6

Dynamic SQL and Metadata

ORACLE

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 (that is, with `EXECUTE IMMEDIATE` statements)**
- **Compare Native Dynamic SQL with the `DBMS_SQL` package approach**
- **Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects**

Lesson Aim

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. You compare Native Dynamic SQL to the `DBMS_SQL` package, which provides similar capabilities.

You learn how to use the `DBMS_METADATA` package to retrieve metadata from the database dictionary as Extensible Markup Language (XML) or creation DDL and to submit the XML to re-create the object.

Execution Flow of SQL

- All SQL statements go through various stages:
  - Parse
  - Bind
  - Execute
  - Fetch
- Some stages may not be relevant for all statements—for example, the fetch phase is applicable to queries.

Note: For embedded SQL statements (`SELECT`, DML, `COMMIT`, and `ROLLBACK`), the parse and bind phases are done at compile time. For dynamic SQL statements, all phases are performed at run time.

Steps to Process SQL Statements

All SQL statements have to go through various stages. However, some stages may not be relevant for all statements. The following are the key stages:

- **Parse:** Every SQL statement must be parsed. Parsing the statement 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.
- **Execute:** At this point, the Oracle server has all necessary information and resources, and the statement is executed. For nonquery 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.

Dynamic SQL

Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:

- **Is constructed and stored as a character string within the application**
- **Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)**
- **Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL**
- **Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package**

Dynamic SQL

The embedded SQL statements available in PL/SQL are limited to `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `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`).

Native Dynamic SQL

- 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
- Is supported by the following PL/SQL statements:
  - `EXECUTE IMMEDIATE`
  - `OPEN-FOR`
  - `FETCH`
  - `CLOSE`

ORACLE

Native Dynamic SQL

In Oracle 8 and earlier releases, the only way to implement dynamic SQL in a PL/SQL application was by using the `DBMS_SQL` package. With Oracle 8*i* and later releases, the PL/SQL environment provides Native Dynamic SQL as an alternative.

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.
- Bind instances of any SQL data types supported in PL/SQL.
- Handle `IN`, `IN OUT`, and `OUT` bind variables that are bound by position, not by name.

Using the `EXECUTE IMMEDIATE` Statement

Use the `EXECUTE IMMEDIATE` statement for Native Dynamic SQL or PL/SQL anonymous blocks:

```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
     [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
     [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- **`INTO` is used for single-row queries and specifies the variables or records into which column values are retrieved.**
- **`USING` is used to hold all bind arguments. The default parameter mode is `IN`, if not specified.**

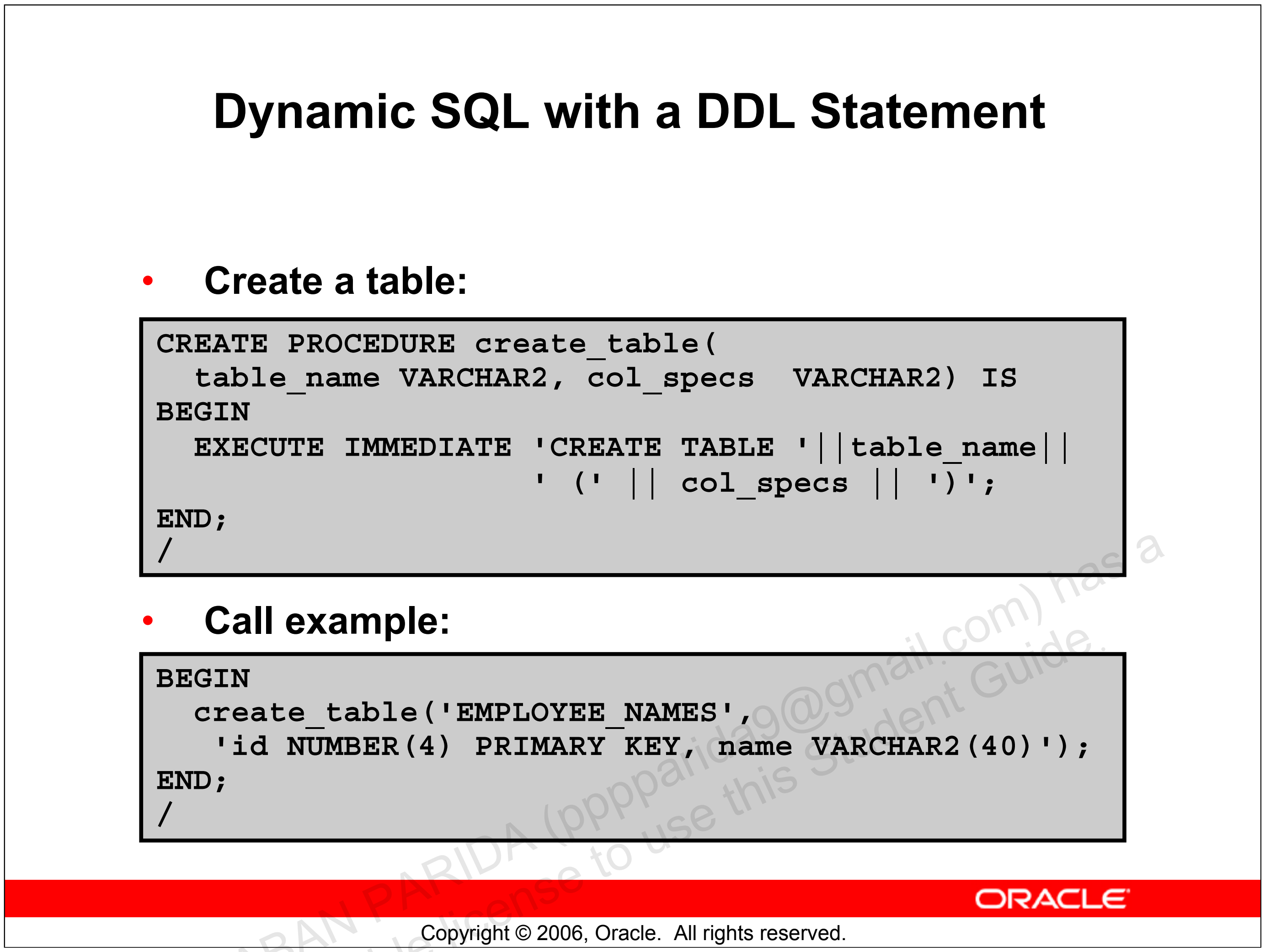
Using the `EXECUTE IMMEDIATE` Statement

The `EXECUTE IMMEDIATE` statement can be used to execute SQL statements or PL/SQL anonymous blocks. The syntactical elements include the following:

- `dynamic_string` is a string expression that represents a dynamic SQL statement (without terminator) or a PL/SQL block (with terminator).
- `define_variable` is a PL/SQL variable that stores the selected column value.
- `record` is a user-defined or `%ROWTYPE` record that stores a selected row.
- `bind_argument` is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.
- The `INTO` clause specifies the variables or record into which column values are retrieved. It is used only for single-row queries. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the `INTO` clause.
- The `USING` clause holds all bind arguments. The default parameter mode is `IN`.

You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (`TRUE`, `FALSE`, and `NULL`).

Note: Use `OPEN-FOR`, `FETCH`, and `CLOSE` for a multirow query. The syntax shown in the slide is not complete because support exists for bulk-processing operations (which is a topic that is not covered in this course).



Dynamic SQL with a DDL Statement

- **Create a table:**

```
CREATE PROCEDURE create_table(
  table_name VARCHAR2, col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '||table_name||
                    ' (' || col_specs || ')';
END;
/
```

- **Call example:**

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

Dynamic SQL with a DDL Statement

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

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

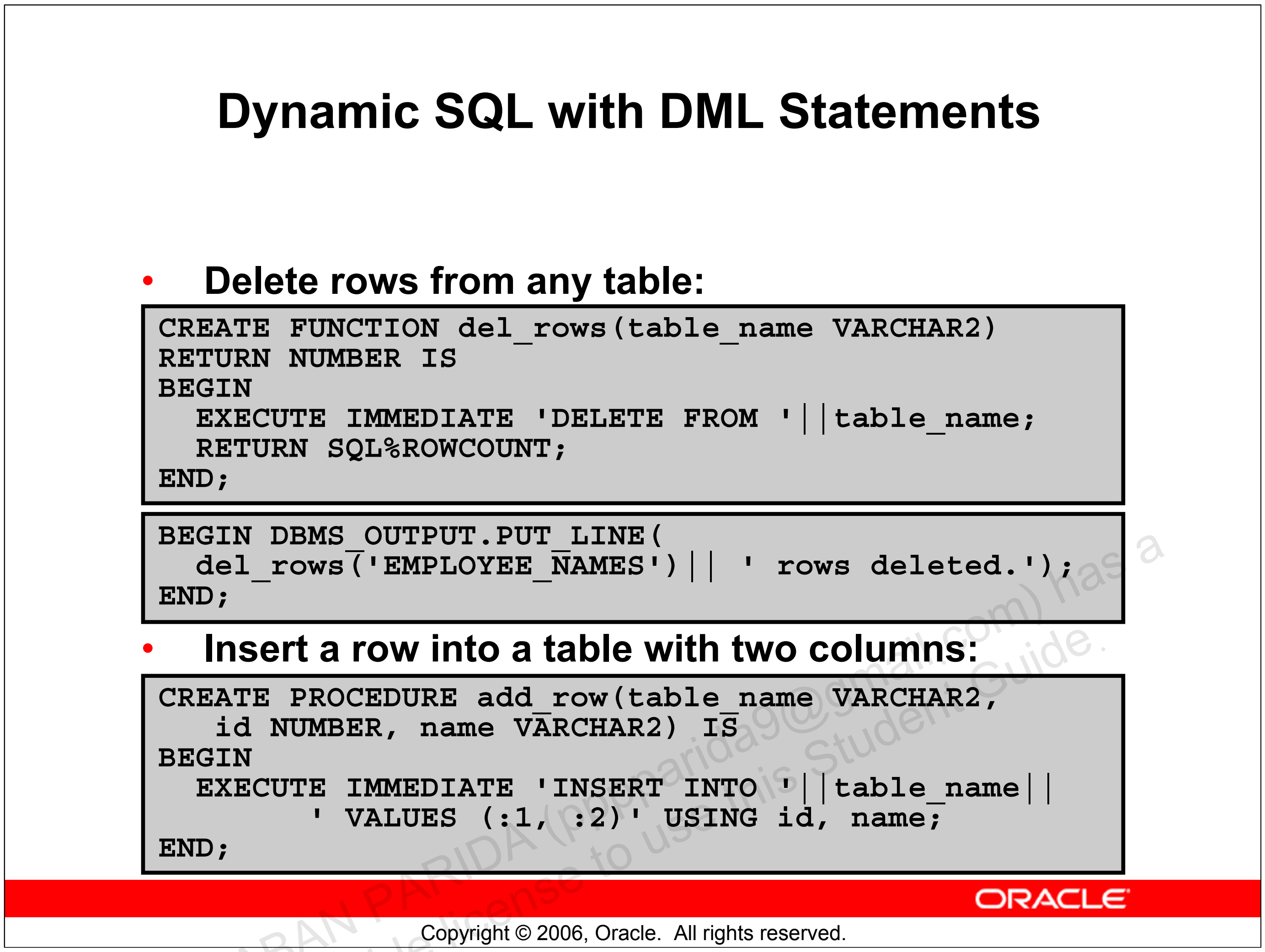
- An ID column with a `NUMBER` data type used as a primary key
- A name column of up to 40 characters for the employee name

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 PROCEDURE add_col(table_name VARCHAR2,
                         col_spec VARCHAR2) IS
  stmt VARCHAR2(100) := 'ALTER TABLE ' || table_name ||
                        ' ADD '||col_spec;
BEGIN
  EXECUTE IMMEDIATE 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(table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM '||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(table_name VARCHAR2,
   id NUMBER, name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '||table_name||
        ' VALUES (:1, :2)' USING id, name;
END;
```

Dynamic SQL with DML Statements

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, but the order of the variable names (`id` and `name`) in the `USING` clause is associated to the bind variables by position, in the order of their respective appearance. Therefore, the PL/SQL variable `id` is assigned to the `:1` placeholder, and the `name` variable is assigned to the `:2` placeholder. Placeholder/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 with a Single-Row Query

Example of a single-row query:

```
CREATE FUNCTION get_emp(emp_id NUMBER)
RETURN employees%ROWTYPE IS
  stmt VARCHAR2(200);
  emprec employees%ROWTYPE;
BEGIN
  stmt := 'SELECT * FROM employees ' ||
          'WHERE employee_id = :id';
  EXECUTE IMMEDIATE stmt INTO emprec USING emp_id;
  RETURN emprec;
END;
/
```

```
DECLARE
  emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '||emprec.last_name);
END;
/
```

Dynamic SQL with a Single-Row Query

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.

Dynamic SQL with a Multirow Query

Use `OPEN-FOR`, `FETCH`, and `CLOSE` processing:

```
CREATE PROCEDURE list_employees(deptid NUMBER) IS
  TYPE emp_refcsr IS REF CURSOR;
  emp_cv  emp_refcsr;
  emprec  employees%ROWTYPE;
  stmt varchar2(200) := 'SELECT * FROM employees';
BEGIN
  IF deptid IS NULL THEN OPEN emp_cv FOR stmt;
  ELSE
    stmt := stmt || ' WHERE department_id = :id';
    OPEN emp_cv FOR stmt USING deptid;
  END IF;
  LOOP
    FETCH emp_cv INTO emprec;
    EXIT WHEN emp_cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emprec.department_id||
                    ' ' ||emprec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
```

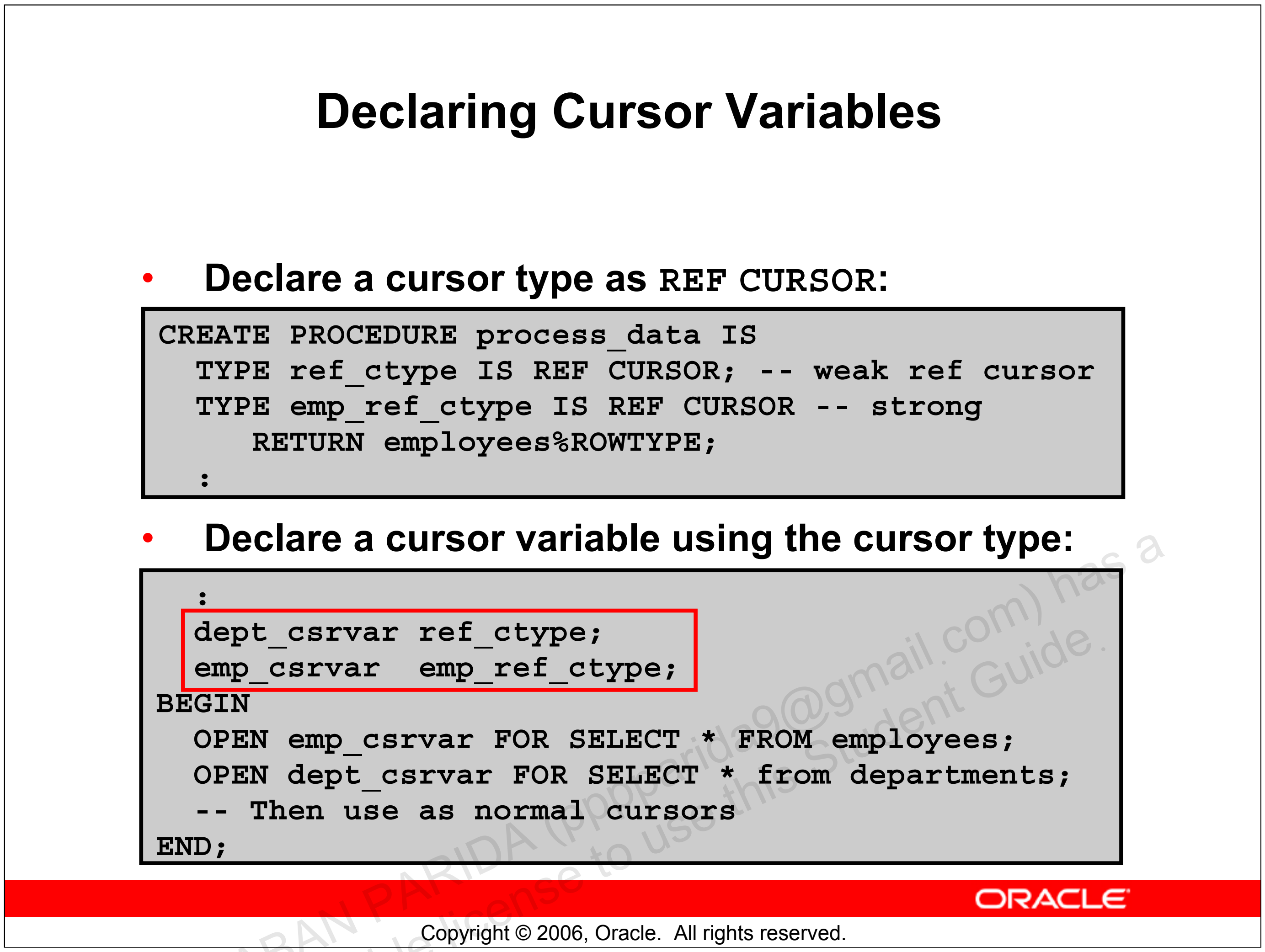
Dynamic SQL with a Multirow Query

The example in the slide shows how to execute a multirow query by performing the following programming steps:

- Declaring a `REF CURSOR` type
- Declaring a cursor variable based on the `REF CURSOR` type name that you declare
- Executing an `OPEN-FOR` statement that uses the cursor variable
- Using a `FETCH` statement referencing the cursor variable until all records are processed
- Executing the `CLOSE` statement by using the cursor variable

This process is the same as using static cursor definitions. However, the `OPEN-FOR` syntax accepts a string literal or variable specifying the `SELECT` statement, which can be dynamically constructed.

Note: The next page provides a brief introduction to the `REF CURSOR` type and cursor variables. An alternative to this is using the `BULK COLLECT` syntax supported by Native Dynamic SQL statements (a topic that is not covered in this course).



Declaring Cursor Variables

- Declare a cursor type as `REF CURSOR`:

```
CREATE PROCEDURE process_data IS
  TYPE ref_ctype IS REF CURSOR; -- weak ref cursor
  TYPE emp_ref_ctype IS REF CURSOR -- strong
     RETURN employees%ROWTYPE;
  :
```

- Declare a cursor variable using the cursor type:

```
  :
  dept_csrvar ref_ctype;
  emp_csrvar  emp_ref_ctype;
BEGIN
  OPEN emp_csrvar FOR SELECT * FROM employees;
  OPEN dept_csrvar FOR SELECT * from departments;
  -- Then use as normal cursors
END;
```

ORACLE

Declaring Cursor Variables

A cursor variable is a PL/SQL identifier whose type name has been declared as a `REF CURSOR` type. Creating a cursor variable involves two steps:

- Declaring a type name as a `REF CURSOR` type
- Declaring a PL/SQL variable by using the type name declared as a `REF CURSOR` type

The slide examples create two reference cursor types:

- The `ref_ctype` is a generic reference cursor, known as a weak reference cursor. A weak reference cursor can be associated with any query.
- The `emp_ref_ctype` is a strong reference cursor type that must be associated with a type-compatible query: the query must return data that is compatible with the type specified after the `RETURN` keyword (for example, an `EMPLOYEES` row type).

After a cursor variable is declared by using a reference cursor type name, the cursor variable that is associated with a query is opened by using the `OPEN-FOR` syntax shown in the slide. The standard `FETCH`, cursor attributes, and `CLOSE` operations used with explicit cursors are also applicable with cursor variables. To compare cursor variables with explicit cursors:

- A cursor variable can be associated with more than one query at run time
- An explicit cursor is associated with one query at compilation time

Dynamically Executing a PL/SQL Block

Execute a PL/SQL anonymous block dynamically:

```
CREATE FUNCTION annual_sal(emp_id NUMBER)
RETURN NUMBER IS
  plsql varchar2(200) :=
    'DECLARE '||
    ' emprec employees%ROWTYPE; '||
    'BEGIN '||
    ' emprec := get_emp(:empid); ' ||
    ' :res := emprec.salary * 12; ' ||
    'END;';
  result NUMBER;
BEGIN
 EXECUTE IMMEDIATE plsql
         USING IN emp_id, OUT result;
  RETURN result;
END;
/
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

Dynamically Executing a PL/SQL Block

The `annual_sal` function dynamically constructs an anonymous PL/SQL block. The PL/SQL block contains bind variables for:

- The input of the employee ID using the `:empid` placeholder
- The output result computing the annual employees' salary using the placeholder called `:res`

Note: This example demonstrates how to use the `OUT` result syntax (in the `USING` clause of the `EXECUTE IMMEDIATE` statement) to obtain the result calculated by the PL/SQL block. The procedure output variables and function return values can be obtained in a similar way from a dynamically executed anonymous PL/SQL block.

Using Native Dynamic SQL to Compile PL/SQL Code

Compile PL/SQL code with the `ALTER` statement:

- `ALTER PROCEDURE name COMPILE`
- `ALTER FUNCTION name COMPILE`
- `ALTER PACKAGE name COMPILE SPECIFICATION`
- `ALTER PACKAGE name COMPILE BODY`

```
CREATE PROCEDURE compile_plsql(name VARCHAR2,
 plsql_type VARCHAR2, options VARCHAR2 := NULL) IS
  stmt varchar2(200) := 'ALTER '|| plsql_type ||
                        ' '|| name || ' COMPILE';
BEGIN
 IF options IS NOT NULL THEN
   stmt := stmt || ' ' || options;
 END IF;
 EXECUTE IMMEDIATE stmt;
END;
/
```

ORACLE

Using Native Dynamic SQL to Compile PL/SQL Code

The `compile_plsql` procedure in the example can be used to compile different PL/SQL code using the `ALTER` DDL statement. Four basic forms of the `ALTER` statement are shown to compile:

- A procedure
- A function
- A package specification
- A package body

Note: If you leave out the keyword `SPECIFICATION` or `BODY` with the `ALTER PACKAGE` statement, then the specification and body are both compiled.

Here are examples of calling the procedure in the slide for each of the four cases, respectively:

```
EXEC compile_plsql ('list_employees', 'procedure')
EXEC compile_plsql ('get_emp', 'function')
EXEC compile_plsql ('mypack', 'package',
'specification')
EXEC compile_plsql ('mypack', 'package', 'body')
```

Here is an example of compiling with `debug` enabled for the `get_emp` function:

```
EXEC compile_plsql ('get_emp', 'function', 'debug')
```

Using the DBMS_SQL Package

The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:

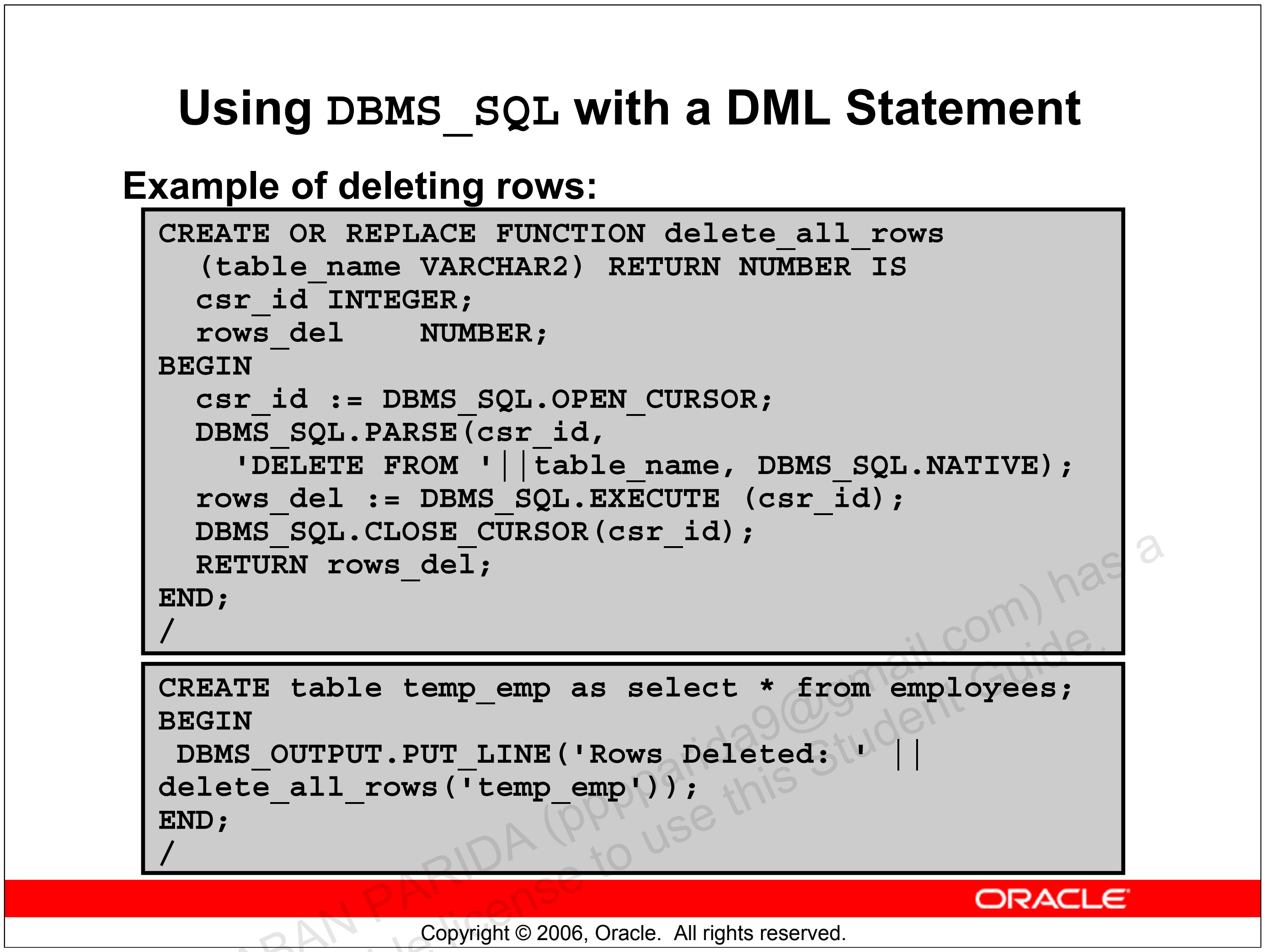
- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

Using the DBMS_SQL Package

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. 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—that is, it checks the statement syntax and associates it with the opened cursor. 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.



Using DBMS_SQL with a DML Statement

Example of deleting rows:

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

```
CREATE table temp_emp as select * from employees;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

ORACLE

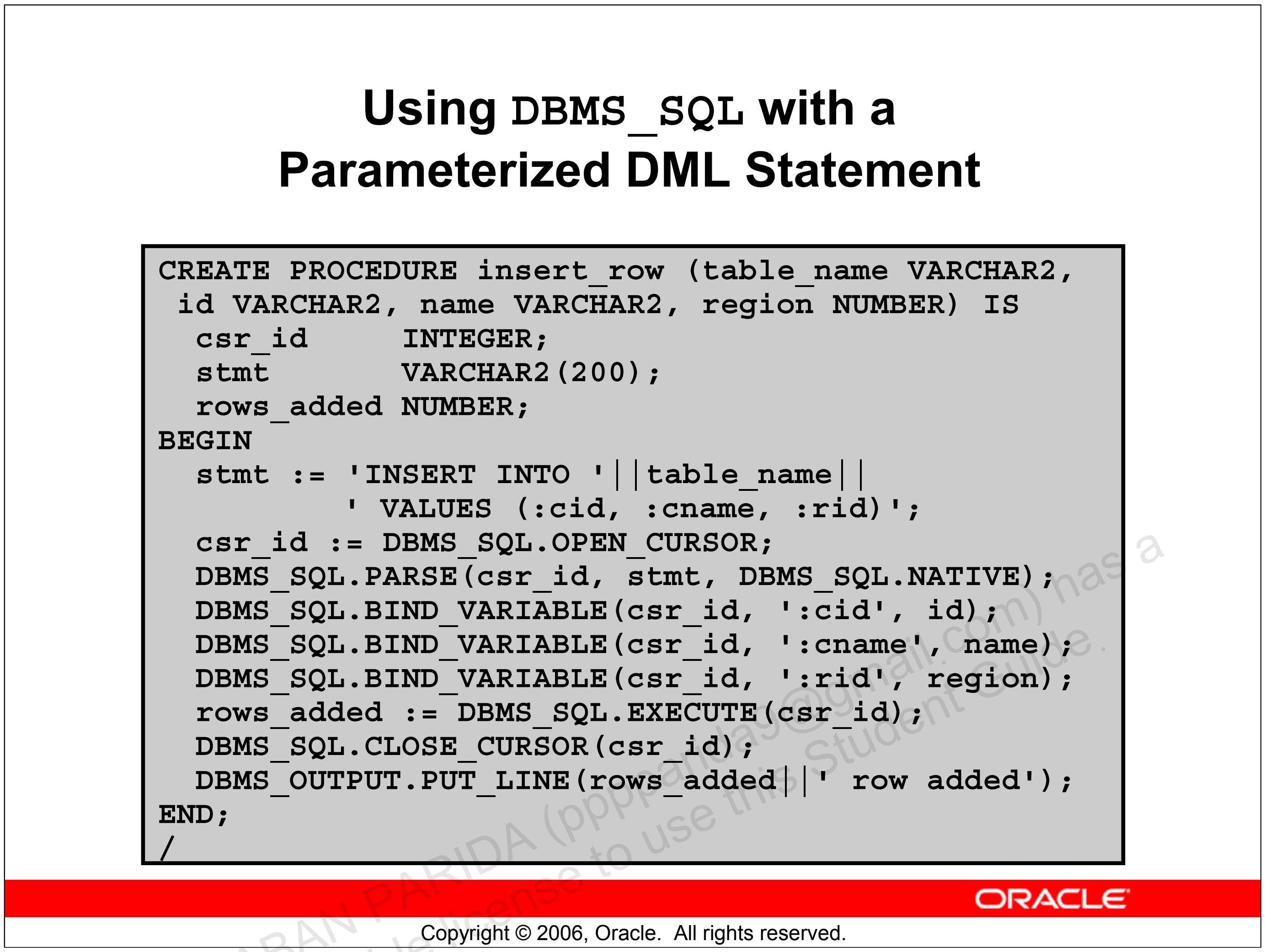
Using DBMS_SQL with a DML Statement

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.



Using DBMS_SQL with a Parameterized DML Statement

```
CREATE PROCEDURE insert_row (table_name VARCHAR2,
 id VARCHAR2, name VARCHAR2, region NUMBER) IS
  csr_id     INTEGER;
  stmt       VARCHAR2(200);
  rows_added NUMBER;
BEGIN
  stmt := 'INSERT INTO '||table_name||
          ' VALUES (:cid, :cname, :rid)';
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cid', id);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cname', name);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':rid', region);
  rows_added := DBMS_SQL.EXECUTE(csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  DBMS_OUTPUT.PUT_LINE(rows_added||' row added');
END;
/
```

Using DBMS_SQL with a Parameterized DML Statement

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', 'ZA', 'South Africa', 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.

Comparison of Native Dynamic SQL and the DBMS_SQL Package

Native Dynamic SQL:

- Is easier to use than DBMS_SQL
- Requires less code than DBMS_SQL
- Enhances performance because the PL/SQL interpreter provides native support for it
- Supports all types supported by static SQL in PL/SQL, including user-defined types
- Can fetch rows directly into PL/SQL records

Comparison of Native Dynamic SQL and the DBMS_SQL Package

Native Dynamic SQL provides the following advantages over the DBMS_SQL package.

Ease of use: Because Native Dynamic SQL is integrated with SQL, you can use it in the same way that you currently use static SQL within PL/SQL code. The code is typically more compact and readable compared with the code written with the DBMS_SQL package.

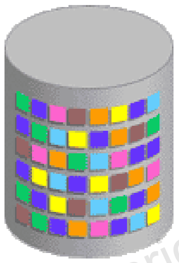
Performance improvement: Native Dynamic SQL performs significantly better than DBMS_SQL, in most circumstances, due to native support provided by the PL/SQL interpreter. The DBMS_SQL approach uses a procedural API and suffers from high procedure call and data copy overhead.

Support for user-defined types: Native Dynamic SQL supports all the types supported by static SQL in PL/SQL. Therefore, Native Dynamic SQL provides support for user-defined types such as user-defined objects, collections, and REFs. The DBMS_SQL package does not support these user-defined types. However, it has limited support for arrays.

Support for fetching into records: With Native Dynamic SQL, the rows resulting from a query can be directly fetched into PL/SQL records. The DBMS_SQL package does not support fetching into records structures.

DBMS_METADATA Package

The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



DBMS_METADATA Package

You can invoke DBMS_METADATA to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

You can use DBMS_METADATA for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column data types, and so on), and then converting the metadata to data definition language (DDL) so that the object can be re-created on the same or another database. In the past, you needed to do this programmatically with problems resulting in each new release.

The DBMS_METADATA functionality is used for the Oracle 10*g* Export/Import replacement, commonly called “the Data Pump.”

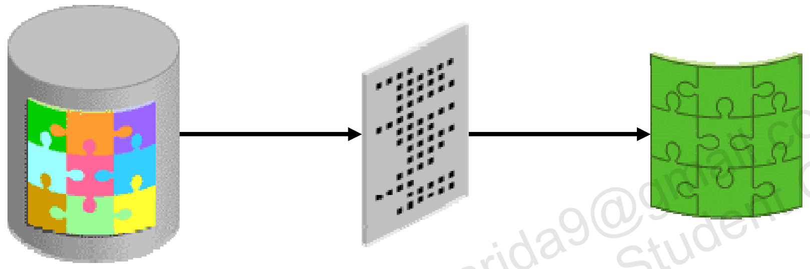
This package was introduced in Oracle9*i* and is further enhanced in Oracle Database 10*g*.

Note: For more information about the DBMS_DATAPUMP package, refer to the Online Course titled *Oracle Database 10g: Reduce Management - Tools and Utilities*.

Metadata API

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.



ORACLE

Metadata API

Every entity in the database is modeled as an object that belongs to an object type. For example, the `EMPLOYEES` table is an object; its object type is `TABLE`. When you fetch an object's metadata, you must specify the object type.

Every object type is implemented by using three entities:

- A user-defined type (UDT) whose attributes comprise all the metadata for objects of the type. An object's XML representation is a translation of a type instance into XML, with the XML tag names derived from the type attribute names. (In the case of tables, several UDTs are needed to represent the different varieties of the object type.)
- An object view of the UDT that populates instances of the object type
- An Extensible Style Sheet Language (XSL) script that converts the XML representation of an object into SQL DDL

Subprograms in DBMS_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved such as the object name or schema
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single FETCH_xxx call
GET_QUERY	Returns the text of the queries that will be used by FETCH_xxx
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned
ADD_TRANSFORM	Specifies a transform that FETCH_xxx applies to the XML representation of the retrieved objects
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by transform_handle
FETCH_*XXX*	Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER
CLOSE	Invalidates the handle returned by OPEN and cleans up the associated state

Subprograms in DBMS_METADATA

The table provides an overview of the procedures and functions available in the DBMS_METADATA package. To retrieve metadata, you can specify:

- The kind of object retrieved, either an object type (a table, index, procedure) or a heterogeneous collection of object types forming a logical unit (such as database export and schema export)
- Selection criteria (owner, name, and so on)
- "parse items" attributes of objects to be parsed and returned separately
- Transformations on the output, implemented by XSLT scripts

The package provides two types of retrieval interface for two types of usage:

- **For programmatic use:** OPEN, SET_FILTER, SET_COUNT, GET_QUERY, SET_PARSE_ITEM, ADD_TRANSFORM, SET_TRANSFORM_PARAM, SET_REMAP_PARAM (new in Oracle Database 10*g*), FETCH_xxx, and CLOSE. These enable flexible selection criteria and the extraction of a stream of objects.
- **For use in SQL queries and for ad hoc browsing:** The GET_*xxx* interfaces (GET_XML and GET_DDL) return metadata for a single named object. The GET_DEPENDENT_xxx and GET_GRANTED_xxx interfaces return metadata for one or more dependent or granted objects. None of these APIs support heterogeneous object types.

FETCH_xxx Subprograms

Name	Description
`FETCH_XML`	This function returns the XML metadata for an object as an XMLType.
`FETCH_DDL`	This function returns the DDL (either to create or to drop the object) into a predefined nested table.
`FETCH_CLOB`	This function returns the objects (transformed or not) as a CLOB.
`FETCH_XML_CLOB`	This procedure returns the XML metadata for the objects as a CLOB in an `IN OUT NOCOPY` parameter to avoid expensive LOB copies.

FETCH_xxx Subprograms

These functions and procedures return metadata for objects meeting the criteria established by the call to the OPEN function that returned the handle, and the subsequent calls to SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on. Each call to FETCH_*xxx* returns the number of objects specified by SET_COUNT (or a smaller number, if fewer objects remain in the current cursor) until all objects have been returned.

SET_FILTER Procedure

- **Syntax:**

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- **Example:**

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME',
'HR');
...
```

SET_FILTER Procedure

You use the SET_FILTER procedure to identify restrictions on objects that are to be retrieved. For example, you can specify restrictions on an object or schema that is being retrieved. This procedure is overloaded with the parameters having the following meanings:

- handle is the handle returned from the OPEN function.
- name is the name of the filter. For each filter, the object type applies to its name, data type (text or Boolean), and meaning or effect (including its default value, if there is one).
- value is the value of the filter. It can be text, Boolean, or a numeric value.
- object_type_path is a path name designating the object types to which the filter applies. By default, the filter applies to the object type of the OPEN handle.

If you use an expression filter, then it is placed to the right of a SQL comparison, and the value is compared with it. The value must contain parentheses and quotation marks where appropriate. A filter value is combined with a particular object attribute to produce a WHERE condition in the query that fetches the objects.

Filters

There are over 70 filters, which are organized into object type categories such as:

- **Named objects**
- **Tables**
- **Objects dependent on tables**
- **Index**
- **Dependent objects**
- **Granted objects**
- **Table data**
- **Index statistics**
- **Constraints**
- **All object types**
- **Database export**

ORACLE

Filters

There are over 70 filters that you can specify when using the SET_FILTER procedure. These filters are organized into object type categories. Some of the new object type categories in Oracle Database 10*g* are listed in the slide.

When using the SET_FILTER procedure, you specify the name of the filter and its respective value.

For example, you can use the SCHEMA filter with a value to identify the schema whose objects are selected. Then use a second call to the SET_FILTER procedure and use a filter named INCLUDE_USER that has a Boolean data type for its value. If it is set to TRUE, then objects containing privileged information about the user are retrieved.

```
DBMS_METADATA.SET_FILTER(handle, SCHEMA, 'HR');
DBMS_METADATA. SET_FILTER(handle, INCLUDE_USER, TRUE);
```

Each call to SET_FILTER causes a WHERE condition to be added to the underlying query that fetches the set of objects. The WHERE conditions are combined using an AND operator, so you can use multiple SET_FILTER calls to refine the set of objects to be returned.

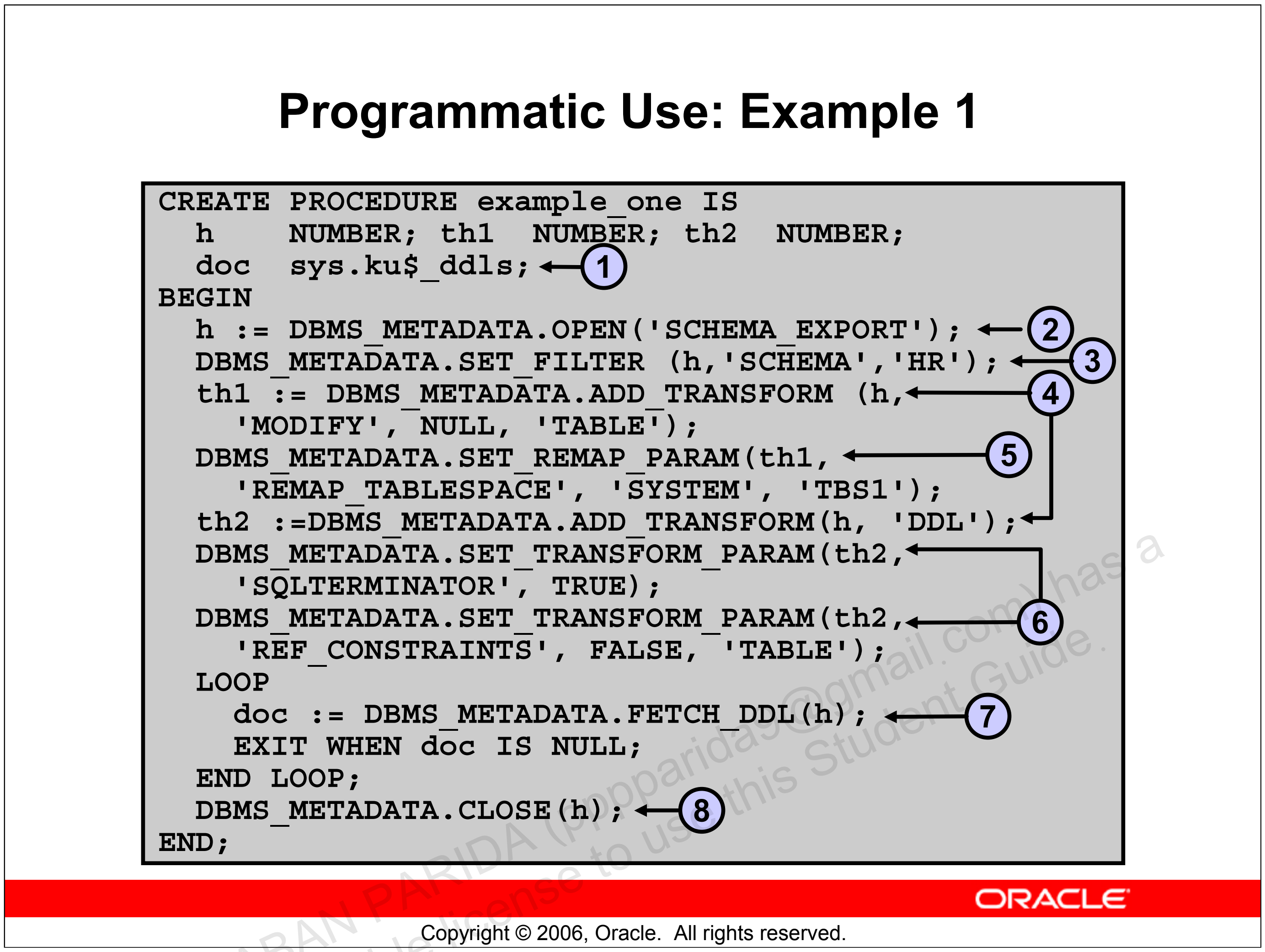
Examples of Setting Filters

Set up the filter to fetch the `HR` schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain `PAYROLL` in the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
  'IN (''PAYROLL'', ''HR'')');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''FUNCTION''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PROCEDURE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PACKAGE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
  'LIKE ''PAYROLL%''', 'VIEW');
```

Examples of Setting Filters

The example shown in the slide calls the SET_FILTER procedure several times to create a WHERE condition that identifies which object types are to be fetched. First, the objects in the PAYROLL and HR schemas are identified as object types to be fetched. Subsequently, the SET_FILTER procedure identifies certain object types (functions, procedures, and packages) and view object names to be excluded.



Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
  h    NUMBER; th1  NUMBER; th2  NUMBER;
  doc  sys.ku$_ddls;
BEGIN
  h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');
  DBMS_METADATA.SET_FILTER (h,'SCHEMA','HR');
  th1 := DBMS_METADATA.ADD_TRANSFORM (h,
    'MODIFY', NULL, 'TABLE');
  DBMS_METADATA.SET_REMAP_PARAM(th1,
    'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
  th2 :=DBMS_METADATA.ADD_TRANSFORM(h, 'DDL');
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,
    'SQLTERMINATOR', TRUE);
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,
    'REF_CONSTRAINTS', FALSE, 'TABLE');
  LOOP
    doc := DBMS_METADATA.FETCH_DDL(h);
    EXIT WHEN doc IS NULL;
  END LOOP;
  DBMS_METADATA.CLOSE(h);
END;
```

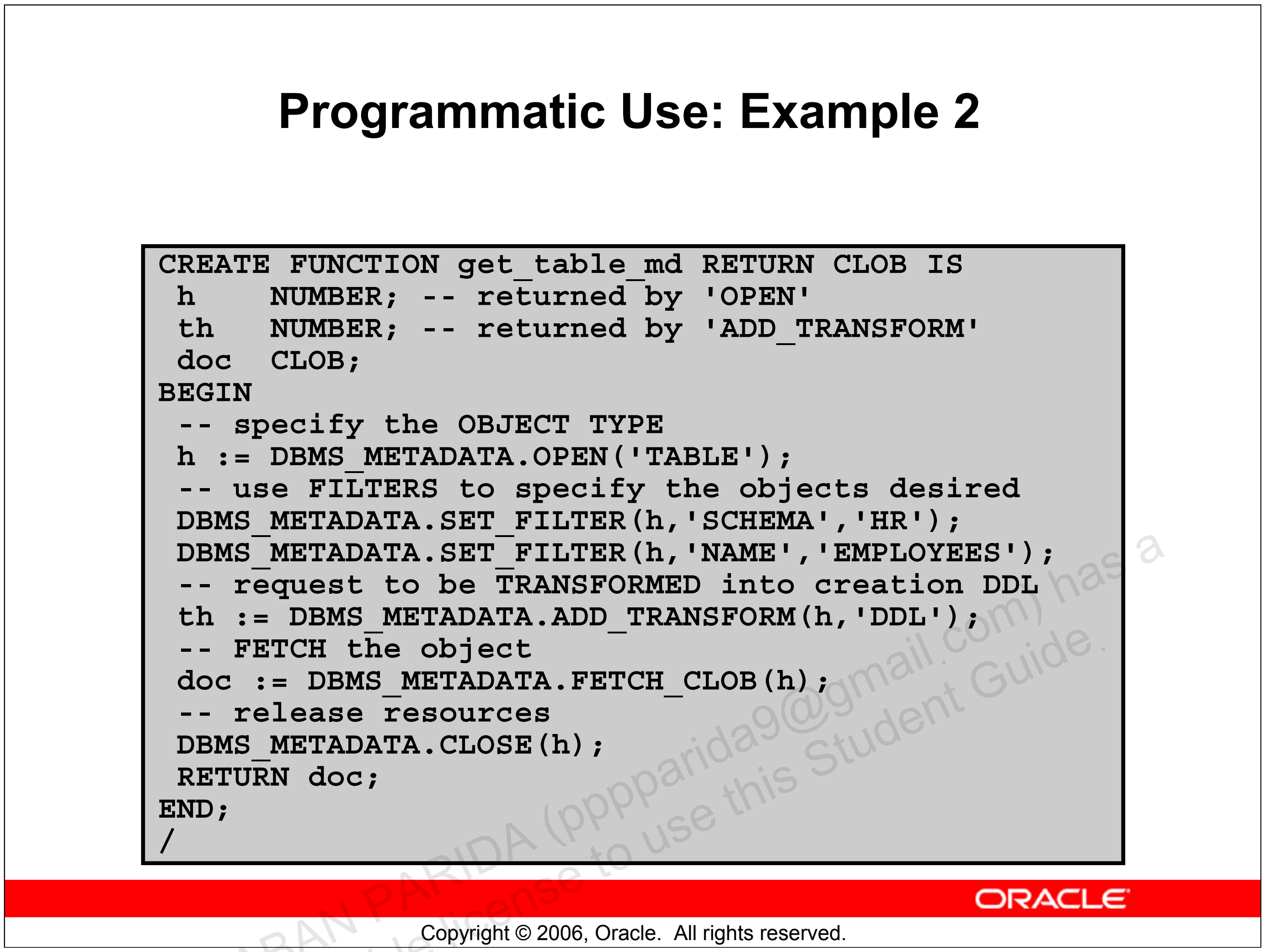
Programmatic Use: Example 1

In this example, all objects are retrieved from the HR schema as creation DDL. The MODIFY transform is used to change the tablespaces for the tables.

1. The DBMS_METADATA package has several predefined types that are owned by SYS. The sys.ku_$ddls type is defined in the DBMS_METADATA package. It is a table type that holds the CLOB type of data.
2. You use the OPEN function to specify the type of object to be retrieved, the version of its metadata, and the object model. It returns a context handle for the set of objects.
   In this example, 'SCHEMA_EXPORT' is the object type, and it indicates all metadata objects in a schema. There are 85 predefined types of objects for the model that you can specify for this parameter. Both the version of metadata and the object model parameters are not identified in this example. The version of metadata parameter defaults to 'COMPATIBLE'. You can also specify 'LATEST' or a specific database version. Currently, the model parameter supports only the Oracle model in Oracle Database 10*g*. This is the default.
3. The SET_FILTER procedure identifies restrictions on the objects to be retrieved.

Programmatic Use: Example 1 (continued)

4. The ADD_TRANSFORM function specifies a transform that FETCH_XXX applies to the XML representation of the retrieved objects. You can have more than one transform. In the example, two transforms occur, one for each of the th1 and th2 program variables. The ADD_TRANSFORM function accepts four parameters and returns a number representing the opaque handle to the transform. The parameters are the handle returned from the OPEN statement, the name of the transform (DDL, DROP, or MODIFY), the encoding name (which is the name of the NLS [national language support] character set in which the style sheet pointed to by the name is encoded), and the object type. If the object type is omitted, the transform applies to all objects; otherwise, it applies only to the object type specified.
   The first transform shown in the program code is the handle returned from the OPEN function. The second transform shown in the code has two parameter values specified. The first parameter is the handle identified from the OPEN function. The second parameter value is DDL, which means the document is transformed to DDL that creates the object. The output of this transform is not an XML document. The third and fourth parameters are not specified. Both take the default values for the encoding and object type parameters.
5. The SET_REMAP_PARAM procedure identifies the parameters to the XSLT style sheet identified by the transform handle, which is the first parameter passed to the procedure. In the example, the second parameter value 'REMAP_TABLESPACE' means that the objects have their tablespaces renamed from an old value to a new value. In the ADD_TRANSFORM function, the choices are DDL, DROP, or MODIFY. For each of these values, the SET_REMAP_PARAM identifies the name of the parameter. REMAP_TABLESPACE means the objects in the document will have their tablespaces renamed from an old value to a new value. The third and fourth parameters identify the old value and new value. In this example, the old tablespace name is SYSTEM, and the new tablespace name is TBS1.
6. SET_TRANSFORM_PARAM works similarly to SET_REMAP_PARAM. In the code shown, the first call to SET_TRANSFORM_PARAM identifies parameters for the th2 variable. The SQLTERMINATOR and TRUE parameter values cause the SQL terminator (; or /) to be appended to each DDL statement.
   The second call to SET_TRANSFORM_PARAM identifies more characteristics for the th2 variable. REF_CONSTRAINTS, FALSE, TABLE means that referential constraints on the tables are not copied to the document.
7. The FETCH_DDL function returns metadata for objects meeting the criteria established by the OPEN, SET_FILTER, ADD_TRANSFORM, SET_REMAP_PARAM, and SET_TRANSFORM_PARAM subroutines.
8. The CLOSE function invalidates the handle returned by the OPEN function and cleans up the associated state. Use this function to terminate the stream of objects established by the OPEN function.



Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
 h    NUMBER; -- returned by 'OPEN'
 th   NUMBER; -- returned by 'ADD_TRANSFORM'
 doc  CLOB;
BEGIN
 -- specify the OBJECT TYPE
 h := DBMS_METADATA.OPEN('TABLE');
 -- use FILTERS to specify the objects desired
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
 DBMS_METADATA.SET_FILTER(h,'NAME','EMPLOYEES');
 -- request to be TRANSFORMED into creation DDL
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');
 -- FETCH the object
 doc := DBMS_METADATA.FETCH_CLOB(h);
 -- release resources
 DBMS_METADATA.CLOSE(h);
 RETURN doc;
END;
/
```

ORACLE

Programmatic Use: Example 2

This example returns the metadata for the EMPLOYEES table. The result is:

```
          set pagesize 0
          set long  1000000
          SELECT get_table_md FROM dual;

          CREATE TABLE "HR"."EMPLOYEES"
             (  "EMPLOYEE_ID" NUMBER(6,0),
                "FIRST_NAME" VARCHAR2(20),
                "LAST_NAME" VARCHAR2(25) CONSTRAINT
        "EMP_LAST_NAME_NN"
                   NOT NULL ENABLE,
                "e-mail" VARCHAR2(25) CONSTRAINT "EMP_e-mail_NN"
                   NOT NULL ENABLE,
                "PHONE_NUMBER" VARCHAR2(20),
                "HIRE_DATE" DATE CONSTRAINT "EMP_HIRE_DATE_NN"
                   NOT NULL ENABLE,
                "JOB_ID" VARCHAR2(10) CONSTRAINT "EMP_JOB_NN"
                   NOT NULL ENABLE,
                "SALARY" NUMBER(8,2),
        ...
```

Programmatic Use: Example 2 (continued)

```
            "COMMISSION_PCT" NUMBER(2,2),
                     "MANAGER_ID" NUMBER(6,0),
                     "DEPARTMENT_ID" NUMBER(4,0),
                      CONSTRAINT "EMP_SALARY_MIN" CHECK (salary > 0)
             ENABLE,
                      CONSTRAINT "EMP_e-mail_UK" UNIQUE ("e-mail")
                 USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
                 STORAGE(INITIAL 65536 NEXT 65536 MINEXTENTS 1
                   MAXEXTENTS 2147483645
                 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                    BUFFER_POOL DEFAULT)
                 TABLESPACE "EXAMPLE"  ENABLE,
                      CONSTRAINT "EMP_EMP_ID_PK" PRIMARY KEY
             ("EMPLOYEE_ID")
                 USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
                 STORAGE(INITIAL 65536 NEXT 65536 MINEXTENTS 1
                   MAXEXTENTS 2147483645
                 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                   BUFFER_POOL DEFAULT)
                 TABLESPACE "EXAMPLE"  ENABLE,
                      CONSTRAINT "EMP_DEPT_FK" FOREIGN KEY
             ("DEPARTMENT_ID")
                       REFERENCES "HR"."DEPARTMENTS"
             ("DEPARTMENT_ID") ENABLE,
                      CONSTRAINT "EMP_JOB_FK" FOREIGN KEY ("JOB_ID")
                       REFERENCES "HR"."JOBS" ("JOB_ID") ENABLE,
                      CONSTRAINT "EMP_MANAGER_FK" FOREIGN KEY
             ("MANAGER_ID")
                       REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID")
             ENABLE
                  ) PCTFREE 0 PCTUSED 40 INITRANS 1 MAXTRANS 255 C
                   OMPRESS LOGGING
                 STORAGE(INITIAL 65536 NEXT 65536 MINEXTENTS 1
                    MAXEXTENTS 2147483645
                 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                   BUFFER_POOL DEFAULT)
                 TABLESPACE "EXAMPLE"
```

You can accomplish the same effect with the browsing interface:

```
SELECT dbms_metadata.get_ddl
       ('TABLE','EMPLOYEES','HR')
   FROM dual;
```

Browsing APIs

Name	Description
`GET_XXX`	The `GET_XML` and `GET_DDL` functions return metadata for a single named object.
`GET_DEPENDENT_XXX`	This function returns metadata for a dependent object.
`GET_GRANTED_XXX`	This function returns metadata for a granted object.

Where *xxx* is:	`DDL` or `XML`

Browsing APIs

The browsing APIs are designed for use in SQL queries and ad hoc browsing. These functions allow you to fetch metadata for objects with a single call. They encapsulate calls to OPEN, SET_FILTER, and so on. Which function you use depends on the characteristics of the object type and whether you want XML or DDL.

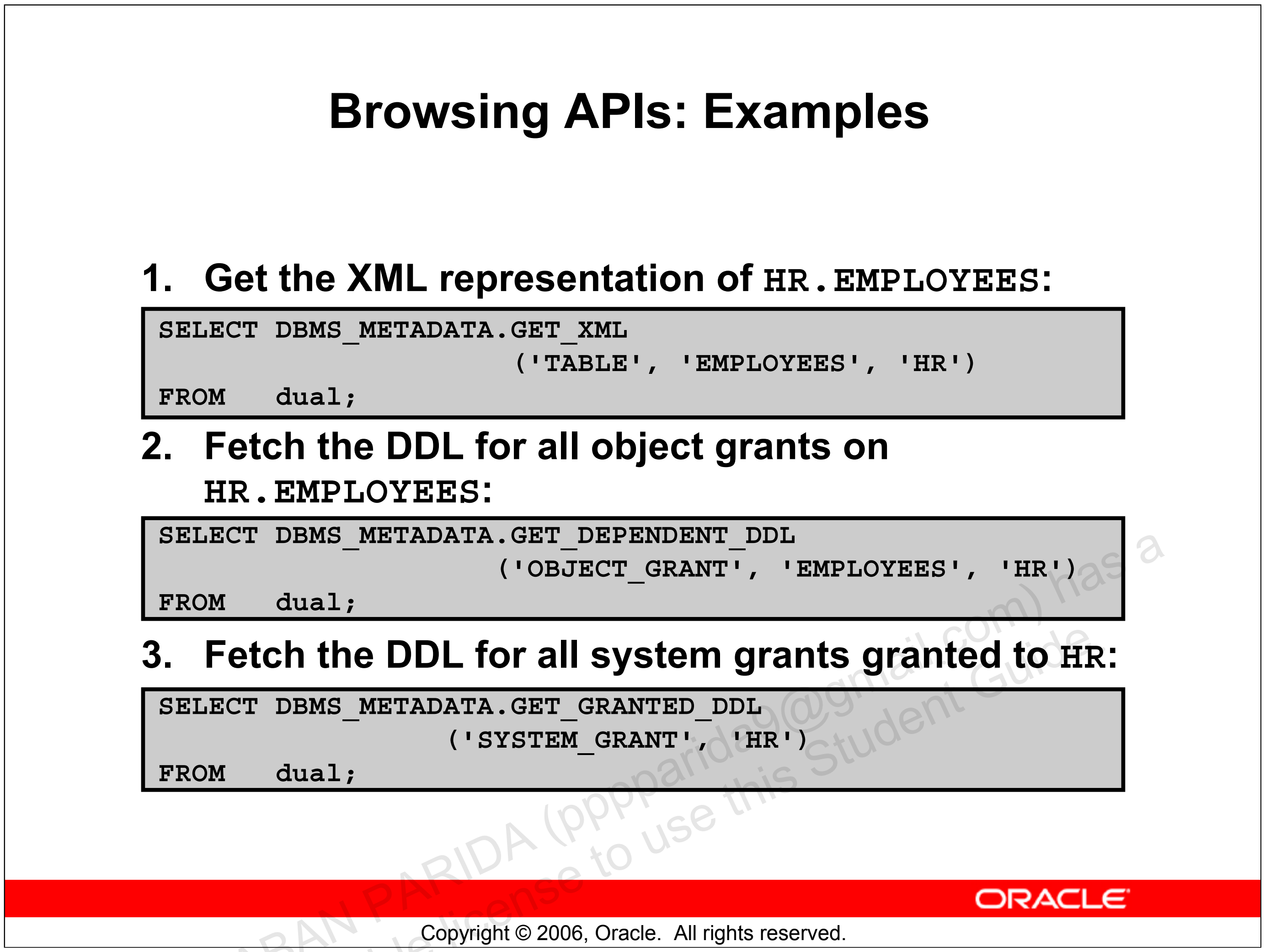
For some object types, you can use more than one function. You can use GET_XXX to fetch an index by name, or GET_DEPENDENT_XXX to fetch the same index by specifying the table on which it is defined.

GET_XXX returns a single object name.

For GET_DEPENDENT_XXX and GET_GRANTED_XXX, an arbitrary number of granted or dependent objects may match the input criteria. You can specify an object count when fetching these objects.

If you invoke these functions from *i*SQL*Plus, then you should use the SET LONG and SET PAGESIZE commands to retrieve complete, uninterrupted output.

```
SET LONG 2000000
SET PAGESIZE 300
```



Browsing APIs: Examples

1. Get the XML representation of `HR.EMPLOYEES`:

```
SELECT DBMS_METADATA.GET_XML
                      ('TABLE', 'EMPLOYEES', 'HR')
FROM   dual;
```

2. Fetch the DDL for all object grants on `HR.EMPLOYEES`:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
                     ('OBJECT_GRANT', 'EMPLOYEES', 'HR')
FROM   dual;
```

3. Fetch the DDL for all system grants granted to `HR`:

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
                  ('SYSTEM_GRANT', 'HR')
FROM   dual;
```

Browsing APIs: Examples

1. Results for fetching the XML representation of `HR.EMPLOYEES`:

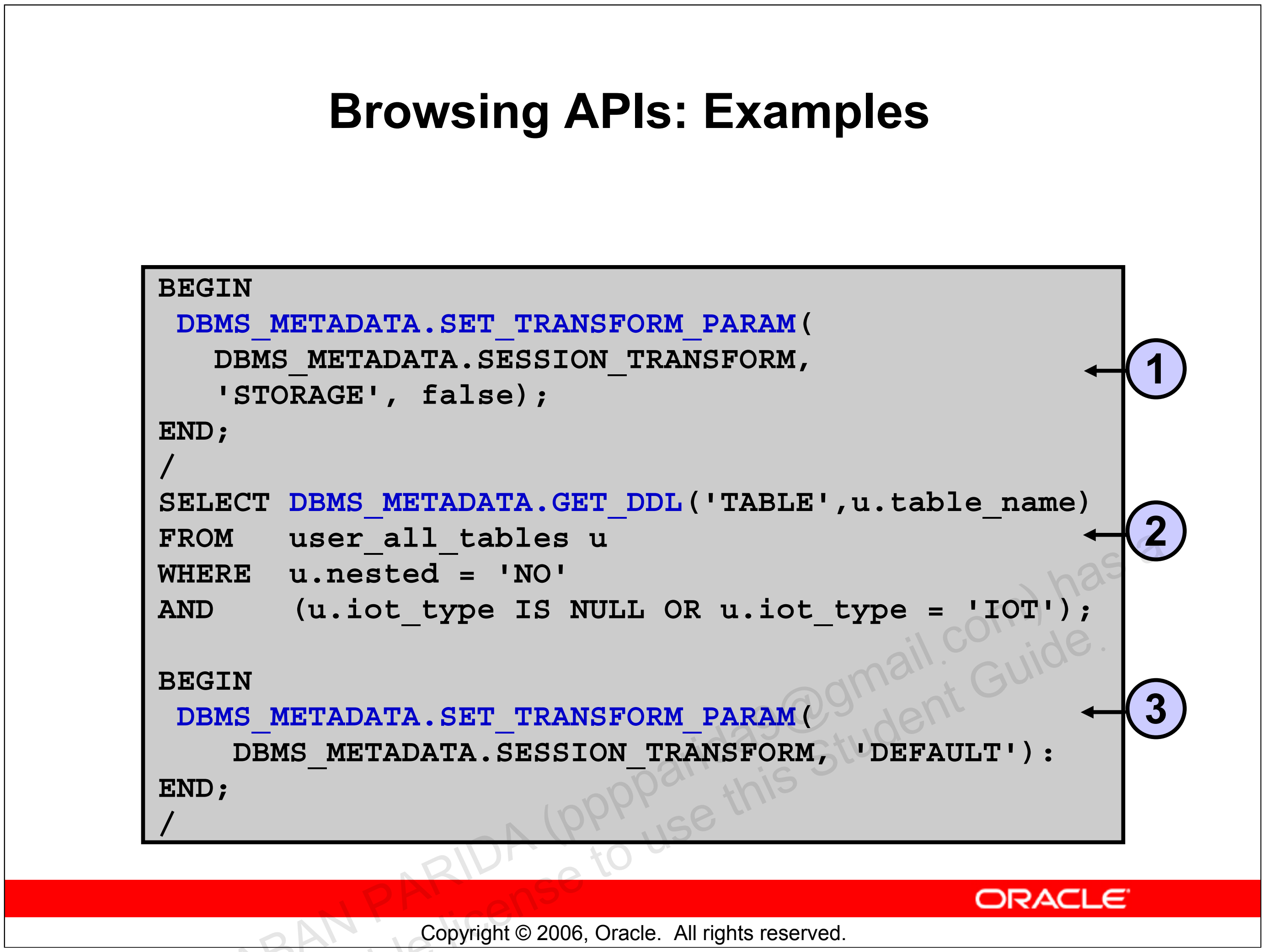
```
DBMS_METADATA.GET_XML('TABLE','EMPLOYEES','HR')
-----------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <TABLE_T>
   <VERS_MAJOR>1</VERS_MAJOR>
```

2. Results for fetching the DDL for all object grants on `HR.EMPLOYEES`:

```
DBMS_METADATA.GET_DEPENDENT_DDL
('OBJECT_GRANT','EMPLOYEES','HR')
------------------------------------------------
GRANT SELECT ON "HR"."EMPLOYEES" TO "OE"
GRANT REFERENCES ON "HR"."EMPLOY
```

3. Results for fetching the DDL for all system grants granted to `HR`:

```
DBMS_METADATA.GET_GRANTED_DDL('SYSTEM_GRANT','HR')
-------------------------------------------------------
-
GRANT UNLIMITED TABLESPACE TO "HR"
```



Browsing APIs: Examples

```
BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
   DBMS_METADATA.SESSION_TRANSFORM,
   'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
FROM   user_all_tables u
WHERE  u.nested = 'NO'
AND    (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT'):
END;
/
```

Browsing APIs: Examples (continued)

The example in the slide shows how to fetch creation DDL for all "complete" tables in the current schema, filtering out nested tables and overflow segments. The steps shown in the slide are as follows:

1. The `SET_TRANSFORM_PARAM` function specifies that the storage clauses are not to be returned in the SQL DDL. The `SESSION_TRANSFORM` function is interpreted to mean "for the current session."
2. Use the `GET_DDL` function to retrieve DDL on all non-nested and non-IOT (index-organized table) tables.

```
CREATE TABLE "HR"."COUNTRIES"
 ( "COUNTRY_ID" CHAR(2)
     CONSTRAINT "COUNTRY_ID_NN" NOT NULL  ENABLE,
   "COUNTRY_NAME" VARCHAR2(40),
   "REGION_ID" NUMBER,
      CONSTRAINT "COUNTRY_C_ID_PK"
      PRIMARY KEY ("COUNTRY_ID") ENABLE,
      CONSTRAINT "COUNTR_REG_FK" FOREIGN KEY
...
```

3. Reset the session-level parameters to their defaults.

Summary

In this lesson, you should have learned how to:

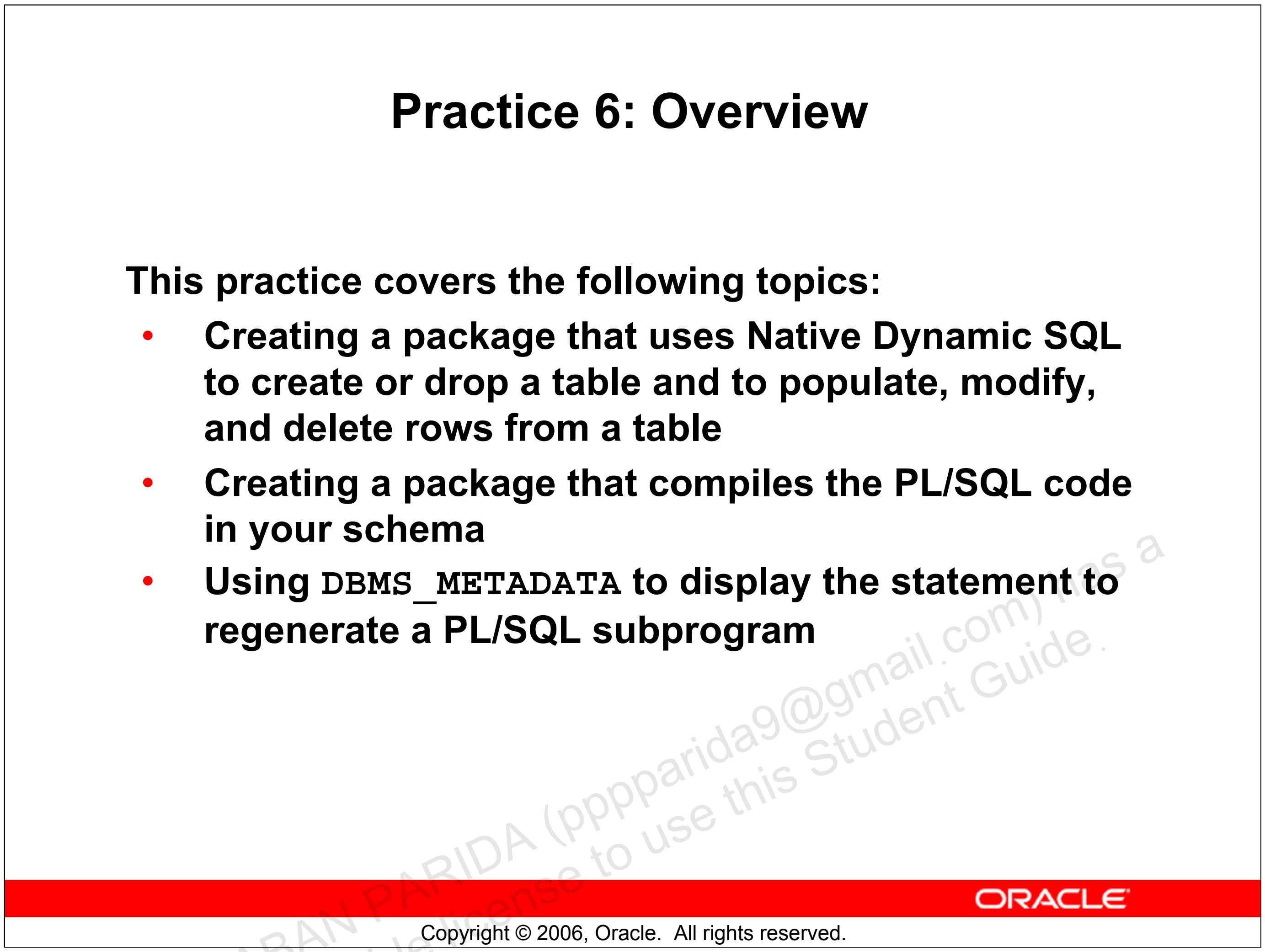
- **Explain the execution flow of SQL statements**
- **Create SQL statements dynamically and execute them using either Native Dynamic SQL statements or the `DBMS_SQL` package**
- **Recognize the advantages of using Native Dynamic SQL compared to the `DBMS_SQL` package**
- **Use `DBMS_METADATA` subprograms to programmatically obtain metadata from the data dictionary**

Summary

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.

The lesson explored some differences and compared using Native Dynamic SQL to the `DBMS_SQL` package. If you are using Oracle8*i* or later releases, you should use Native Dynamic SQL for new projects.

The lesson also discussed using the `DBMS_METADATA` package to retrieve metadata from the database dictionary with results presented in XML or creational DDL format. The resulting XML data can be used for re-creating the object.



Practice 6: Overview

This practice covers the following topics:

- **Creating a package that uses Native Dynamic SQL to create or drop a table and to populate, modify, and delete rows from a table**
- **Creating a package that compiles the PL/SQL code in your schema**
- **Using `DBMS_METADATA` to display the statement to regenerate a PL/SQL subprogram**

Practice 6: Overview

In this practice, you write code to perform the following tasks:

- Create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.
- Create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an `INVALID` status in the `USER_OBJECTS` table.
- Use `DBMS_METADATA` to regenerate PL/SQL code for any procedure that you have in your schema.

Practice 6

1. Create a package called `TABLE_PKG` that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.
   a. Create a package specification with the following procedures:
   ```
   PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2)
   PROCEDURE add_row(table_name VARCHAR2, col_values
   VARCHAR2,
     cols VARCHAR2 := NULL)
   PROCEDURE upd_row(table_name VARCHAR2, set_values
   VARCHAR2,
     conditions VARCHAR2 := NULL)
   PROCEDURE del_row(table_name VARCHAR2,
    conditions VARCHAR2 := NULL);
   PROCEDURE remove(table_name VARCHAR2)
   ```
   Ensure that subprograms manage optional default parameters with `NULL` values.
   b. Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the `remove` procedure that should be written using the `DBMS_SQL` package.
   c. Execute the package `MAKE` procedure to create a table as follows:
   ```
   make('my_contacts', 'id number(4), name varchar2(40)');
   ```
   d. Describe the `MY_CONTACTS` table structure.
   e. Execute the `ADD_ROW` package procedure to add the following rows:
   ```
   add_row('my_contacts','1,''Geoff Gallus''','id, name');
   add_row('my_contacts','2,''Nancy''','id, name');
   add_row('my_contacts','3,''Sunitha Patel''','id,name');
   add_row('my_contacts','4,''Valli Pataballa''','id,name');
   ```
   f. Query the `MY_CONTACTS` table contents.
   g. Execute the `DEL_ROW` package procedure to delete a contact with `ID` value 1.
   h. Execute the `UPD_ROW` procedure with the following row data:
   ```
   upd_row('my_contacts','name=''Nancy Greenberg''','id=2');
   ```
   i. Select the data from the `MY_CONTACTS` table again to view the changes.
   j. Drop the table by using the `remove` procedure and describe the `MY_CONTACTS` table.
2. Create a `COMPILE_PKG` package that compiles the PL/SQL code in your schema.
   a. In the specification, create a package procedure called `MAKE` that accepts the `name` of a PL/SQL program unit to be compiled.
   b. In the body, the `MAKE` procedure should call a private function named `GET_TYPE` to determine the PL/SQL object type from the data dictionary, and return the type name (use `PACKAGE` for a package with a body) if the object exists; otherwise, it should return a `NULL`. If the object exists, `MAKE` dynamically compiles it with the `ALTER` statement.
   c. Use the `COMPILE_PKG.MAKE` procedure to compile the `EMPLOYEE_REPORT` procedure, the `EMP_PKG` package, and a nonexistent object called `EMP_DATA`.

Practice 6 (continued)

3. Add a procedure to the `COMPILE_PKG` that uses the `DBMS_METADATA` to obtain a DDL statement that can regenerate a named PL/SQL subprogram, and writes the DDL statement to a file by using the `UTL_FILE` package.
   a. In the package specification, create a procedure called `REGENERATE` that accepts the name of a PL/SQL component to be regenerated. Declare a public `VARCHAR2` variable called `dir` initialized with the directory alias value `'UTL_FILE'`. Compile the specification.
   b. In the package body, implement the `REGENERATE` procedure so that it uses the `GET_TYPE` function to determine the PL/SQL object type from the supplied name. If the object exists, then obtain the DDL statement used to create the component using the `DBMS_METADATA.GET_DDL` procedure, which must be provided with the object name in uppercase text.
      Save the DDL statement in a file by using the `UTL_FILE.PUT` procedure. Write the file in the directory path stored in the public variable called `dir` (from the specification). Construct a file name (in lowercase characters) by concatenating the `USER` function, an underscore, and the object name with a `.sql` extension. For example: `ora1_myobject.sql`. Compile the body.
   c. Execute the `COMPILE_PKG.REGENERATE` procedure by using the name of the `TABLE_PKG` created in the first task of this practice.
   d. Use Putty FTP to get the generated file from the server to your local directory. Edit the file to insert a `/` terminator character at the end of a `CREATE` statement (if required). Cut and paste the results into the *i*SQL*Plus buffer and execute the statement.