1

Creating Stored Procedures

ORACLE

Objectives

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

- **Describe and create a procedure**
- **Create procedures with parameters**
- **Differentiate between formal and actual parameters**
- **Use different parameter-passing modes**
- **Invoke a procedure**
- **Handle exceptions in procedures**
- **Remove a procedure**

ORACLE

Lesson Aim

In this lesson, you learn to create, execute, and remove procedures with or without parameters. Procedures are the foundation of modular programming in PL/SQL. 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` parameters.

To make your programs robust, you should always manage exception conditions by using the exception-handling features of PL/SQL.

What Is a Procedure?

A procedure:

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

ORACLE

Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments). Generally, you use a procedure to perform an action. It has a header, a declaration section, an executable section, and an optional exception-handling section. A procedure is invoked by using the procedure name in the execution section of another PL/SQL block.

A procedure is compiled and stored in the database as a schema object. If you are using the procedures with Oracle Forms and Reports, then they can be compiled within the Oracle Forms or Oracle Reports executables.

Procedures promote reusability and maintainability. When validated, they can be used in any number of applications. If the requirements change, only the procedure needs to be updated.

Syntax for Creating Procedures

- 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` statement, and an `END` statement (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

ORACLE

Syntax for Creating Procedures

You create new procedures with the `CREATE PROCEDURE` statement, which may declare a list of parameters and must define the actions to be performed by the standard PL/SQL block. The `CREATE` clause enables you to create stand-alone procedures that are stored in an Oracle database.

- PL/SQL blocks start with `BEGIN`, optionally preceded by the declaration of local variables. PL/SQL blocks end with either `END` or `END` *`procedure_name`*.
- The `REPLACE` option indicates that if the procedure exists, it is dropped and replaced with the new version created by the statement.

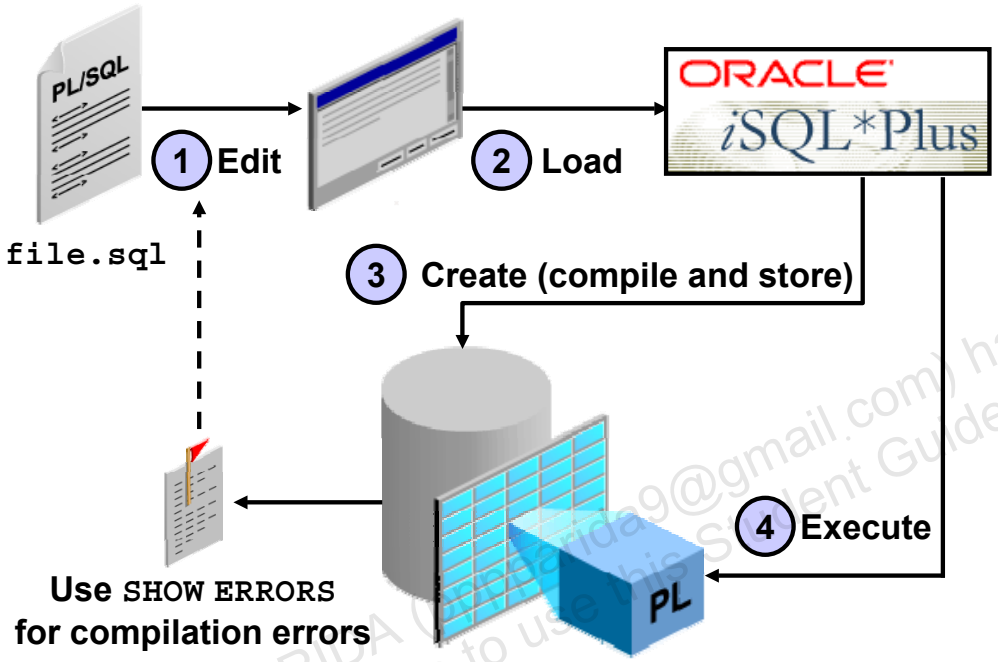
Other Syntactic Elements

- *parameter1* represents the name of a parameter.
- The *mode* option defines how a parameter is used: `IN` (default), `OUT`, or `IN OUT`.
- *datatype1* specifies the parameter data type, without any precision.

Note: Parameters can be considered as local variables. Substitution and host (bind) variables cannot be referenced anywhere in the definition of a PL/SQL stored procedure.

The `OR REPLACE` option does not require any change in object security, as long as you own the object and have the `CREATE [ANY] PROCEDURE` privilege.

Developing Procedures



Developing Procedures

To develop a stored procedure, perform the following steps:

1. Write the code to create a procedure in an editor or a word processor, and then save it as a SQL script file (typically with an `.sql` extension).
2. Load the code into one of the development tools such as SQL*Plus or *i*SQL*Plus.
3. Create the procedure in the database. The `CREATE PROCEDURE` statement compiles and stores source code and the compiled *m-code* in the database. If compilation errors exist, then the *m-code* is not stored and you must edit the source code to make corrections. You cannot invoke a procedure that contains compilation errors. To view the compilation errors in SQL*Plus or *i*SQL*Plus, use:
   - `SHOW ERRORS` for the most recently (last) compiled procedure
   - `SHOW ERRORS PROCEDURE procedure_name` for any procedure compiled previously
4. After successful compilation, execute the procedure to perform the desired action. Use the `EXECUTE` command from *i*SQL*Plus or an anonymous PL/SQL block from environments that support PL/SQL.

Note: If compilation errors occur, use a `CREATE OR REPLACE PROCEDURE` statement to overwrite the existing code if you previously used a `CREATE PROCEDURE` statement. Otherwise, `DROP` the procedure first and then execute the `CREATE PROCEDURE` statement.

What Are Parameters?

Parameters:

- **Are declared after the subprogram name in the PL/SQL header**
- **Pass or communicate data between the caller and the subprogram**
- **Are used like local variables but are dependent on their parameter-passing mode:**
  - **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 may be returned (output) as a modified value.**

ORACLE

What Are Parameters?

Parameters are used to transfer data values to and from the calling environment and the procedure (or subprogram). Parameters are declared in the subprogram header, after the name and before the declaration section for local variables.

Parameters are subject to one of the three parameter-passing modes: `IN`, `OUT`, or `IN OUT`.

- An `IN` parameter passes a constant value from the calling environment into the procedure.
- An `OUT` parameter passes a value from the procedure to the calling environment.
- An `IN OUT` parameter passes a value from the calling environment to the procedure and a possibly different value from the procedure back to the calling environment using the same parameter.

Parameters can be thought of as a special form of local 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.

Formal and Actual Parameters

- **Formal parameters: Local variables declared in the parameter list of a subprogram specification**

  Example:

```
CREATE PROCEDURE raise_sal(id NUMBER,sal NUMBER) IS
BEGIN ...
END raise_sal;
```

- **Actual parameters: Literal values, variables, and expressions used in the parameter list of the called subprogram**

  Example:

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

ORACLE

Formal and Actual Parameters

Formal parameters are local variables that are declared in the parameter list of a subprogram specification. In the first example, in the `raise_sal` procedure, the variable `id` and `sal` identifiers represent the formal parameters.

The actual parameters can be literal values, variables, and expressions that are provided in the parameter list of a called subprogram. In the second example, a call is made to `raise_sal`, where the `emp_id` variable provides the actual parameter value for the `id` formal parameter and `2000` is supplied as the actual parameter value for `sal`. Actual parameters:

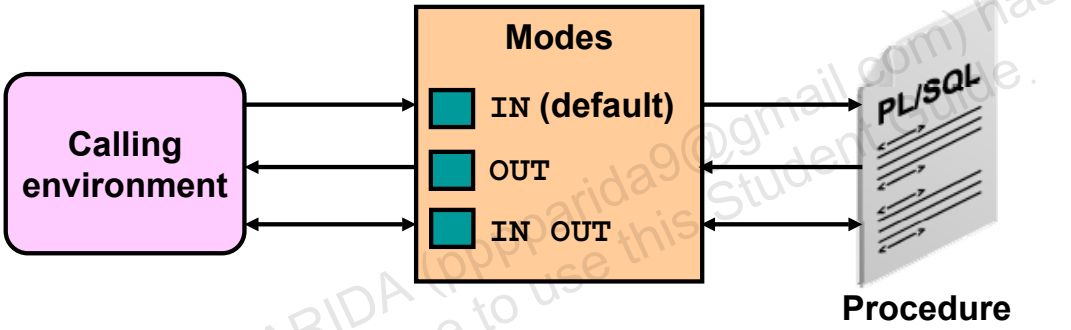
- Are associated with formal parameters during the subprogram call
- Can also be expressions, as in the following example:
  `raise_sal(emp_id, raise+100);`

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.

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



Procedural Parameter Modes

When you create the procedure, the formal parameter defines a variable name whose value is used in the executable section of the PL/SQL block. The actual parameter is used when invoking the procedure to provide input values or receive output results.

The parameter mode `IN` is the default passing mode—that is, if no mode is specified with a parameter declaration, the parameter is considered to be an `IN` parameter. The parameter modes `OUT` and `IN OUT` must be explicitly specified in their parameter declarations.

The *`datatype`* parameter is specified without a size specification. It can be specified:

- As an explicit data type
- Using the `%TYPE` definition
- Using the `%ROWTYPE` definition

Note: One or more formal parameters can be declared, with each separated by a comma.

Using IN Parameters: Example

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

EXECUTE raise_salary(176,10)
```


Using IN Parameters: Example

The example shows a procedure with two IN parameters. Running this first statement in *i*SQL*Plus creates the raise_salary procedure in the database. The second example invokes raise_salary and provides the first parameter value of 176 for the employee ID, and a percentage salary increase of 10 percent for the second parameter value.

To invoke a procedure in *i*SQL*Plus, use the following EXECUTE command:

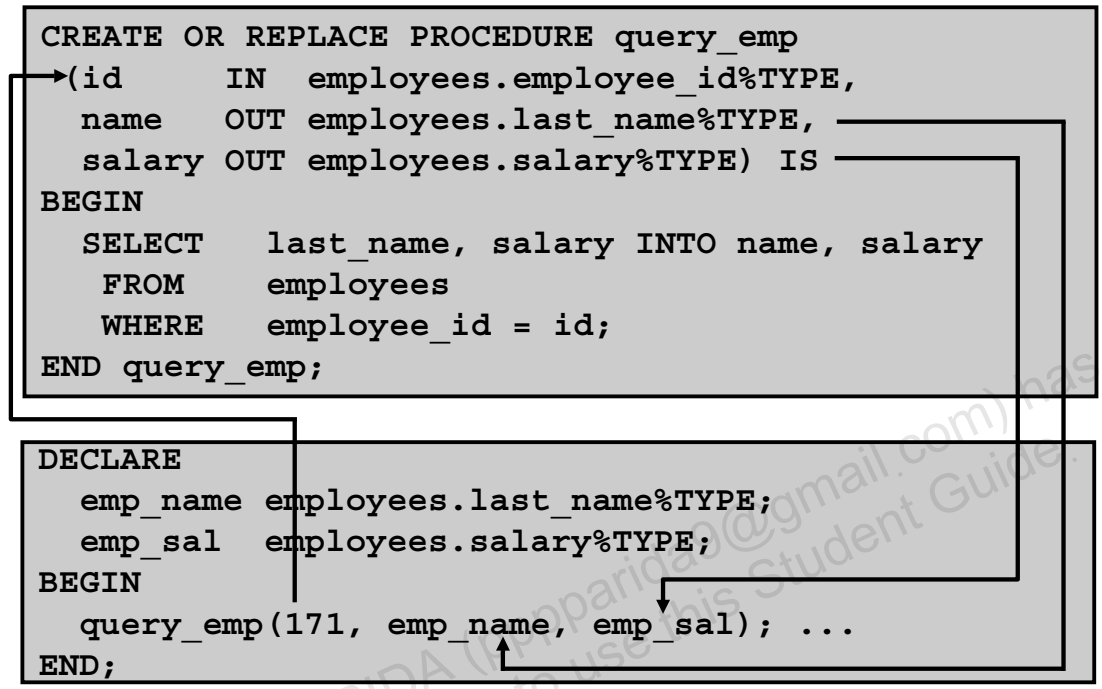
```
EXECUTE raise_salary (176, 10)
```

To invoke a procedure from another procedure, use a direct call inside an executable section of the calling block. At the location of calling the new procedure, enter the procedure name and actual parameters. For example:

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

Note: IN parameters are passed as read-only values from the calling environment into the procedure. Attempts to change the value of an IN parameter result in a compile-time error.

Using OUT Parameters: Example



Using OUT Parameters: Example

In this example, you create a procedure with OUT parameters to retrieve information about an employee. The procedure accepts the value 171 for employee ID and retrieves the name and salary of the employee with ID 171 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 in the lower portion of the slide. The procedure accepts an employee ID value through the id parameter. The emp_name and emp_salary variables are populated with the information retrieved from the query into their two corresponding OUT parameters.

If you print the values returned into PL/SQL variables of the calling block shown in the second block of code, then the variables contain the following:

- emp_name holds the value Smith.
- emp_salary holds the value 7600.

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

Attempting to use or read OUT parameters inside the procedure that declares them results in a compilation error. The OUT parameters can be assigned values only in the body of the procedure in which they are declared.

Viewing OUT Parameters with *i*SQL*Plus

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

```
SET SERVEROUTPUT ON
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || emp_sal);
END;
```

- **Use *i*SQL*Plus host variables, execute QUERY_EMP using host variables, and print the host variables.**

```
VARIABLE name VARCHAR2(25)
VARIABLE sal  NUMBER
EXECUTE query_emp(171, :name, :sal)
PRINT name sal
```

ORACLE

Viewing OUT Parameters with *i*SQL*Plus

The examples show two ways to view the values returned from OUT parameters.

- The first technique uses PL/SQL variables in an anonymous block to retrieve the OUT parameter values. The DBMS_OUPUT.PUT_LINE procedure is called to print the values held in the PL/SQL variables. The SET SERVEROUPUT must be ON.
- The second technique shows how to use *i*SQL*Plus variables that are created using the VARIABLE command. The *i*SQL*Plus variables are external to the PL/SQL block and are known as host or bind variables. To reference host variables from a PL/SQL block, you must prefix their names with a colon (:). To display the values stored in the host variables, you must use the *i*SQL*Plus PRINT command followed by the name of the *i*SQL*Plus variable (without the colon because this is not a PL/SQL command or context).

To use *i*SQL*Plus and host variables when calling a procedure with OUT parameters, perform the following steps:

1. Create an *i*SQL*Plus script file by using an editor.
2. Add commands to create the variables, execute the procedure, and print the variables.
3. Load and execute the *i*SQL*Plus script file.

Note: For details about the VARIABLE command, see the *i*SQL*Plus Command Reference.

Calling PL/SQL Using Host Variables

A host variable (also known as a *bind* or a *global* variable):
- Is declared and exists externally to the PL/SQL subprogram. A host variable can be created in:
  – *i*SQL*Plus by using the `VARIABLE` command
  – Oracle Forms internal and UI variables
  – Java variables
- Is preceded by a colon (`:`) when referenced in PL/SQL code
- Can be referenced in an anonymous block but not in a stored subprogram
- Provides a value to a PL/SQL block and receives a value from a PL/SQL block

Calling PL/SQL Using Host Variables

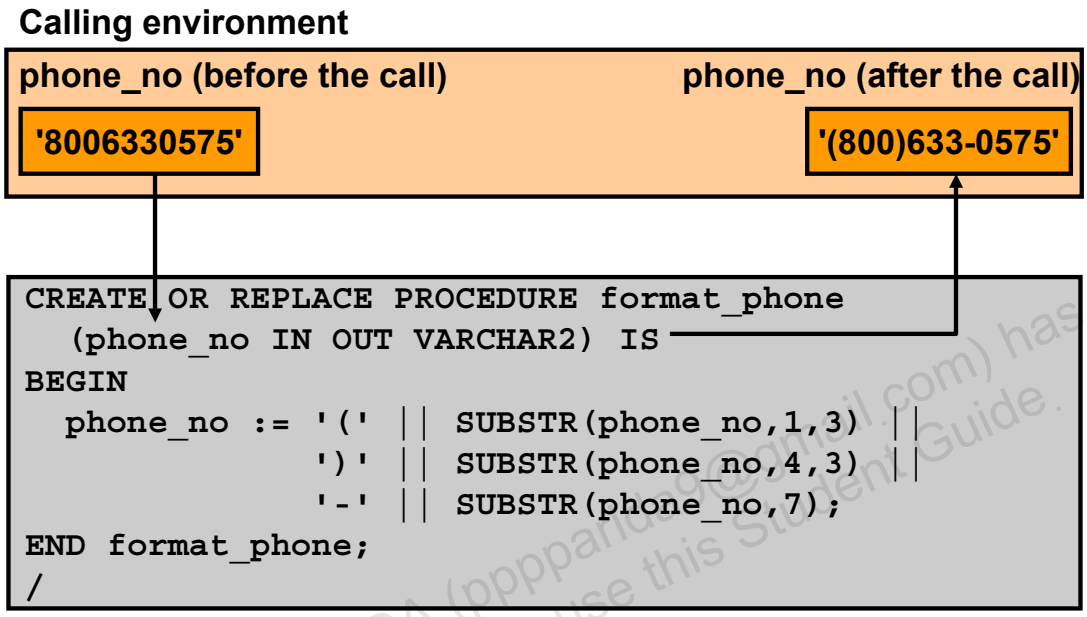
The PL/SQL code that is stored in the database can be called from a variety of environments, such as:
- SQL*Plus or *i*SQL*Plus
- Oracle Forms and Oracle Reports
- Java and C applications

Each of the preceding environments provides ways to declare variables to store data in memory. The variable values in these applications are defined and held external to stored PL/SQL code. Each environment provides a way to pass the variable data to PL/SQL and receive updated values from the PL/SQL code. In general, most languages host calls to PL/SQL blocks or subprograms. The PL/SQL engine uses a technique called binding to associate values supplied from external locations to PL/SQL variables or parameters declared in the PL/SQL subprograms.

Unlike in Java, PL/SQL recognizes host variables by the presence of a colon prefixed to the external variable name when it is used in a PL/SQL block.

You cannot store PL/SQL code with host variables because the compiler cannot resolve references to host variables. The binding process is done at run time.

Using IN OUT Parameters: Example



ORACLE

Using IN OUT Parameters: Example

Using an IN OUT parameter, you can pass a value into a procedure that can be updated. 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

Note: An IN OUT parameter acts as an initialized variable.

The example in the 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.

The following code uses the phone_no host variable of *i*SQL*Plus to provide the input value passed to the FORMAT_PHONE procedure. The procedure is executed and returns an updated string in the phone_no host variable.

```
VARIABLE phone_no VARCHAR2(15)
EXECUTE :phone_no := '8006330575'
PRINT phone_no
EXECUTE format_phone (:phone_no)
PRINT phone_no
```

Syntax for Passing Parameters

- **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**
- **Combination:**
  - **Lists some of the actual parameters as positional and some as named**

ORACLE

Syntax for Passing Parameters

For a procedure that contains multiple parameters, you can use a number of methods to specify the values of the parameters. The methods are:

- **Positional:** Lists the actual parameter values in the order in which the formal parameters are declared
- **Named:** Lists the actual values in arbitrary order and uses the association operator to associate each actual parameter with its formal parameter by name. The PL/SQL association operator is an "equal" sign followed by an "is greater than" sign, without spaces: =>.
- **Combination:** Lists the first parameter values by their position and the remainder by using the special syntax of the named method

The next page shows some examples of the first two methods.

Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(
  name IN departments.department_name%TYPE,
  loc  IN departments.location_id%TYPE) IS
BEGIN
  INSERT INTO departments(department_id,
              department_name, location_id)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
/
```

- **Passing by positional notation:**

```
EXECUTE add_dept ('TRAINING', 2500)
```

- **Passing by named notation:**

```
EXECUTE add_dept (loc=>2400, name=>'EDUCATION')
```

ORACLE

Parameter Passing: Examples

In the example, the `add_dept` procedure declares two `IN` parameters: `name` and `loc`. The values of these parameters are used in the `INSERT` statement to set the `department_name` and `location_id` columns, respectively.

Passing parameters by position is shown in the first call to execute `add_dept` below the procedure definition. The first actual parameter supplies the value `TRAINING` for the `name` parameter. The second actual parameter value of `2500` is assigned by position to the `loc` parameter.

Passing parameters using the named notation is shown in the last example. Here, the `loc` parameter, which is declared as the second formal parameter, is referenced by name in the call, where it is associated to the actual value of `2400`. The `name` parameter is associated to the value `EDUCATION`. The order of the actual parameters is irrelevant if all parameter values are specified.

Note: You must provide a value for each parameter unless the formal parameter is assigned a default value. Specifying default values for formal parameters is discussed next.

Using the DEFAULT Option for Parameters

- **Defines default values for parameters:**

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

- **Provides flexibility by combining the positional and named parameter-passing syntax:**

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

ORACLE

Using the DEFAULT Option for Parameters

The code examples in the slide show two ways of assigning a default value to an IN parameter. The two ways shown use:

- The assignment operator (:=), as shown for the name parameter
- The DEFAULT option, as shown for the loc parameter

When default values are assigned to formal parameters, you can call the procedure without supplying an actual parameter value for the parameter. Thus, you can pass different numbers of actual parameters to a subprogram, either by accepting or by overriding the default values as required. It is recommended that you declare parameters without default values first. Then, you can add formal parameters with default values without having to change every call to the procedure.

Note: You cannot assign default values to OUT and IN OUT parameters.

The slide shows 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 loc parameter.

Using the DEFAULT Option for Parameters (continued)

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.

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:

```
EXECUTE add_dept(name=>'new dept', 'new location')
```

The following error message is generated:

```
ERROR at line 1:
ORA-06550: line 1, column 34:
PLS-00312: a positional parameter association may not follow a named association
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

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

ORACLE

Summary of Parameter Modes

The `IN` parameter mode is the default mode if no mode is specified in the declaration. The `OUT` and `IN OUT` parameter modes must be explicitly specified with the parameter declaration.

A formal parameter of `IN` mode cannot be assigned a value and cannot be modified in the body of the procedure. By default, the `IN` parameter is passed by reference. An `IN` parameter can be assigned a default value in the formal parameter declaration, in which case the caller need not provide a value for the parameter if the default applies.

An `OUT` or `IN OUT` parameter must be assigned a value before returning to the calling environment. The `OUT` and `IN OUT` parameters cannot be assigned default values. To improve performance with `OUT` and `IN OUT` parameters, the `NOCOPY` compiler hint can be used to request to pass by reference.

Note: Using `NOCOPY` is discussed later in this course.

Invoking Procedures

You can invoke procedures by:

- **Using anonymous blocks**
- **Using another procedure, as in the following example:**

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

ORACLE

Invoking Procedures

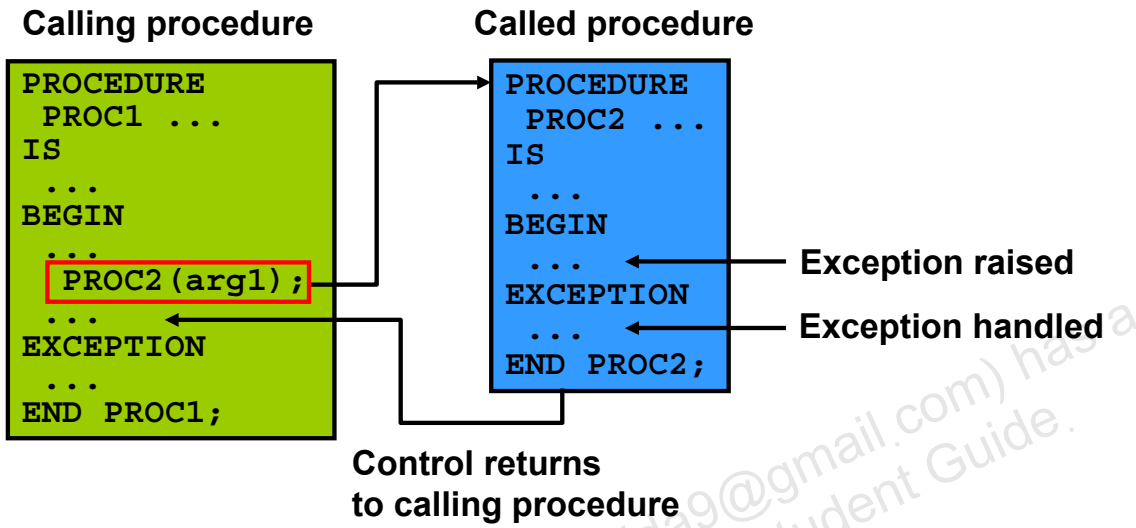
You can invoke procedures by using:

- Anonymous blocks
- Another procedure or PL/SQL subprogram

Examples on the preceding pages have illustrated how to use anonymous blocks (or the `EXECUTE` command in *i*SQL*Plus).

This example shows you how to invoke a procedure from another stored procedure. The `PROCESS_EMPLOYEES` stored procedure uses a cursor to process all the records in the `EMPLOYEES` table and passes each employee's ID to the `RAISE_SALARY` procedure, which results in a 10% salary increase across the company.

Handled Exceptions



ORACLE

Handled Exceptions

When you develop procedures that are called from other procedures, you should be aware of the effects that handled and unhandled exceptions have on the transaction and the calling procedure.

When an exception is raised in a called procedure, the control immediately goes to the exception section of that block. An exception is considered handled if the exception section provides a handler for the exception raised.

When an exception occurs and is handled, the following code flow takes place:

1. The exception is raised.
2. Control is transferred to the exception handler.
3. The block is terminated.
4. The calling program/block continues to execute as if nothing has happened.

If a transaction was started (that is, if any data manipulation language [DML] statements executed before executing the procedure in which the exception was raised), then the transaction is unaffected. A DML operation is rolled back if it was performed within the procedure before the exception.

Note: You can explicitly end a transaction by executing a `COMMIT` or `ROLLBACK` operation in the exception section.

Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
EXCEPTION
 WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Err: adding dept: '||name);
END;
```

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


Handled Exceptions: Example

The two procedures in the example are the following:

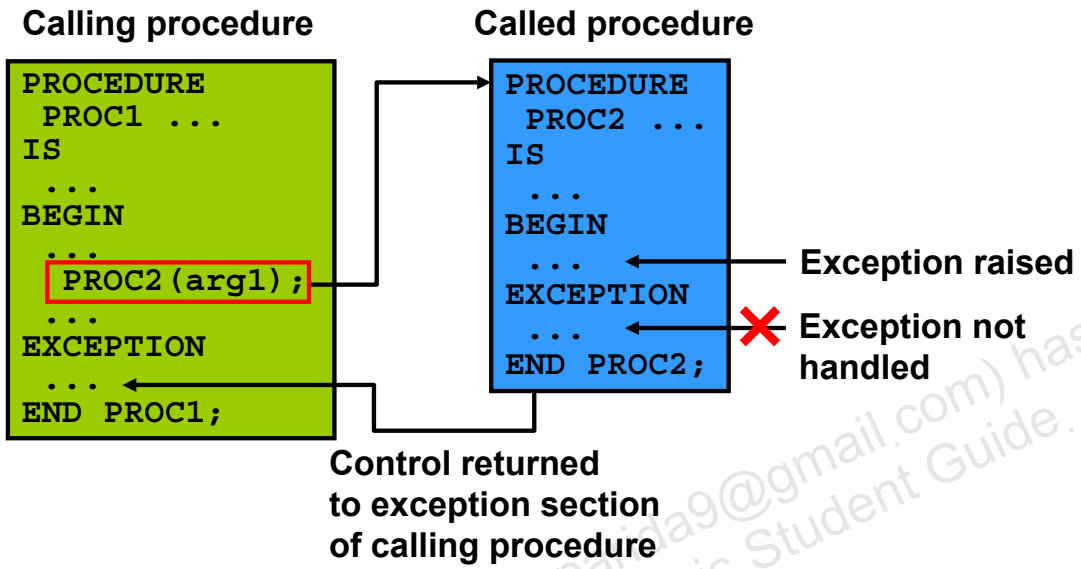
- The `add_department` procedure creates a new department record by allocating a new department number from an Oracle sequence, and sets the `department_name`, `manager_id`, and `location_id` column values using the `name`, `mgr`, and `loc` parameters, respectively.
- The `create_departments` procedure creates more than one department by using calls to the `add_department` procedure.

The `add_department` procedure catches all raised exceptions in its own handler. When `create_departments` is executed, the following output is generated:

```
Added Dept: Media
Err: Adding Dept: Editing
Added Dept: Advertising
```

The `Editing` department with `manager_id` of `99` is not inserted because of a foreign key integrity constraint violation on the `manager_id`. Because the exception was handled in the `add_department` procedure, the `create_department` procedure continues to execute. A query on the `DEPARTMENTS` table where the `location_id` is `1800` shows that `Media` and `Advertising` are added but the `Editing` record is not.

Exceptions Not Handled



ORACLE

Exceptions Not Handled

As discussed, when an exception is raised in a called procedure, control immediately goes to the exception section of that block. If the exception section does not provide a handler for the raised exception, then it is not handled. The following code flow occurs:

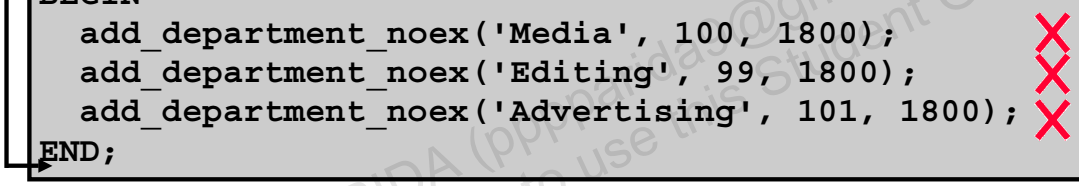
1. The exception is raised.
2. The block terminates because no exception handler exists; any DML operations performed within the procedure are rolled back.
3. The exception propagates to the exception section of the calling procedure—that is, control is returned to the exception section of the calling block, if one exists.

If an exception is not handled, then all the DML statements in the calling procedure and the called procedure are rolled back along with any changes to any host variables. The DML statements that are not affected are statements that were executed before calling the PL/SQL code whose exceptions are not handled.

Exceptions Not Handled: Example

```
SET SERVEROUTPUT ON
CREATE PROCEDURE add_department_noex(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||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;
```



ORACLE

Exceptions Not Handled: Example

The code example in the slide shows `add_department_noex`, which does not have an exception section. In this case, the exception occurs when the `Editing` department is added. Because of the lack of exception handling in either of the subprograms, no new department records are added into the `DEPARTMENTS` table. Executing the `create_departments_noex` procedure produces a result that is similar to the following:

```
Added Dept: Media
BEGIN create_departments_noex; END;

*
ERROR at line 1:
ORA-02291: integrity constraint (ORA1.DEPT_MGR_FK)
violated - parent key not
found
ORA-06512: at "ORA1.ADD_DEPARTMENT_NOEX", line 4
ORA-06512: at "ORA1.CREATE_DEPARTMENTS_NOEX", line 4
ORA-06512: at line 1
```

Although the results show that the `Media` department was added, its operation is rolled back because the exception was not handled in either of the subprograms invoked.

Removing Procedures

You can remove a procedure that is stored in the database.

- **Syntax:**

```
DROP PROCEDURE procedure_name
```

- **Example:**

```
DROP PROCEDURE raise_salary;
```


ORACLE

Removing Procedures

When a stored procedure is no longer required, you can use the `DROP PROCEDURE` SQL statement to remove it.

Note: Whether successful or not, executing a data definition language (DDL) command such as `DROP PROCEDURE` commits any pending transactions that cannot be rolled back.

Viewing Procedures in the Data Dictionary

Information for PL/SQL procedures is saved in the following data dictionary views:

- **View source code in the `USER_SOURCE` table to view the subprograms that you own, or the `ALL_SOURCE` table for procedures that are owned by others who have granted you the `EXECUTE` privilege.**

```
SELECT text
FROM   user_source
WHERE  name='ADD_DEPARTMENT' and type='PROCEDURE'
ORDER BY line;
```

- **View the names of procedures in `USER_OBJECTS`.**

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'PROCEDURE';
```

ORACLE

Viewing Procedures in the Data Dictionary

The source code for PL/SQL subprograms is stored in the data dictionary tables. The source code is accessible to PL/SQL procedures that are successfully or unsuccessfully compiled. To view the PL/SQL source code stored in the data dictionary, execute a `SELECT` statement on the following tables:

- The `USER_SOURCE` table to display PL/SQL code that you own
- The `ALL_SOURCE` table to display PL/SQL code to which you have been granted the `EXECUTE` right by the owner of that subprogram code

The query example shows all the columns provided by the `USER_SOURCE` table:

- The `TEXT` column holds a line of PL/SQL source code.
- The `NAME` column holds the name of the subprogram in uppercase text.
- The `TYPE` column holds the subprogram type, such as `PROCEDURE` or `FUNCTION`.
- The `LINE` column stores the line number for each source code line.

The `ALL_SOURCE` table provides an `OWNER` column in addition to the preceding columns.

Note: You cannot display the source code for Oracle PL/SQL built-in packages, or PL/SQL whose source code has been wrapped by using a `WRAP` utility. The `WRAP` utility converts the PL/SQL source code into a form that cannot be deciphered by humans.

Benefits of Subprograms

- Easy maintenance
- Improved data security and integrity
- Improved performance
- Improved code clarity

ORACLE

Benefits of Subprograms

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

- **Easy maintenance** is realized because subprograms are located in one place. Modifications need to be done in only one place to affect multiple applications and minimize excessive testing.
- **Improved data security** can be achieved by controlling indirect access to database objects from nonprivileged users with security privileges. The subprograms are by default executed with definer's right. The execute privilege does not allow a calling user direct access to objects that are accessible to the subprogram.
- **Data integrity** is managed by having related actions performed together or not at all.
- **Improved performance** can be realized from reuse of parsed PL/SQL code that becomes available in the shared SQL area of the server. Subsequent calls to the subprogram avoid parsing the code again. Because PL/SQL code is parsed at compile time, the parsing overhead of SQL statements is avoided at run time. Code can be written to reduce the number of network calls to the database, and therefore, decrease network traffic.
- **Improved code clarity** can be attained by using appropriate names and conventions to describe the action of the routines, thereby reducing the need for comments and enhancing the clarity of the code.

Summary

In this lesson, you should have learned how to:

- **Write a procedure to perform a task or an action**
- **Create, compile, and save procedures in the database by using the `CREATE PROCEDURE` SQL command**
- **Use parameters to pass data from the calling environment to the procedure by using three different parameter modes: `IN` (the default), `OUT`, and `IN OUT`**
- **Recognize the effect of handling and not handling exceptions on transactions and calling procedures**

ORACLE

Summary

A procedure is a subprogram that performs a specified action. You can compile and save a procedure as a stored procedure in the database. A procedure can return zero or more values through its parameters to its calling environment. There are three parameter modes: `IN`, `OUT`, and `IN OUT`.

You should be able to handle and not handle exceptions, and you should understand how managing exceptions affects transactions and calling procedures. The exceptions are handled in the exception section of a subprogram.

Summary

In this lesson, you should have learned how to:

- **Remove procedures from the database by using the `DROP PROCEDURE` SQL command**
- **Modularize your application code by using procedures as building blocks**

ORACLE

Summary (continued)

You can modify and remove procedures. Procedures are modular components that form the building blocks of an application. You can also create client-side procedures that can be used by client-side applications.

Practice 1: Overview

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**

ORACLE

Practice 1: Overview

In this practice, you create procedures that issue DML and query commands.

If you encounter compilation errors when you are using *i*SQL*Plus, use the `SHOW ERRORS` command.

If you correct any compilation errors in *i*SQL*Plus, do so in the original script file (rather than in the buffer) and then rerun the new version of the file. This saves a new version of the procedure to the data dictionary.

Note: It is recommended to use *i*SQL*Plus for this practice.

Practice 1

Note: You can find table descriptions and sample data in Appendix B, "Table Descriptions and Data." Click the Save Script button to save your subprograms as `.sql` files in your local file system.

Remember to enable `SERVEROUTPUT` if you have previously disabled it.

1. Create and invoke the `ADD_JOB` procedure and consider the results.
   a. Create a procedure called `ADD_JOB` to insert a new job into the `JOBS` table. Provide the ID and title of the job using two parameters.
   b. Compile the code; invoke the procedure with `IT_DBA` as job ID and `Database Administrator` as job title. Query the `JOBS` table to view the results.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Database Administrator		

   c. Invoke your procedure again, passing a job ID of `ST_MAN` and a job title of `Stock Manager`. What happens and why?

      _____

      _____

2. Create a procedure called `UPD_JOB` to modify a job in the `JOBS` table.
   a. Create a procedure called `UPD_JOB` to update the job title. Provide the job ID and a new title using two parameters. Include the necessary exception handling if no update occurs.
   b. Compile the code; invoke the procedure to change the job title of the job ID `IT_DBA` to `Data Administrator`. Query the `JOBS` table to view the results.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		

      Also check the exception handling by trying to update a job that does not exist. (You can use the job ID `IT_WEB` and the job title `Web Master`.)

3. Create a procedure called `DEL_JOB` to delete a job from the `JOBS` table.
   a. Create a procedure called `DEL_JOB` to delete a job. Include the necessary exception handling if no job is deleted.
   b. Compile the code; invoke the procedure using the job ID `IT_DBA`. Query the `JOBS` table to view the results.

```
no rows selected
```

      Also, check the exception handling by trying to delete a job that does not exist. (Use the `IT_WEB` job ID.) You should get the message that you used in the exception-handling section of the procedure as output.

Practice 1 (continued)

4. Create a procedure called `GET_EMPLOYEE` to query the `EMPLOYEES` table, retrieving the salary and job ID for an employee when provided with the employee ID.
   a. Create a procedure that returns a value from the `SALARY` and `JOB_ID` columns for a specified employee ID. Compile the code and remove the syntax errors.
   b. Execute the procedure using host variables for the two `OUT` parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

SALARY
8000

JOB
ST_MAN

   c. Invoke the procedure again, passing an `EMPLOYEE_ID` of 300. What happens and why?

_____

_____