10

Introducing Stored Procedures and Functions

Objectives

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

- Differentiate between anonymous blocks and subprograms
- Create a simple procedure and invoke it from an anonymous block
- Create a simple function
- Create a simple function that accepts a parameter
- Differentiate between procedures and functions

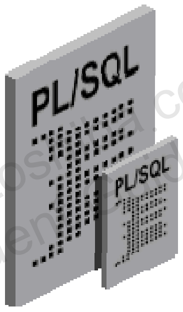
You learned about anonymous blocks. This lesson introduces you to named blocks, which are also called *subprograms*. Procedures and functions are PL/SQL subprograms. In the lesson, you learn to differentiate between anonymous blocks and subprograms.

Agenda

- Introducing procedures and functions
- Previewing procedures
- Previewing functions

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without the `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



Up to this point, anonymous blocks were the only examples of PL/SQL code covered in this course. As the name indicates, *anonymous* blocks are unnamed executable PL/SQL blocks. Because they are unnamed, they can be neither reused nor stored for later use.

Procedures and functions are named PL/SQL blocks that are also known as *subprograms*. These subprograms are compiled and stored in the database. The block structure of the subprograms is similar to the structure of anonymous blocks. Subprograms can be declared not only at the schema level but also within any other PL/SQL block. A subprogram contains the following sections:

- **Declarative section:** Subprograms can have an optional declarative section. However, unlike anonymous blocks, the declarative section of a subprogram does not start with the `DECLARE` keyword. The optional declarative section follows the `IS` or `AS` keyword in the subprogram declaration.
- **Executable section:** This is the mandatory section of the subprogram, which contains the implementation of the business logic. Looking at the code in this section, you can easily determine the business functionality of the subprogram. This section begins and ends with the `BEGIN` and `END` keywords, respectively.
- **Exception section:** This is an optional section that is included to handle exceptions.

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	If functions, must return values
Cannot take parameters	Can take parameters

The table in the slide not only shows the differences between anonymous blocks and subprograms, but also highlights the general benefits of subprograms.

Anonymous blocks are not persistent database objects. They are compiled every time they are to be executed. They are not stored in the database for reuse. If you want to reuse them, you must rerun the script that creates the anonymous block, which causes recompilation and execution.

Procedures and functions are compiled and stored in the database in a compiled form. They are recompiled only when they are modified. Because they are stored in the database, any application can make use of these subprograms based on appropriate permissions. The calling application can pass parameters to the procedures if the procedure is designed to accept parameters. Similarly, a calling application can retrieve a value if it invokes a function or a procedure.

Agenda

- Introducing procedures and functions
- Previewing procedures
- Previewing functions

Procedure: Syntax

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

The slide shows the syntax for creating procedures. In the syntax:

procedure_name	Is the name of the procedure to be created
argument	Is the name given to the procedure parameter. Every argument is associated with a mode and data type. You can have any number of arguments separated by commas.

mode	Mode of argument:
	`IN` (default)
	`OUT`
	`IN OUT`

datatype	Is the data type of the associated parameter. The data type of parameters cannot have explicit size; instead, use `%TYPE`.
Procedure_body	Is the PL/SQL block that makes up the code

The argument list is optional in a procedure declaration. You learn about procedures in detail in the course titled *Oracle Database: Develop PL/SQL Program Units*.

Creating a Procedure

```
...
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;
```

In the code example, the `add_dept` procedure inserts a new department with department ID `280` and department name `ST-Curriculum`.

In addition, the example shows the following:

- The declarative section of a procedure starts immediately after the procedure declaration and does not begin with the `DECLARE` keyword.
- The procedure declares two variables, `dept_id` and `dept_name`.
- The procedure uses the implicit cursor attribute or the `SQL%ROWCOUNT` SQL attribute to verify that the row was successfully inserted. A value of 1 should be returned in this case.

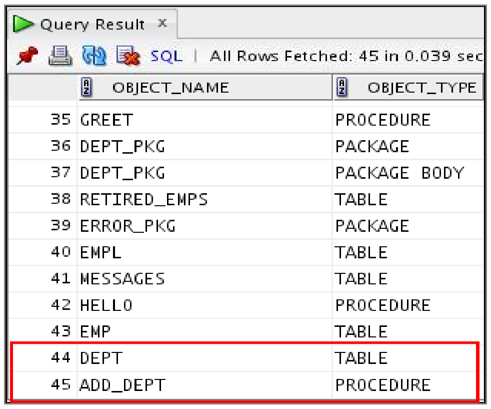
Note: See the following page for more notes on the example.

Procedure: Example

Note

- When you create any object, the entries are made to the `user_objects` table. When the code in the slide is executed successfully, you can check the `user_objects` table for the new objects by issuing the following command:

```
SELECT object_name,object_type FROM user_objects;
```

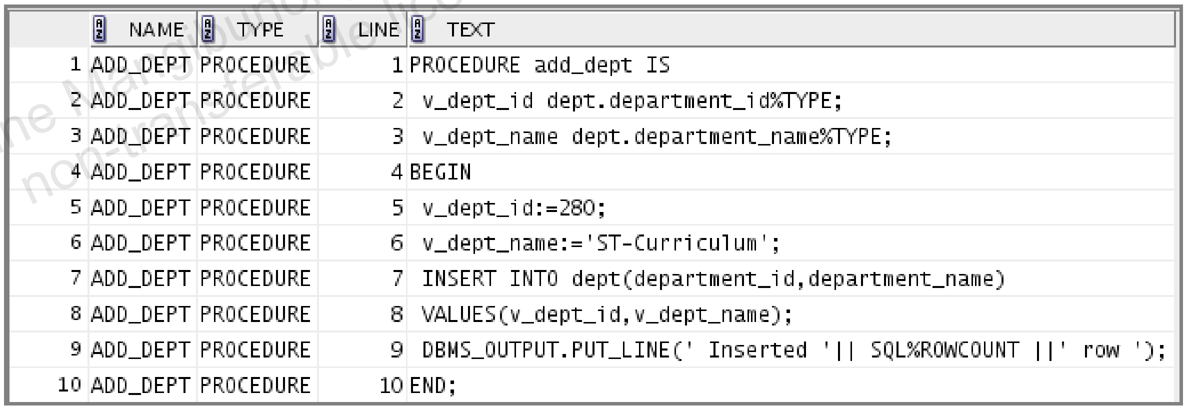


Query Result ×
SQL | All Rows Fetched: 45 in 0.039 sec

	OBJECT_NAME	OBJECT_TYPE
35	GREET	PROCEDURE
36	DEPT_PKG	PACKAGE
37	DEPT_PKG	PACKAGE BODY
38	RETIRED_EMPS	TABLE
39	ERROR_PKG	PACKAGE
40	EMPL	TABLE
41	MESSAGES	TABLE
42	HELLO	PROCEDURE
43	EMP	TABLE
44	DEPT	TABLE
45	ADD_DEPT	PROCEDURE

- The source of the procedure is stored in the `user_source` table. You can check the source for the procedure by issuing the following command:

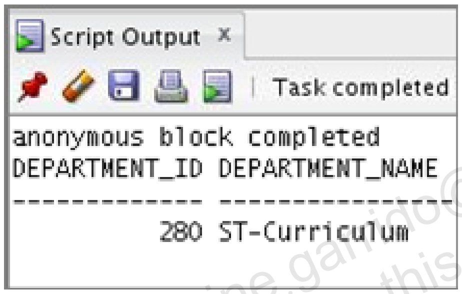
```
SELECT * FROM user_source WHERE name='ADD_DEPT';
```



	NAME	TYPE	LINE	TEXT
1	ADD_DEPT	PROCEDURE	1	PROCEDURE add_dept IS
2	ADD_DEPT	PROCEDURE	2	v_dept_id dept.department_id%TYPE;
3	ADD_DEPT	PROCEDURE	3	v_dept_name dept.department_name%TYPE;
4	ADD_DEPT	PROCEDURE	4	BEGIN
5	ADD_DEPT	PROCEDURE	5	v_dept_id:=280;
6	ADD_DEPT	PROCEDURE	6	v_dept_name:='ST-Curriculum';
7	ADD_DEPT	PROCEDURE	7	INSERT INTO dept(department_id,department_name)
8	ADD_DEPT	PROCEDURE	8	VALUES(v_dept_id,v_dept_name);
9	ADD_DEPT	PROCEDURE	9	DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT ||' row ');
10	ADD_DEPT	PROCEDURE	10	END;

Invoking a Procedure

```
...
BEGIN
 add_dept;
END;
/
SELECT department_id, department_name FROM dept
WHERE department_id=280;
```



```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME
------------- ----------------
          280 ST-Curriculum
```

The slide shows how to invoke a procedure from an anonymous block. You must include the call to the procedure in the executable section of the anonymous block. Similarly, you can invoke the procedure from any application, such as a Forms application or a Java application. The `SELECT` statement in the code checks to see whether the row was successfully inserted.

You can also invoke a procedure with the SQL statement `CALL` <*procedure_name*>.

Agenda

- Introducing procedures and functions
- Previewing procedures
- Previewing functions

Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
function_body;
```

The slide shows the syntax for creating a function. In the syntax:

function_name	Is the name of the function to be created
argument	Is the name given to the function parameter (Every argument is associated with a mode and data type. You can have any number of arguments separated by a comma. You pass the argument when you invoke the function.)
mode	Is the type of parameter (Only `IN` parameters should be declared.)
datatype	Is the data type of the associated parameter
`RETURN` *datatype*	Is the data type of the value returned by the function
function_body	Is the PL/SQL block that makes up the function code

The argument list is optional in the function declaration. The difference between a procedure and a function is that a function must return a value to the calling program. Therefore, the syntax contains *`return_type`*, which specifies the data type of the value that the function returns. A procedure may return a value via an `OUT` or `IN OUT` parameter.

Creating a Function

```
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;
```

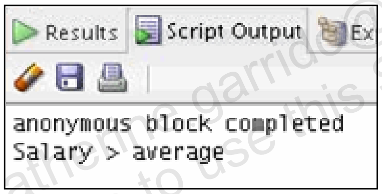
Function: Example

The `check_sal` function is written to determine whether the salary of a particular employee is greater than or less than the average salary of all employees working in the same department. The function returns `TRUE` if the salary of the employee is greater than the average salary of the employees in the department; if not, it returns `FALSE`. The function returns `NULL` if a `NO_DATA_FOUND` exception is thrown.

Note that the function checks for the employee with the employee ID 205. The function is hard-coded to check only for this employee ID. If you want to check for any other employees, you must modify the function itself. You can solve this problem by declaring the function such that it accepts an argument. You can then pass the employee ID as parameter.

Invoking a Function

```
BEGIN
 IF (check_sal IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
END;
/
```



```
anonymous block completed
Salary > average
```

You include the call to the function in the executable section of the anonymous block. The function is invoked as a part of a statement. Remember that the `check_sal` function returns `Boolean` or `NULL`. Thus the call to the function is included as the conditional expression for the `IF` block.

Note: You can use the `DESCRIBE` command to check the arguments and return type of the function, as in the following example:

```
DESCRIBE check_sal;
```

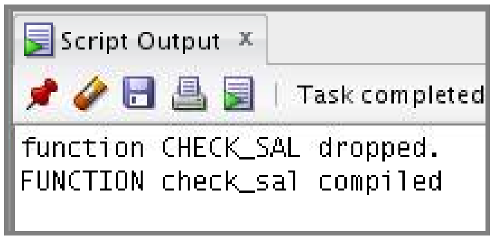
Passing a Parameter to the Function

```
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
  ...
```

Remember that the function was hard-coded to check the salary of the employee with employee ID 205. The code shown in the slide removes that constraint because it is rewritten to accept the employee number as a parameter. You can now pass different employee numbers and check for the employee's salary.

You learn more about functions in the course titled *Oracle Database: Develop PL/SQL Program Units*.

The output of the code example in the slide is as follows:

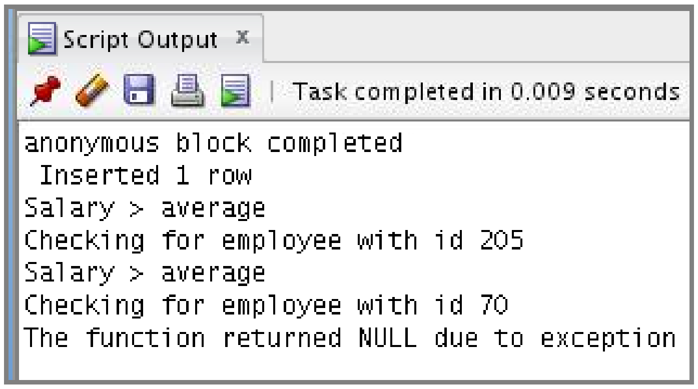


```
function CHECK_SAL dropped.
FUNCTION check_sal compiled
```

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;
/
```

The code in the slide invokes the function twice by passing parameters. The output of the code is as follows:



```
anonymous block completed
 Inserted 1 row
Salary > average
Checking for employee with id 205
Salary > average
Checking for employee with id 70
The function returned NULL due to exception
```

Quiz

Subprograms:

a. Are named PL/SQL blocks and can be invoked by other applications
b. Are compiled only once
c. Are stored in the database
d. Do not have to return values if they are functions
e. Can take parameters

Answer: a, b, c, e

Summary

In this lesson, you should have learned to:

- Create a simple procedure
- Invoke the procedure from an anonymous block
- Create a simple function
- Create a simple function that accepts parameters
- Invoke the function from an anonymous block

You can use anonymous blocks to design any functionality in PL/SQL. However, the major constraint with anonymous blocks is that they are not stored and, therefore, cannot be reused.

Instead of creating anonymous blocks, you can create PL/SQL subprograms. Procedures and functions are called subprograms, which are named PL/SQL blocks. Subprograms express reusable logic by virtue of parameterization. The structure of a procedure or function is similar to the structure of an anonymous block. These subprograms are stored in the database and are, therefore, reusable.

Practice 10: Overview

This practice covers the following topics:

- Converting an existing anonymous block to a procedure
- Modifying the procedure to accept a parameter
- Writing an anonymous block to invoke the procedure