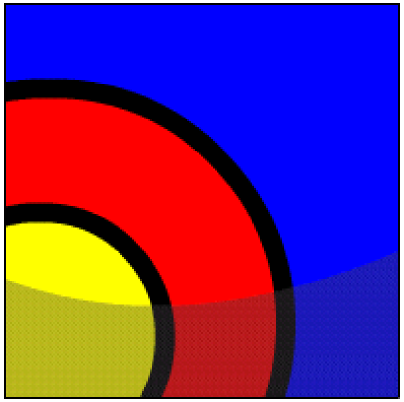
Creating Procedures

What Will I Learn?

In this lesson, you will learn to:

- Differentiate between anonymous blocks and subprograms
- Identify the benefits of subprograms
- Define a stored procedure
- Create a procedure
- Describe how a stored procedure is invoked
- List the development steps for creating a procedure



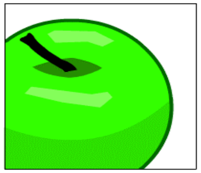
Why Learn It?

Up to now in this course, you have learned how to write and execute anonymous PL/SQL blocks. Anonymous blocks are written as part of the application program.

In this and the next two sections, you will learn how to create, execute, and manage PL/SQL subprograms. These are stored in the database, giving many benefits, such as shareability, better security, and faster performance.

There are two kinds of PL/SQL subprograms: procedures and functions. In this lesson, you learn how to create and execute stored procedures.

Tell Me / Show Me



Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks

The only kind of PL/SQL blocks that have been introduced in this course so far are anonymous blocks. As the word "anonymous" indicates, anonymous blocks are unnamed executable PL/SQL blocks. Because they are unnamed, they can neither be reused nor stored in the database for later use.

While you can store anonymous blocks on your PC, the database is not aware of them, so no one else can share them.

Tell Me / Show Me

Differences Between Anonymous Blocks and Subprograms

Subprograms

Procedures and functions are named PL/SQL blocks. They 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.

While subprograms can be explicitly shared, the default is to make them private to the owner's schema.

Later subprograms become the building blocks of packages and triggers.

Tell Me / Show Me

Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks

```
DECLARE   (Optional)
   Variables, cursors, etc.;
BEGIN     (Mandatory)
   SQL and PL/SQL statements;
EXCEPTION (Optional)
   WHEN exception-handling actions;
END;      (Mandatory)
```

Subprograms (Procedures)

```
CREATE [OR REPLACE] PROCEDURE name [parameters] IS|AS
(Mandatory)
  Variables, cursors, etc.; (Optional)
BEGIN     (Mandatory)
   SQL and PL/SQL statements;
EXCEPTION (Optional)
   WHEN exception-handling actions;
END [name]; (Mandatory)
```

Tell Me / Show Me

Differences Between Anonymous Blocks and Subprograms (continued)

Anonymous Blocks	**Subprograms**
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled on every execution	Compiled only once, when created
Not stored in the database	Stored in the database
Cannot be invoked by other applications	They are 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

Tell Me / Show Me

Benefits of Subprograms

Procedures and functions have many benefits due to the modularizing of the code:

- Easy maintenance: Modifications need only be done once to improve multiple applications and minimize testing.

- Code reuse: Subprograms are located in one place. When compiled and validated, they can be used and reused in any number of applications.

Tell Me / Show Me

Benefits of Subprograms (continued)

- Improved data security: Indirect access to database objects is permitted by the granting of security privileges on the subprograms. By default, subprograms run with the privileges of the subprogram owner, not the privileges of the user.

- Data integrity: Related actions can be grouped into a block and are performed together (“Statement Processed”) or not at all.

Tell Me / Show Me

Benefits of Subprograms (Continued)

- Improved performance: You can reuse compiled PL/SQL code that is stored in the shared SQL area cache of the serve. Subsequent calls to the subprogram avoid compiling the code again. Also, many users can share a single copy of the subprogram code in memory.

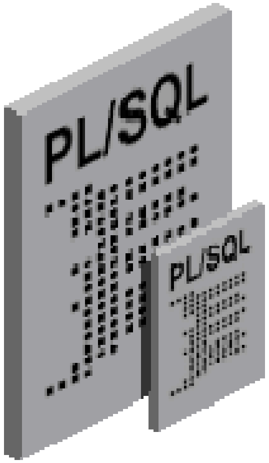
- Improved code clarity: By using appropriate names and conventions to describe the action of the routines, you can reduce the need for comments, and enhance the clarity of the code.

Tell Me / Show Me

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional parameters
  - Optional declarative section (but the `DECLARE` keyword changes to `IS` or `AS`)
  - Mandatory executable section
  - Optional section to handle exceptions

This section focuses on procedures.



Tell Me / Show Me

What Is a Procedure?

- A procedure is a named PL/SQL block that can accept parameters.
- Generally, you use a procedure to perform an action (sometimes called a “side-effect”).
- A procedure is compiled and stored in the database as a schema object.
  - Shows up in `USER_OBJECTS` as an object type of `PROCEDURE`
  - More details in `USER_PROCEDURES`
  - Detailed PL/SQL code in `USER_SOURCE`

Tell Me / Show Me

Syntax for Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode1] datatype1,
   parameter2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```

- Parameters are optional
- Mode defaults to `IN`
- Datatype can be either explicit (for example, `VARCHAR2`) or implicit with `%TYPE`
- Body is the same as an anonymous block

Tell Me / Show Me

Syntax for Creating Procedures (continued)

- Use CREATE PROCEDURE followed by the name, optional parameters, and keyword IS or AS.
- Add the OR REPLACE option to overwrite an existing procedure.
- Write a PL/SQL block containing local variables, a BEGIN, and an END (or END *procedure_name*).

```
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

Tell Me / Show Me

Procedure: Example

In the following example, the add_dept procedure inserts a new department with the department_id 280 and department_name ST-Curriculum. The procedure declares two variables, v_dept_id and v_dept_name, in the declarative section.

```
CREATE TABLE dept AS SELECT * FROM departments;
CREATE OR REPLACE 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;
```

Tell Me / Show Me

Procedure: Example (continued)

The declarative section of a procedure starts immediately after the procedure declaration and does not begin with the keyword DECLARE. This procedure uses the SQL%ROWCOUNT cursor attribute to check if the row was successfully inserted. SQL%ROWCOUNT should return 1 in this case.

```
CREATE TABLE dept AS SELECT * FROM departments;
CREATE OR REPLACE 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;
```

Tell Me / Show Me

Invoking Procedures

You can invoke (execute) a procedure from:

- An anonymous block
- Another procedure
- A calling application

Note: You CANNOT invoke a procedure from inside a SQL statement such as SELECT.

Tell Me / Show Me

Invoking the Procedure from Application Express

To invoke (execute) a procedure in Oracle Application Express, write and run a small anonymous block that invokes the procedure. For example:

```
CREATE OR REPLACE PROCEDURE add_dept IS ...

BEGIN
  add_dept;
END;

SELECT department_id, department_name FROM dept
WHERE department_id=280;
```

The select statement at the end confirms that the row was successfully inserted.

Tell Me / Show Me

Correcting Errors in CREATE PROCEDURE Statements

If compilation errors exist, Application Express displays them in the output portion of the SQL Commands window. You must edit the source code to make corrections. The procedure is still created even though it contains errors.

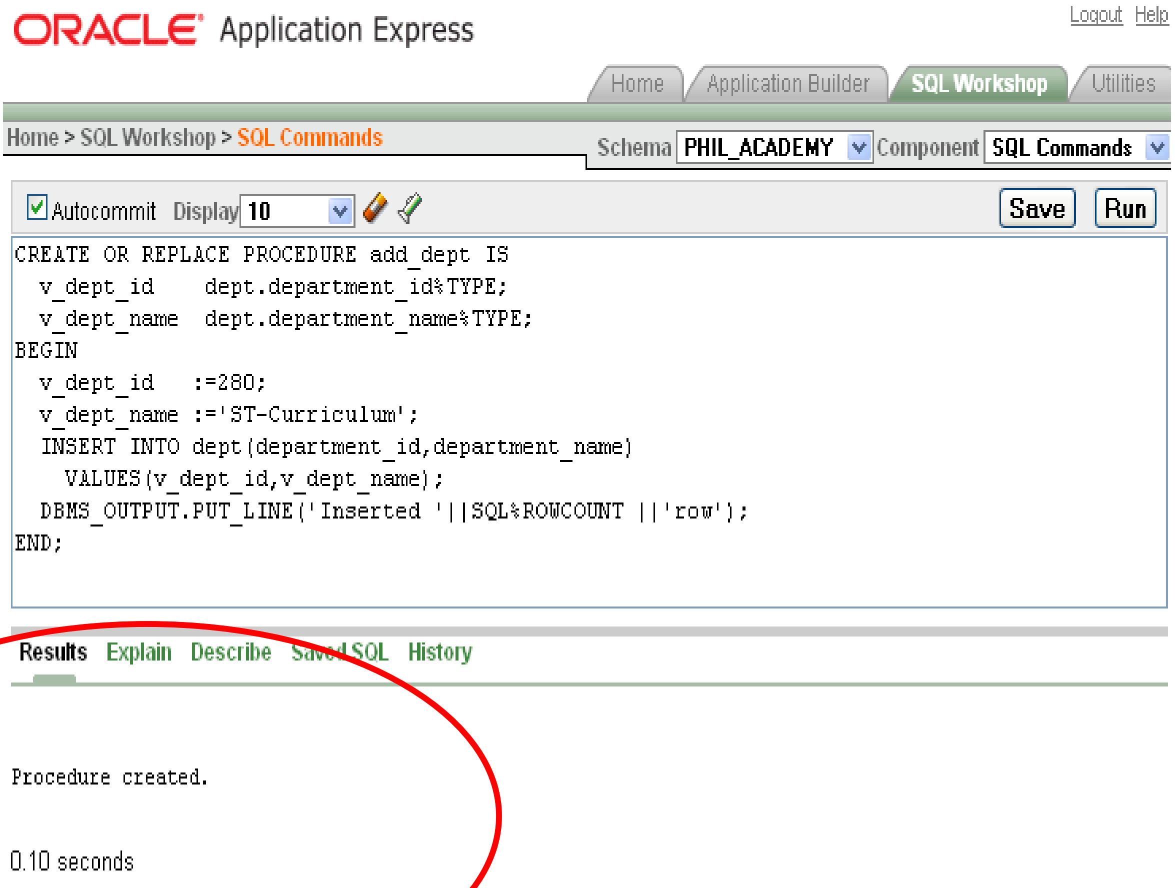
After you have corrected the error in the code, you need to recreate the procedure. There are two ways to do this:

- Use a CREATE or REPLACE PROCEDURE statement to overwrite the existing code (most common)
- DROP the procedure first and then execute the CREATE PROCEDURE statement (less common).

Tell Me / Show Me

Saving Your Work

Once a procedure has been created successfully, you should save its definition in case you need to modify the code later.



```
CREATE OR REPLACE 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;
```

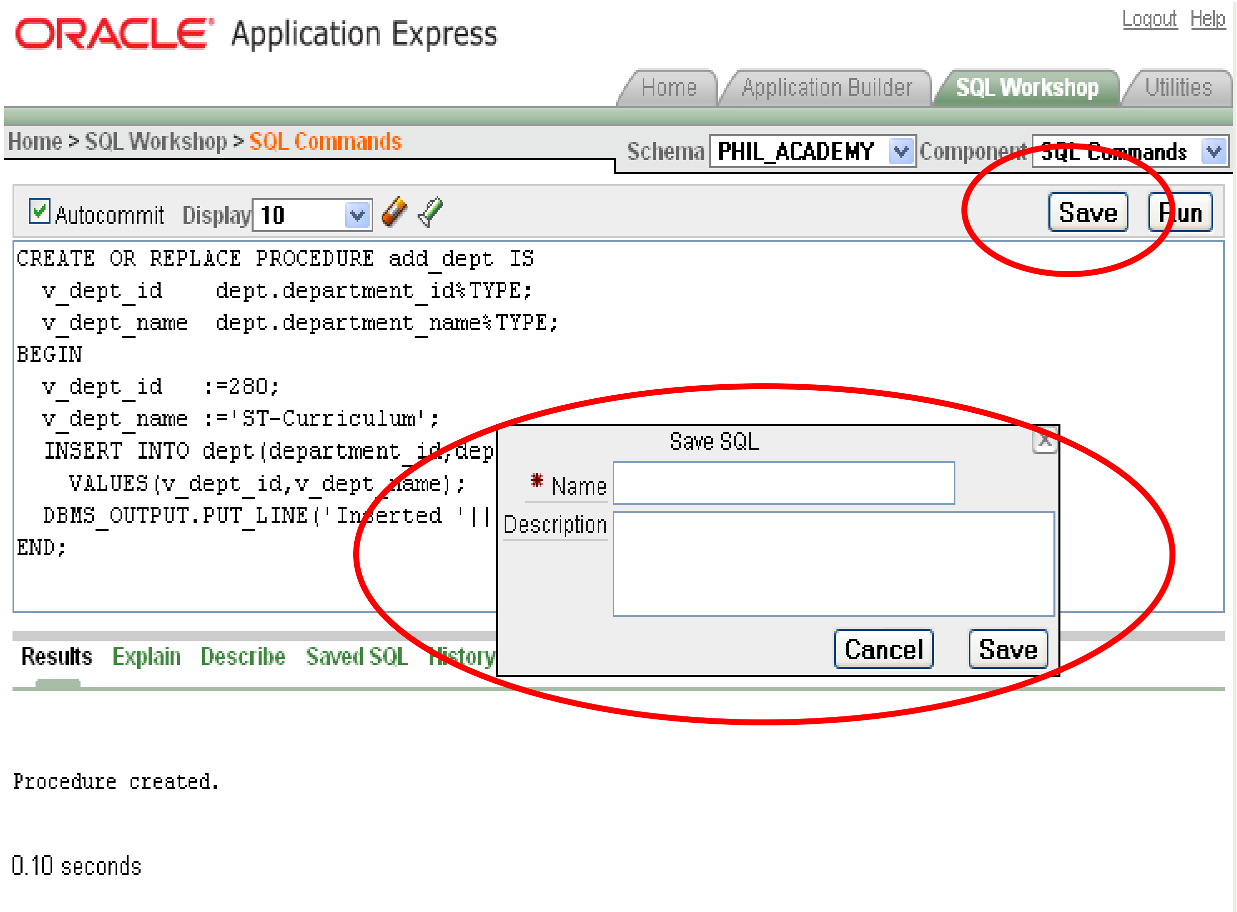
```
Procedure created.

0.10 seconds
```

Tell Me / Show Me

Saving Your Work

In the Application Express SQL Commands window, click the SAVE button and enter a name and optional description for your code.



Schema PHIL_ACADEMY Component SQL Commands

☑ Autocommit Display 10

```
CREATE OR REPLACE 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,dep
    VALUES(v_dept_id,v_dept_name);
  DBMS_OUTPUT.PUT_LINE('Inserted '||
END;
```

* Name

Description

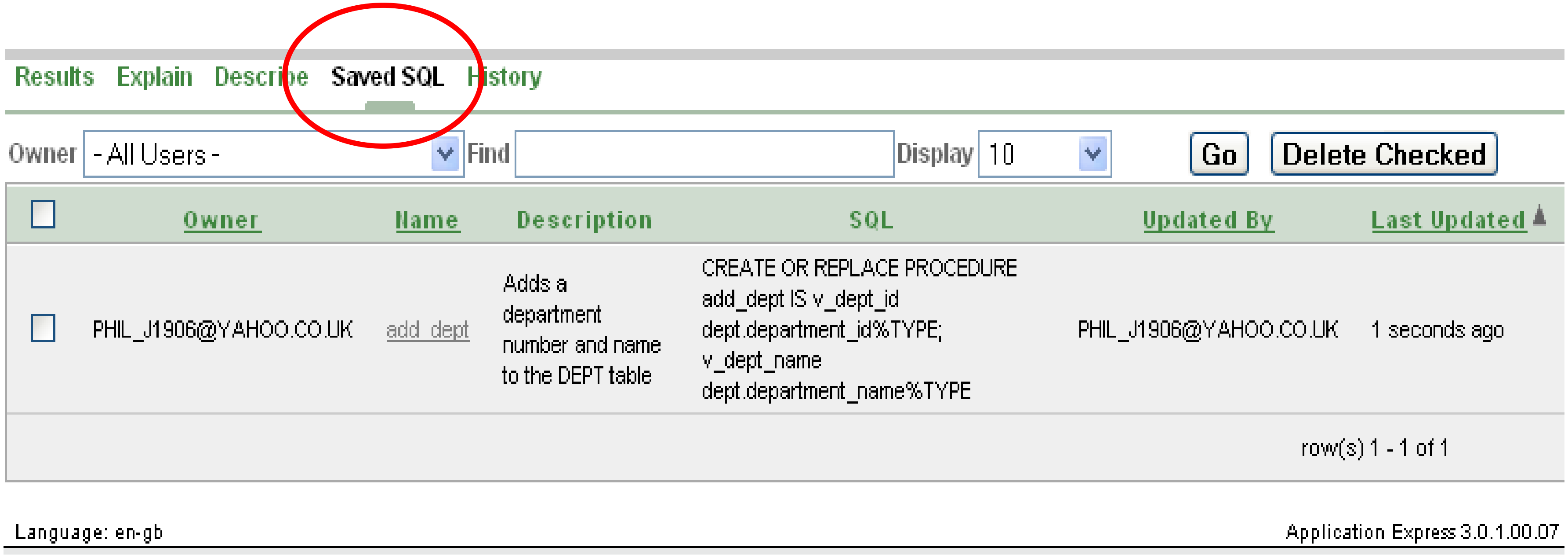
```
Procedure created.

0.10 seconds
```

Tell Me / Show Me

Saving Your Work

You can view and reload your code later by clicking on the Saved SQL button in the SQL Commands window.



□	Owner	Name	Description	SQL	Updated By	Last Updated
□	PHIL_J1906@YAHOO.CO.UK	add_dept	Adds a department number and name to the DEPT table	CREATE OR REPLACE PROCEDURE add_dept IS v_dept_id dept.department_id%TYPE; v_dept_name dept.department_name%TYPE	PHIL_J1906@YAHOO.CO.UK	1 seconds ago

row(s) 1 - 1 of 1

Language: en-gb

Application Express 3.0.1.00.07

Tell Me / Show Me

Alternative Tools for Developing Procedures

If you end up writing PL/SQL procedures for a living, there are other free tools that can make this process easier. For instance, Oracle tools, such as SQL Developer and JDeveloper assist you by:

- Color-coding **`commands`** vs **`variables`** vs **`constants`**
- Highlighting matched and mismatched **((**parentheses**)**
- Displaying errors more graphically
- Enhancing code with standard indentations and capitalization
- Completing commands when typing
- Completing column names from tables

Tell Me / Show Me

Terminology

Key terms used in this lesson include:

Anonymous blocks

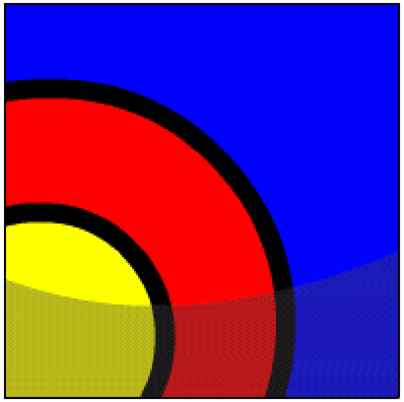
Subprograms

Procedures

Summary

In this lesson, you learned to:

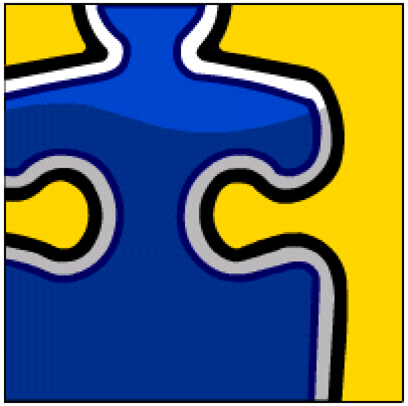
- Differentiate between anonymous blocks and subprograms
- Identify the benefits of subprograms
- Define a stored procedure
- Create a procedure
- Describe how a stored procedure is invoked
- List the development steps for creating a procedure



Try It / Solve It

The exercises in this lesson cover the following topics:

- Differentiating between anonymous blocks and subprograms
- Identifying the benefits of subprograms
- Defining a stored procedure
- Creating a procedure
- Describing how a stored procedure is invoked
- Listing the development steps for creating a procedure

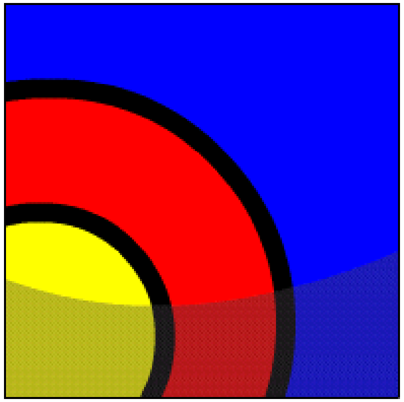


Using Parameters in Procedures

What Will I Learn?

In this lesson, you will learn to:

- Describe how parameters contribute to a procedure
- Define a parameter
- Create a procedure using a parameter
- Invoke a procedure that has parameters
- Differentiate between formal and actual parameters



Why Learn It?

To make procedures more flexible, it is important that varying data is either calculated or passed into a procedure by using input parameters. Calculated results can be returned to the caller of a procedure by using `OUT` or `IN OUT` parameters.

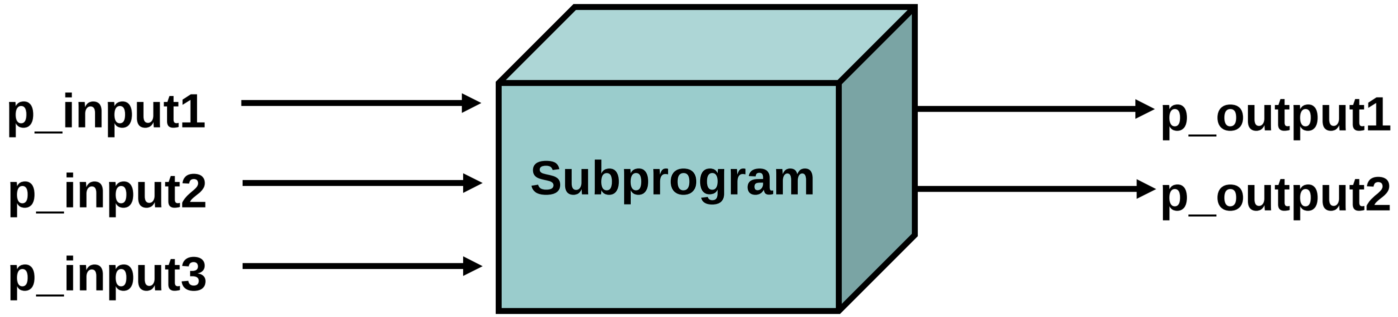
Tell Me / Show Me

What Are Parameters?

Parameters pass or communicate data between the caller and the subprogram.

You can think of parameters as a special form of a variable, whose input values are initialized by the calling environment when the subprogram is called, and whose output values are returned to the calling environment when the subprogram returns control to the caller.

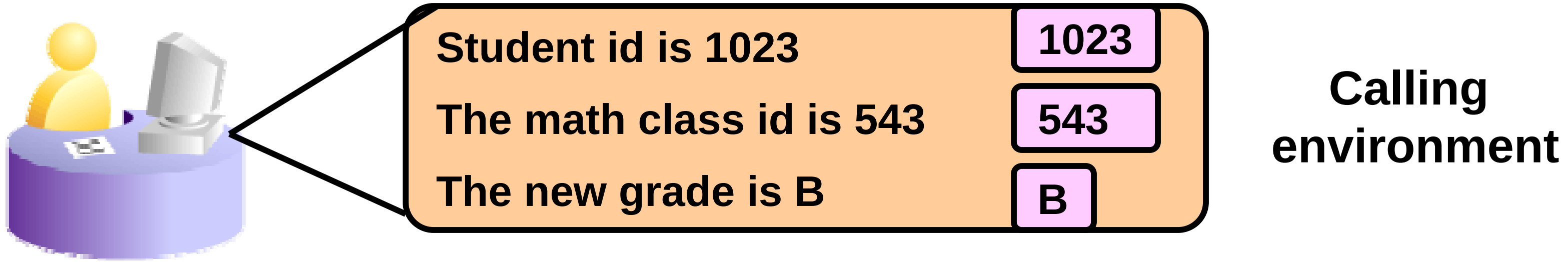
By convention, parameters are often named with a “p_” prefix.



Tell Me / Show Me

What Are Parameters? (continued)

Consider the following example where a math teacher needs to change a student's grade from a C to a B in the student administration system.



Student id is 1023 1023

The math class id is 543 543

The new grade is B B

Calling environment

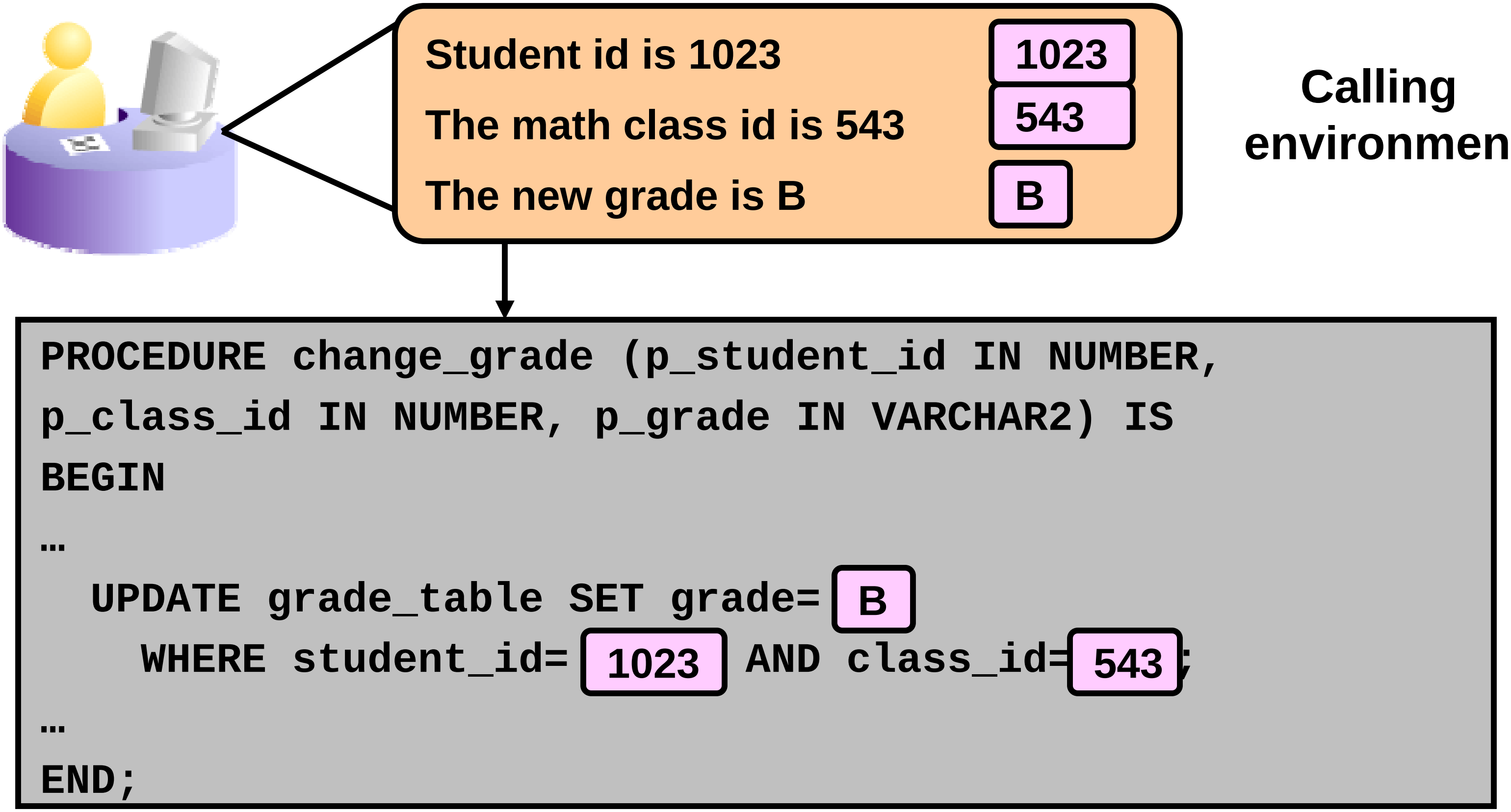
In this example, the calling system is passing values for student id, class id, and grade to a subprogram.

Do you need to know the old (before) value for the grade? Why or why not?

Tell Me / Show Me

What Are Parameters? (continued)

The `change_grade` procedure accepts three parameters: `p_student_id`, `p_class_id`, and `p_grade`. These parameters act like local variables in the `change_grade` procedure.



Student id is 1023 | 1023

The math class id is 543 | 543

The new grade is B | B

Calling environment

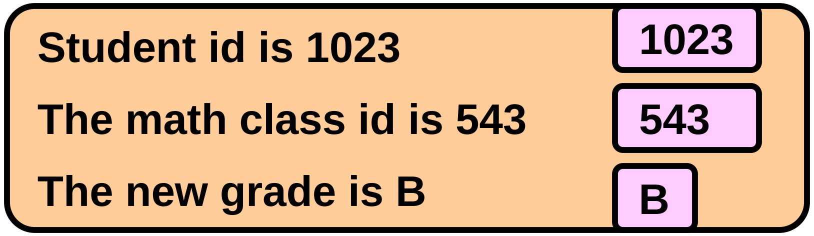
```
PROCEDURE change_grade (p_student_id IN NUMBER,
p_class_id IN NUMBER, p_grade IN VARCHAR2) IS
BEGIN
…
  UPDATE grade_table SET grade= B
    WHERE student_id= 1023 AND class_id= 543;
…
END;
```

Tell Me / Show Me

What Are Arguments?

Parameters are commonly referred to as arguments. However, arguments are more appropriately thought of as the actual values assigned to the parameter variables when the subprogram is called at runtime.

In the previous example, 1023 is an argument passed in to the `p_student_id` parameter.



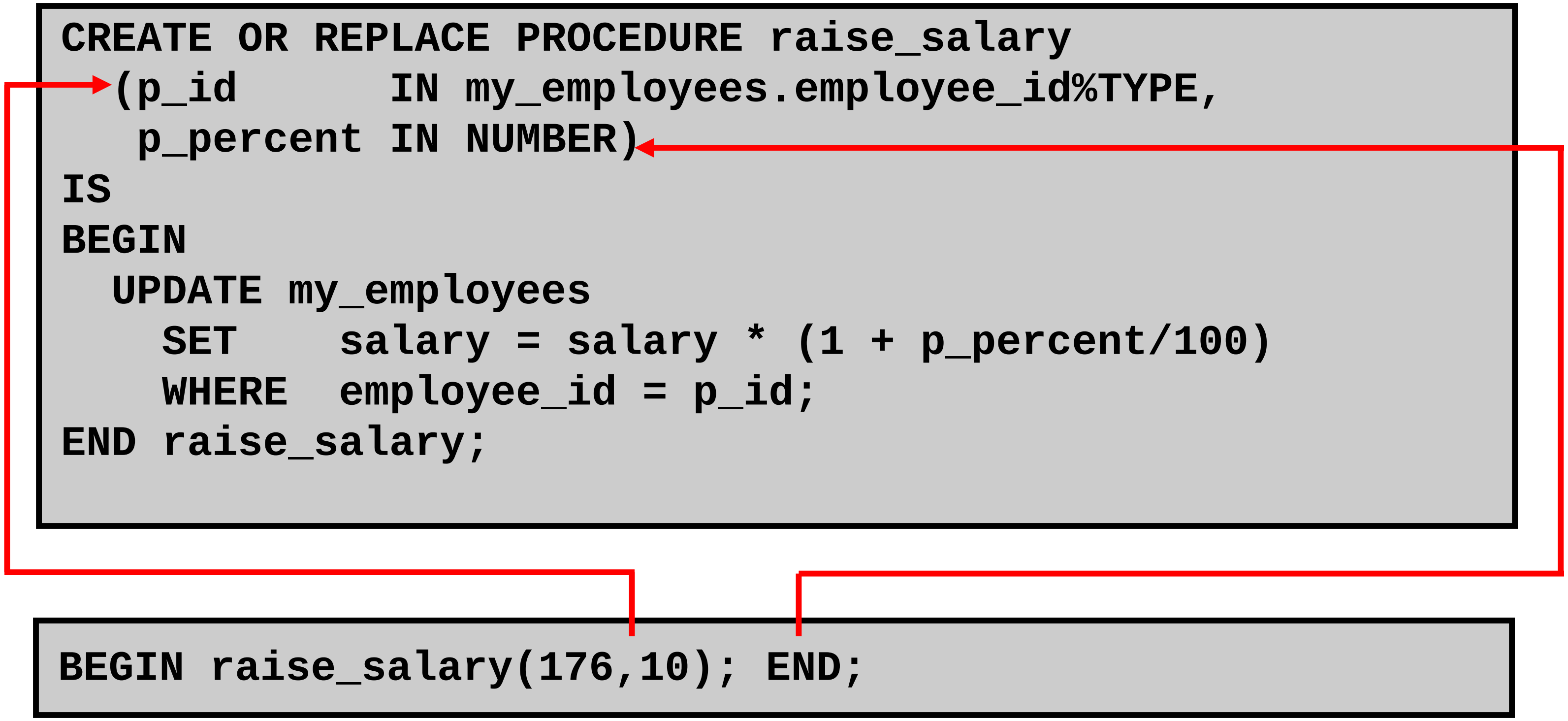
Student id is 1023	1023
The math class id is 543	543
The new grade is B	B

Even though parameters are a kind of variable, IN parameter arguments act as constants and cannot be changed by the subprogram.

Tell Me / Show Me

Creating Procedures with Parameters

The example shows a procedure with two parameters. Running this first statement creates the `raise_salary` procedure in the database. The second example executes the procedure, passing the arguments 176 and 10 to the two parameters.



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

```
BEGIN raise_salary(176,10); END;
```

Tell Me / Show Me

Invoking Procedures with Parameters

To invoke a procedure from Oracle Application Express, create an anonymous block and use a direct call inside the executable section of the block. Where you want to call the new procedure, enter the procedure name and parameter values (arguments). For example:

```
BEGIN
  raise_salary (176, 10);
END;
```

You must enter the arguments in the same order as they are declared in the procedure.

Tell Me / Show Me

Invoking Procedures with Parameters

To invoke a procedure from another procedure, use a direct call inside an executable section of the block. At the location of calling the new procedure, enter the procedure name and parameter arguments.

```
CREATE OR REPLACE PROCEDURE process_employees
IS
  CURSOR emp_cursor IS
    SELECT employee_id
      FROM   my_employees;
BEGIN
   FOR v_emp_rec IN emp_cursor
   LOOP
     raise_salary(v_emp_rec.employee_id, 10);
   END LOOP;
   COMMIT;
END process_employees;
```

Tell Me / Show Me

Types of Parameters

There are two types of parameters: Formal and Actual.

A parameter-name declared in the procedure heading is called a formal parameter. The corresponding parameter-name (or value) in the calling environment is called an actual parameter.

In the following example, can you guess which parameter is the formal parameter and which parameter is the actual parameter?

```
CREATE OR REPLACE PROCEDURE fetch_emp
  (p_emp_id IN employees.employee_id%TYPE) IS ...
END;
/* Now call the procedure from an anonymous block */
BEGIN      fetch_emp(v_emp_id);    END;
```

Tell Me / Show Me

Formal Parameters

Formal parameters are variables that are declared in the parameter list of a subprogram specification. In the following example, in the procedure raise_sal, the identifiers p_id and p_sal represent formal parameters.

```
CREATE PROCEDURE raise_sal(p_id IN NUMBER, p_sal IN
NUMBER) IS
BEGIN ...
END raise_sal;
```

Notice that the formal parameter datatypes do not have sizes. For instance p_sal is NUMBER, not NUMBER(6,2).

Tell Me / Show Me

Actual Parameters

Actual parameters can be literal values, variables, or expressions that are provided in the parameter list of a called subprogram. In the following example, a call is made to `raise_sal`, where the `a_emp_id` variable is the actual parameter for the `p_id` formal parameter, and 100 is the argument (the actual passed value).

```
a_emp_id := 100;
raise_sal(a_emp_id, 2000);
```

Actual parameters:

- Are associated with formal parameters during the subprogram call
- Can also be expressions, as in the following example: `raise_sal(a_emp_id, v_raise+100);`

Tell Me / Show Me

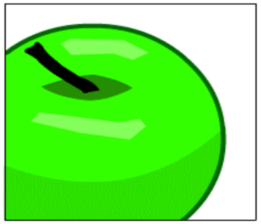
Formal and Actual Parameters

The formal and actual parameters should be of compatible data types. If necessary, before assigning the value, PL/SQL converts the data type of the actual parameter value to that of the formal parameter.

For instance, you can pass in a salary of `'1000.00'` in single quotes, so it is coming in as the *letter* 1 and the *letters* zero etc., which gets converted into the *number* one thousand. This is slower and should be avoided if possible.

You can find out the datatypes that are expected by using the command `DESCRIBE proc_name`.

Tell Me / Show Me



Terminology

Key terms used in this lesson include:

Parameters

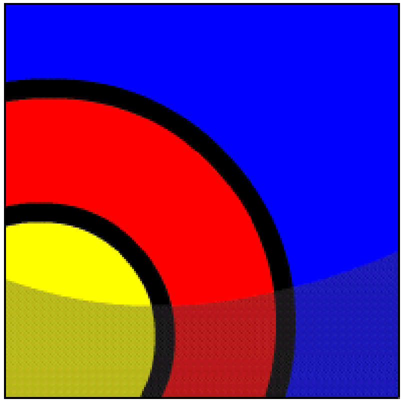
Formal parameter

Actual parameter

Summary

In this lesson, you learned to:

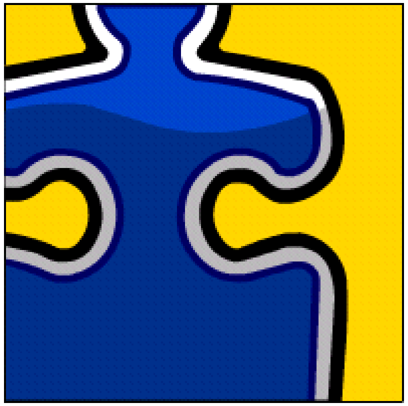
- Describe how parameters contribute to a procedure
- Define a parameter
- Create a procedure using a parameter
- Invoke a procedure that has parameters
- Differentiate between formal and actual parameters



Try It / Solve It

The exercises in this lesson cover the following topics:

- Defining what parameters are and why they are needed
- Creating a procedure with parameters
- Invoking a procedure that has parameters
- Differentiating between formal and actual parameters

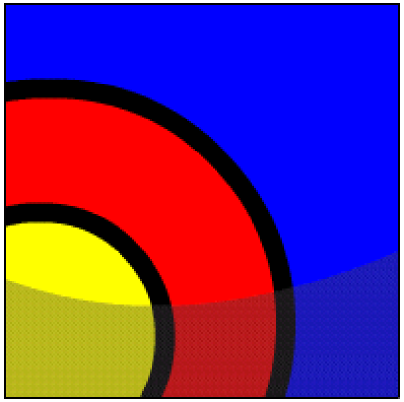


Passing Parameters

What Will I Learn?

In this lesson, you will learn to:

- List the types of parameter modes
- Create a procedure that passes parameters
- Identify three methods for passing parameters
- Describe the DEFAULT option for parameters



Why Learn It?

To make procedures more flexible, it is important that varying data is either calculated or passed into a procedure by using input parameters.

Calculated results can be returned to the caller of a procedure by using `OUT` or `IN OUT` parameters.

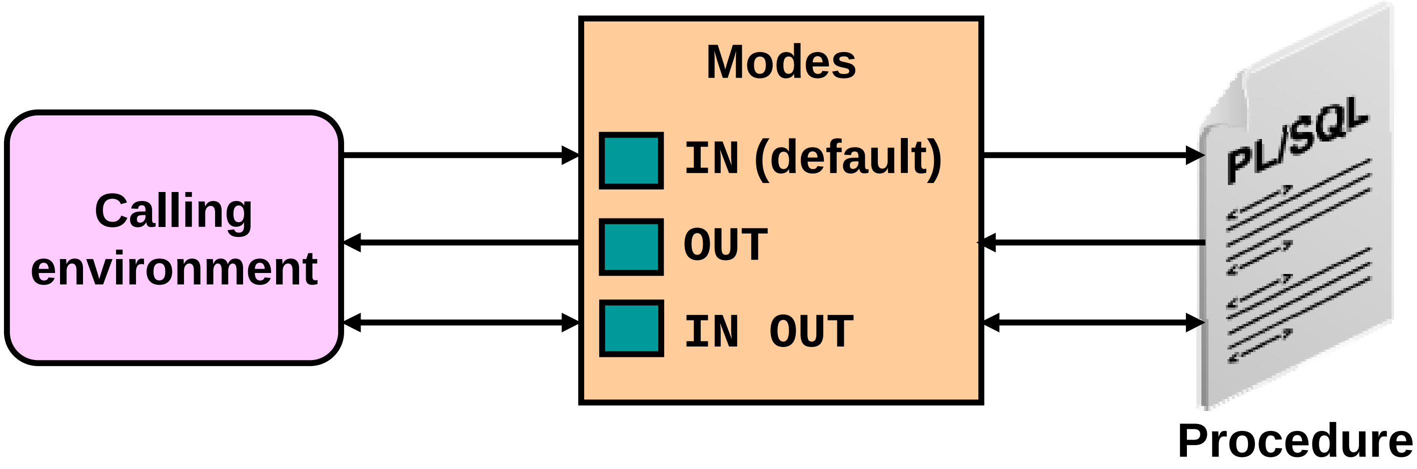
Tell Me / Show Me

Procedural Parameter Modes

Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.

Parameter-passing modes:

- An `IN` parameter (the default) provides values for a subprogram to process.
- An `OUT` parameter returns a value to the caller.
- An `IN OUT` parameter supplies an input value, which can be returned (output) as a modified value.



Tell Me / Show Me

The `IN` mode is the default if no mode is specified.

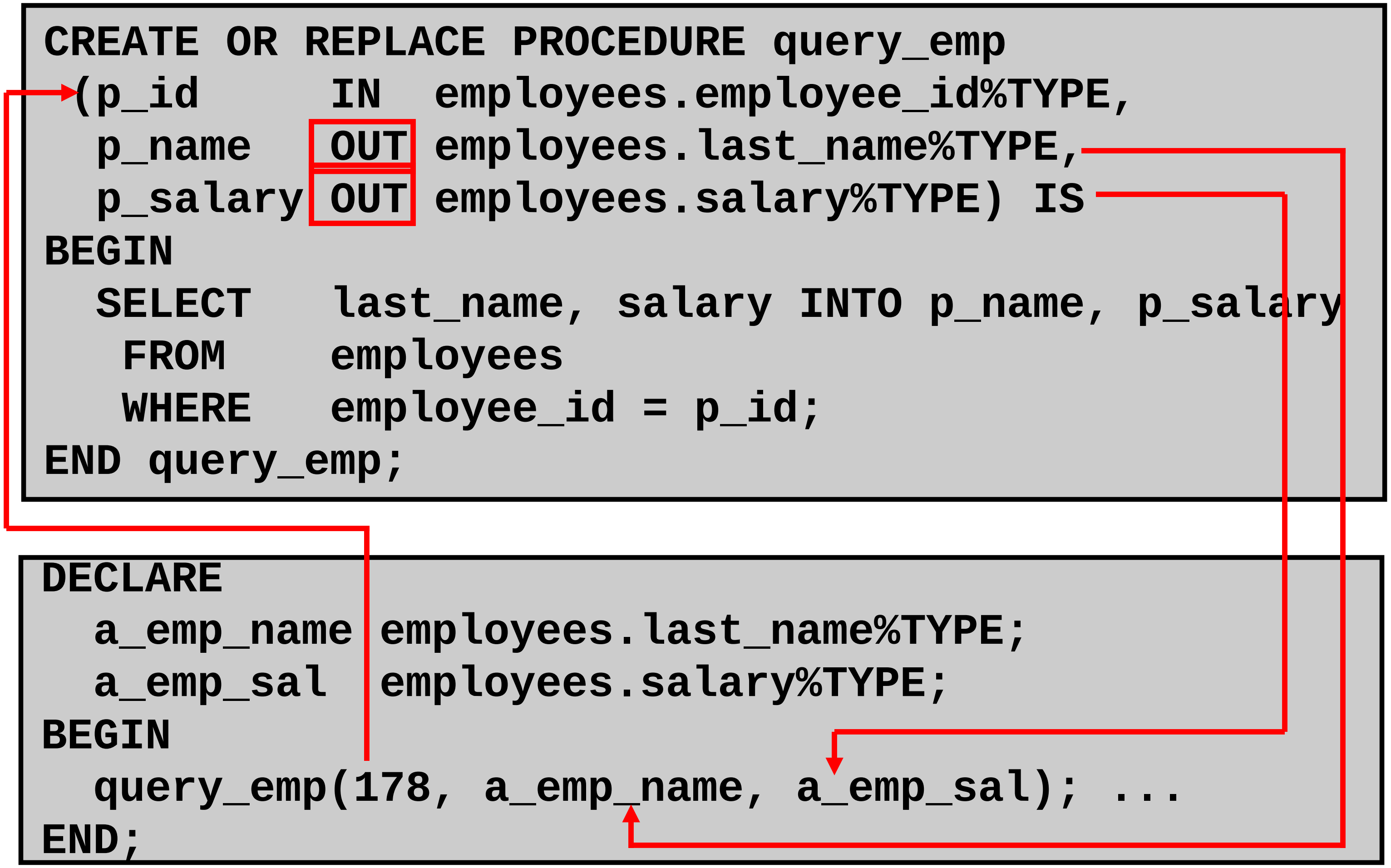
```
CREATE PROCEDURE procedure(param [mode] datatype)
...
```

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

`IN` parameters can only be read within the procedure. They cannot be modified.

Tell Me / Show Me

Using OUT Parameters: 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;
```

```
DECLARE
  a_emp_name employees.last_name%TYPE;
  a_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(178, a_emp_name, a_emp_sal); ...
END;
```

Tell Me / Show Me

Using the Previous `OUT` Example

Create a procedure with `OUT` parameters to retrieve information about an employee. The procedure accepts the value `178` for employee ID and retrieves the name and salary of the employee with ID `178` into the two `OUT` parameters. The `query_emp` procedure has three formal parameters. Two of them are `OUT` parameters that return values to the calling environment, shown in the code box at the bottom of the previous slide. The procedure accepts an employee ID value through the `p_id` parameter. The `a_emp_name` and `a_emp_sal` variables are populated with the information retrieved from the query into their two corresponding `OUT` parameters.

Note: Make sure that the data type for the actual parameter variables used to retrieve values from `OUT` parameters has a size large enough to hold the data values being returned.

Tell Me / Show Me

Viewing OUT Parameters in Application Express

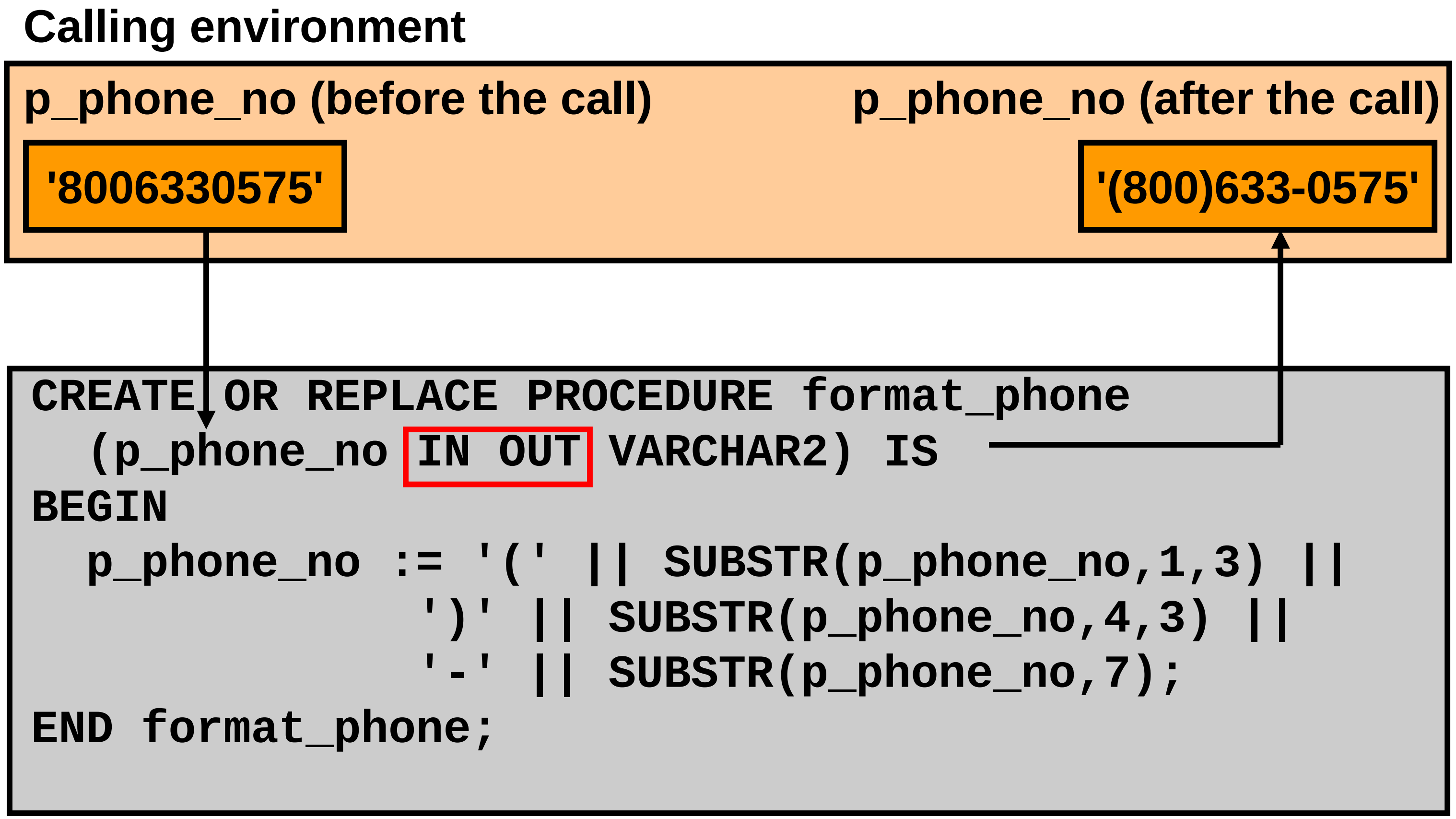
Use PL/SQL variables that are displayed with calls to the `DBMS_OUTPUT.PUT_LINE` procedure.

```
DECLARE
  a_emp_name employees.last_name%TYPE;
  a_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(178, a_emp_name, a_emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || a_emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || a_emp_sal);
END;
```

```
Name: Grant
Salary: 7700
```

Tell Me / Show Me

Using IN OUT Parameters: Example



Calling environment

p_phone_no (before the call)

'8006330575'

p_phone_no (after the call)

'(800)633-0575'

```
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;
```

Tell Me / Show Me

Using the Previous IN OUT Example

Using an `IN OUT` parameter, you can pass a value into a procedure that can be updated within the procedure. The actual parameter value supplied from the calling environment can return as either of the following:

- The original unchanged value
- A new value that is set within the procedure

The example in the previous slide creates a procedure with an `IN OUT` parameter to accept a 10-character string containing digits for a phone number. The procedure returns the phone number formatted with parentheses around the first three characters and a hyphen after the sixth digit. For example, the phone string ‘8006330575’ is returned as ‘(800)633-0575’.

Tell Me / Show Me

Calling the Previous IN OUT Example

The following code creates an anonymous block that declares a_phone_no, assigns the unformatted phone number to it, and passes it as an actual parameter to the FORMAT_PHONE procedure. The procedure is executed and returns an updated string in the a_phone_no variable, which is then displayed.

```
DECLARE
  a_phone_no VARCHAR2(13);
BEGIN
  a_phone_no := '8006330575' ;
  format_phone (a_phone_no);
  DBMS_OUTPUT.PUT_LINE('The formatted phone number is: '
                        || a_phone_no);
END;
```

Tell Me / Show Me

Summary of Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; 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

Tell Me / Show Me

Syntax for Passing Parameters

There are three ways of passing parameters from the calling environment:

- 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 (‘=>' which is an equal and an arrow together) to associate a named formal parameter with its actual parameter
- Combination:
  - Lists some of the actual parameters as positional (no special operator) and some as named (with the => operator).

Tell Me / Show Me

Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(
  p_name IN my_depts.department_name%TYPE,
  p_loc  IN my_depts.location_id%TYPE) IS
BEGIN
  INSERT INTO my_depts(department_id,
              department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
```

- Passing by positional notation

```
add_dept ('EDUCATION', 1400);
```

- Passing by named notation

```
add_dept (p_loc=>1400, p_name=>'EDUCATION');
```

- Passing by combination notation

```
add_dept ('EDUCATION', p_loc=>1400);
```

Tell Me / Show Me

Parameter Passing

Will the following call execute successfully?

```
add_dept (p_loc => 1400, 'EDUCATION');
```

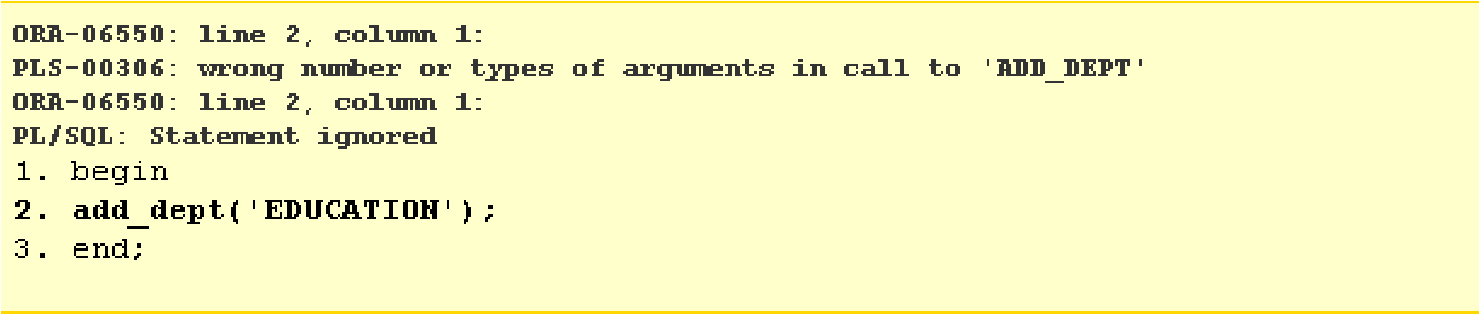
Answer: **No**, because when using the combination notation, positional notation parameters must be listed before named notation parameters.

Tell Me / Show Me

Parameter Passing

Will the following call execute successfully?

```
add_dept ('EDUCATION');
```



```
ORA-06550: line 2, column 1:
PLS-00306: wrong number or types of arguments in call to 'ADD_DEPT'
ORA-06550: line 2, column 1:
PL/SQL: Statement ignored
1. begin
2. add_dept('EDUCATION');
3. end;
```

No: You must provide a value for each parameter unless the formal parameter is assigned a default value. But what if you really want to omit an actual parameter, or you don’t know a value for the parameter? Specifying default values for formal parameters is discussed next.

Tell Me / Show Me

Using the DEFAULT Option for IN Parameters

You can assign a default value for formal IN parameters. This provides flexibility when passing parameters.

```
CREATE OR REPLACE PROCEDURE add_dept(
  p_name my_depts.department_name%TYPE:='Unknown',
  p_loc  my_depts.location_id%TYPE DEFAULT 1400)
IS
BEGIN
  INSERT INTO my_depts (...)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
```

The code shows two ways of assigning a default value to an `IN` parameter. The two ways shown use:

- The assignment operator (:=), as shown for the p_name parameter
- The `DEFAULT` option, as shown for the `p_loc` parameter

Tell Me / Show Me

Using the DEFAULT Option for Parameters

The following are three ways of invoking the add_dept procedure:

- The first example assigns the default values for each parameter.
- The second example illustrates a combination of position and named notation to assign values. In this case, using named notation is presented as an example.
- The last example uses the default value for the name parameter and the supplied value for the p_loc parameter.

```
add_dept;
add_dept ('ADVERTISING', p_loc => 1400);
add_dept (p_loc => 1400);
```

Tell Me / Show Me

Guidelines for Using the DEFAULT Option for Parameters

- You cannot assign default values to OUT and IN OUT parameters *in the header*, but you can in the body of the procedure.

- Usually, you can use named notation to override the default values of formal parameters. However, you cannot skip providing an actual parameter if there is no default value provided for a formal parameter.

- A parameter inheriting a DEFAULT value is different from NULL.

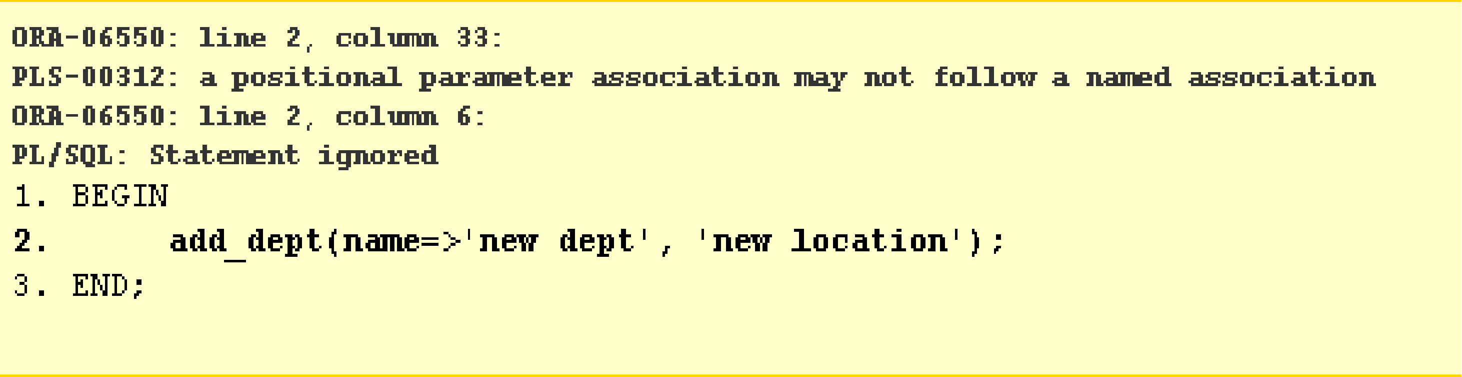
Tell Me / Show Me

Working with Parameter Errors During Runtime

- **Note:** All the positional parameters should precede the named parameters in a subprogram call. Otherwise, you receive an error message, as shown in the following example:

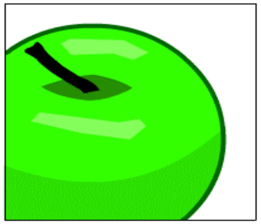
```
BEGIN
  add_dept (name =>'new dept', 'new location');
END;
```

- The following error message is generated:



```
ORA-06550: line 2, column 33:
PLS-00312: a positional parameter association may not follow a named association
ORA-06550: line 2, column 6:
PL/SQL: Statement ignored
1. BEGIN
2.      add_dept(name=>'new dept', 'new location');
3. END;
```

Tell Me / Show Me



Terminology

Key terms used in this lesson include:

`IN` parameter

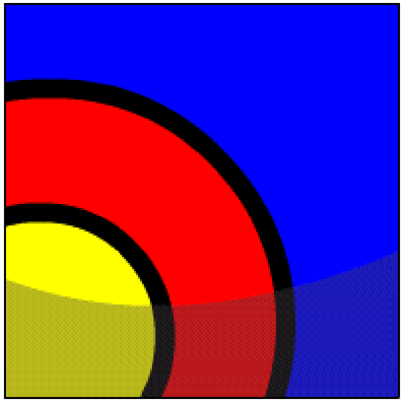
`OUT` parameter

`IN OUT` parameter

Summary

In this lesson, you learned to:

- List the types of parameter modes
- Create a procedure that passes parameters
- Identify three methods for passing parameters
- Describe the `DEFAULT` option for parameters

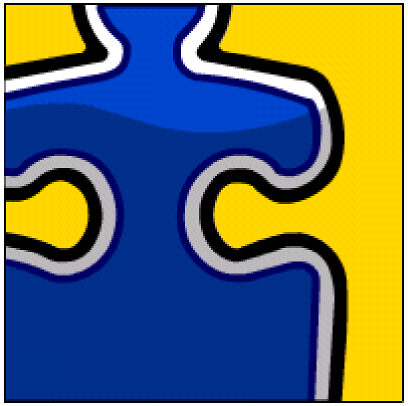


Copyright © 2008, Oracle. All rights reserved.

Try It / Solve It

The exercises in this lesson cover the following topics:

- Listing the types of parameter modes
- Creating a procedure with parameters
- Identifying three methods for passing parameters
- Describing the DEFAULT option for parameters

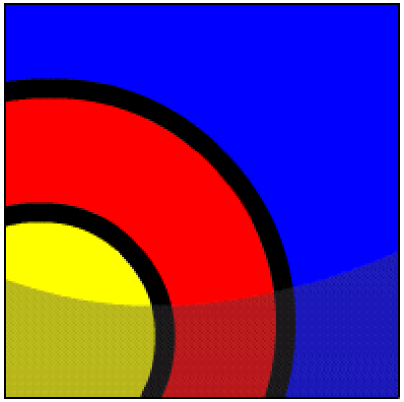


Creating Functions

What Will I Learn?

In this lesson, you will learn to:

- Define a stored function
- Create a PL/SQL block containing a function
- List ways in which you can invoke a function
- Create a PL/SQL block that invokes a function that has parameters
- List the development steps for creating a function
- Describe the differences between procedures and functions



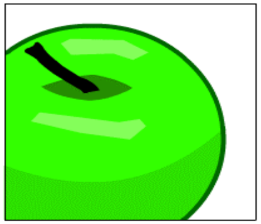
Why Learn It?

In this lesson, you learn how to create and invoke functions. A function is a subprogram that must return exactly one value.

A procedure is a standalone executable statement, whereas a function can only exist as part of an executable statement.

Functions are an integral part of modular code. Business rules and/or formulas can be placed in functions so that they can be easily reused.

Tell Me / Show Me



What Is a Stored Function?

- A function is a named PL/SQL block (a subprogram) that can accept optional IN parameters and must return a single output value.

- Functions are stored in the database as schema objects for repeated execution.

Tell Me / Show Me

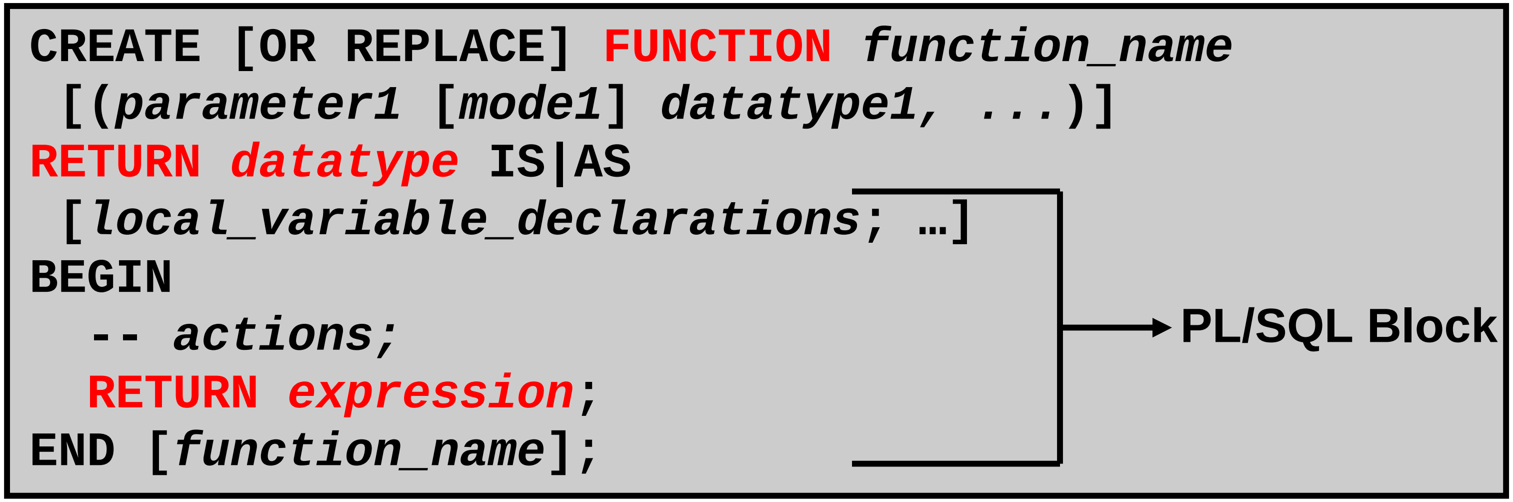
What Is a Stored Function? (continued)

- A function can be called as part of an SQL expression or as part of a PL/SQL expression.
  - Certain return types, for example, Boolean, prevent a function from being called as part of a `SELECT`.
- In SQL expressions, a function must obey specific rules to control side effects. Side effects to be avoided are:
  - Any kind of DML or DDL
  - `COMMIT` or `ROLLBACK`
  - Altering global variables
- In PL/SQL expressions, the function identifier acts like a variable whose value depends on the parameters passed to it.

Tell Me / Show Me

Syntax for 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

The header is like a PROCEDURE header with two differences:

1. The parameter mode should only be IN.
2. The RETURN clause is used instead of an OUT mode.

Tell Me / Show Me

Syntax for Creating Functions (continued)

- A function is a PL/SQL subprogram that returns a single value. You must provide a `RETURN` statement to return a value with a data type that is consistent with the function declaration type.

- You create new functions with the `CREATE [OR REPLACE] FUNCTION` statement, which can declare a list of parameters, must return exactly one value, and must define the actions to be performed by the PL/SQL block.

Tell Me / Show Me

Stored Function With a Parameter: Example

- Create the function:

```
CREATE OR REPLACE FUNCTION get_sal
 (p_id employees.employee_id%TYPE)
  RETURN NUMBER IS
  v_sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
    INTO  v_sal
    FROM  employees
    WHERE employee_id = p_id;
  RETURN v_sal;
END get_sal;
```

- Invoke the function as an expression or as a parameter value:

```
 ... v_salary := get_sal(100);
```

Tell Me / Show Me

You can RETURN from the executable section and/or from the EXCEPTION section.

- Create the function

```
CREATE OR REPLACE FUNCTION get_sal
 (p_id employees.employee_id%TYPE) RETURN NUMBER IS
  v_sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary INTO v_sal
    FROM employees WHERE employee_id = p_id;
  RETURN v_sal;
EXCEPTION
  WHEN NO_DATA_FOUND THEN RETURN NULL;
END get_sal;
```

- Invoke the function as an expression with a bad parameter

```
... v_salary := get_sal(999);
```

Tell Me / Show Me

Ways to Invoke (or Execute) Functions With Parameters

- Invoke as part of a PL/SQL expression, using a local variable to store the returned result

```
DECLARE v_sal employees.salary%type;
BEGIN
  v_sal := get_sal(100); ...
END;
```

A

- Use as a parameter to another subprogram

```
... DBMS_OUTPUT.PUT_LINE(get_sal(100));
```

B

- Use in an SQL statement (subject to restrictions)

```
SELECT job_id, get_sal(employee_id) FROM employees;
```

C

Tell Me / Show Me

Ways to Invoke (or Execute) Functions With Parameters

If functions are designed thoughtfully, they can be powerful constructs. You can invoke functions in the following ways:

- As part of PL/SQL expressions: (A) Uses a local variable in an anonymous block to hold the returned value from a function.
- As a parameter to another subprogram: (B) Demonstrates this usage. The `get_sal` function with all its arguments is nested in the parameter required by the `DBMS_OUTPUT.PUT_LINE` procedure.
- As an expression in an SQL statement: (C) Shows how you can use a function as a single-row function in an SQL statement.

Note: The restrictions that apply to functions when used in an SQL statement are discussed in the next lesson.

Tell Me / Show Me

Invoking Functions Without Parameters

Most functions have parameters, but not all. The following are system functions USER and SYSDATE without parameters.

- Invoke as part of a PL/SQL expression, using a local variable to obtain the result

```
DECLARE v_today DATE;
BEGIN
  v_today := SYSDATE; ...
END;
```

- Use as a parameter to another subprogram

```
... DBMS_OUTPUT.PUT_LINE(USER);
```

- Use in an SQL statement (subject to restrictions)

```
SELECT job_id, SYSDATE-hiredate FROM employees;
```

Tell Me / Show Me

Benefits and Restrictions That Apply to Functions

+ Try things quickly: Functions allow you to temporarily display a value in a new format: a different case, annually vs. monthly (times 12), concatenated or with substrings.
+ Extend functionality: Add new features, such as spell checking and parsing.

– Restrictions: PL/SQL types do not completely overlap with SQL types. What is fine for PL/SQL (for example, `BOOLEAN`, `RECORD`) might be invalid for a `SELECT`.

– Restrictions: PL/SQL sizes are not the same as SQL sizes. For instance, a PL/SQL `VARCHAR2` variable can be up to 32 KB, whereas an SQL `VARCHAR2` column can be only up to 4 KB.

Tell Me / Show Me

Syntax Differences Between Procedures and Functions

Procedures

```
CREATE [OR REPLACE] PROCEDURE name [parameters] IS|AS (Mandatory)
  Variables, cursors, etc. (Optional)
BEGIN          (Mandatory)
  SQL and PL/SQL statements;
EXCEPTION      (Optional)
  WHEN exception-handling actions;
END [name];      (Mandatory)
```

Functions

```
CREATE [OR REPLACE] FUNCTION name [parameters] (Mandatory)
  RETURN datatype IS|AS         (Mandatory)
  Variables, cursors, etc. (Optional)
BEGIN          (Mandatory)
  SQL and PL/SQL statements;
  RETURN ...; (One Mandatory, more optional)
EXCEPTION      (Optional)
  WHEN exception-handling actions;
END [name];      (Mandatory)
```

Tell Me / Show Me

Differences/Similarities Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can return values (if any) in output parameters	Must return a single value
Can contain a RETURN statement without a value	Must contain at least one RETURN statement

Both can have zero or more IN parameters that can be passed from the calling environment.

Both have the standard block structure including exception handling.

Tell Me / Show Me

Differences Between Procedures and Functions

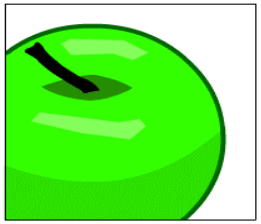
Procedures

- You create a procedure to store a series of actions for later execution. A procedure does not have to return a value. A procedure can call a function to assist with its actions. **Note:** A procedure containing a single `OUT` parameter might be better rewritten as a function returning the value.

Functions

- You create a function when you want to compute a value that must be returned to the calling environment. Functions return only a single value, and the value is returned through a `RETURN` statement. The functions used in SQL statements cannot use `OUT` or `IN OUT` modes. Although a function using `OUT` can be invoked from a PL/SQL procedure or anonymous block, it cannot be used in SQL statements.

Tell Me / Show Me



Terminology

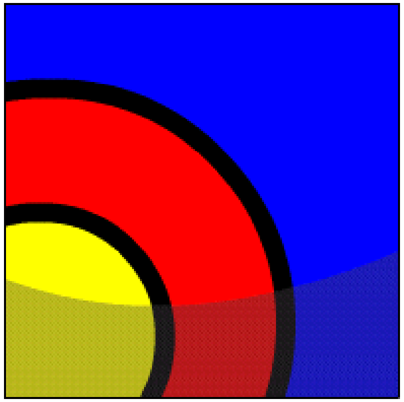
Key terms used in this lesson include:

Stored function

Summary

In this lesson, you learned to:

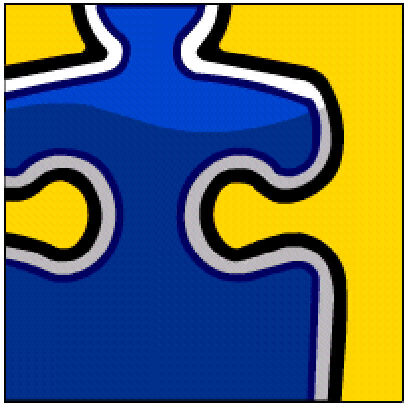
- Define a stored function
- Create a PL/SQL block containing a function
- List ways in which a function can be invoked
- Create a PL/SQL block that invokes a function that has parameters
- List the development steps for creating a function
- Describe the differences between procedures and functions



Try It / Solve It

The exercises in this lesson cover the following topics:

- Defining a stored function
- Creating a function
- Listing how a function can be invoked
- Invoking a function that has parameters
- Listing the development steps for creating a function
- Describing the differences between procedures and functions

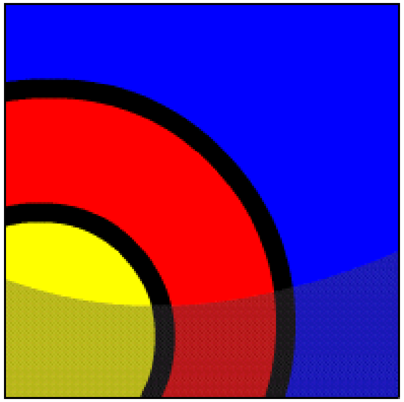


Using Functions in SQL Statements

What Will I Learn?

In this lesson, you will learn to:

- List the advantages of user-defined functions in SQL statements
- List where user-defined functions can be called from within an SQL statement
- Describe the restrictions on calling functions from SQL statements



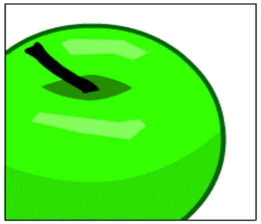
Why Learn It?

In this lesson, you learn how to use functions within SQL statements.

If the SQL statement processes many rows in a table, the function executes once for each row processed by the SQL statement.

For example, you could calculate the tax to be paid by every employee using just one function.

Tell Me / Show Me



What Is a User-Defined Function?

A user-defined function is a function that is created by the PL/SQL programmer. `GET_DEPT_NAME` and `CALCULATE_TAX` are examples of user-defined functions, whereas `UPPER`, `LOWER`, and `LPAD` are examples of system-defined functions automatically provided by Oracle.

Most system functions, such as `UPPER`, `LOWER`, and `LPAD` are stored in a package named `SYS.STANDARD`. Packages are covered in a later section.

These system functions are often called built-in functions.

Tell Me / Show Me

Advantages of Functions in SQL Statements

- In the WHERE clause of a SELECT statement, functions can increase efficiency by eliminating unwanted rows before the data is sent to the application.

For example in a school administrative system, you want to fetch only those students whose last names are stored entirely in uppercase. This could be a small minority of the table’s rows. You code:

```
SELECT * FROM students
  WHERE student_name = UPPER(student_name);
```

Without the UPPER function, you would have to fetch all the student rows, transmit them across the network, and eliminate the unwanted ones within the application.

Tell Me / Show Me

Advantages of Functions in SQL Statements (continued)

- Can manipulate data values

For example, for an end-of-semester social event, you want (just for fun) to print out the name of every teacher with the characters reversed, so “Mary Jones” becomes “senoJ yraM.” You can create a user-defined function called REVERSE_NAME, which does this, then code:

```
SELECT name, reverse_name(name) FROM teachers;
```

Tell Me / Show Me

Advantages of Functions in SQL Statements (continued)

- User-defined functions in particular can extend SQL where activities are too complex, too awkward, or unavailable with regular SQL

For example, you want to calculate how long an employee has been working for your business, rounded to a whole number of months. You could create a user-defined function called `HOW_MANY_MONTHS` to do this. Then, the application programmer can code:

```
SELECT employee_id, how_many_months(hire_date)
  FROM employees;
```

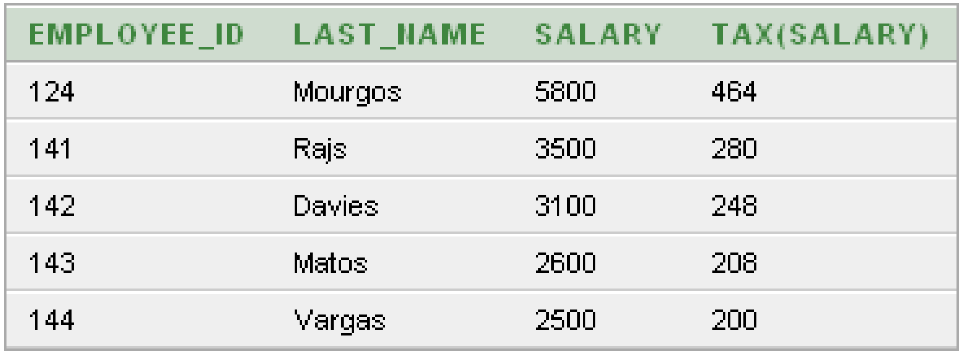
Tell Me / Show Me

Function in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
```

```
Function created.
```

```
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 50;
```



EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
124	Mourgos	5800	464
141	Rajs	3500	280
142	Davies	3100	248
143	Matos	2600	208
144	Vargas	2500	200

Tell Me / Show Me

Where Can You Use User-Defined Functions in an SQL Statement?

User-defined functions act like built-in single-row functions, such as `UPPER`, `LOWER` and `LPAD`. They can be used in:

- The `SELECT` column-list of a query
- Conditional expressions in the `WHERE` and `HAVING` clauses
- The `ORDER BY` and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

In short, they can be used *ANYWHERE* that you have a value or expression!

Tell Me / Show Me

Where Can You Use User-Defined Functions in an SQL Statement? (continued)

This example shows the user-defined function TAX being used in four places within a single SQL statement.

```
SELECT employee_id, tax(salary)
  FROM   employees
  WHERE  tax(salary) > (SELECT MAX(tax(salary))
                        FROM employees
                        WHERE department_id = 20)
  ORDER BY tax(salary) DESC;
```

Tell Me / Show Me

Restrictions on Using Functions in SQL Statements

To use a user-defined function within a SQL statement, the function must conform to the rules and restrictions of the SQL language.

- The function can accept only valid SQL datatypes as `IN` parameters, and must `RETURN` a valid SQL datatype.

  - PL/SQL-specific types, such as `BOOLEAN` and `%ROWTYPE` are not allowed

  - SQL size limits must not be exceeded (PL/SQL allows a VARCHAR2 variable to be up to 32 KB in size, but SQL allows only 4 KB)

Tell Me / Show Me

Restrictions on Using Functions in SQL Statements (continued)

- Parameters must be specified with positional notation. Named notation (=>) is not allowed.

Example:

```
SELECT employee_id, tax(salary)
  FROM   employees;

SELECT employee_id, tax(p_value => salary)
  FROM   employees;
```

The second SELECT statement causes an error.

Tell Me / Show Me

Restrictions on Using Functions in SQL Statements (continued)

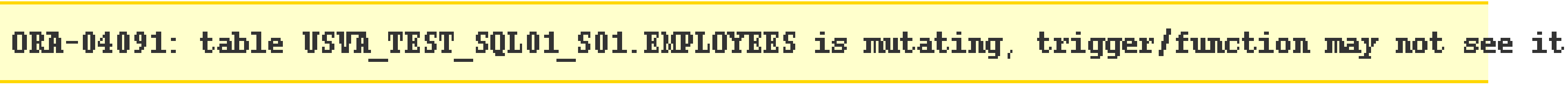
- Functions called from a `SELECT` statement cannot contain DML statements
- Functions called from an `UPDATE` or `DELETE` statement on a table cannot query or contain DML on the same table
- Functions called from any SQL statement cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)
- Functions called from any SQL statement cannot issue DDL (for example, `CREATE TABLE`) or DCL (for example, `ALTER SESSION`) because they also do an implicit `COMMIT`
- Calls to subprograms that break these restrictions are also not allowed in the function.

Tell Me / Show Me

Restrictions on Using Functions in SQL Statements: Example 1

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_sal NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name, email,
                        hire_date, job_id, salary)
  VALUES(1, 'Frost', 'jfrost@company.com',
         SYSDATE, 'SA_MAN', p_sal);
  RETURN (p_sal + 100);
END dml_call_sql;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 174;
```



```
ORA-04091: table USVA_TEST_SQL01_S01.EMPLOYEES is mutating, trigger/function may not see it
```

Tell Me / Show Me

Restrictions on Using Functions in SQL Statements: Example 2

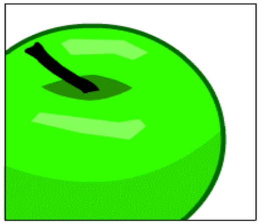
The following function queries the EMPLOYEES table.

```
CREATE OR REPLACE FUNCTION query_max_sal (p_dept_id NUMBER)
  RETURN NUMBER IS
  v_num NUMBER;
 BEGIN
   SELECT MAX(salary) INTO v_num FROM employees
     WHERE department_id = p_dept_id;
   RETURN (v_num);
 END;
```

When used within the following DML statement, it returns the “mutating table” error message similar to the error message shown in the previous slide.

```
UPDATE employees SET salary = query_max_sal(department_id)
  WHERE employee_id = 174;
```

Tell Me / Show Me



Terminology

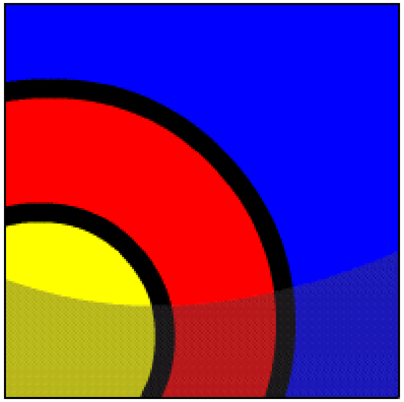
Key terms used in this lesson include:

User-defined function

Summary

In this lesson, you learned to:

- List the advantages of user-defined functions in SQL statements
- List where user-defined functions can be called from within an SQL statement
- Describe the restrictions on calling functions from SQL statements



Try It / Solve It

The exercises in this lesson cover the following topics:

- Listing the advantages of user-defined functions in SQL statements
- Listing where user-defined functions can be called from within an SQL statement
- Describing the restrictions on calling functions from SQL statements