema (schema owner), or database (all users).



Trigger Event Types

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

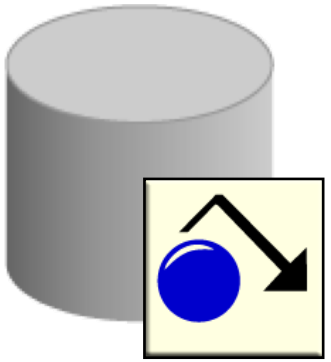
- Une manipulation de la base de données (DML) (`DELETE`, `INSERT`, or `UPDATE`).
- Une requête de définition de base de données (DDL) (`CREATE`, `ALTER`, or `DROP`).
- Une opération de base de données tels que `SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`.



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

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

Available Trigger Types

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

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

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
```

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.

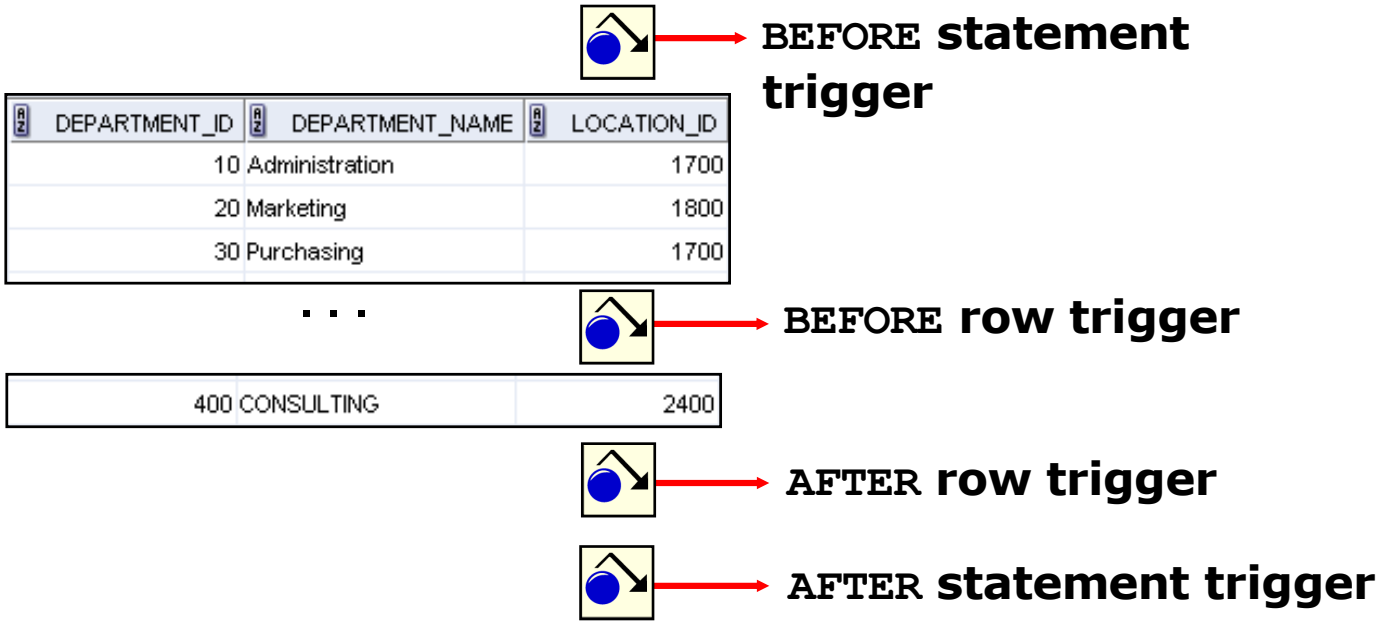
Statement-Level Triggers Versus Row-Level Triggers

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

Trigger-Firing Sequence: Single-Row Manipulation

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

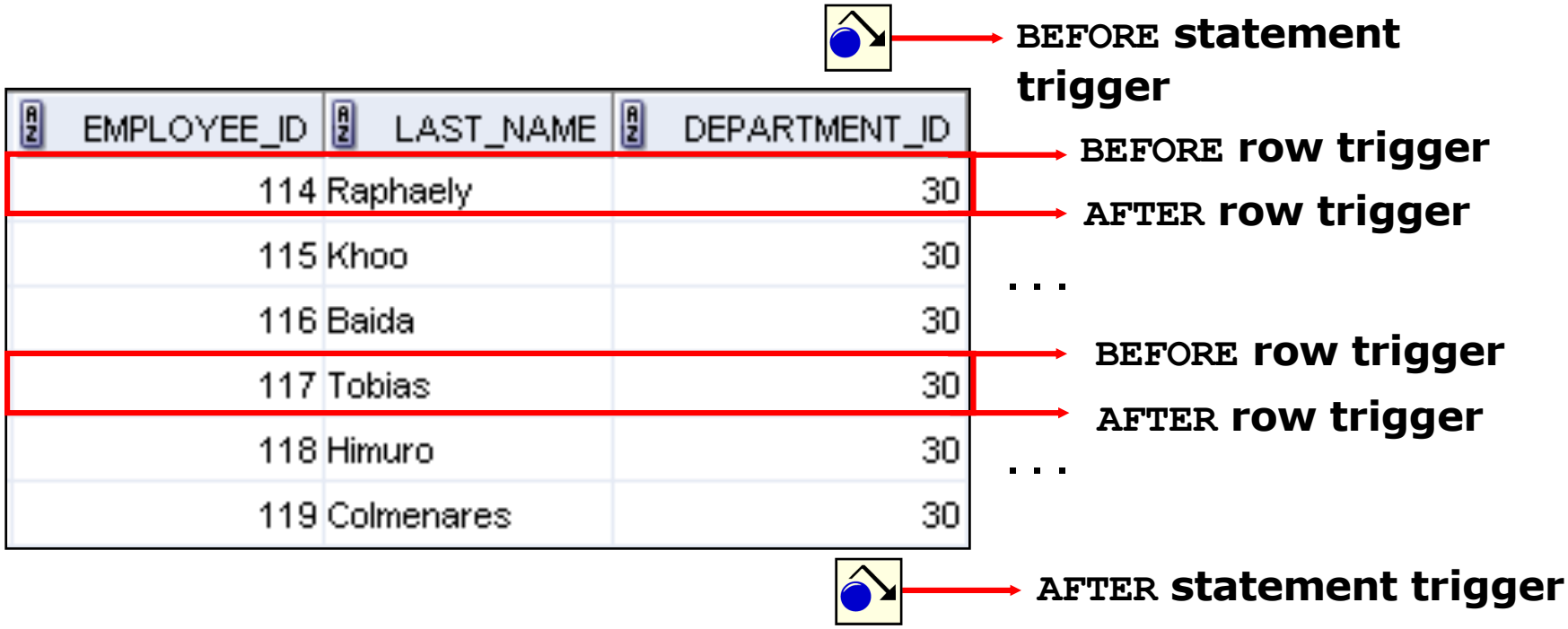
```
INSERT INTO departments
   (department_id,department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```



Trigger-Firing Sequence: Multirow Manipulation

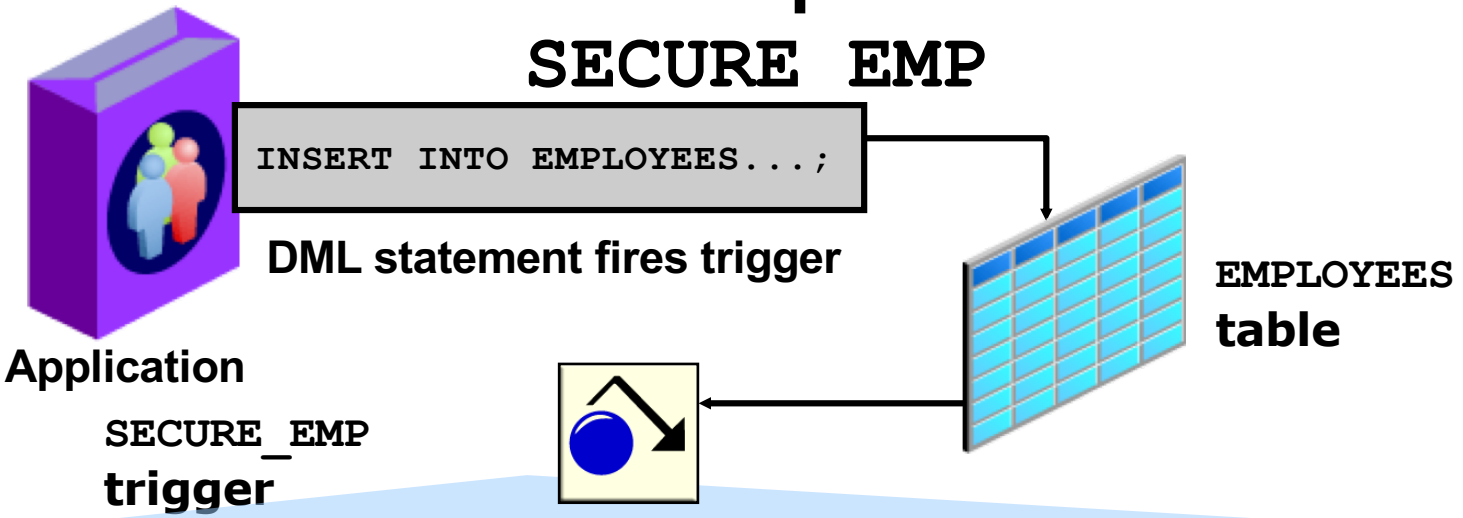
Use the following firing sequence for a trigger on a table when many rows are manipulated:

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



Creating a DML Statement Trigger

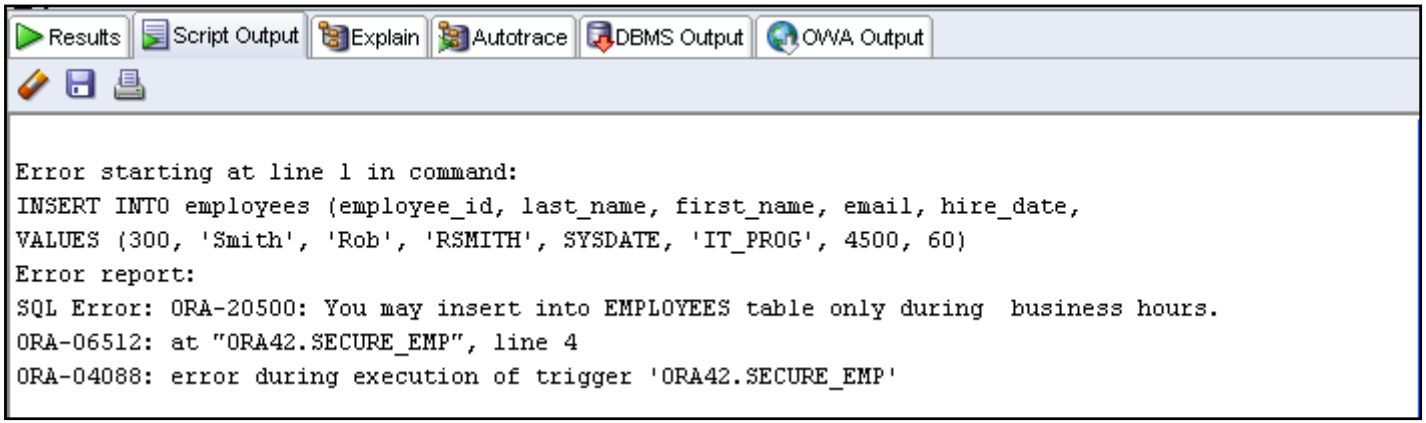
Example:




```
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);
```



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

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;
/
```

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP

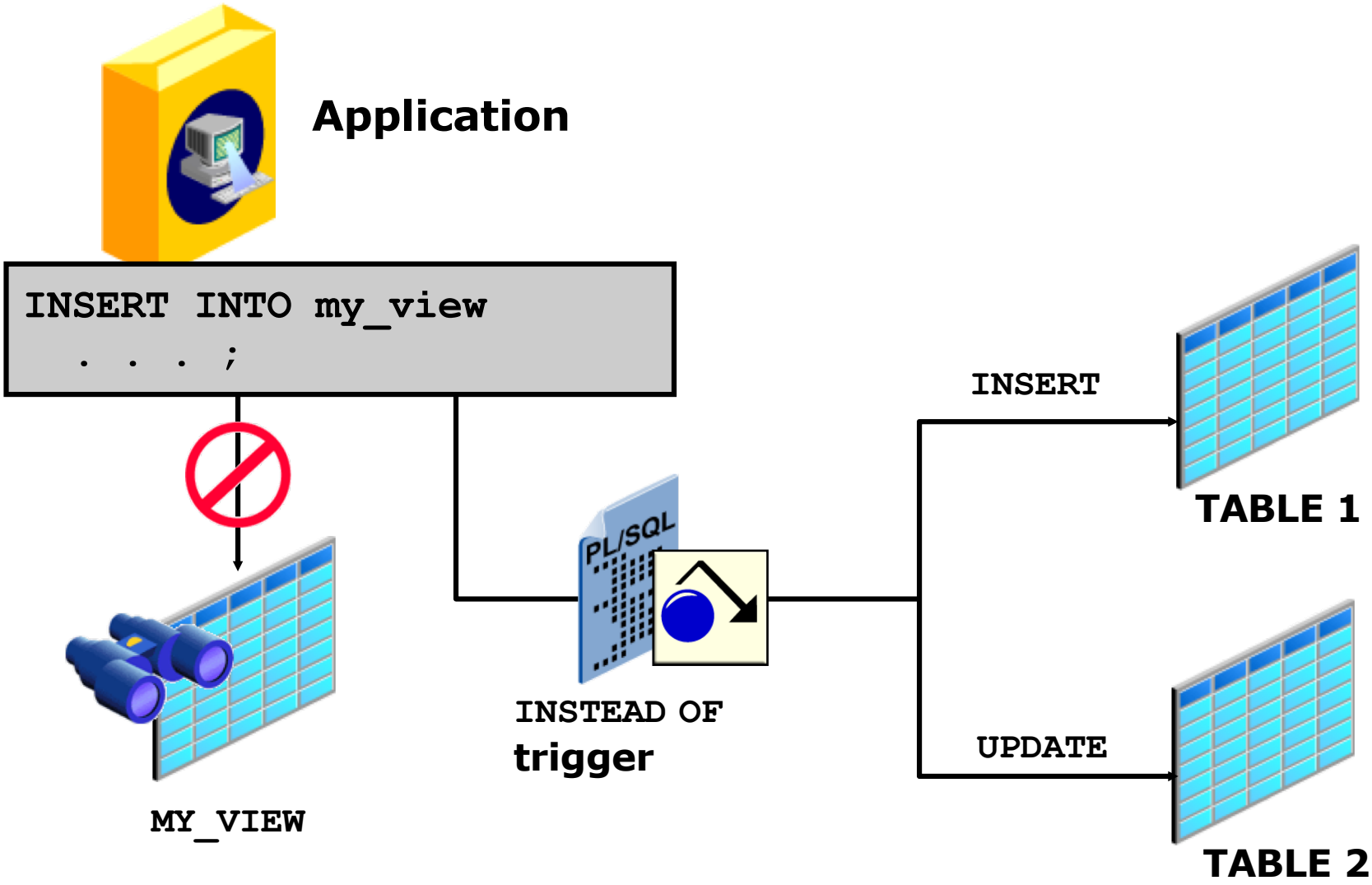
```
INSERT INTO employees (employee_id, last_name, job_id,
salary, email, hire_date)
VALUES (999, 'Temp emp', 'SA_REP', 6000, 'TEMPEMP',
TRUNC(SYSDATE));
/
UPDATE employees
 SET salary = 7000, last_name = 'Smith'
 WHERE employee_id = 999;
/
SELECT *
FROM  audit_emp;
```

	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

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

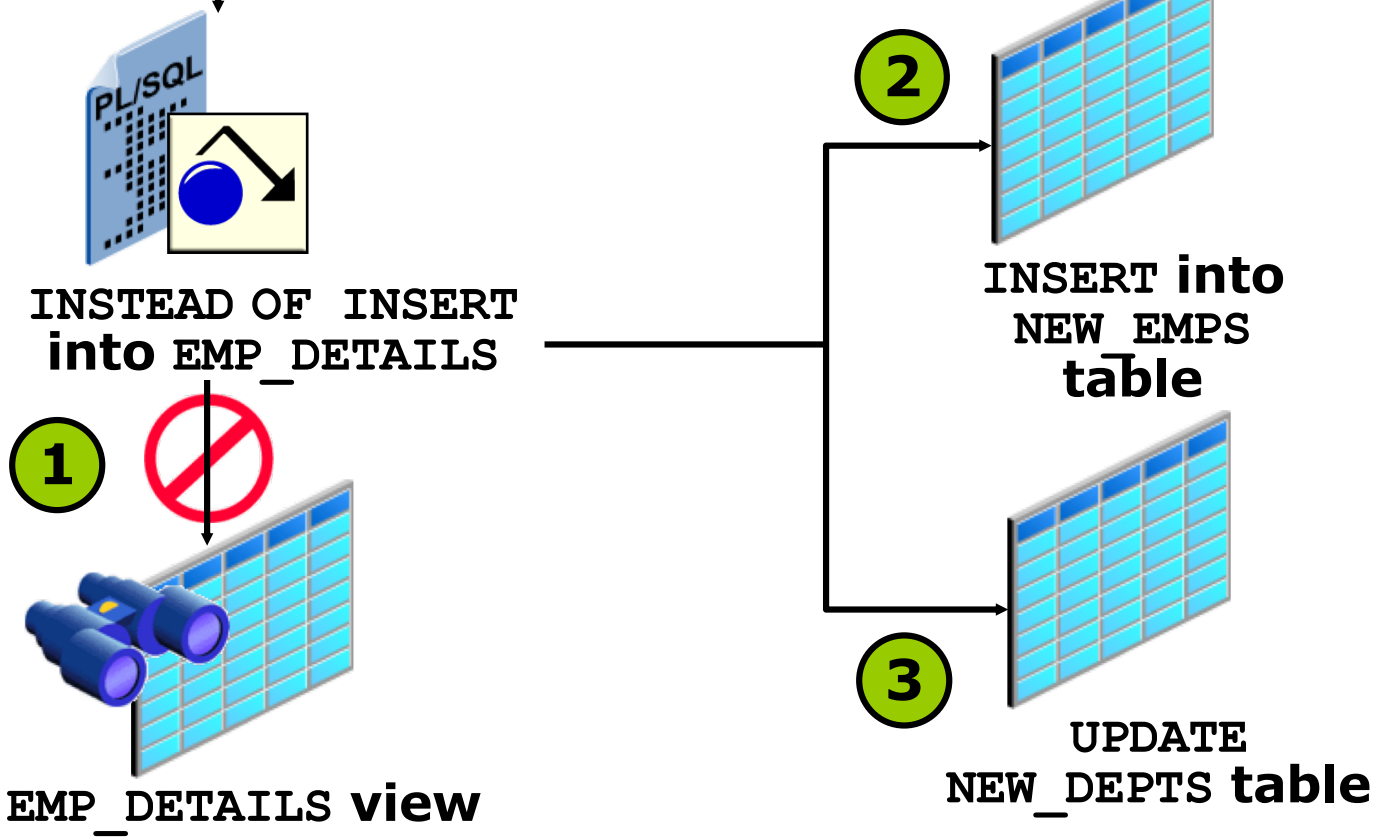
```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
 IF INSERTING THEN
   :NEW.commission_pct := 0;
 ELSIF :OLD.commission_pct IS NULL THEN
   :NEW.commission_pct := 0;
 ELSE
   :NEW.commission_pct := :OLD.commission_pct+0.05;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

INSTEAD OF Triggers



Creating an INSTEAD OF Trigger: Example

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



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
GROUP BY d.department_id,d.department_name;

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;
```

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;
```

```
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;
```

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

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';
```

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.

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.