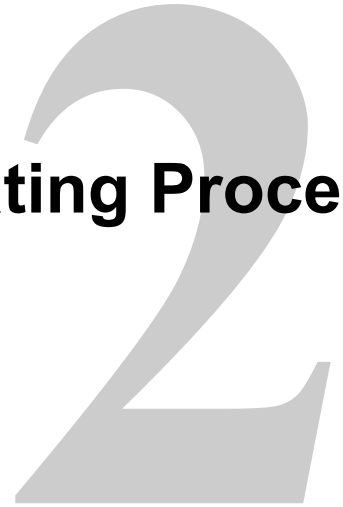


Creating Procedures

Objectives

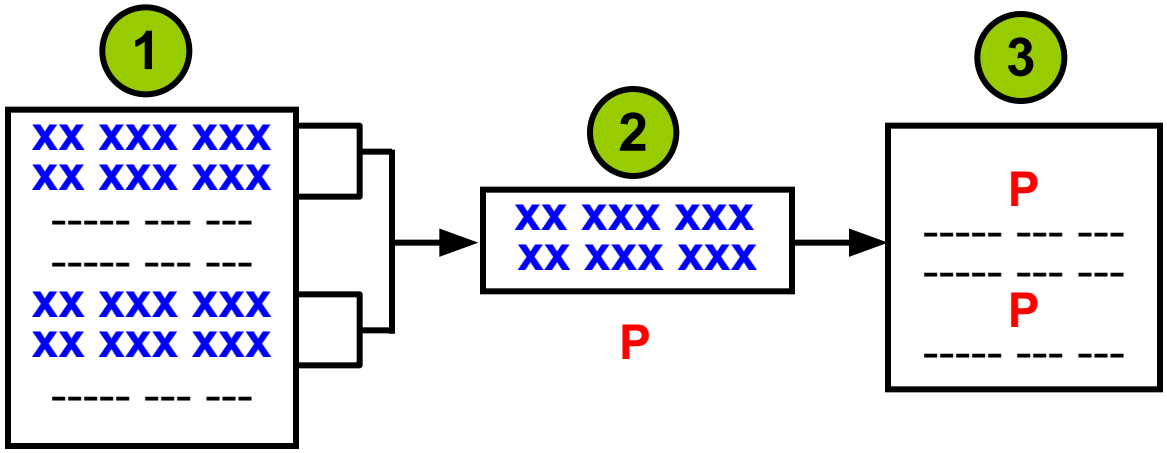
After completing this lesson, you should be able to do the following:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure and display its information

Lesson Agenda

- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
  - Creating and calling procedures
  - Identifying the available parameter-passing modes
  - Using formal and actual parameters
  - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedure's information

Creating a Modularized Subprogram Design



Modularize code into subprograms.

1. Locate code sequences repeated more than once.
2. Create subprogram P containing the repeated code
3. Modify original code to invoke the new subprogram.

Creating a Layered Subprogram Design

Create subprogram layers for your application.

- Data access subprogram layer with SQL logic
- Business logic subprogram layer, which may or may not use the data access layer

Modularizing Development with PL/SQL Blocks

- PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:
  - Anonymous blocks
  - Procedures and functions
  - Packages
  - Database triggers
- The benefits of using modular program constructs are:
  - Easy maintenance
  - Improved data security and integrity
  - Improved performance
  - Improved code clarity

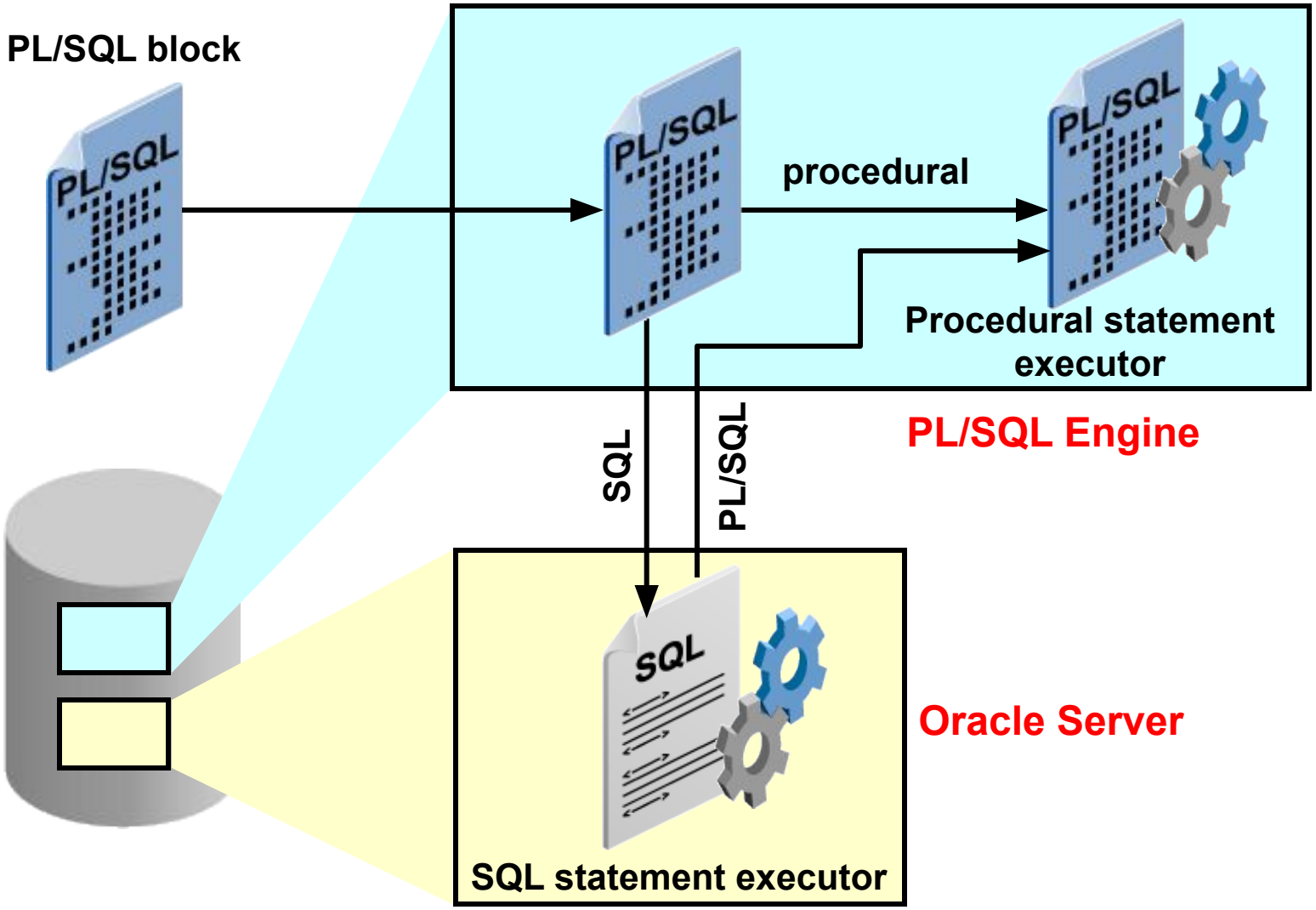
Anonymous Blocks: Overview

Anonymous blocks:

- Form the basic PL/SQL block structure
- Initiate PL/SQL processing tasks from applications
- Can be nested within the executable section of any PL/SQL block

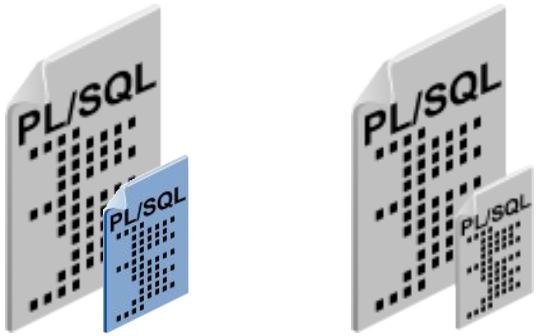
```
[DECLARE      -- Declaration Section (Optional)
  variable declarations; ... ]
BEGIN         -- Executable Section (Mandatory)
  SQL or PL/SQL statements;
[EXCEPTION    -- Exception Section (Optional)
  WHEN exception THEN statements; ]
END;          -- End of Block (Mandatory)
```

PL/SQL Runtime Architecture

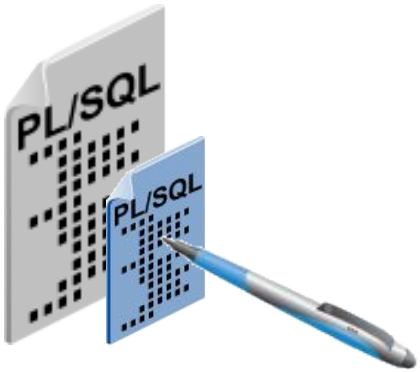


What Are PL/SQL Subprograms?

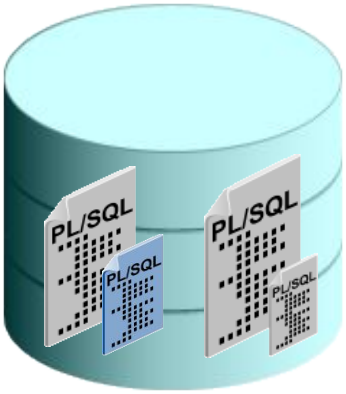
- A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters.
- You can declare and define a subprogram within either a PL/SQL block or another subprogram.
- A subprogram consists of a specification and a body.
- A subprogram can be a procedure or a function.
- Typically,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



The Benefits of Using PL/SQL Subprograms



Easy maintenance



Subprograms:
Stored procedures
and functions

Improved data security
and integrity

Improved code clarity

Improved performance

Differences Between Anonymous Blocks and Subprograms

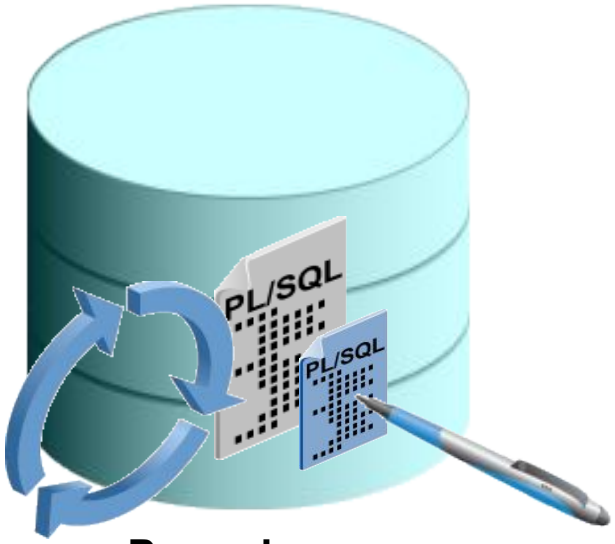
Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	Subprograms called functions must return values.
Cannot take parameters	Can take parameters

Lesson Agenda

- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
  - Creating and calling procedures
  - Identifying the available parameter-passing modes
  - Using formal and actual parameters
  - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information

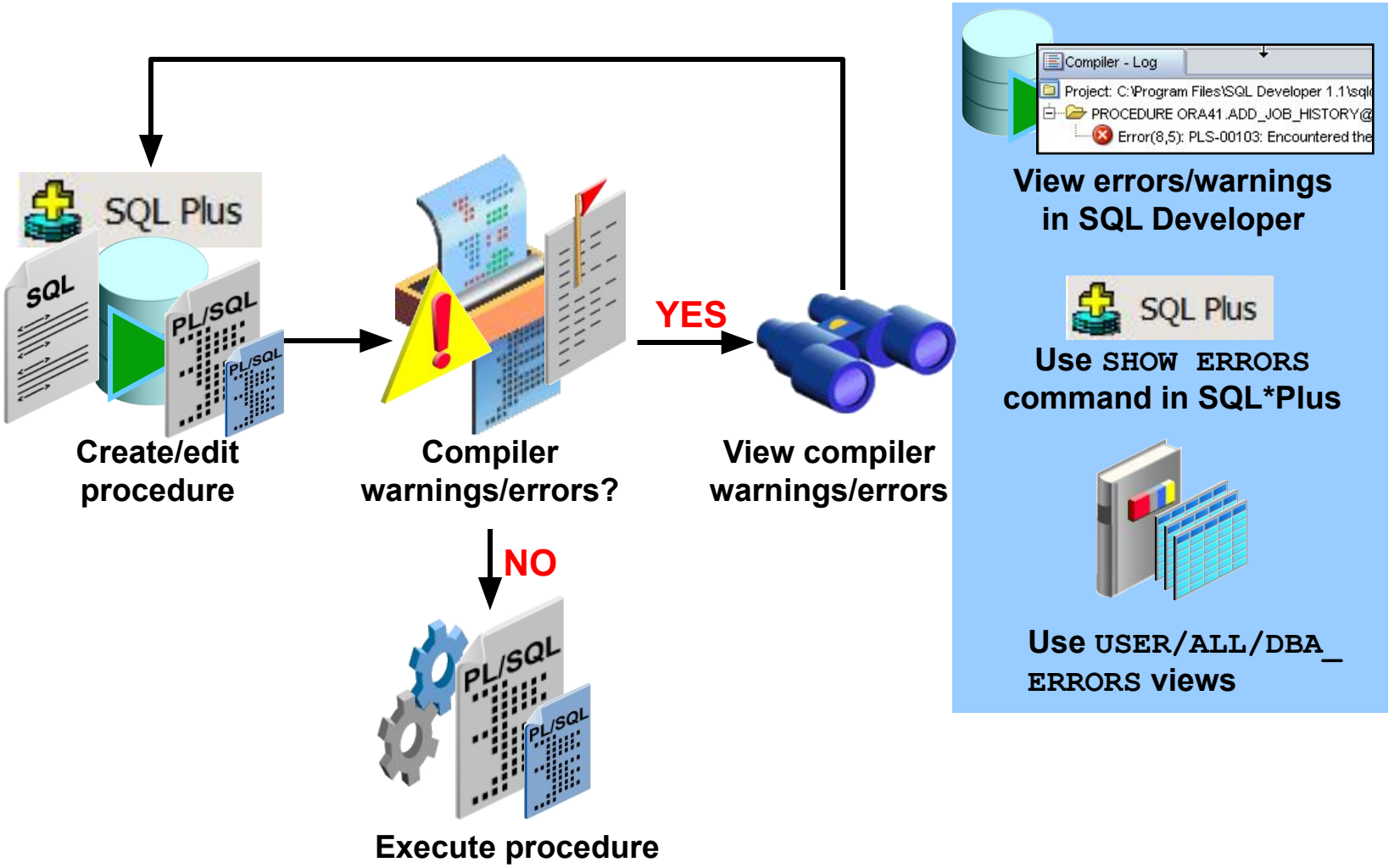
What Are Procedures?

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



Procedures

Creating Procedures: Overview



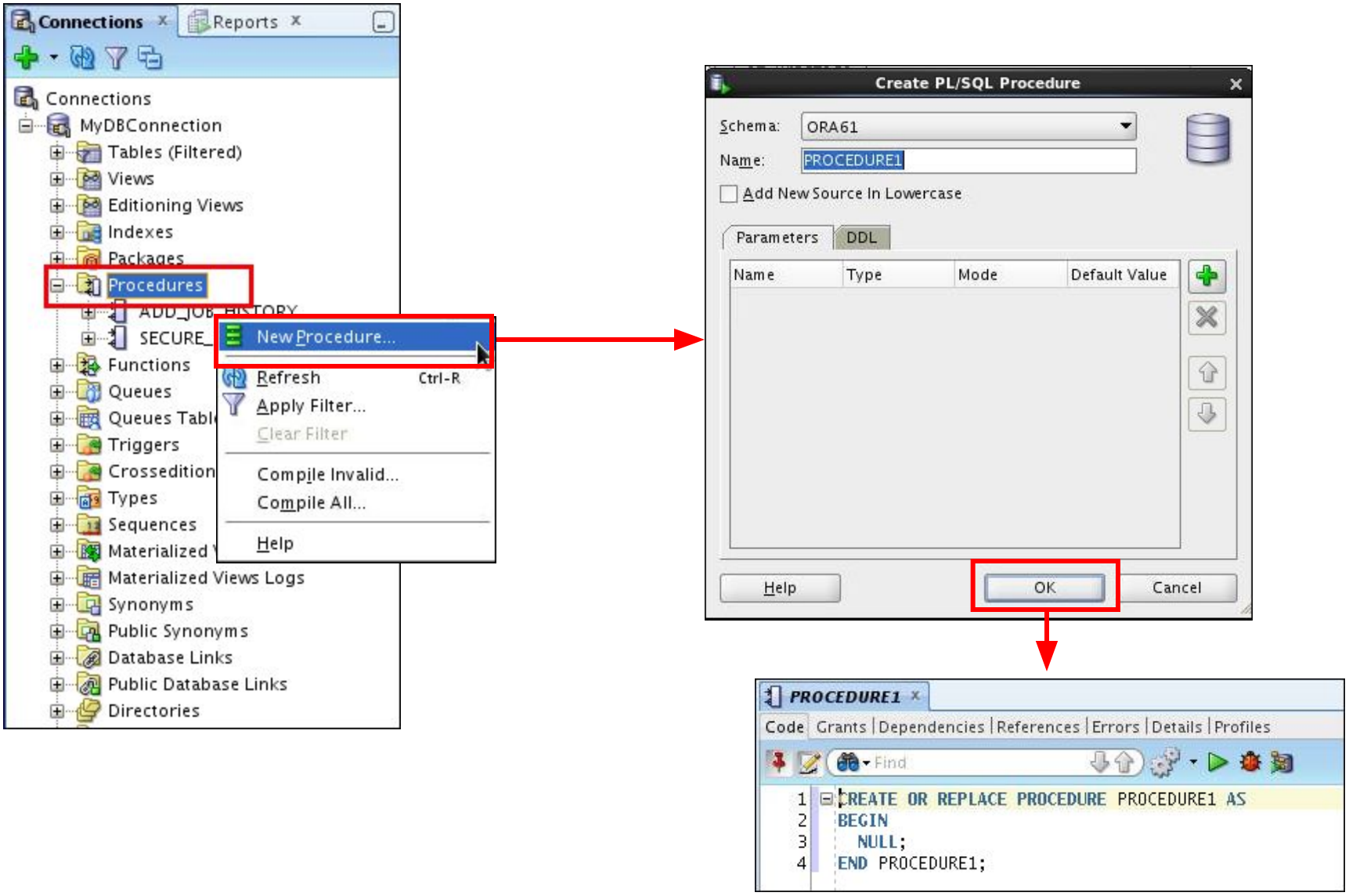
Creating Procedures with the SQL `CREATE OR REPLACE` Statement

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

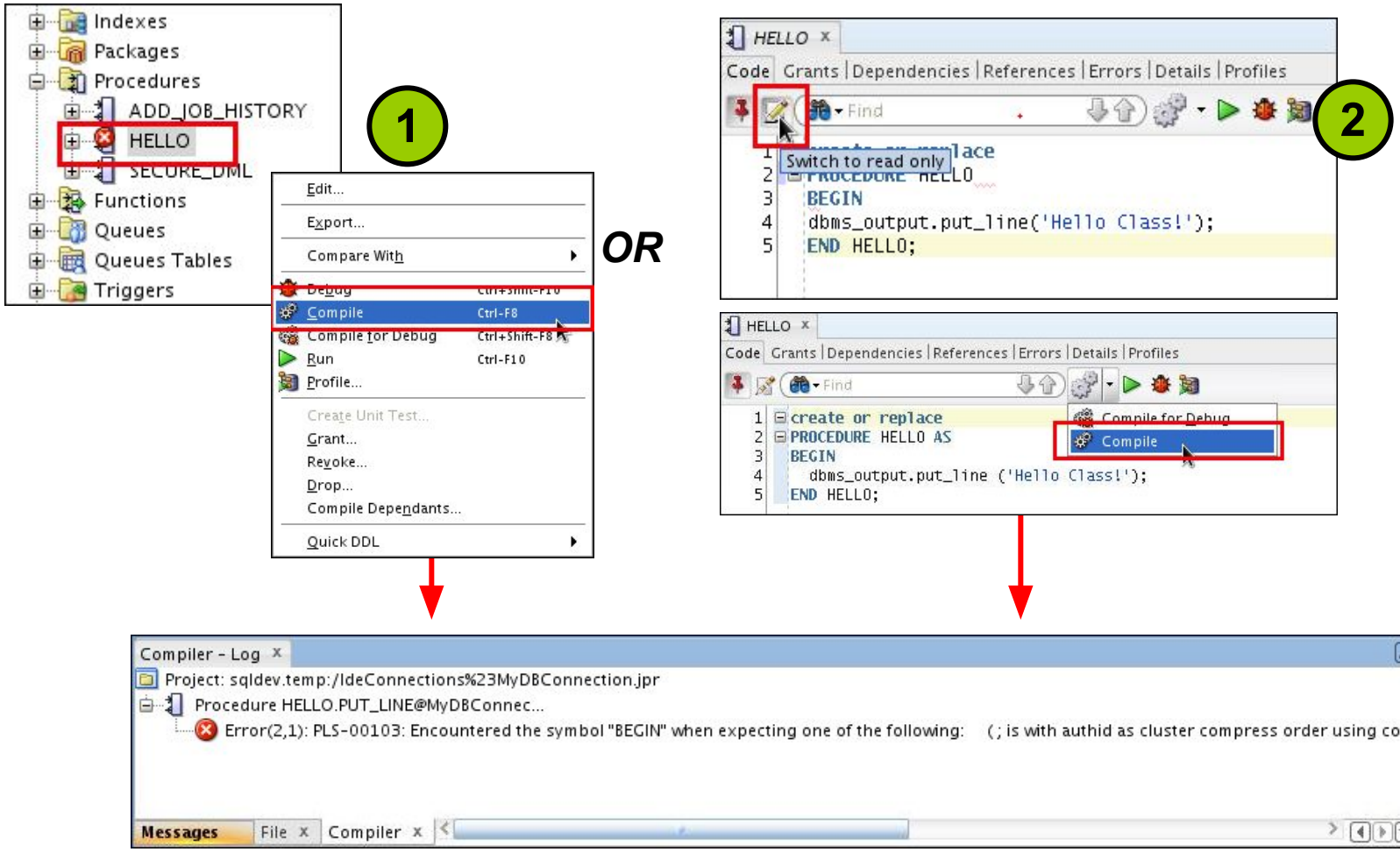
```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; ...]
BEGIN
      -- actions;
END [procedure_name];
```

PL/SQL block

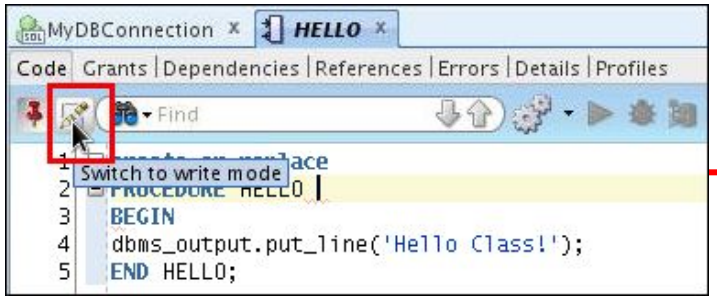
Creating Procedures by Using SQL Developer



Compiling Procedures and Displaying Compilation Errors in SQL Developer

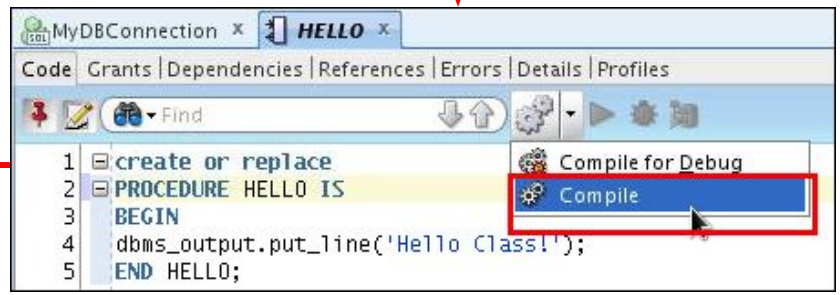


Correcting Compilation Errors in SQL Developer

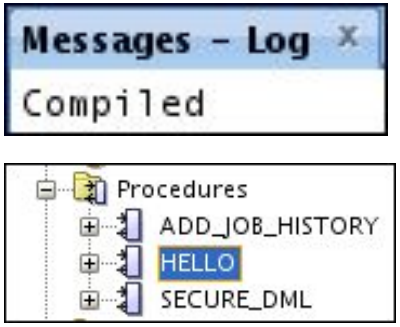


1. Edit procedure

2. Correct error (add keyword `IS`)



3. Recompile procedure



4. Recompilation successful

Naming Conventions of PL/SQL Structures Used in This Course

PL/SQL Structure	Convention	Example
Variable	**v_***variable_name*	v_rate
Constant	**c_***constant_name*	c_rate
Subprogram parameter	**p_***parameter_name*	p_id
Bind (host) variable	**b_***bind_name*	b_salary
Cursor	**cur_***cursor_name*	cur_emp
Record	**rec_***record_name*	rec_emp
Type	**type_**name_*type*	ename_table_type
Exception	**e_***exception_name*	e_products_invalid
File handle	**f_***file_handle_name*	f_file

What Are Parameters and Parameter Modes?

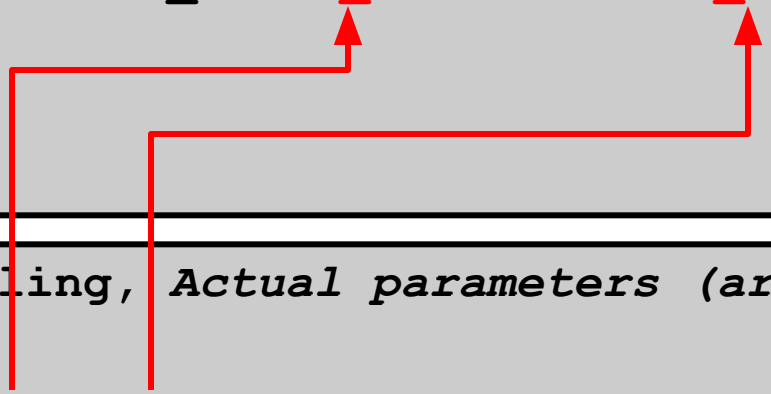
- Are declared after the subprogram name in the PL/SQL header
- Pass or communicate data between the calling environment and the subprogram
- Are used like local variables but are dependent on their parameter-passing mode:
  - An `IN` parameter mode (the default) provides values for a subprogram to process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

Formal and Actual Parameters

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

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

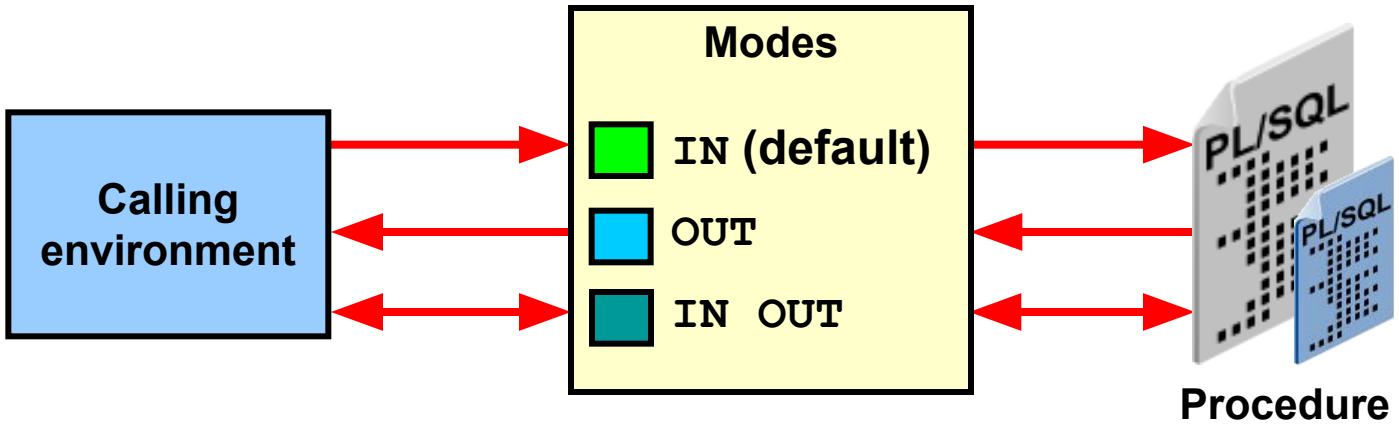
```
-- Procedure calling, Actual parameters (arguments)
v_emp_id := 100;
raise_sal(v_emp_id, 2000)
```



Procedural Parameter Modes

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

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

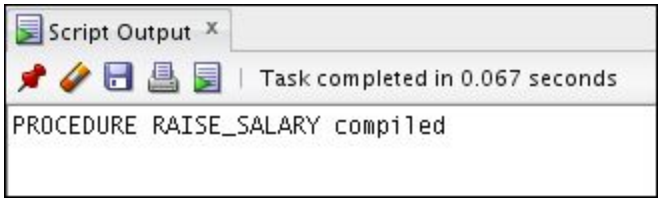


Comparing the Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Value is returned to the calling environment	Value passed into sub-program; value returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

Using the IN Parameter Mode: Example

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




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

Using the OUT Parameter Mode: Example

```
CREATE OR REPLACE PROCEDURE query_emp
 (p_id      IN  employees.employee_id%TYPE,
  p_name    OUT employees.last_name%TYPE,
  p_salary OUT employees.salary%TYPE) IS
BEGIN
  SELECT  last_name, salary INTO p_name, p_salary
  FROM    employees
  WHERE   employee_id = p_id;
END query_emp;
/
```

```
SET SERVEROUTPUT ON
DECLARE
  v_emp_name employees.last_name%TYPE;
  v_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, v_emp_name, v_emp_sal);
  DBMS_OUTPUT.PUT_LINE(v_emp_name||' earns '||
    to_char(v_emp_sal, '$999,999.00'));
END;
/
```

Using the IN OUT Parameter Mode: Example

Calling environment


```
CREATE OR REPLACE PROCEDURE format_phone
   (p_phone_no IN OUT VARCHAR2) IS
BEGIN
  p_phone_no := '('   || SUBSTR(p_phone_no,1,3) ||
                ') '  || SUBSTR(p_phone_no,4,3) ||
                '-'   || SUBSTR(p_phone_no,7);
END format_phone;
/
```

```
anonymous block completed
B_PHONE_NO
----------
8006330575

anonymous block completed
B_PHONE_NO
--------------
(800) 633-0575
```

Viewing the OUT Parameters: Using the DBMS_OUTPUT.PUT_LINE Subroutine

Use PL/SQL variables that are printed with calls to the DBMS_OUTPUT.PUT_LINE procedure.

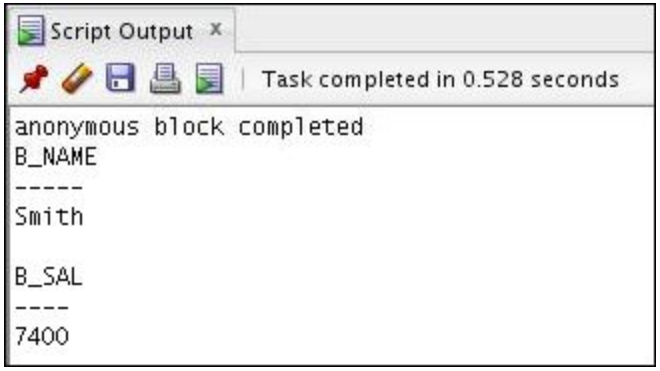
```
SET SERVEROUTPUT ON

DECLARE
  v_emp_name employees.last_name%TYPE;
  v_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, v_emp_name, v_emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || v_emp_sal);
END;
```

```
anonymous block completed
Name: Smith
Salary: 7400
```

Viewing OUT Parameters: Using SQL*Plus Host Variables

1. Use SQL*Plus host variables.
2. Execute QUERY_EMP using host variables.
3. Print the host variables.

```
VARIABLE b_name  VARCHAR2(25)
VARIABLE b_sal       NUMBER
EXECUTE query_emp(171, :b_name, :b_sal)
PRINT b_name b_sal
```

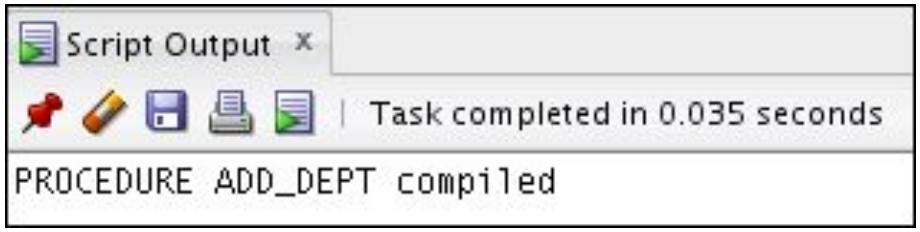


Available Notations for Passing Actual Parameters

- When calling a subprogram, you can write the actual parameters using the following notations:
  - Positional: Lists the actual parameters in the same order as the formal parameters
  - Named: Lists the actual parameters in arbitrary order and uses the association operator (=>) to associate a named formal parameter with its actual parameter
  - Mixed: Lists some of the actual parameters as positional and some as named
- Prior to Oracle Database 11*g*, only the positional notation is supported in calls from SQL.
- Starting in Oracle Database 11*g*, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements.

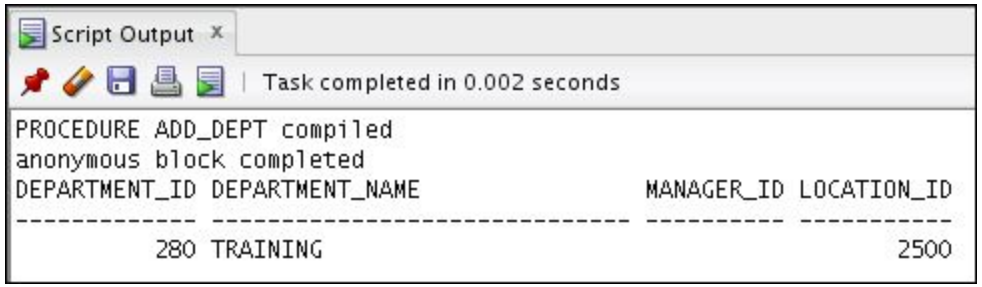
Passing Actual Parameters: Creating the add_dept Procedure

```
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;
/
```

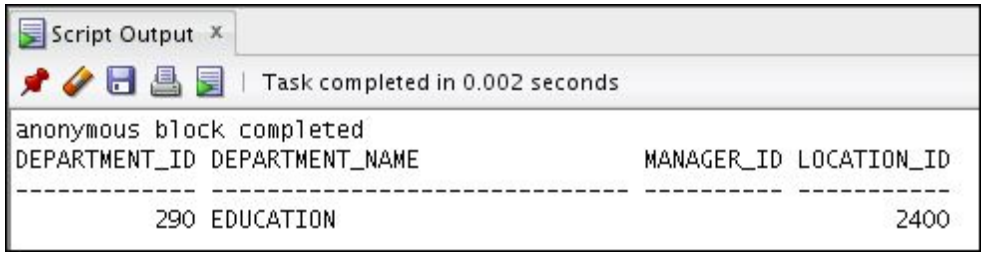


Passing Actual Parameters: Examples

```
-- 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')
```



Using the DEFAULT Option for the Parameters

- Defines default values for parameters
- Provides flexibility by combining the positional and named parameter-passing syntax

```
CREATE OR REPLACE PROCEDURE add_dept(
 p_name departments.department_name%TYPE:='Unknown',
 p_loc  departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments (department_id,
    department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
```

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', p_loc => 1200)
EXECUTE add_dept (p_loc => 1200)
```

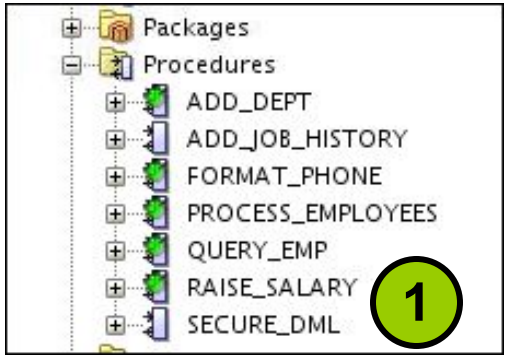
Calling Procedures

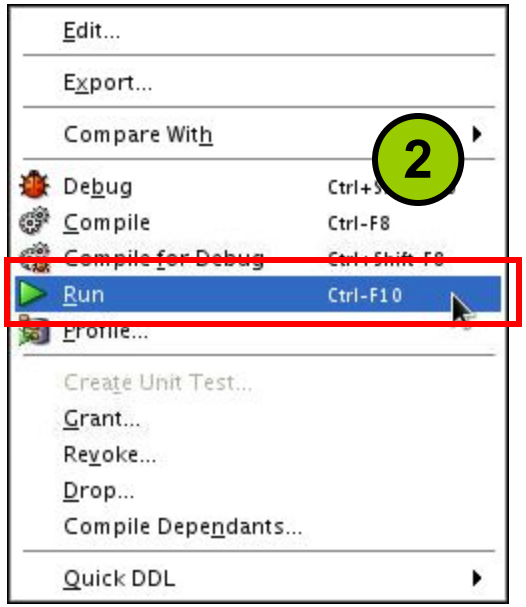
- You can call procedures using anonymous blocks, another procedure, or packages.
- You must own the procedure or have the `EXECUTE` privilege.

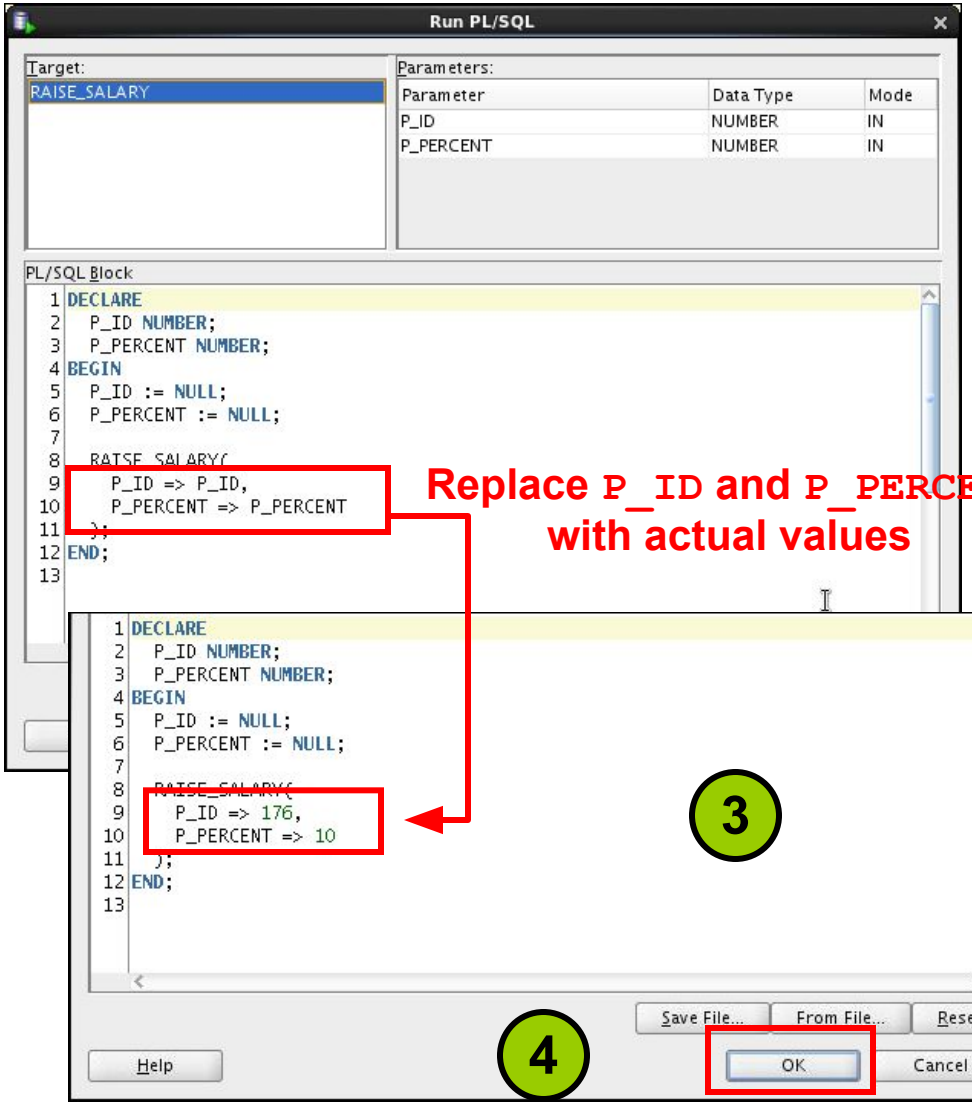
```
CREATE OR REPLACE PROCEDURE process_employees
IS
   CURSOR cur_emp_cursor IS
      SELECT employee_id
      FROM   employees;
BEGIN
   FOR emp_rec IN cur_emp_cursor
   LOOP
     raise_salary(emp_rec.employee_id, 10);
   END LOOP;

   COMMIT;
END process_employees;
/
```

```
PROCEDURE PROCESS_EMPLOYEES compiled
```

Calling Procedures Using SQL Developer



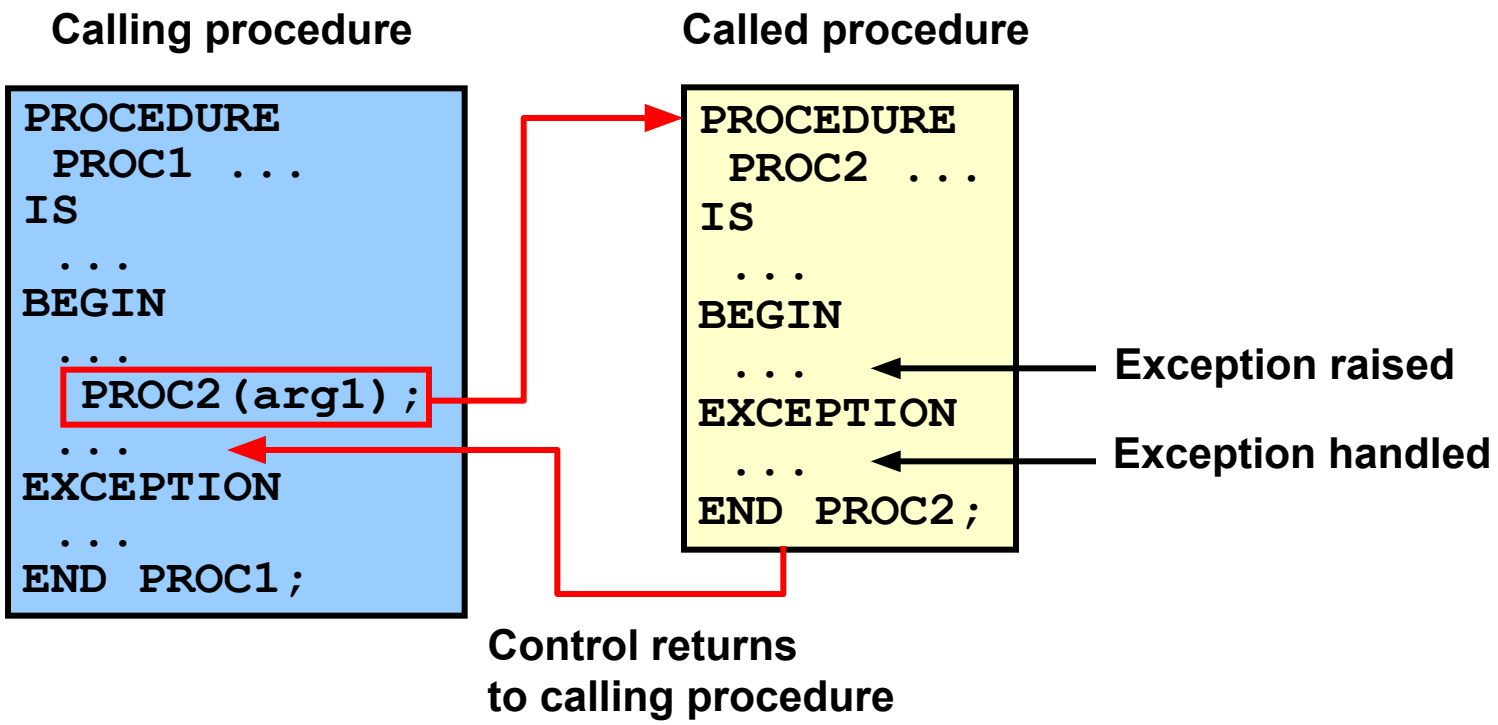




Lesson Agenda

- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
  - Creating and calling procedures
  - Identifying the available parameter-passing modes
  - Using formal and actual parameters
  - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedure's information

Handled Exceptions

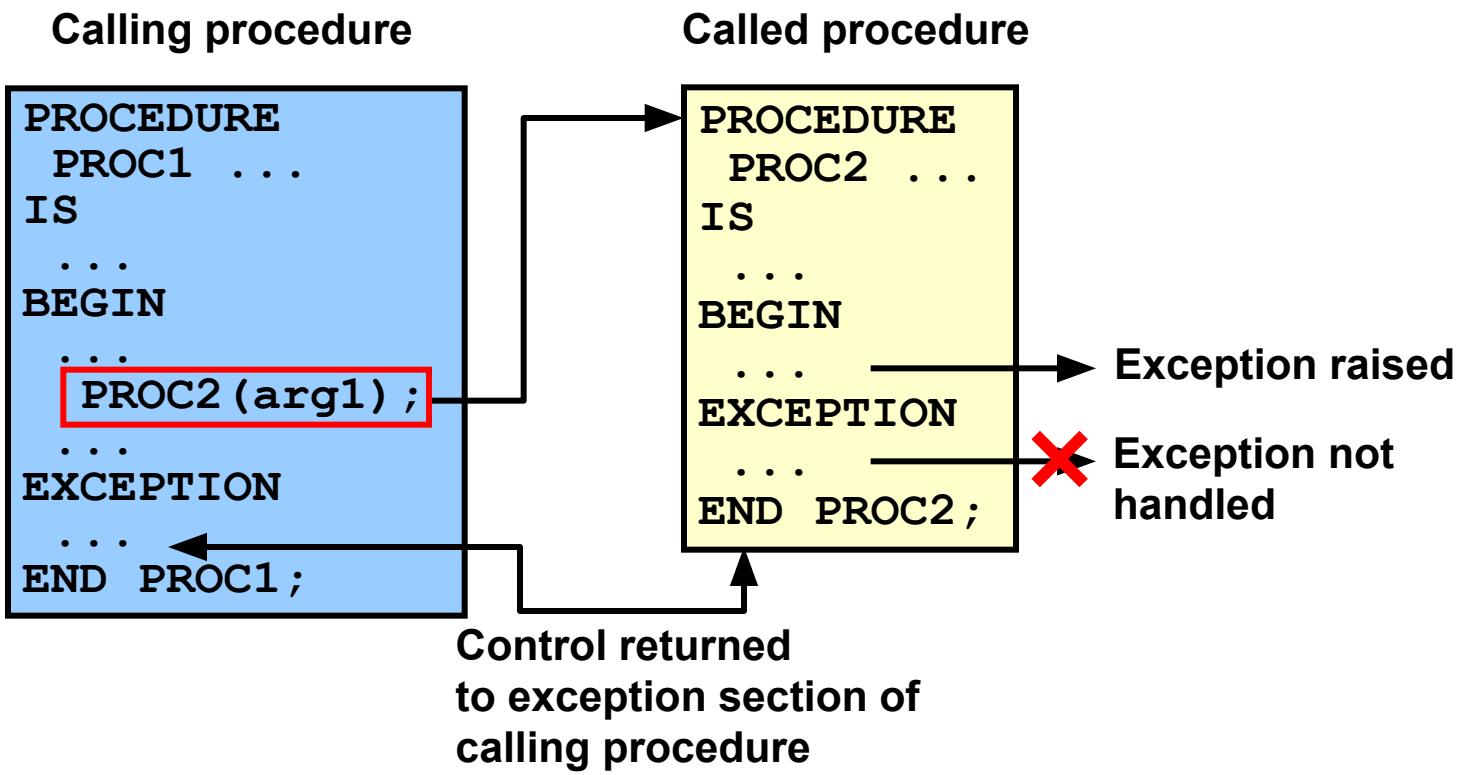


Handled Exceptions: Example

```
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;
```

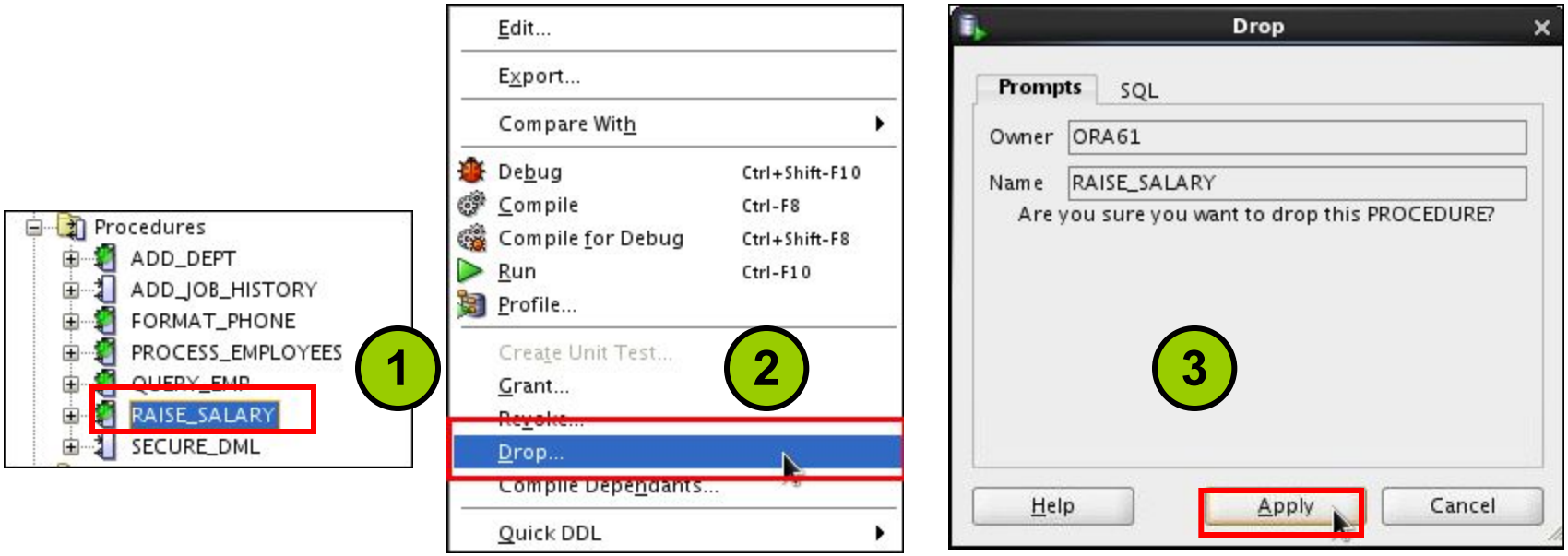
```
CREATE PROCEDURE create_departments_noex IS
BEGIN
  add_department_noex('Media', 100, 1800);
  add_department_noex('Editing', 99, 1800);
  add_department_noex('Advertising', 101, 1800);
END;
```

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

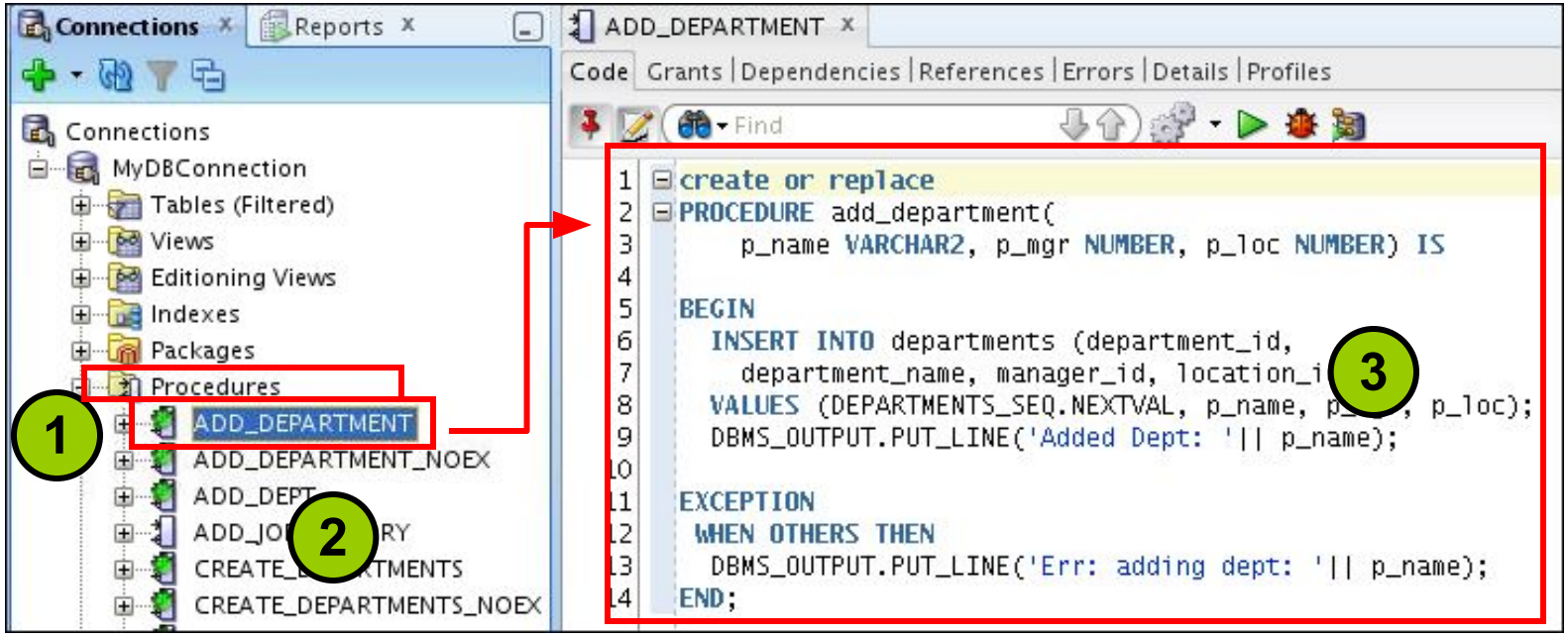
```
DESCRIBE user_source
```

```
DESCRIBE user_source
Name Null Type
---- ---- -------------
NAME      VARCHAR2(128)
TYPE      VARCHAR2(12)
LINE      NUMBER
TEXT      VARCHAR2(4000)
```

```
SELECT text
FROM   user_source
WHERE  name = 'ADD_DEPT' AND type = 'PROCEDURE'
ORDER BY line;
```

	TEXT
1	PROCEDURE add_dept(
2	p_name departments.department_name%TYPE:='Unknown',
3	p_loc departments.location_id%TYPE DEFAULT 1700) 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;

Viewing Procedures Information Using SQL Developer



Quiz

Formal parameters are literal values, variables, and expressions used in the parameter list of the calling subprogram.

a. True

b. False

Summary

In this lesson, you should have learned how to:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure
- Display the procedure's information

Practice 2 Overview: Creating, Compiling, and Calling Procedures

This practice covers the following topics:

- Creating stored procedures to:
  - Insert new rows into a table using the supplied parameter values
  - Update data in a table for rows that match the supplied parameter values
  - Delete rows from a table that match the supplied parameter values
  - Query a table and retrieve data based on supplied parameter values
- Handling exceptions in procedures
- Compiling and invoking procedures 8880342444