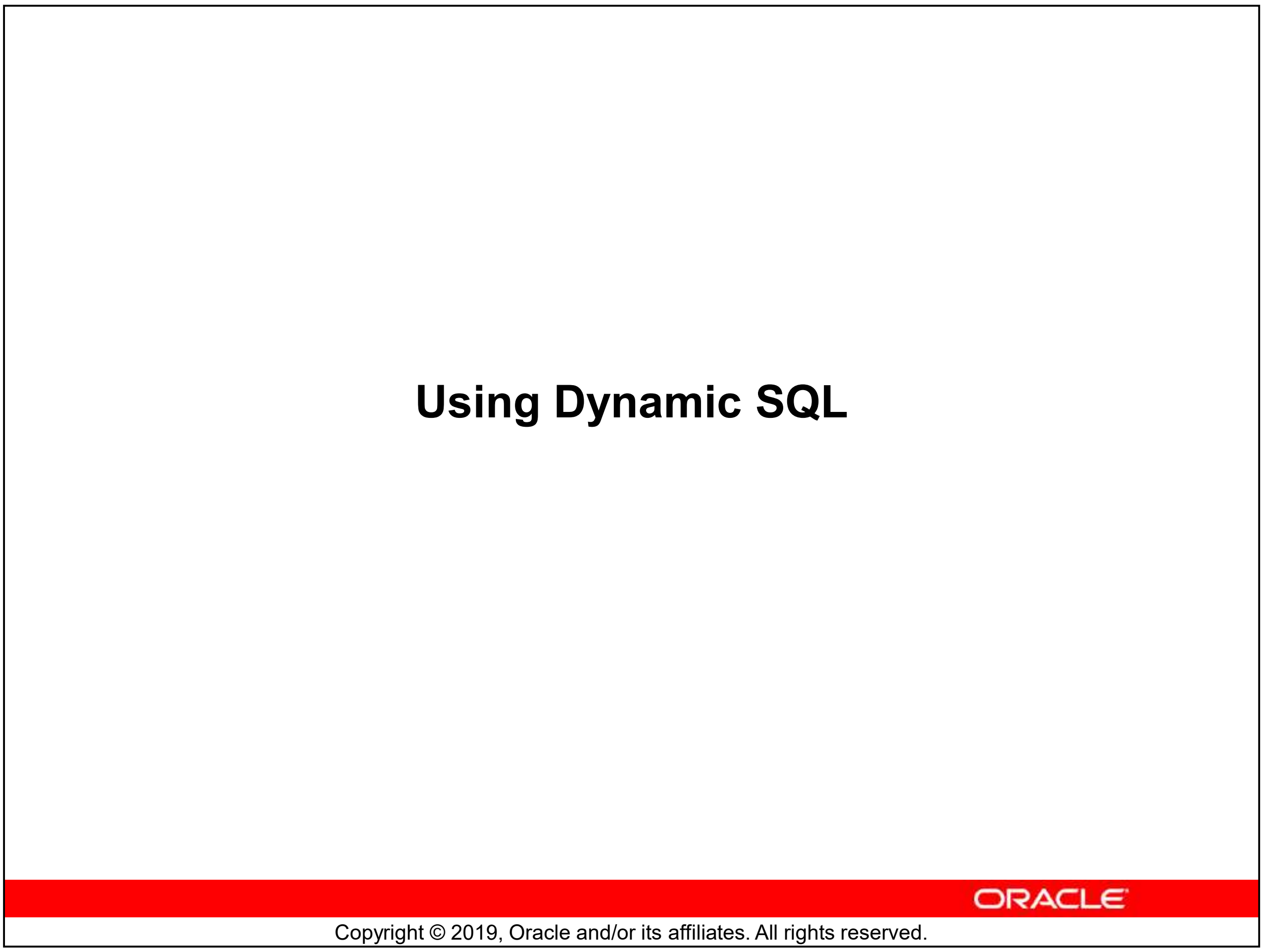
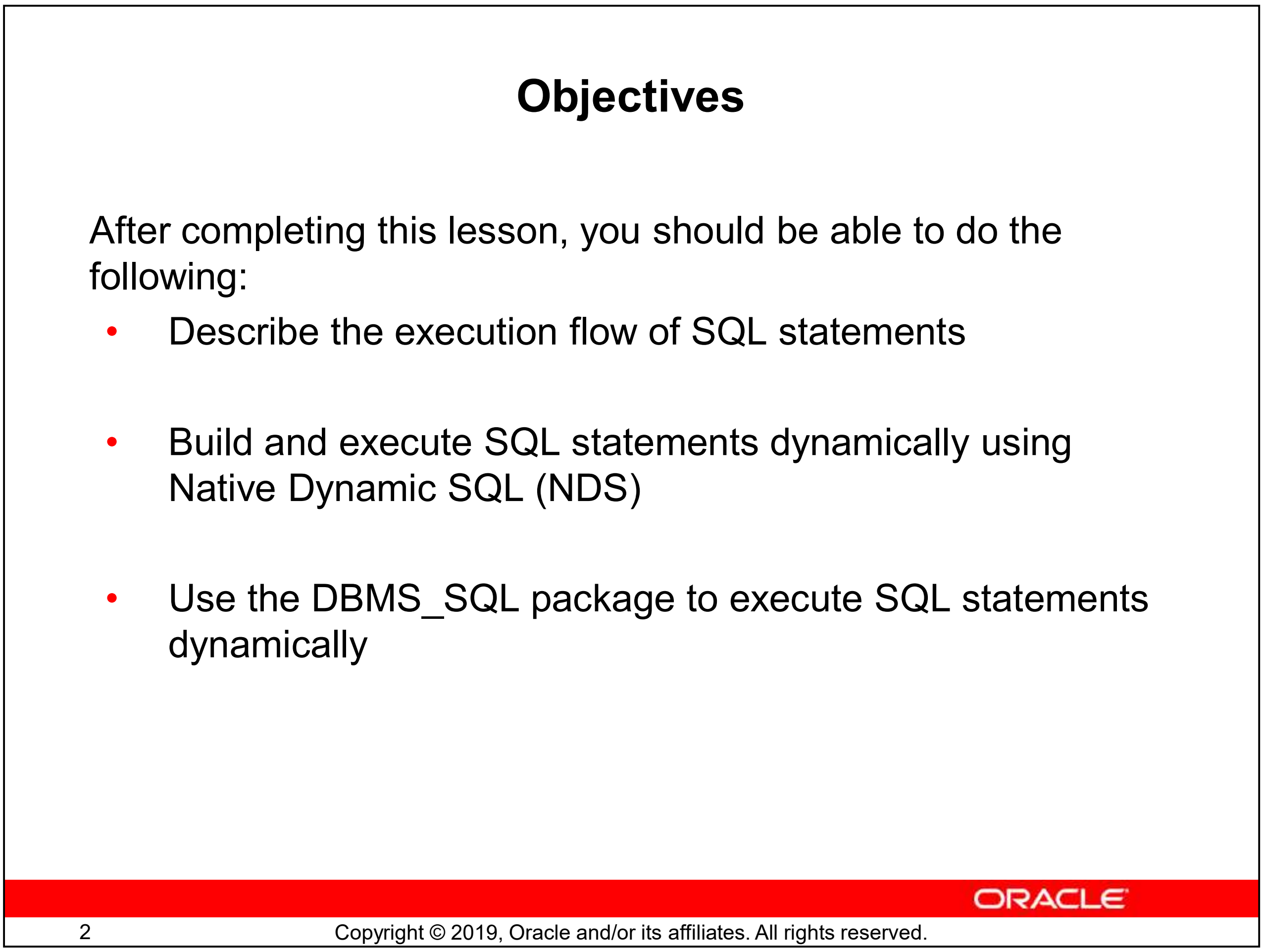


Using Dynamic SQL

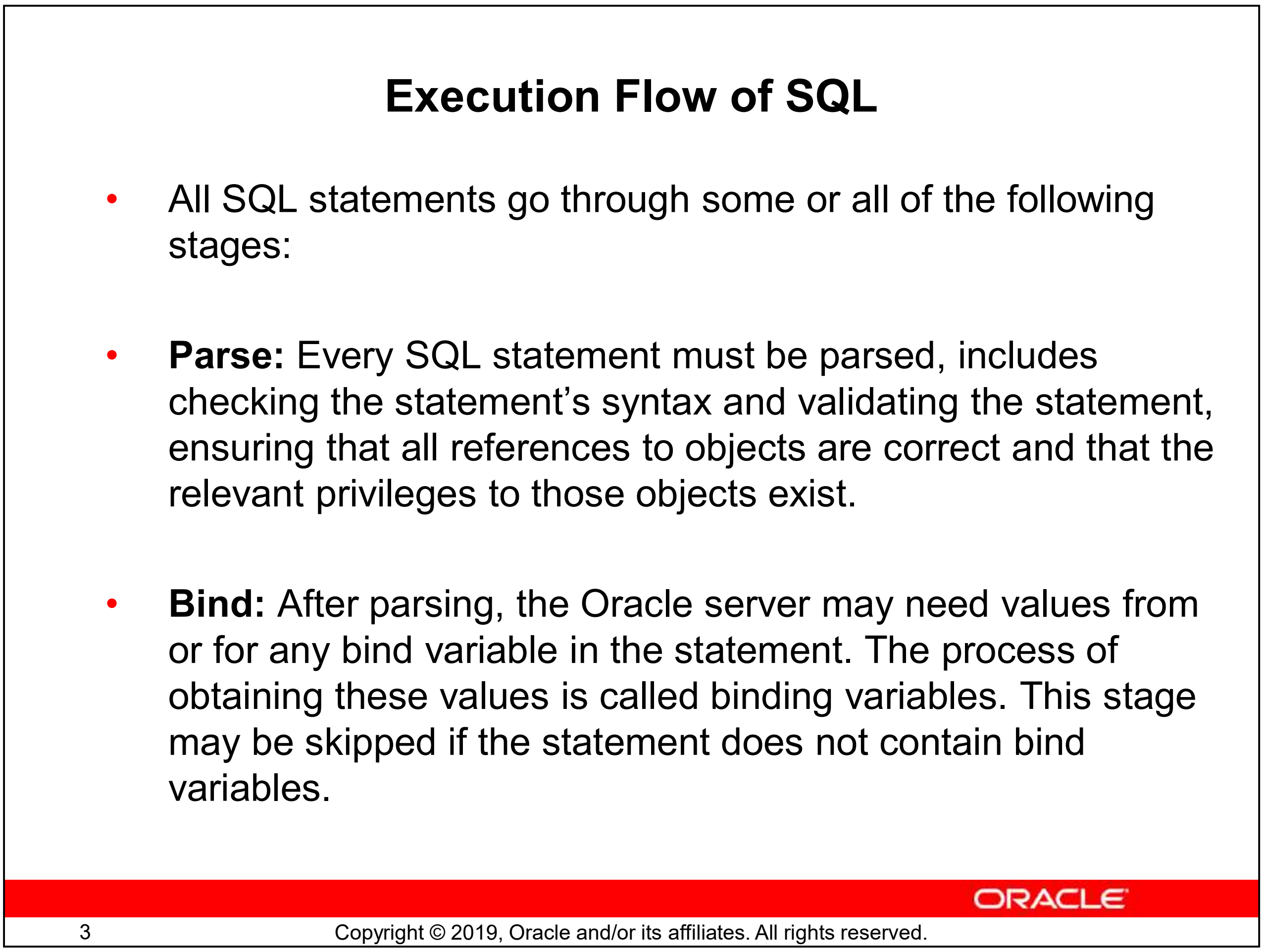


Objectives

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

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (NDS)
- Use the DBMS_SQL package to execute SQL statements dynamically

In this lesson, you learn to construct and execute SQL statements dynamically—that is, at run time using the Native Dynamic SQL statements in PL/SQL.



Execution Flow of SQL

- All SQL statements go through some or all of the following stages:

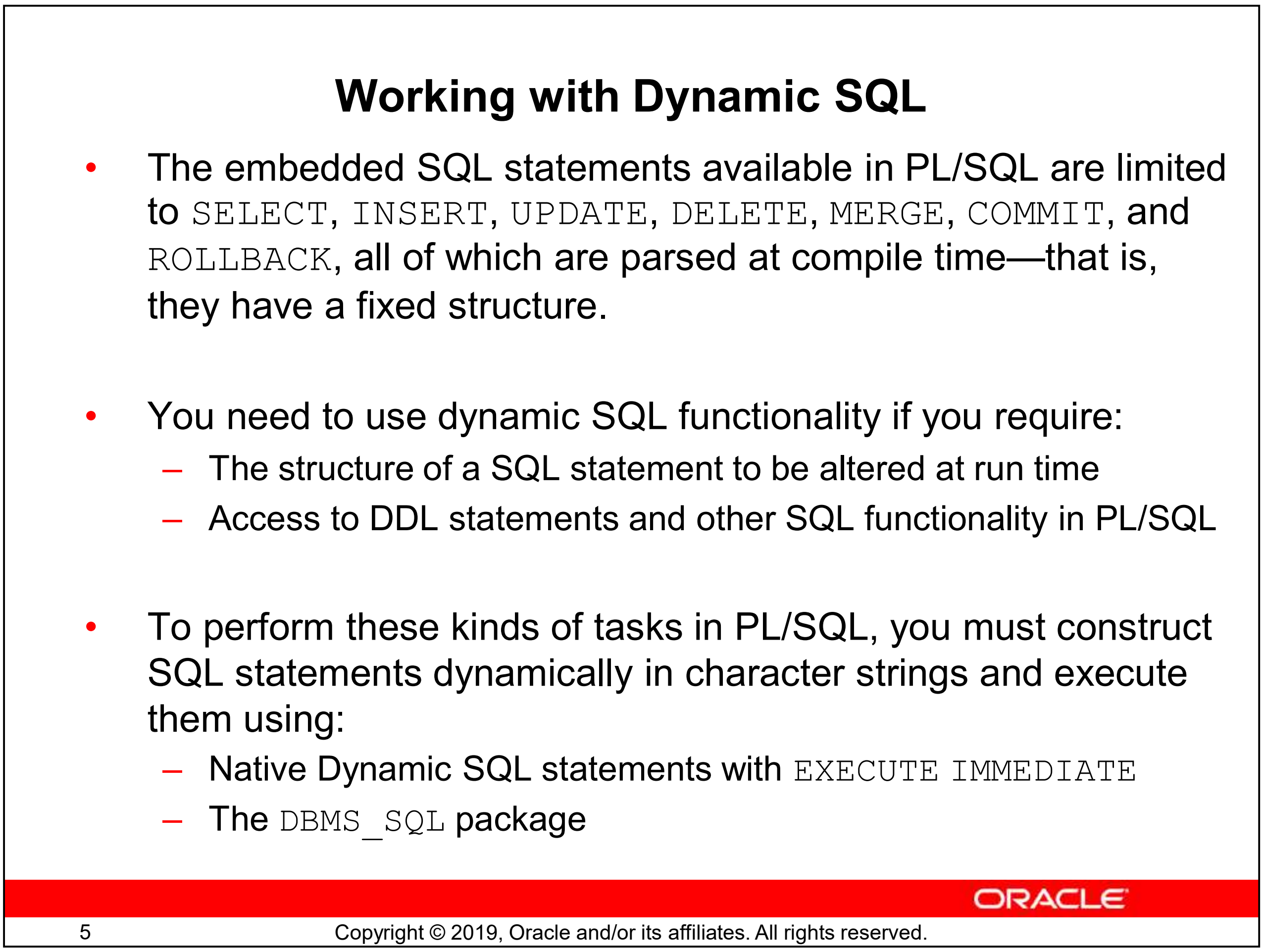
- **Parse:** Every SQL statement must be parsed, includes checking the statement’s syntax and validating the statement, ensuring that all references to objects are correct and that the relevant privileges to those objects exist.

- **Bind:** After parsing, the Oracle server may need values from or for any bind variable in the statement. The process of obtaining these values is called binding variables. This stage may be skipped if the statement does not contain bind variables.

Execution Flow of SQL

- **Execute:** At this point, the Oracle server has all necessary information and resources, and the statement is executed. For non-query statements, this is the last phase.
- **Fetch:** In the fetch stage, which is applicable to queries, the rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched.

- Some stages may not be relevant for all statements:
  - The fetch phase is applicable to queries.
  - For embedded SQL statements such as `SELECT`, DML, `COMMIT`, `SAVEPOINT`, and `ROLLBACK`, the parse and bind phases are done at compile time.
  - For dynamic SQL statements, all phases are performed at run time.



Working with Dynamic SQL

- The embedded SQL statements available in PL/SQL are limited to `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `COMMIT`, and `ROLLBACK`, all of which are parsed at compile time—that is, they have a fixed structure.
- You need to use dynamic SQL functionality if you require:
  - The structure of a SQL statement to be altered at run time
  - Access to DDL statements and other SQL functionality in PL/SQL
- To perform these kinds of tasks in PL/SQL, you must construct SQL statements dynamically in character strings and execute them using:
  - Native Dynamic SQL statements with `EXECUTE IMMEDIATE`
  - The `DBMS_SQL` package

Dynamic SQL

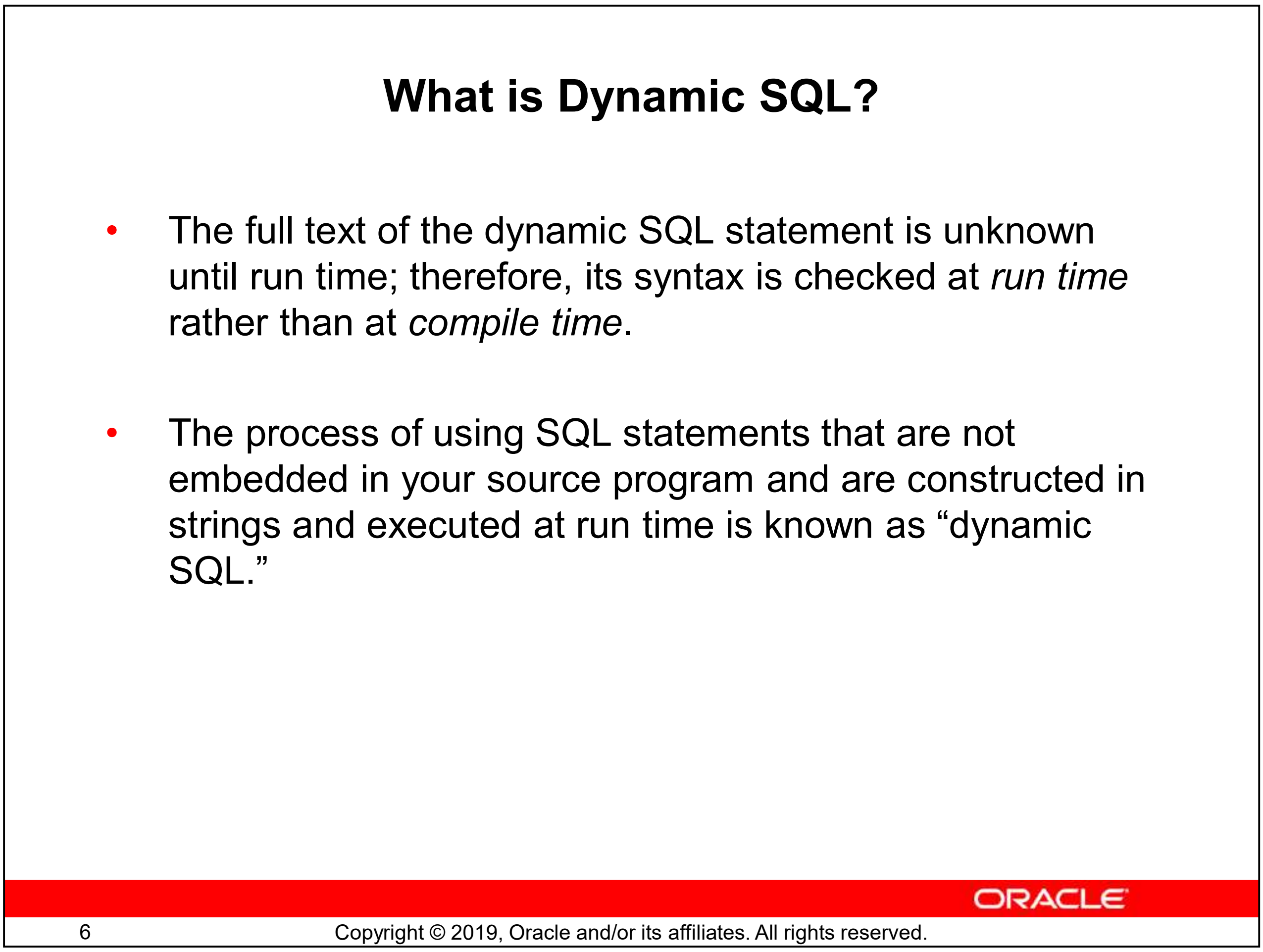
The embedded SQL statements available in PL/SQL are limited to `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `COMMIT`, and `ROLLBACK`, all of which are parsed at compile time—that is, they have a fixed structure. You need to use dynamic SQL functionality if you require:

- The structure of a SQL statement to be altered at run time
- Access to data definition language (DDL) statements and other SQL functionality in PL/SQL

To perform these kinds of tasks in PL/SQL, you must construct SQL statements dynamically in character strings and execute them using either of the following:

- Native Dynamic SQL statements with `EXECUTE IMMEDIATE`
- The `DBMS_SQL` package

The process of using SQL statements that are not embedded in your source program and are constructed in strings and executed at run time is known as “dynamic SQL.” The SQL statements are created dynamically at run time and can access and use PL/SQL variables. For example, you create a procedure that uses dynamic SQL to operate on a table whose name is not known until run time, or execute a DDL statement (such as `CREATE TABLE`), a data control statement (such as `GRANT`), or a session control statement (such as `ALTER SESSION`).



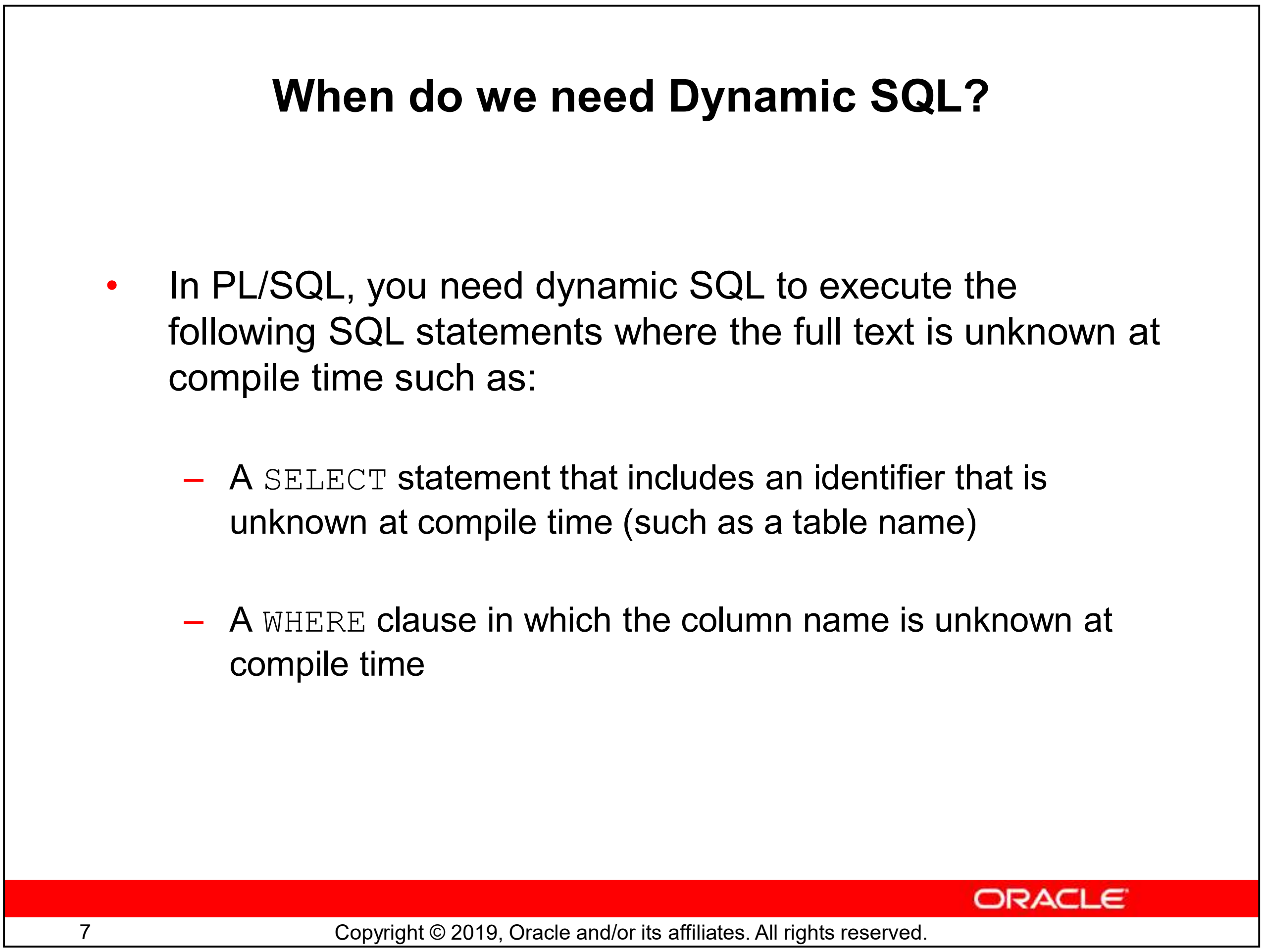
What is Dynamic SQL?

- The full text of the dynamic SQL statement is unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.
- The process of using SQL statements that are not embedded in your source program and are constructed in strings and executed at run time is known as “dynamic SQL.”

Note

For additional information about dynamic SQL, see the following resources:

- *Pro*C/C++ Programmer’s Guide*
    - *Lesson 13, Oracle Dynamic SQL,* covers the four available methods that you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, and then offers guidelines for choosing the right method. Later sections in the same guide show you how to use the methods, and include example programs that you can study.
    - *Lesson 15*, *Oracle Dynamic SQL: Method 4*, contains very detailed information about Method 4 when defining dynamic SQL statements.
- *Oracle PL/SQL Programming* book by Steven Feuerstein and Bill Pribyl. *Lesson 16*, *Dynamic SQL and Dynamic PL/SQL,* contains additional information about dynamic SQL.



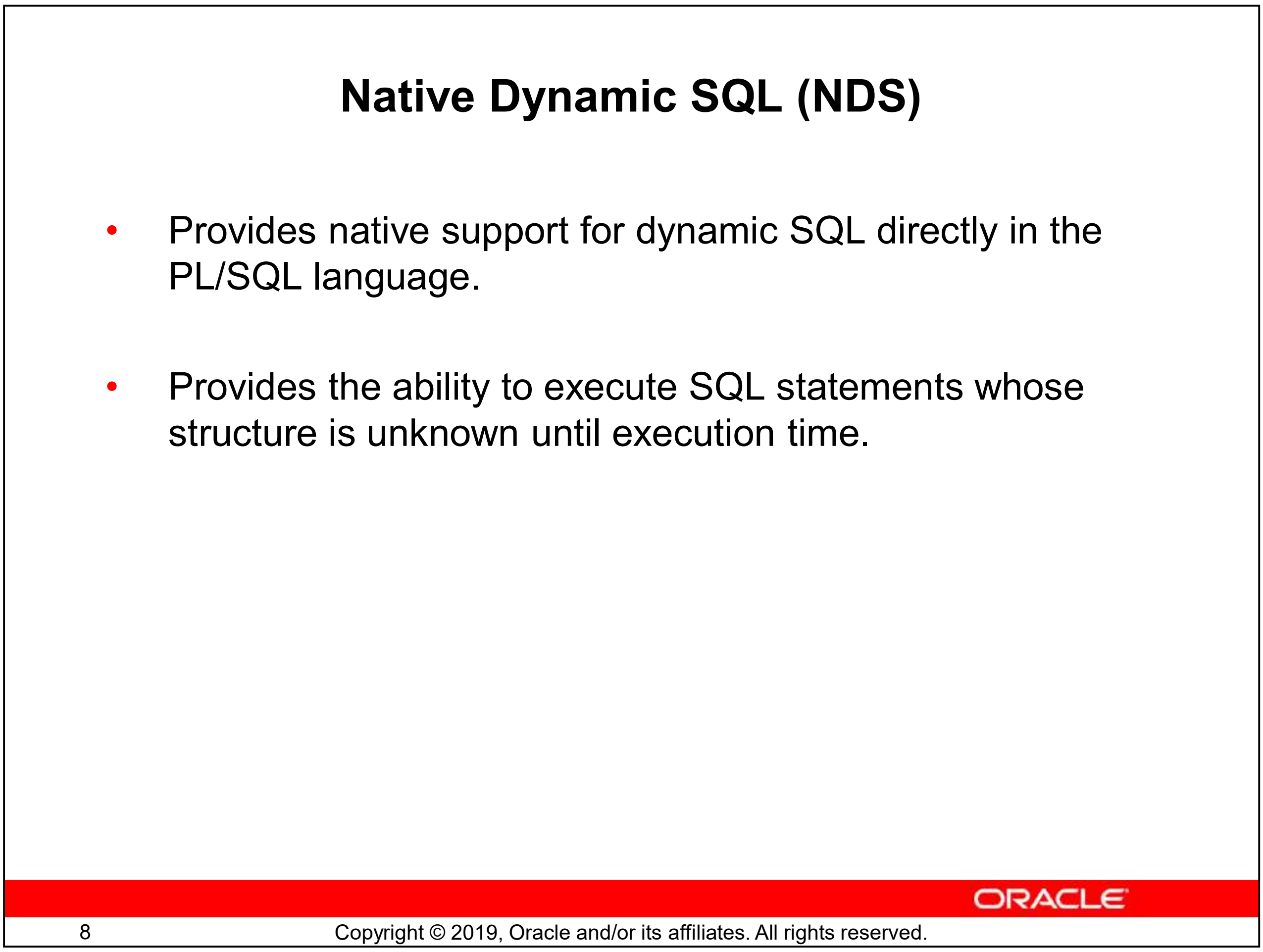
When do we need Dynamic SQL?

- In PL/SQL, you need dynamic SQL to execute the following SQL statements where the full text is unknown at compile time such as:
  - A `SELECT` statement that includes an identifier that is unknown at compile time (such as a table name)
  - A `WHERE` clause in which the column name is unknown at compile time

Note

For additional information about dynamic SQL, see the following resources:

- *Pro*C/C++ Programmer's Guide*
  - *Lesson 13, Oracle Dynamic SQL,* covers the four available methods that you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, and then offers guidelines for choosing the right method. Later sections in the same guide show you how to use the methods, and include example programs that you can study.
  - *Lesson 15*, *Oracle Dynamic SQL: Method 4*, contains very detailed information about Method 4 when defining dynamic SQL statements.
- *Oracle PL/SQL Programming* book by Steven Feuerstein and Bill Pribyl. *Lesson 16*, *Dynamic SQL and Dynamic PL/SQL,* contains additional information about dynamic SQL.



Native Dynamic SQL (NDS)

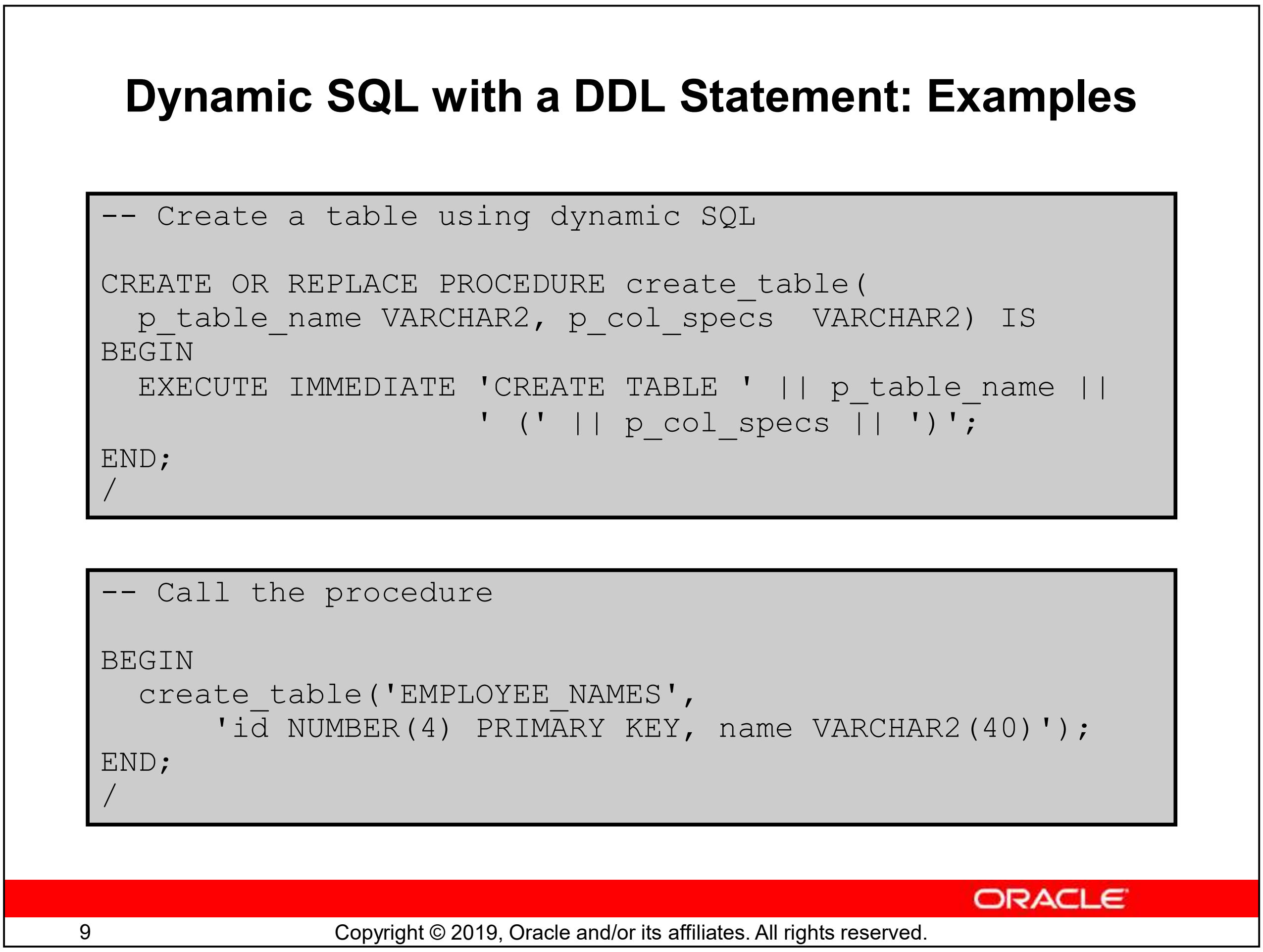
- Provides native support for dynamic SQL directly in the PL/SQL language.
- Provides the ability to execute SQL statements whose structure is unknown until execution time.

Native Dynamic SQL provides the ability to dynamically execute SQL statements whose structure is constructed at execution time. The following statements have been added or extended in PL/SQL to support Native Dynamic SQL:

- **`EXECUTE IMMEDIATE`:** Prepares a statement, executes it, returns variables, and then deallocates resources
- **`OPEN-FOR`:** Prepares and executes a statement using a cursor variable
- **`FETCH`:** Retrieves the results of an opened statement by using the cursor variable
- **`CLOSE`:** Closes the cursor used by the cursor variable and deallocates resources

You can use bind variables in the dynamic parameters in the `EXECUTE IMMEDIATE` and `OPEN` statements. Native Dynamic SQL includes the following capabilities:

- Define a dynamic SQL statement.
- Handle `IN`, `IN OUT`, and `OUT` bind variables that are bound by position, not by name.



Dynamic SQL with a DDL Statement: Examples

```
-- Create a table using dynamic SQL

CREATE OR REPLACE PROCEDURE create_table(
  p_table_name VARCHAR2, p_col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE ' || p_table_name ||
                    ' (' || p_col_specs || ')';
END;
/
```

```
-- Call the procedure

BEGIN
  create_table('EMPLOYEE_NAMES',
      'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```

The code examples show the creation of a `create_table` procedure that accepts the table name and column definitions (specifications) as parameters.

The procedure call shows the creation of a table called `EMPLOYEE_NAMES` with two columns:

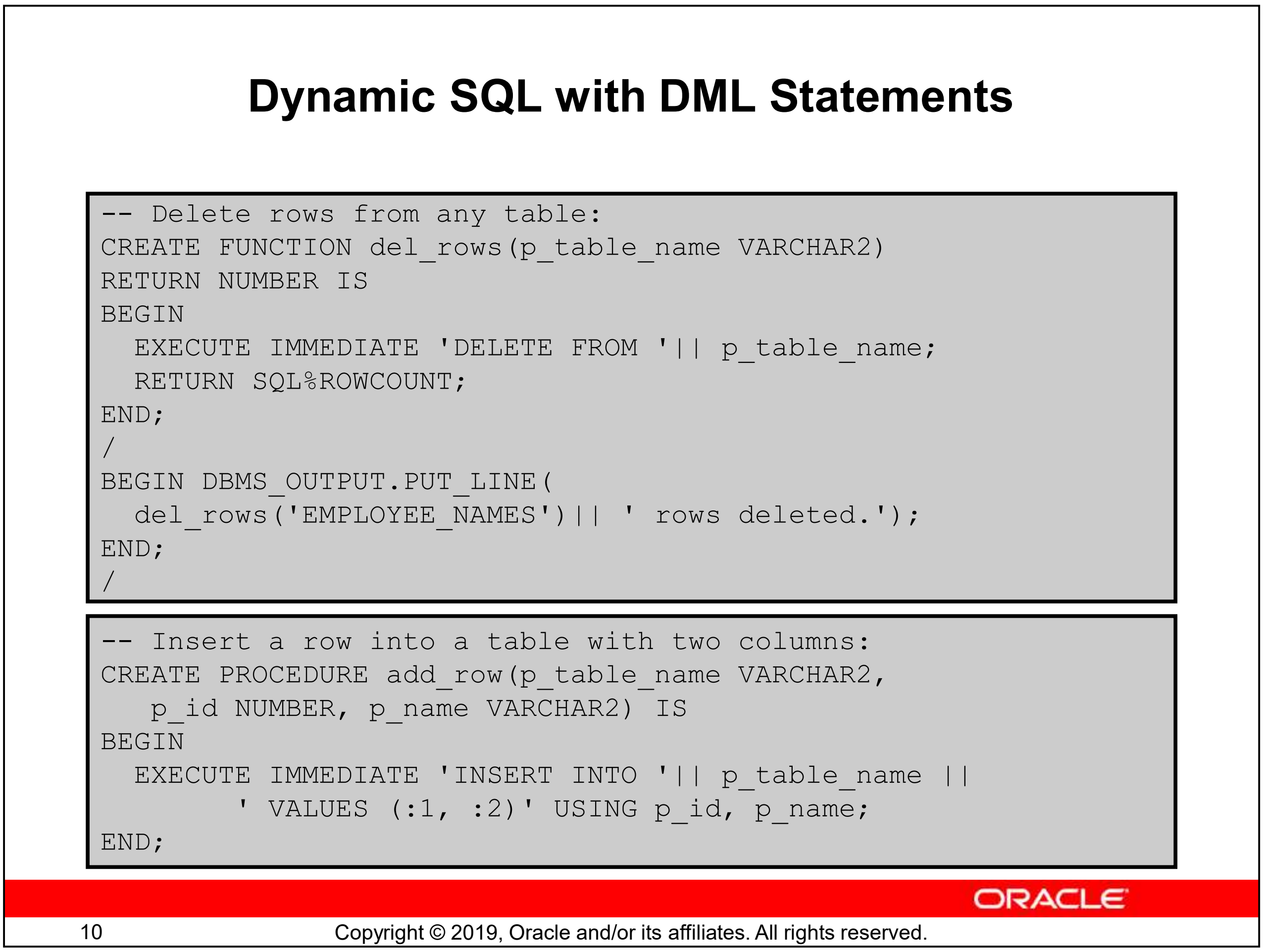
- An ID column with a `NUMBER` data type used as a primary key
- A name column of up to 40 characters for the employee name

Any DDL statement can be executed by using the syntax shown in the slide, whether the statement is dynamically constructed or specified as a literal string. You can create and execute a statement that is stored in a PL/SQL string variable, as in the following example:

```
CREATE OR REPLACE PROCEDURE add_col(p_table_name VARCHAR2,
                         p_col_spec   VARCHAR2) IS
  v_stmt VARCHAR2(100) := 'ALTER TABLE ' || p_table_name ||
                          ' ADD '|| p_col_spec;
BEGIN
  EXECUTE IMMEDIATE v_stmt;
END;
/
```

To add a new column to a table, enter the following:

```
EXECUTE add_col('employee_names', 'salary number(8,2)')
```



Dynamic SQL with DML Statements

```
-- Delete rows from any table:
CREATE FUNCTION del_rows(p_table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM '|| p_table_name;
  RETURN SQL%ROWCOUNT;
END;
/
BEGIN DBMS_OUTPUT.PUT_LINE(
  del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');
END;
/
```

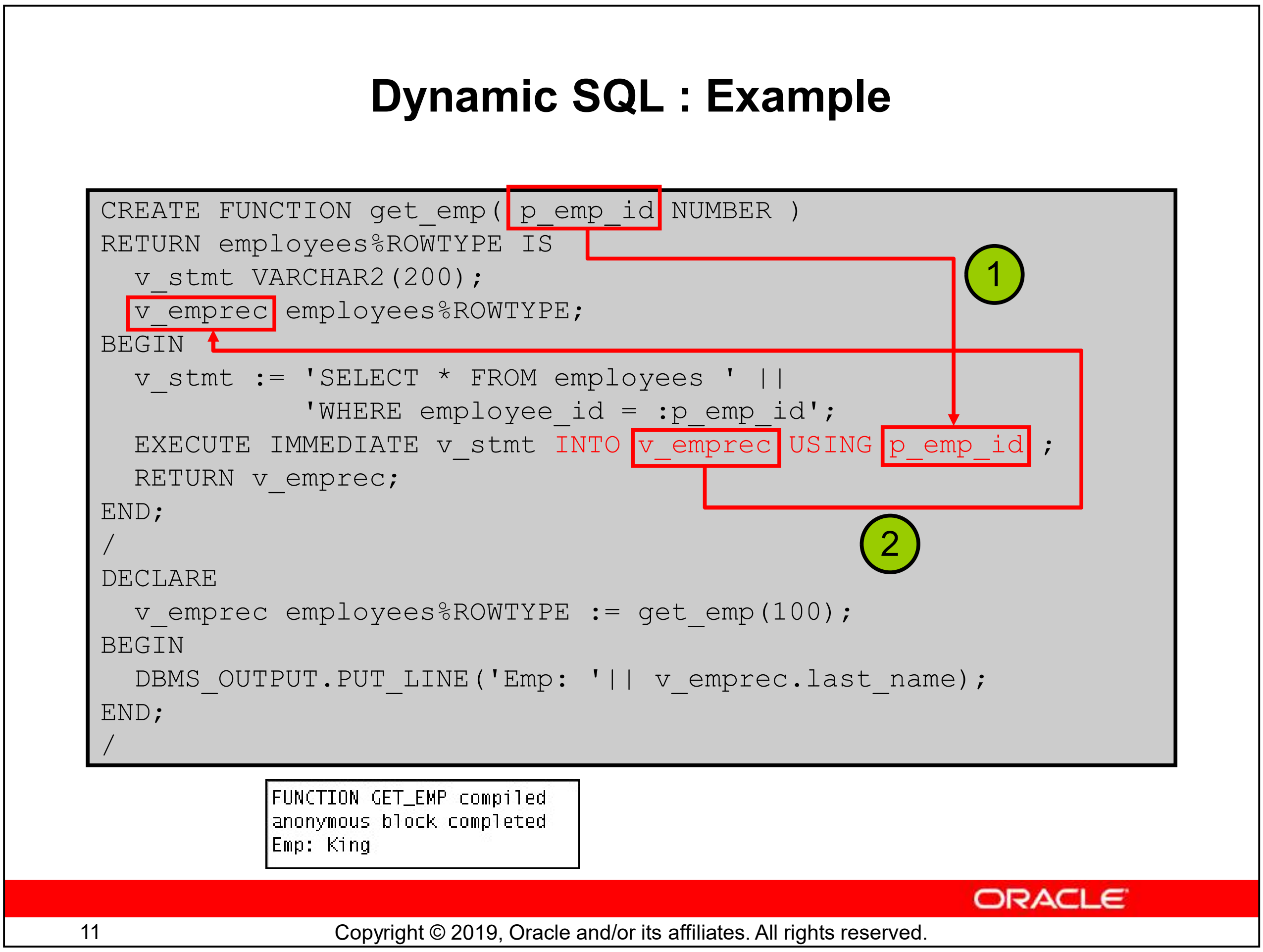
```
-- Insert a row into a table with two columns:
CREATE PROCEDURE add_row(p_table_name VARCHAR2,
   p_id NUMBER, p_name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '|| p_table_name ||
        ' VALUES (:1, :2)' USING p_id, p_name;
END;
```

The first code example in the slide defines a dynamic SQL statement using Method 1—that is, nonquery without host variables. The examples in the slide demonstrate the following:

- The del_rows function deletes rows from a specified table and returns the number of rows deleted by using the implicit SQL cursor %ROWCOUNT attribute. Executing the function is shown below the example for creating a function.
- The add_row procedure shows how to provide input values to a dynamic SQL statement with the USING clause. The bind variable names :1 and :2 are not important; however, the order of the parameter names (p_id and p_name) in the USING clause is associated with the bind variables by position, in the order of their respective appearance. Therefore, the PL/SQL parameter p_id is assigned to the :1 placeholder, and the p_name parameter is assigned to the :2 placeholder. Placeholder or bind variable names can be alphanumeric but must be preceded with a colon.

Note: The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes the dynamic SQL statement. Dynamic SQL statements are always parsed.

Also, note that a COMMIT operation is not performed in either of the examples. Therefore, the operations can be undone with a ROLLBACK statement.



Dynamic SQL : Example

```
CREATE FUNCTION get_emp( p_emp_id NUMBER )
RETURN employees%ROWTYPE IS
  v_stmt VARCHAR2(200);
  v_emprec employees%ROWTYPE;
BEGIN
  v_stmt := 'SELECT * FROM employees ' ||
            'WHERE employee_id = :p_emp_id';
  EXECUTE IMMEDIATE v_stmt INTO v_emprec USING p_emp_id ;
  RETURN v_emprec;
END;
/
DECLARE
  v_emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '|| v_emprec.last_name);
END;
/
```

```
FUNCTION GET_EMP compiled
anonymous block completed
Emp: King
```

The code example in the slide is an example of defining a dynamic SQL statement using Method 3 with a single row queried—that is, query with a known number of select-list items and input host variables.

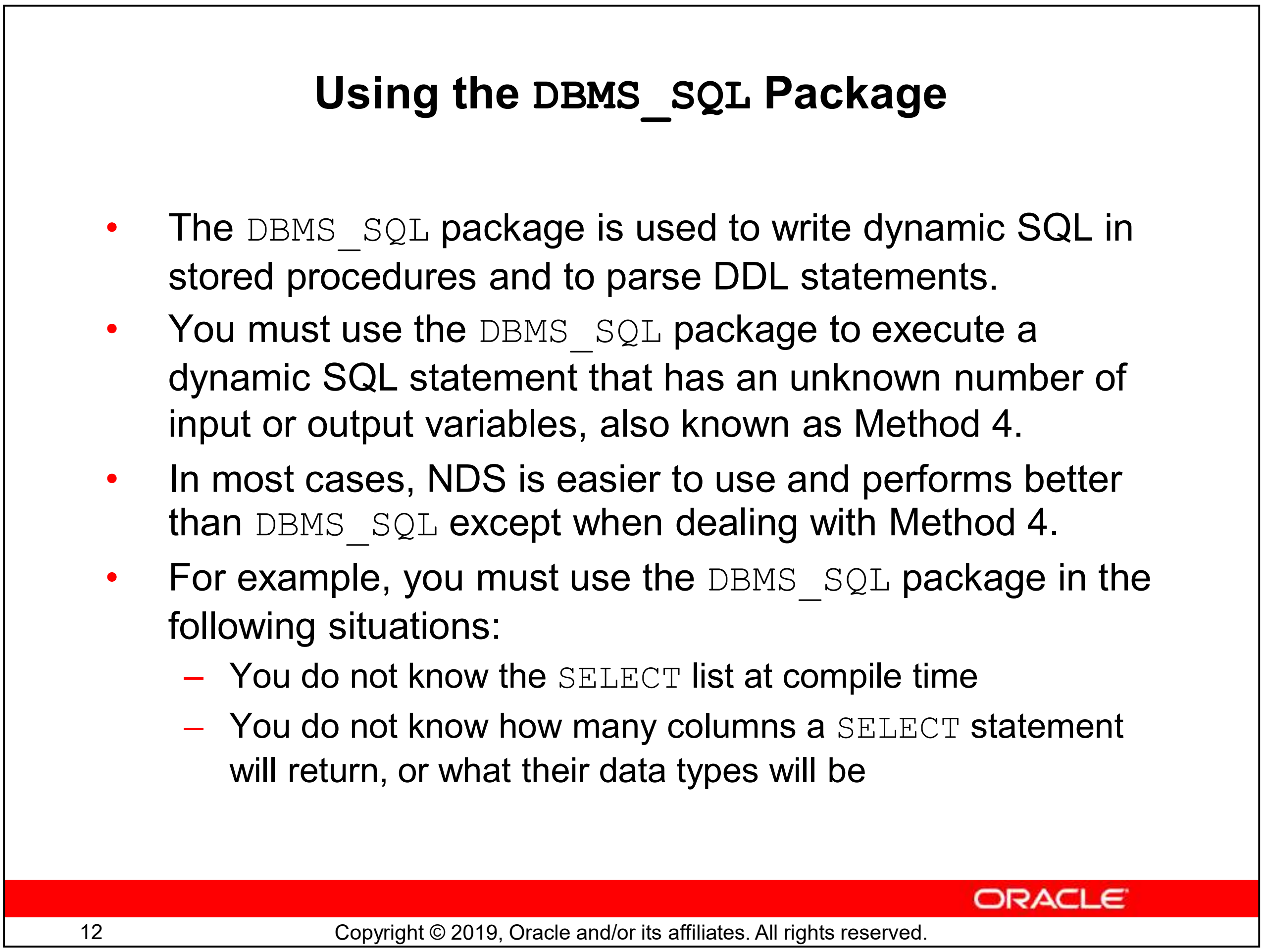
The single-row query example demonstrates the `get_emp` function that retrieves an `EMPLOYEES` record into a variable specified in the `INTO` clause. It also shows how to provide input values for the `WHERE` clause.

The anonymous block is used to execute the `get_emp` function and return the result into a local `EMPLOYEES` record variable.

The example could be enhanced to provide alternative `WHERE` clauses depending on input parameter values, making it more suitable for dynamic SQL processing.

Note

- For an example of “Dynamic SQL with a Multirow Query: Example” using `REF CURSORS`, see the `demo_07_13_a` in the `/home/oracle/labs/plpu/demo` folder.
- For an example on using `REF CURSORS`, see the `demo_07_13_b` in the `/home/oracle/labs/plpu/demo` folder.
- `REF CURSORS` are covered in the *Oracle Database: Advanced PL/SQL* course.



Using the DBMS_SQL Package

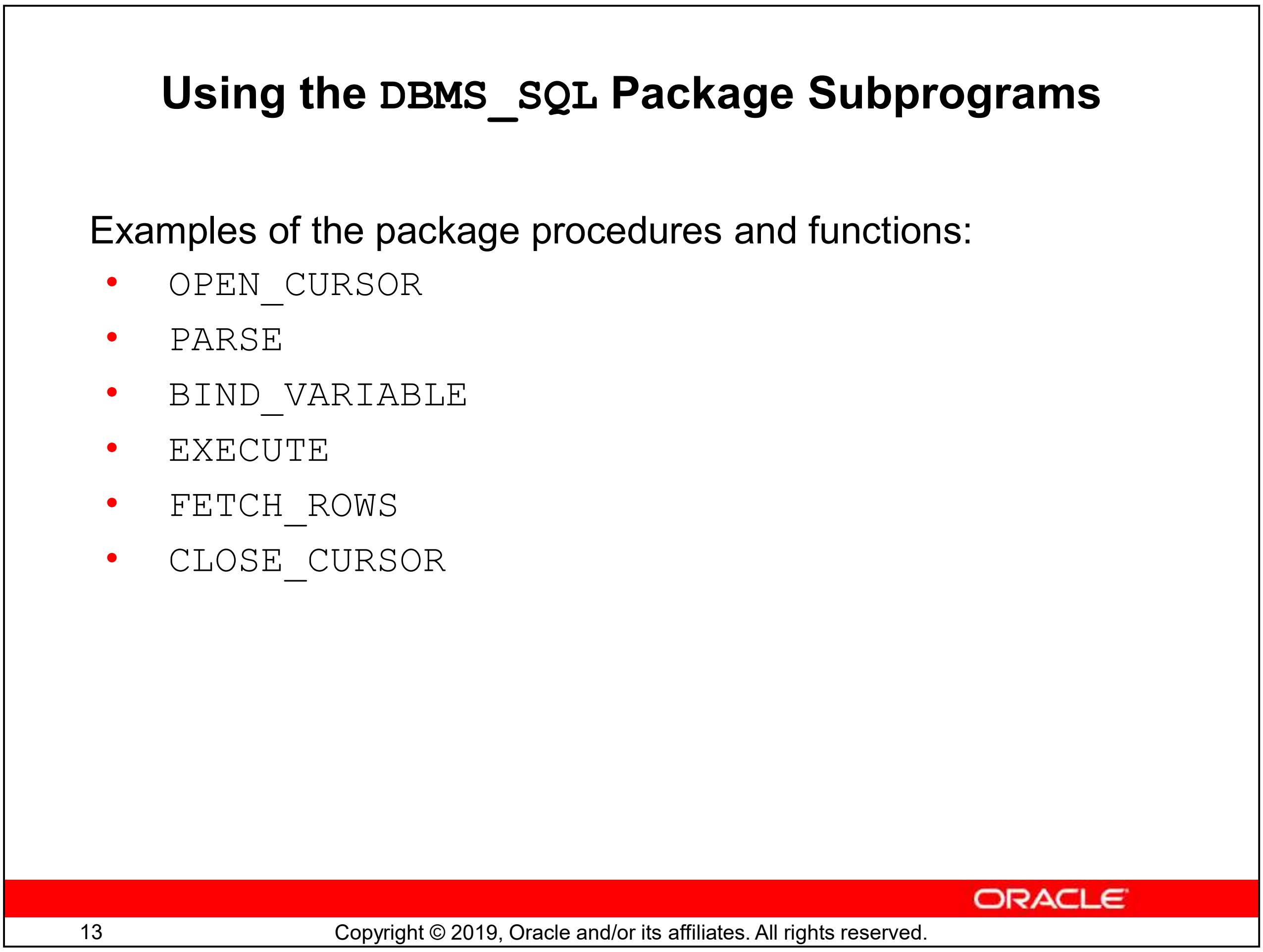
- The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements.
- You must use the DBMS_SQL package to execute a dynamic SQL statement that has an unknown number of input or output variables, also known as Method 4.
- In most cases, NDS is easier to use and performs better than DBMS_SQL except when dealing with Method 4.
- For example, you must use the DBMS_SQL package in the following situations:
  - You do not know the SELECT list at compile time
  - You do not know how many columns a SELECT statement will return, or what their data types will be

Using DBMS_SQL, you can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL, such as executing DDL statements in PL/SQL—for example, executing a DROP TABLE statement. The operations provided by this package are performed under the current user, not under the package owner SYS.

Method 4: Method 4 refers to situations where, in a dynamic SQL statement, the number of columns selected for a query or the number of bind variables set is not known until run time. In this case, you should use the DBMS_SQL package.

When generating dynamic SQL, you can either use the DBMS_SQL supplied package when dealing with Method 4 situations, or you can use native dynamic SQL. Before Oracle Database 11*g*, each of these methods had functional limitations. In Oracle Database 11*g*, functionality is added to both methods to make them more complete.

The features for executing dynamic SQL from PL/SQL had some restrictions in Oracle Database 10*g*. DBMS_SQL was needed for Method 4 scenarios but it could not handle the full range of data types and its cursor representation was not usable by a client to the database. Native dynamic SQL was more convenient for non–Method 4 scenarios, but it did not support statements bigger than 32 KB. Oracle Database 11*g* removes these and other restrictions to make the support of dynamic SQL from PL/SQL functionally complete.



Using the DBMS_SQL Package Subprograms

Examples of the package procedures and functions:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

The DBMS_SQL package provides the following subprograms to execute dynamic SQL:

- OPEN_CURSOR to open a new cursor and return a cursor ID number
- PARSE to parse the SQL statement. Every SQL statement must be parsed by calling the PARSE procedures. Parsing the statement checks the statement’s syntax and associates it with the cursor in your program. You can parse any DML or DDL statement. DDL statements are immediately executed when parsed.
- BIND_VARIABLE to bind a given value to a bind variable identified by its name in the statement being parsed. This is not needed if the statement does not have bind variables.
- EXECUTE to execute the SQL statement and return the number of rows processed
- FETCH_ROWS to retrieve the next row for a query (use in a loop for multiple rows)
- CLOSE_CURSOR to close the specified cursor

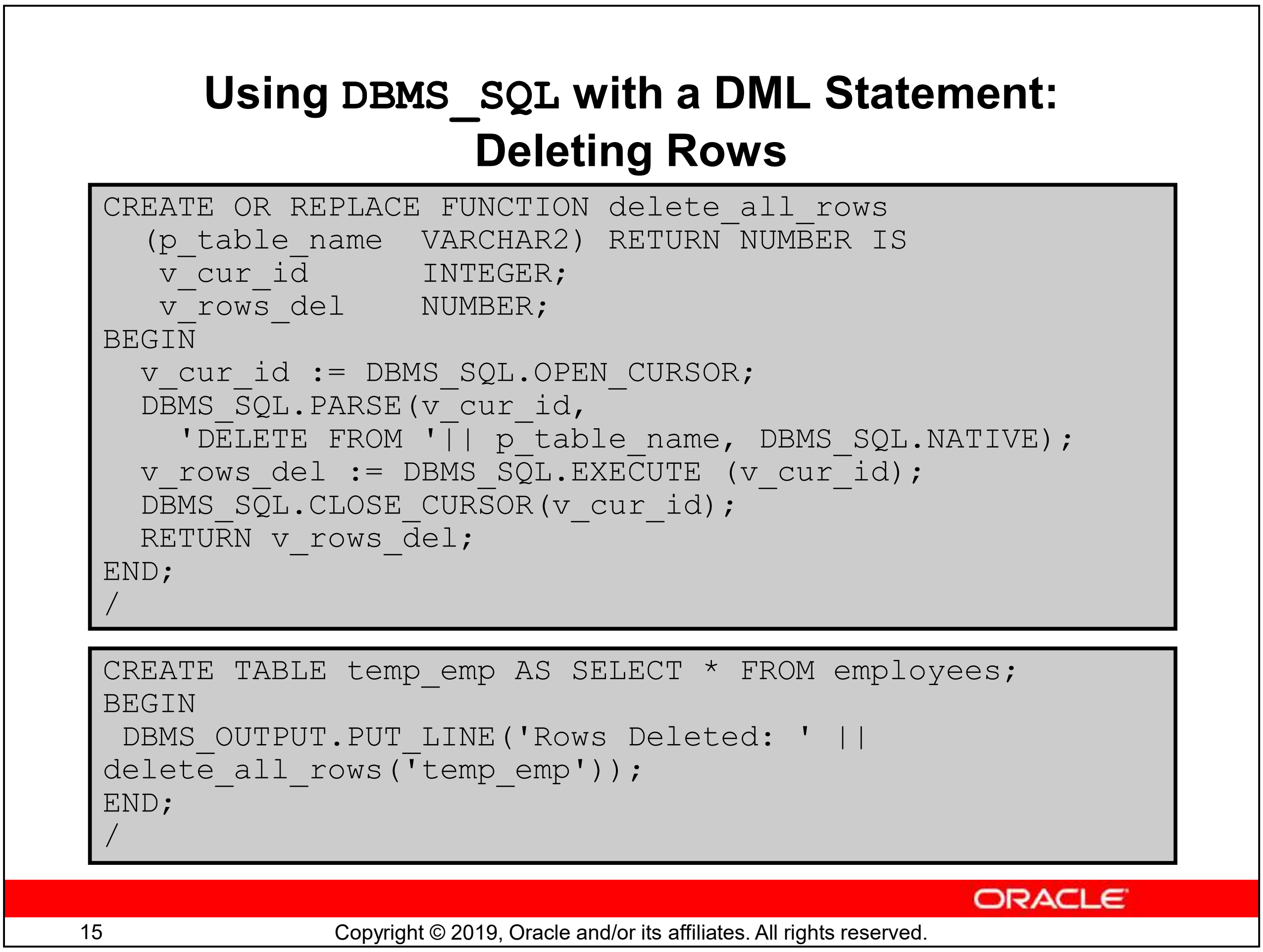
Note: Using the DBMS_SQL package to execute DDL statements can result in a deadlock. For example, the most likely reason is that the package is being used to drop a procedure that you are still using.

The PARSE Procedure Parameters

The LANGUAGE_FLAG parameter of the PARSE procedure determines how Oracle handles the SQL statement—that is, using behavior associated with a specific Oracle database version. Using NATIVE (or 1) for this parameter specifies using the normal behavior associated with the database to which the program is connected.

If the LANGUAGE_FLAG parameter is set to V6 (or 0), that specifies version 6 behavior. If the LANGUAGE_FLAG parameter is set to V7 (or 2), that specifies Oracle database version 7 behavior.

Note: For additional information, see *Oracle Database PL/SQL Packages and Types Reference*.



Using DBMS_SQL with a DML Statement: Deleting Rows

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (p_table_name  VARCHAR2) RETURN NUMBER IS
   v_cur_id      INTEGER;
   v_rows_del    NUMBER;
BEGIN
  v_cur_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_cur_id,
    'DELETE FROM '|| p_table_name, DBMS_SQL.NATIVE);
  v_rows_del := DBMS_SQL.EXECUTE (v_cur_id);
  DBMS_SQL.CLOSE_CURSOR(v_cur_id);
  RETURN v_rows_del;
END;
/
```

```
CREATE TABLE temp_emp AS SELECT * FROM employees;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

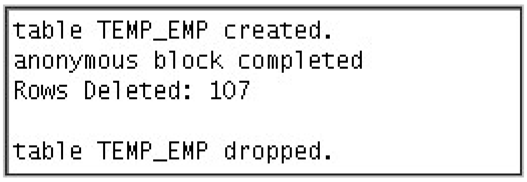
Using DBMS_SQL with a DML Statement

In the slide, the table name is passed into the delete_all_rows function. The function uses dynamic SQL to delete rows from the specified table, and returns a count representing the number of rows that are deleted after successful execution of the statement.

To process a DML statement dynamically, perform the following steps:

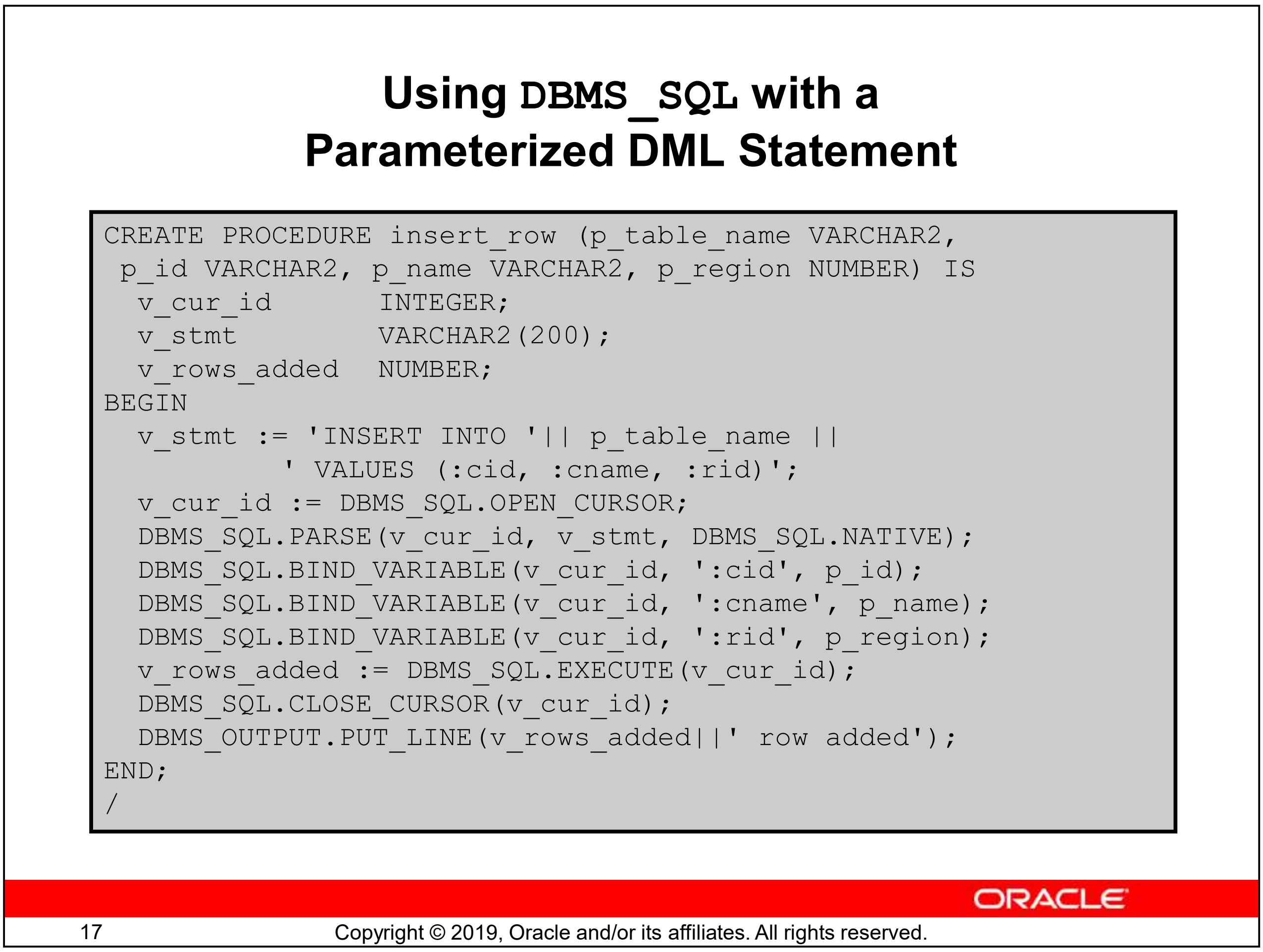
1. Use OPEN_CURSOR to establish an area in memory to process a SQL statement.
2. Use PARSE to establish the validity of the SQL statement.
3. Use the EXECUTE function to run the SQL statement. This function returns the number of rows processed.
4. Use CLOSE_CURSOR to close the cursor.

The steps to execute a DDL statement are similar; but step 3 is optional because a DDL statement is immediately executed when the `PARSE` is successfully done—that is, the statement syntax and semantics are correct. If you use the `EXECUTE` function with a DDL statement, then it does not do anything and returns a value of `0` for the number of rows processed because DDL statements do not process rows.



```
table TEMP_EMP created.
anonymous block completed
Rows Deleted: 107

table TEMP_EMP dropped.
```



Using DBMS_SQL with a Parameterized DML Statement

```
CREATE PROCEDURE insert_row (p_table_name VARCHAR2,
 p_id VARCHAR2, p_name VARCHAR2, p_region NUMBER) IS
  v_cur_id       INTEGER;
  v_stmt         VARCHAR2(200);
  v_rows_added   NUMBER;
BEGIN
  v_stmt := 'INSERT INTO '|| p_table_name ||
          ' VALUES (:cid, :cname, :rid)';
  v_cur_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_cur_id, v_stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(v_cur_id, ':cid', p_id);
  DBMS_SQL.BIND_VARIABLE(v_cur_id, ':cname', p_name);
  DBMS_SQL.BIND_VARIABLE(v_cur_id, ':rid', p_region);
  v_rows_added := DBMS_SQL.EXECUTE(v_cur_id);
  DBMS_SQL.CLOSE_CURSOR(v_cur_id);
  DBMS_OUTPUT.PUT_LINE(v_rows_added||' row added');
END;
/
```

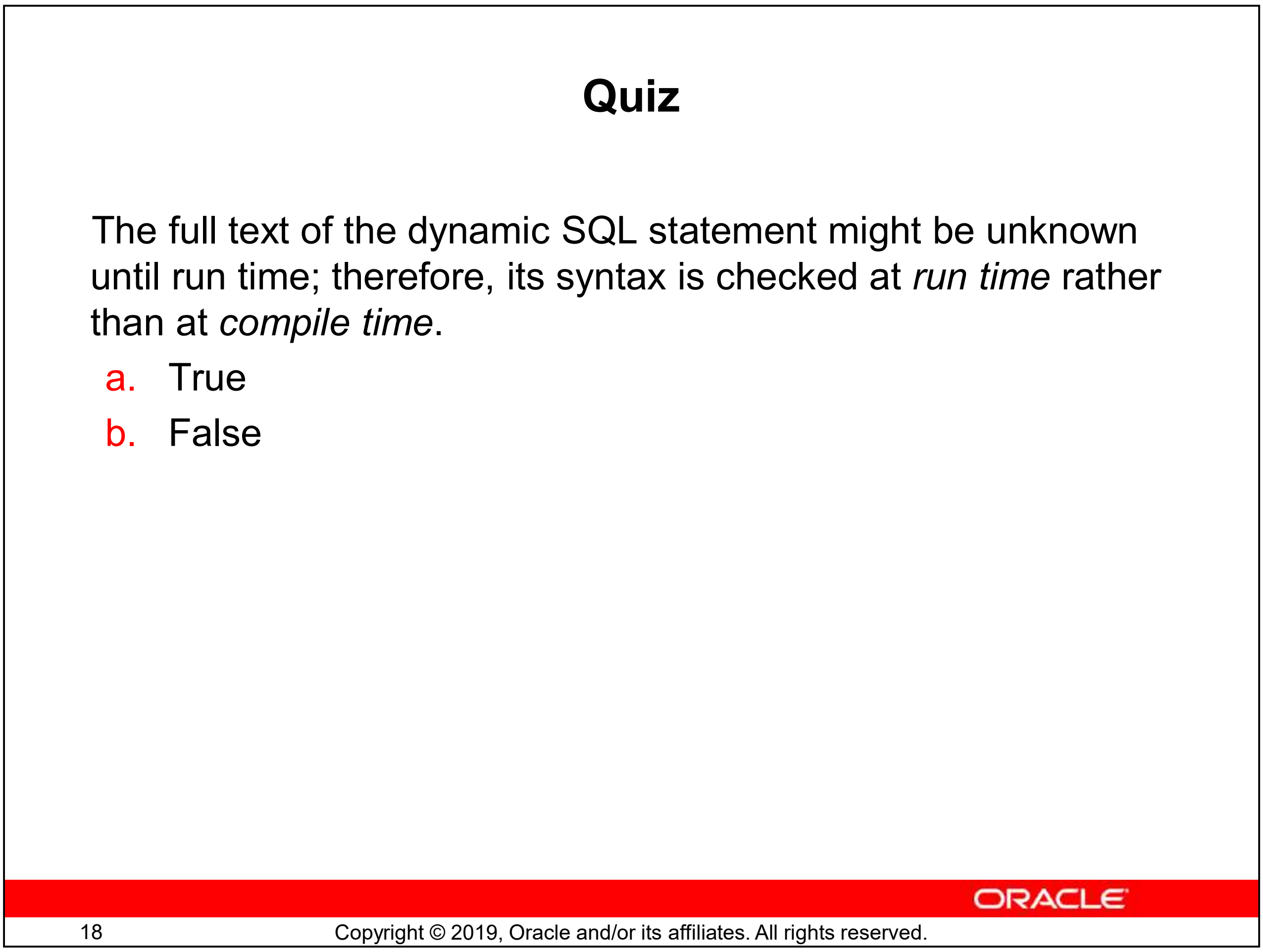
The example in the slide performs the DML operation to insert a row into a specified table. The example demonstrates the extra step required to associate values to bind variables that exist in the SQL statement. For example, a call to the procedure shown in the slide is:

```
EXECUTE insert_row('countries', 'LB', 'Lebanon', 4)
```

After the statement is parsed, you must call the DBMS_SQL.BIND_VARIABLE procedure to assign values for each bind variable that exists in the statement. The binding of values must be done before executing the code. To process a SELECT statement dynamically, perform the following steps after opening and before closing the cursor:

1. Execute DBMS_SQL.DEFINE_COLUMN for each column selected.
2. Execute DBMS_SQL.BIND_VARIABLE for each bind variable in the query.
3. For each row, perform the following steps:
   a. Execute DBMS_SQL.FETCH_ROWS to retrieve a row and return the number of rows fetched. Stop additional processing when a zero value is returned.
   b. Execute DBMS_SQL.COLUMN_VALUE to retrieve each selected column value into each PL/SQL variable for processing.

Although this coding process is not complex, it is more time consuming to write and is prone to error compared with using the Native Dynamic SQL approach.



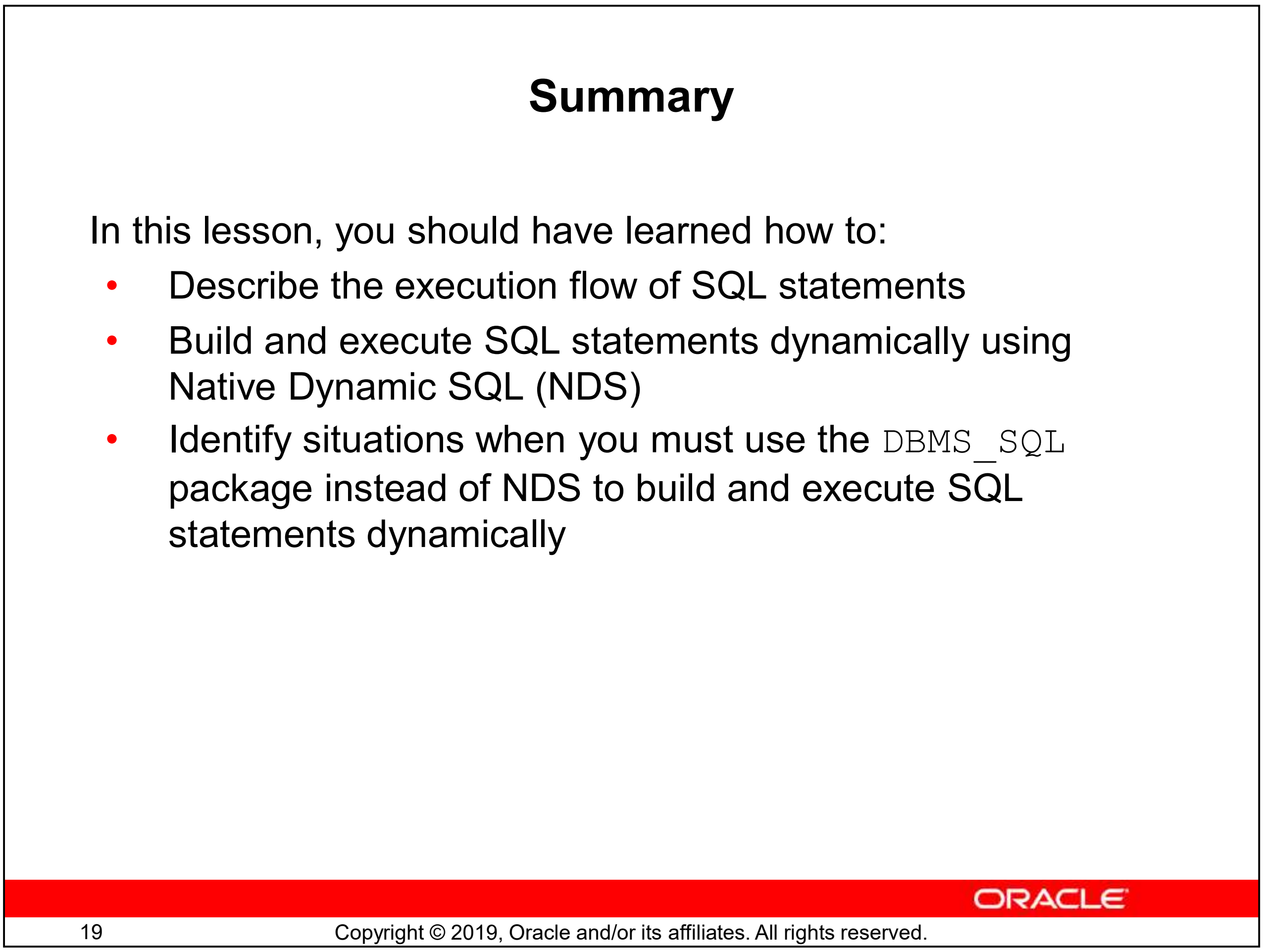
Quiz

The full text of the dynamic SQL statement might be unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.

a. True

b. False

Answer: a



Summary

In this lesson, you should have learned how to:

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (NDS)
- Identify situations when you must use the `DBMS_SQL` package instead of NDS to build and execute SQL statements dynamically

In this lesson, you discovered how to dynamically create any SQL statement and execute it using the Native Dynamic SQL statements. Dynamically executing SQL and PL/SQL code extends the capabilities of PL/SQL beyond query and transactional operations. For earlier releases of the database, you could achieve similar results with the `DBMS_SQL` package.