8

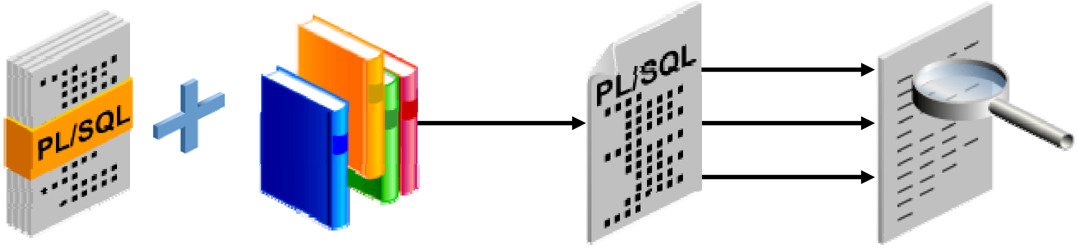
Analyzing PL/SQL Code

ORACLE

Objectives

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

- Find information about your PL/SQL code
- Trace PL/SQL program execution
- Profile PL/SQL applications



Supplied packages

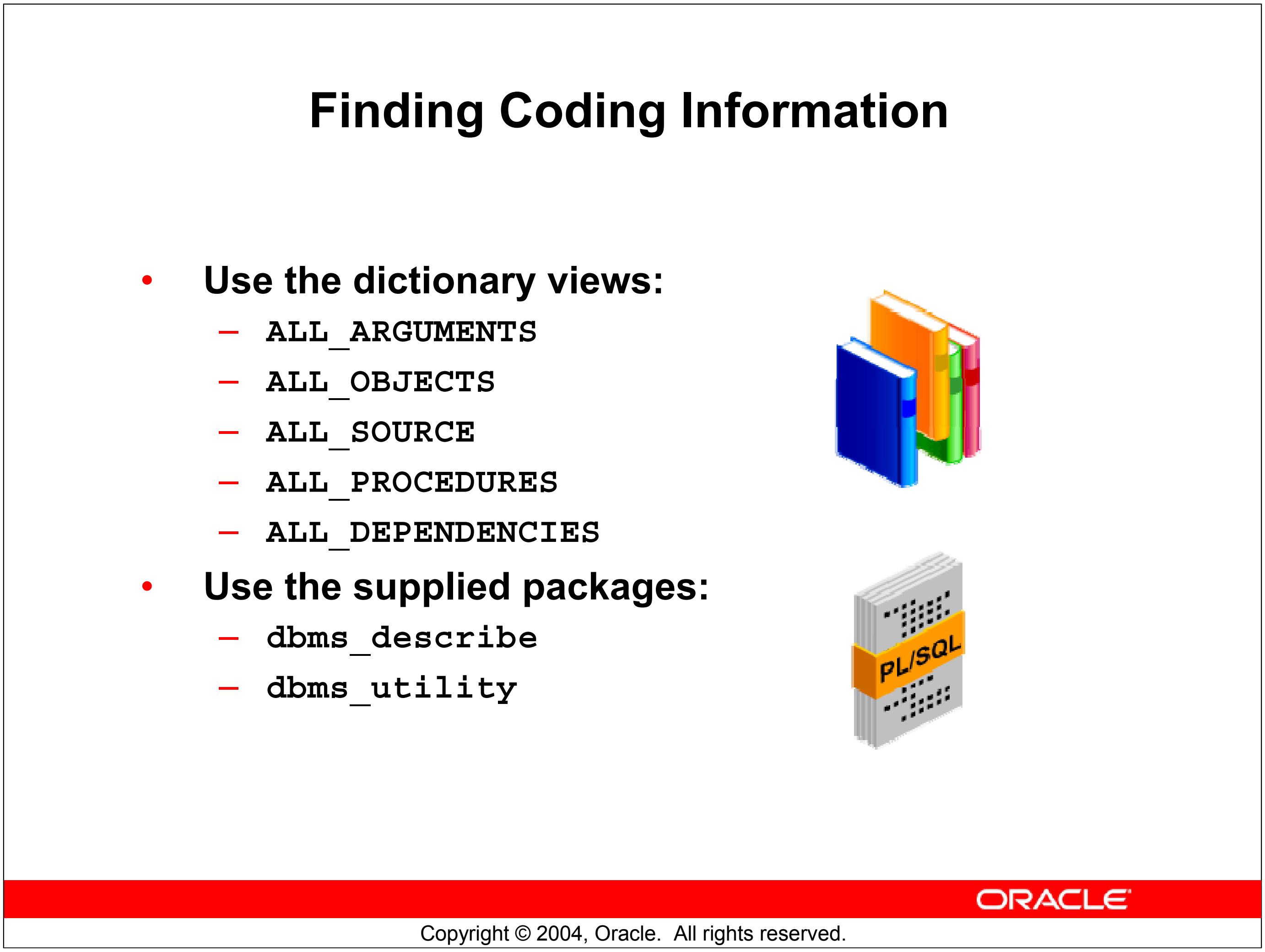
Dictionary views

Code analysis

Interpret information

Objectives

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications. You are introduced to testing PL/SQL code, tracing PL/SQL code, and profiling PL/SQL code.



Finding Coding Information

- Use the dictionary views:
  - ALL_ARGUMENTS
  - ALL_OBJECTS
  - ALL_SOURCE
  - ALL_PROCEDURES
  - ALL_DEPENDENCIES
- Use the supplied packages:
  - dbms_describe
  - dbms_utility

Finding Information on Your PL/SQL Code

The Oracle dictionary views store information on your compiled PL/SQL code. You can write SQL statements against the views to find information about your code.

Dictionary View	Description
ALL_SOURCE	Includes the lines of source code for all the programs you modify
ALL_ARGUMENTS	Includes information about the parameters to the procedures and functions you can call
ALL_PROCEDURES	Contains the list of procedures and functions you can execute
ALL_DEPENDENCIES	Is one of the several views that give you information about dependencies between database objects.

You can also use the Oracle-supplied DBMS_DESCRIBE package to obtain information about a PL/SQL object. The package contains the DESCRIBE_PROCEDURE procedure, which provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

You can use the DBMS_UTILITY supplied package to follow a call stack and an exception stack.

Finding Coding Information

Find all instances of CHAR in your code:

```
SELECT NAME, line, text
FROM         user_source
WHERE        INSTR (UPPER(text), ' CHAR') > 0
             OR INSTR (UPPER(text), ' CHAR(') > 0
             OR INSTR (UPPER(text), ' CHAR (') > 0;

NAME              LINE TEXT
----------------- ---- --------------------------------
CUST_ADDRESS_TYP     6    , country_id         CHAR(2)
```

ORACLE

Finding Data Types

You may want to find all occurrences of the CHAR data type. The CHAR data type is fixed in length and can cause false negatives on comparisons to VARCHAR2 strings. By finding the CHAR data type, you can modify the object, if appropriate, and change it to VARCHAR2.

Finding Coding Information

Create a package with various queries that you can easily call:

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
  PROCEDURE find_text_in_code (str IN VARCHAR2);
  PROCEDURE encap_compliance ;
END query_code_pkg;
/
```

Creating a Package to Query Code

A better idea is to create a package to hold various queries that you can easily call. The QUERY_CODE_PKG will hold two validation procedures:

The FIND_TEXT_IN_CODE procedure displays all programs with a specified character string. It queries USER_SOURCE to find occurrences of a text string passed as a parameter. For efficiency, the BULK COLLECT statement is used to retrieve all matching rows into the collection variable.

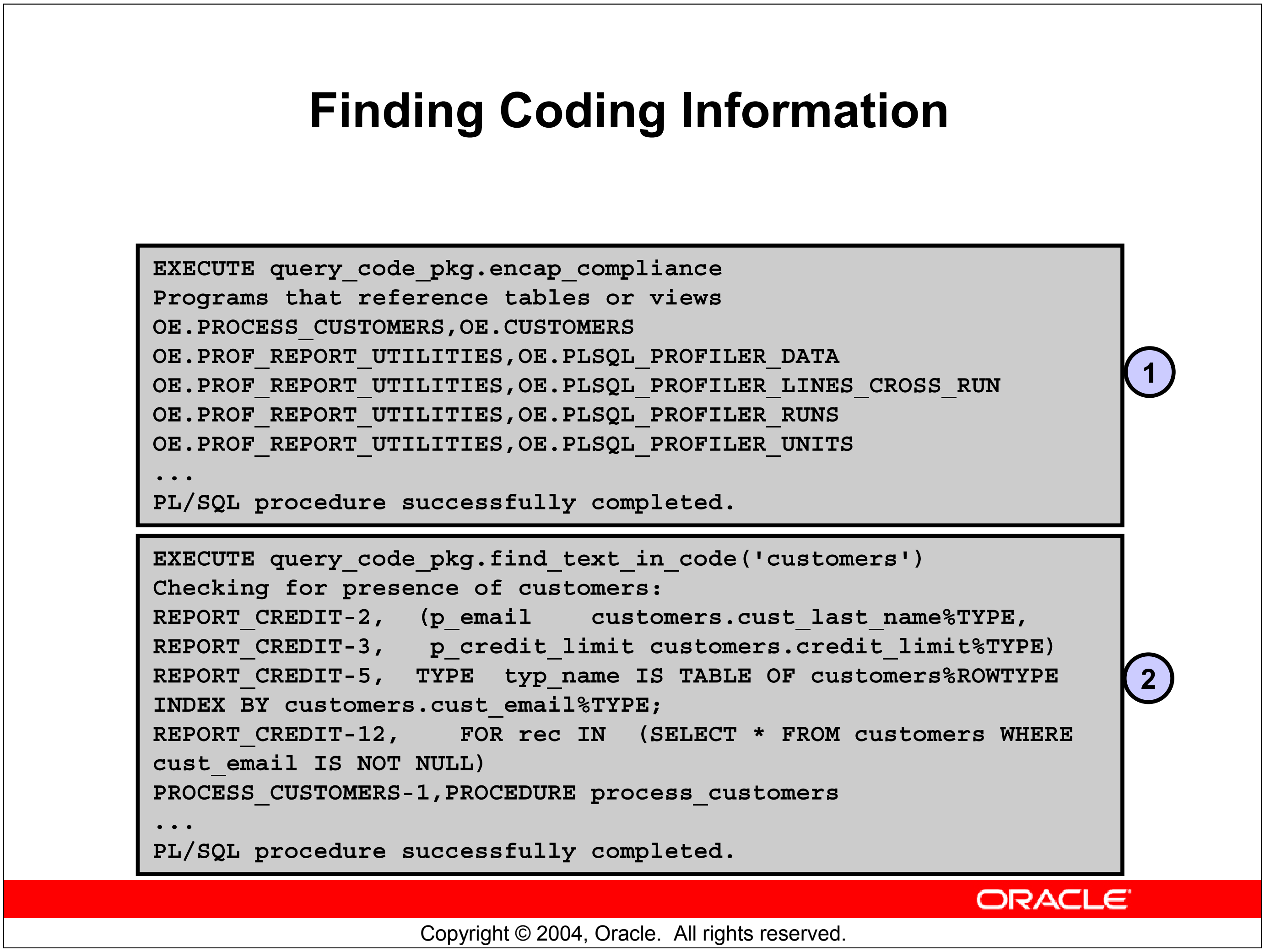
The ENCAP_COMPLIANCE procedure identifies programs that reference a table directly. This procedure queries the ALL_DEPENDENCIES view to find PL/SQL code objects that directly reference a table or a view.

You can also include a procedure to validate a set of standards for exception handling.

Creating a Package to Query Code (continued)

QUERY_CODE_PKG Code

```
CREATE OR REPLACE PACKAGE BODY query_code_pkg IS
  PROCEDURE find_text_in_code (str IN VARCHAR2)
  IS
    TYPE info_rt IS RECORD (NAME user_source.NAME%TYPE,
      text user_source.text%TYPE );
    TYPE info_aat IS TABLE OF info_rt INDEX BY PLS_INTEGER;
    info_aa info_aat;
  BEGIN
    SELECT NAME || '-' || line, text
    BULK COLLECT INTO info_aa FROM user_source
      WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
      AND NAME != 'VALSTD' AND NAME != 'ERRNUMS';
    DBMS_OUTPUT.PUT_LINE ('Checking for presence of '||
                          str || ':');
    FOR indx IN info_aa.FIRST .. info_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
          info_aa (indx).NAME|| ',' || info_aa (indx).text);
    END LOOP;
  END find_text_in_code;

  PROCEDURE encap_compliance IS
    SUBTYPE qualified_name_t IS VARCHAR2 (200);
    TYPE refby_rt IS RECORD (NAME qualified_name_t,
         referenced_by qualified_name_t );
    TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
    refby_aa refby_aat;
  BEGIN
    SELECT owner || '.' || NAME refs_table
          , referenced_owner || '.' || referenced_name
          AS table_referenced
    BULK COLLECT INTO refby_aa
      FROM all_dependencies
      WHERE owner = USER
      AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
                   'PROCEDURE', 'FUNCTION')
      AND referenced_type IN ('TABLE', 'VIEW')
      AND referenced_owner NOT IN ('SYS', 'SYSTEM')
     ORDER BY owner, NAME, referenced_owner, referenced_name;
    DBMS_OUTPUT.PUT_LINE ('Programs that reference
                          tables or views');
    FOR indx IN refby_aa.FIRST .. refby_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (refby_aa (indx).NAME || ',' ||
            refby_aa (indx).referenced_by);
    END LOOP;
 END encap_compliance;
END query_code_pkg;
/
```



Finding Coding Information

```
EXECUTE query_code_pkg.encap_compliance
Programs that reference tables or views
OE.PROCESS_CUSTOMERS,OE.CUSTOMERS
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_DATA
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_LINES_CROSS_RUN
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_RUNS
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_UNITS
...
PL/SQL procedure successfully completed.
```

1

```
EXECUTE query_code_pkg.find_text_in_code('customers')
Checking for presence of customers:
REPORT_CREDIT-2,  (p_email    customers.cust_last_name%TYPE,
REPORT_CREDIT-3,   p_credit_limit customers.credit_limit%TYPE)
REPORT_CREDIT-5,  TYPE  typ_name IS TABLE OF customers%ROWTYPE
INDEX BY customers.cust_email%TYPE;
REPORT_CREDIT-12,    FOR rec IN  (SELECT * FROM customers WHERE
cust_email IS NOT NULL)
PROCESS_CUSTOMERS-1,PROCEDURE process_customers
...
PL/SQL procedure successfully completed.
```

2

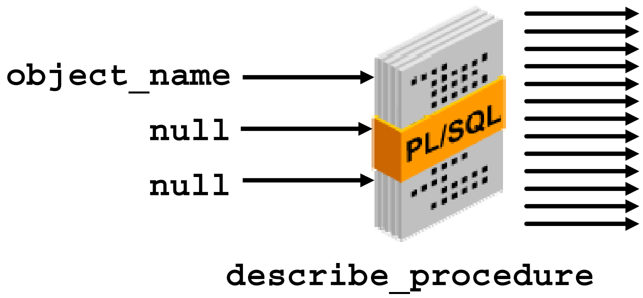
`QUERY_CODE_PKG` Examples

In the first example, the `ENCAP_COMPLIANCE` procedure displays all PL/SQL code objects that reference a table or view directly. Both the code name and table or view name are listed in the output.

In the second example, the `FIND_TEXT_IN_CODE` procedure returns all PL/SQL code objects that contain the “customers” text string. The code name, line number, and line are listed in the output.

Using DBMS_DESCRIBE

- Use it to get information about a PL/SQL object.
- It contains one procedure: DESCRIBE_PROCEDURE.
- Includes:
  - Three scalar IN parameters
  - One scalar OUT parameter
  - Twelve associative array OUT parameters



object_name

null

null

PL/SQL

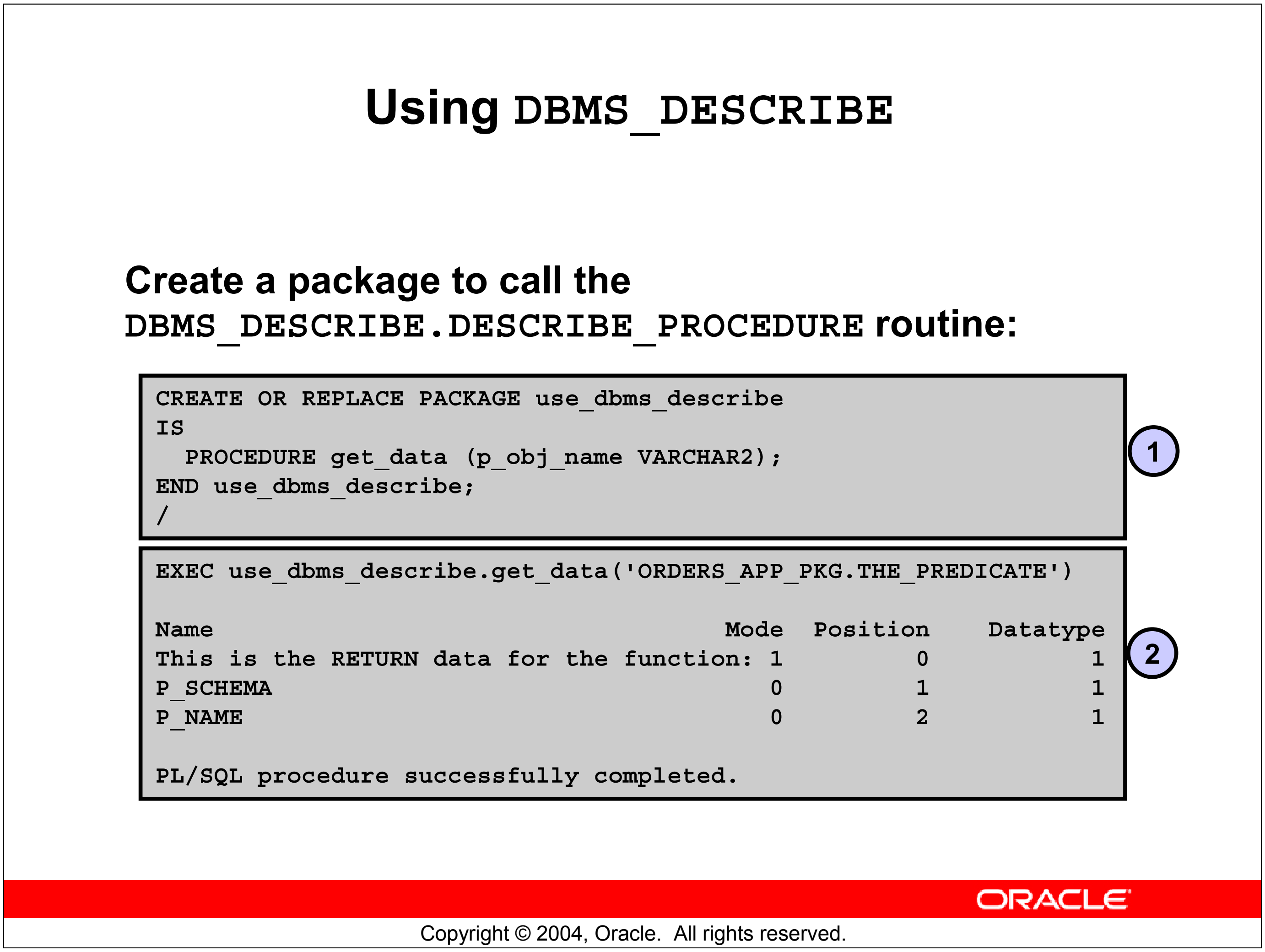
describe_procedure

The DBMS_DESCRIBE Package

You can use the DBMS_DESCRIBE package to find information about your procedures. It contains one procedure, named DESCRIBE_PROCEDURE. This routine accepts the name of the procedure that you are inquiring about. It returns detailed parameter information in a set of associative arrays. The details are numerically coded. You can find the following information from the results returned:

- **Overload:** If overloaded, it holds a value for each version of the procedure.
- **Position:** Position of the argument in the parameter list. 0 is reserved for the RETURN information of a function.
- **Level:** For composite types only; it holds the level of the data type
- **Argument name:** Name of the argument
- **Data type:** A numerically coded value representing a data type
- **Default value:** 0 for no default value, 1 if the argument has a default value
- **Parameter mode:** 0 = IN, 1 = OUT, 2 = IN OUT

Note: This is not the complete list of values returned from the DESCRIBE_PROCEDURE routine. For a complete list, see the *PL/SQL Packages and Types Reference 10g Release 1* reference manual.



Using DBMS_DESCRIBE

Create a package to call the DBMS_DESCRIBE.DESCRIBE_PROCEDURE routine:

```
CREATE OR REPLACE PACKAGE use_dbms_describe
IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
```

1

```
EXEC use_dbms_describe.get_data('ORDERS_APP_PKG.THE_PREDICATE')

Name                                   Mode  Position    Datatype
This is the RETURN data for the function: 1         0           1
P_SCHEMA                                  0         1           1
P_NAME                                    0         2           1

PL/SQL procedure successfully completed.
```

2

The DESCRIBE_PROCEDURE Routine

Because the DESCRIBE_PROCEDURE returns information about your parameters in a set of associative arrays, it is easiest to define a package to call and handle the information returned from it.

In the first example shown on the slide above, the specification for the USE_DBMS_DESCRIBE package is defined. This package holds one procedure, GET_DATA. This GET_DATA routine calls the DBMS_DESCRIBE.DESCRIBE_PROCEDURE routine. The implementation of the USE_DBMS_DESCRIBE package is shown on the next page. Note that several associative array variables are defined to hold the values returned via OUT parameters from the DESCRIBE_PROCEDURE routine. Each of these arrays uses the predefined package types:

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
  INDEX BY BINARY_INTEGER;
TYPE NUMBER_TABLE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

In the call to the DESCRIBE_PROCEDURE routine, you need to pass three parameters: the name of the procedure that you are inquiring about and two null values. These null values are reserved for future use.

In the second example shown on the slide above, the results are displayed for the parameters of the ORDERS_APP_PKG.THE_PREDICATE function. Data type of 1 indicates it is a VARCHAR2 data type.

The DESCRIBE_PROCEDURE Routine (continued)

Calling DBMS_DESCRIBE.DESCRIBE_PROCEDURE

```
CREATE OR REPLACE PACKAGE use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
CREATE OR REPLACE PACKAGE BODY use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2)
  IS
    v_overload     DBMS_DESCRIBE.NUMBER_TABLE;
    v_position     DBMS_DESCRIBE.NUMBER_TABLE;
    v_level        DBMS_DESCRIBE.NUMBER_TABLE;
    v_arg_name     DBMS_DESCRIBE.VARCHAR2_TABLE;
    v_datatype     DBMS_DESCRIBE.NUMBER_TABLE;
    v_def_value    DBMS_DESCRIBE.NUMBER_TABLE;
    v_in_out       DBMS_DESCRIBE.NUMBER_TABLE;
    v_length       DBMS_DESCRIBE.NUMBER_TABLE;
    v_precision    DBMS_DESCRIBE.NUMBER_TABLE;
    v_scale        DBMS_DESCRIBE.NUMBER_TABLE;
    v_radix        DBMS_DESCRIBE.NUMBER_TABLE;
    v_spare        DBMS_DESCRIBE.NUMBER_TABLE;
  BEGIN
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE
    (p_obj_name, null, null, -- these are the 3 in parameters
     v_overload, v_position, v_level, v_arg_name,
     v_datatype, v_def_value, v_in_out, v_length,
     v_precision, v_scale, v_radix, v_spare, null);
    IF v_in_out.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE ('No arguments to report.');
    ELSE
      DBMS_OUTPUT.PUT
      ('Name                                        Mode');
      DBMS_OUTPUT.PUT_LINE('  Position    Datatype ');
      FOR i IN v_arg_name.FIRST .. v_arg_name.LAST LOOP
        IF v_position(i) = 0 THEN
          DBMS_OUTPUT.PUT('This is the RETURN data for
          the function: ');
        ELSE
          DBMS_OUTPUT.PUT (
            rpad(v_arg_name(i), LENGTH(v_arg_name(i)) +
                 42-LENGTH(v_arg_name(i)), ' '));
        END IF;
        DBMS_OUTPUT.PUT( '     ' ||
          v_in_out(i) || '         ' || v_position(i) ||
          '           ' || v_datatype(i) );
        DBMS_OUTPUT.NEW_LINE;
      END LOOP;
    END IF;
  END get_data;
END use_dbms_describe;
```

Using ALL_ARGUMENTS

Query the ALL_ARGUMENTS view to find information about arguments for procedures and functions:

```
SELECT object_name, argument_name, in_out, position, data_type
FROM   all_arguments
WHERE  package_name = 'ORDERS_APP_PKG';

OBJECT_NAME       ARGUMENT_NAME   IN_OUT    POSITION DATA_TYPE
----------------- --------------- -------- --------- -----------
THE_PREDICATE     P_NAME          IN               2 VARCHAR2
THE_PREDICATE     P_SCHEMA        IN               1 VARCHAR2
THE_PREDICATE                     OUT              0 VARCHAR2
SET_APP_CONTEXT                   IN               1
SHOW_APP_CONTEXT                  IN               1
```

Using the ALL_ARGUMENTS Dictionary View

You can also query the ALL_ARGUMENTS dictionary view to find out information about the arguments of procedures and functions to which you have access. Similar to using DBMS_DESCRIBE, the ALL_ARGUMENTS view returns information in textual rather than numeric form. There is overlap between the two, but there is also unique information to be found both in DBMS_DESCRIBE and ALL_ARGUMENTS.

In the example shown above, the argument name, mode, position, and data type are returned for the ORDERS_APP_PKG. Note the following:

- A position of 1 and a sequence and level of 0 indicates that the procedure has no arguments.
- For a function that has no arguments, it is displayed as a single row for the RETURN clause, with a position of 0.
- The argument name for the RETURN clause is NULL.
- If programs are overloaded, the OVERLOAD column (not shown above) indicates the *N*th overloading; otherwise, it is NULL.
- The DATA_LEVEL column (not shown above) value of 0 identifies a parameter as it appears in the program specification.

Using ALL_ARGUMENTS

Other column information:

- Details about the data type are found in the DATA_TYPE and TYPE_ columns.
- All arguments in the parameter list are at level 0.
- For composite parameters, the individual elements of the composite are assigned levels, starting at 1.
- The POSITION-DATA_LEVEL column combination is unique only for a level 0 argument (the actual parameter, not its subtypes if it is a composite).

ORACLE

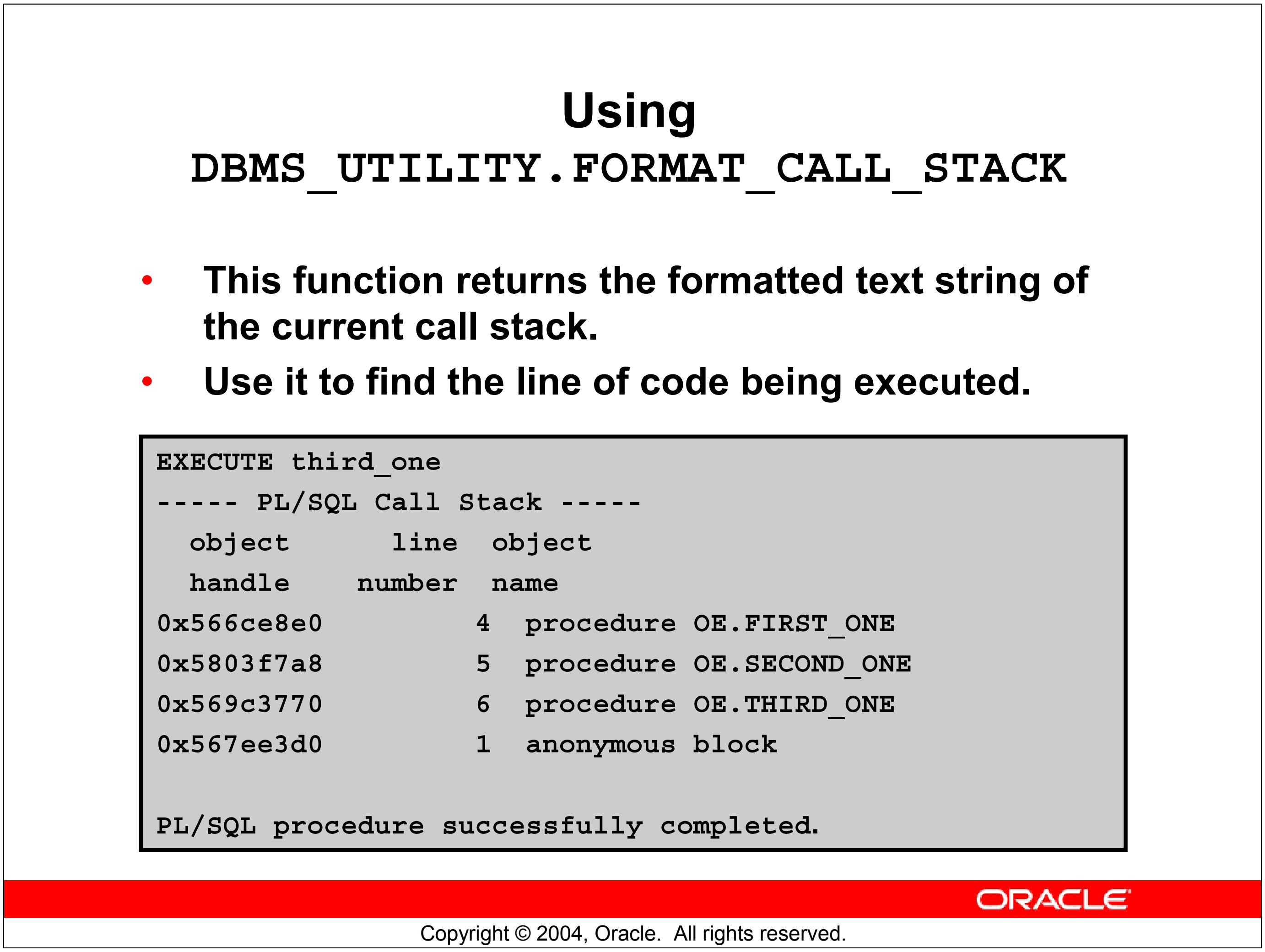
Using the ALL_ARGUMENTS Dictionary View (continued)

The DATA_TYPE column holds the generic PL/SQL data type. To find more information about the data type, query the TYPE_ columns.

- **TYPE_NAME:** Holds the name of the type of the argument. If the type is a package local type (that is, it is declared in a package specification), then this column displays the name of the package.
- **TYPE_SUBNAME:** Is relevant only for package local types. Displays the name of the type declared in the package identified in the TYPE_NAME column. For example, if the data type is a PL/SQL table, you can find out which type of table only by looking at the TYPE_SUBNAME column.

Note: The DEFAULT_VALUE and DEFAULT_LENGTH columns are reserved for future use and do not currently contain information about a parameter's default value. You can use DBMS_DESCRIBE to find some default value information. In this package, the parameter DEFAULT_VALUE returns 1 if there is a default value; otherwise, it returns 0.

By combining the information from DBMS_DESCRIBE and ALL_ARGUMENTS, you can find valuable information about parameters, as well as about how your PL/SQL routines are overloaded.



Using DBMS_UTILITY.FORMAT_CALL_STACK

- This function returns the formatted text string of the current call stack.
- Use it to find the line of code being executed.

```
EXECUTE third_one
----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
0x566ce8e0         4  procedure OE.FIRST_ONE
0x5803f7a8         5  procedure OE.SECOND_ONE
0x569c3770         6  procedure OE.THIRD_ONE
0x567ee3d0         1  anonymous block

PL/SQL procedure successfully completed.
```

The DBMS_UTILITY.FORMAT_CALL_STACK Function

Another tool available to you is the FORMAT_CALL_STACK function within the DBMS_UTILITY supplied package. It returns the call stack in a formatted character string. The results shown above were generated based on the following routines:

```
SET SERVEROUT ON
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
  dbms_output.put_line(
    substr(dbms_utility.format_call_Stack, 1, 255));
END;
/

CREATE OR REPLACE PROCEDURE second_one
IS
BEGIN
  null;
  first_one;
END;
/
-- continued on next page
```

The DBMS_UTILITY.FORMAT_CALL_STACK Function (continued)

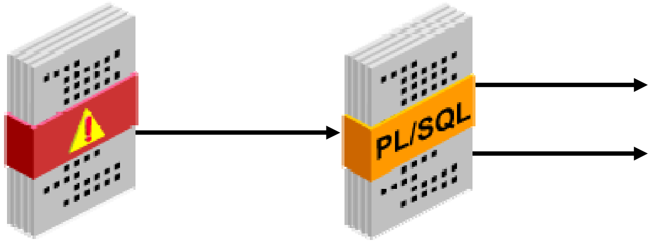
```
-- continued from previous page

CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
  null;
  null;
  second_one;
END;
/
```

The output from the FORMAT_CALL_STACK function shows you the object handle number, line number from where a routine is called, and the routine that is called. Note that the NULL; statements added into the procedures shown are used to emphasize the line number from where the routine is called.

Finding Error Information

`DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`:

- **Shows you the call stack at the point where an exception is raised.**
- **Returns:**
  - **The backtrace string**
  - **A null string if there are no errors being handled**



`Backtrace string`

`Null string`

Raised exception

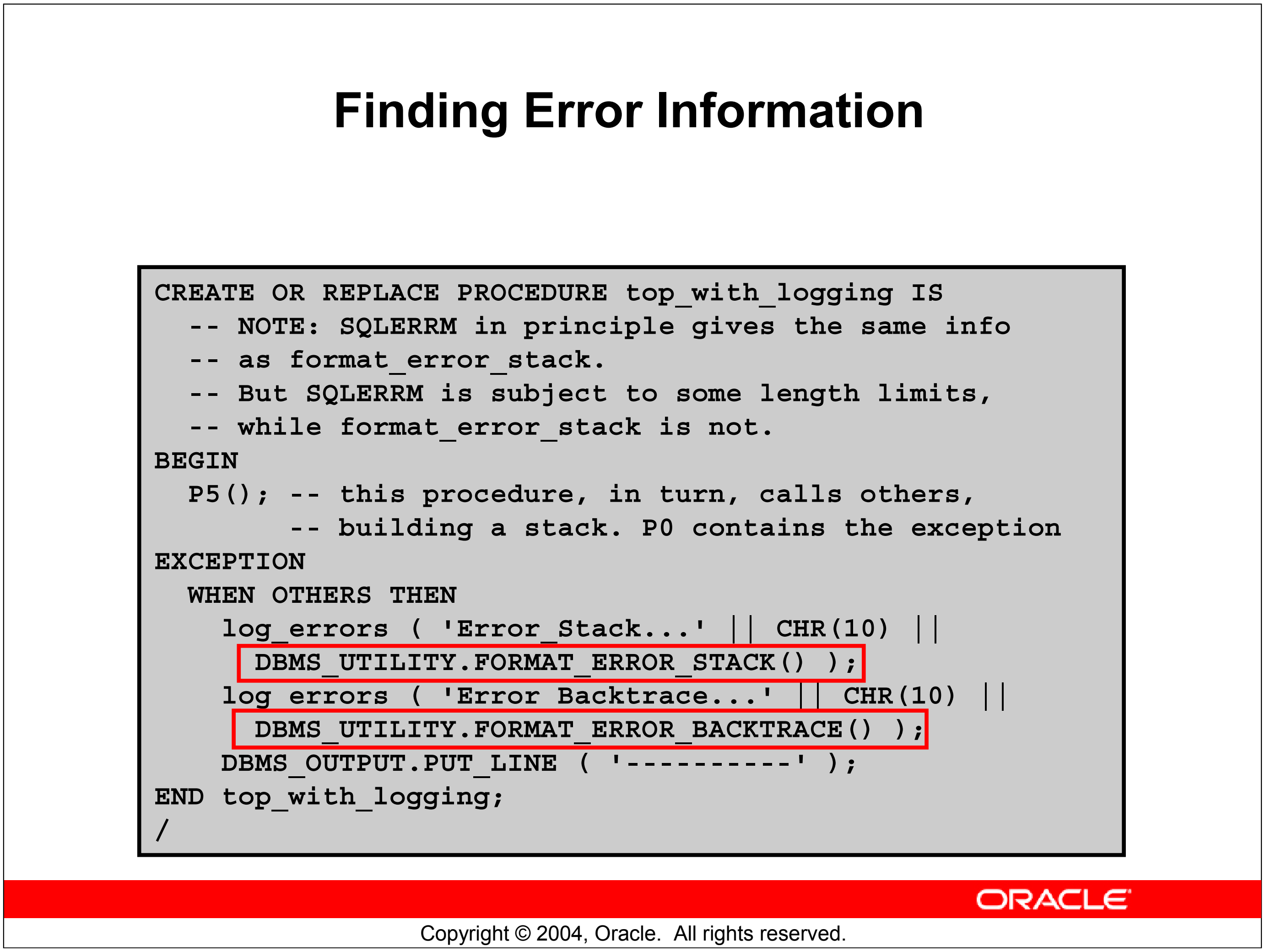
`format_error_backtrace`

Using `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`

You can use this function to display the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output returned is similar to the output of the SQLERRM function, but not subject to the same size limitation.

Using `DBMS_UTILITY.FORMAT_ERROR_STACK`

You can use this function to format the current error stack. It can be used in exception handlers to view the full error stack. The function returns the error stack, up to 2,000 bytes.



Finding Error Information

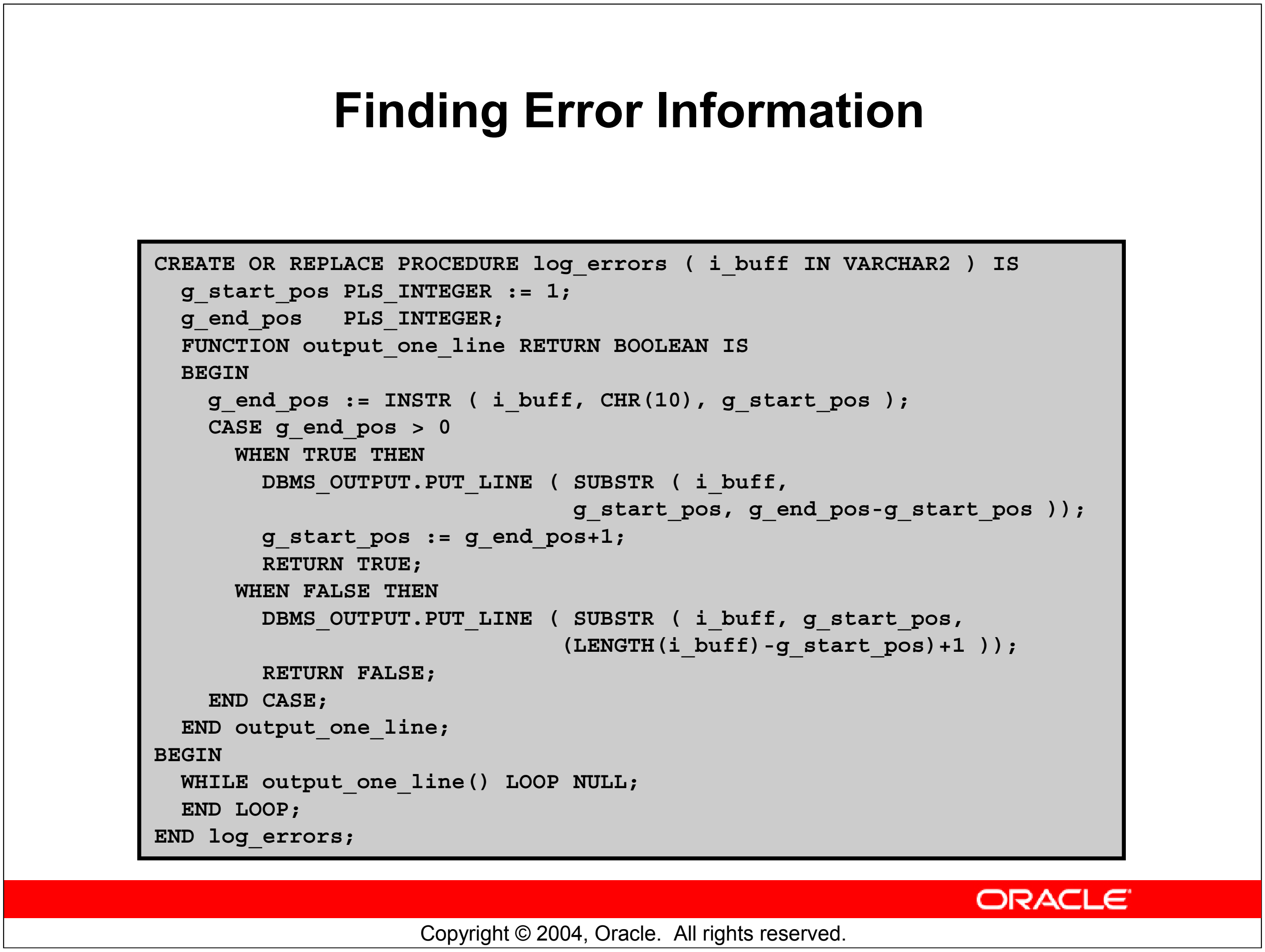
```
CREATE OR REPLACE PROCEDURE top_with_logging IS
  -- NOTE: SQLERRM in principle gives the same info
  -- as format_error_stack.
  -- But SQLERRM is subject to some length limits,
  -- while format_error_stack is not.
BEGIN
  P5(); -- this procedure, in turn, calls others,
        -- building a stack. P0 contains the exception
EXCEPTION
  WHEN OTHERS THEN
    log_errors ( 'Error_Stack...' || CHR(10) ||
     DBMS_UTILITY.FORMAT_ERROR_STACK() );
    log errors ( 'Error Backtrace...' || CHR(10) ||
     DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '----------' );
END top_with_logging;
/
```

ORACLE

Using FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE

To show you the functionality of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions, a TOP_WITH_LOGGING procedure is created. This procedure calls the LOG_ERRORS procedure and passes to it the results of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions.

The LOG_ERRORS procedure is shown on the next page.



Finding Error Information

```
CREATE OR REPLACE PROCEDURE log_errors ( i_buff IN VARCHAR2 ) IS
  g_start_pos PLS_INTEGER := 1;
  g_end_pos   PLS_INTEGER;
  FUNCTION output_one_line RETURN BOOLEAN IS
  BEGIN
    g_end_pos := INSTR ( i_buff, CHR(10), g_start_pos );
    CASE g_end_pos > 0
      WHEN TRUE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff,
                               g_start_pos, g_end_pos-g_start_pos ));
        g_start_pos := g_end_pos+1;
        RETURN TRUE;
      WHEN FALSE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff, g_start_pos,
                              (LENGTH(i_buff)-g_start_pos)+1 ));
        RETURN FALSE;
    END CASE;
  END output_one_line;
BEGIN
  WHILE output_one_line() LOOP NULL;
  END LOOP;
END log_errors;
```

The LOG_ERRORS Example

This procedure takes the return results of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions as an IN string parameter, and reports it back to you using DBMS_OUTPUT.PUT_LINE. The LOG_ERRORS procedure is called twice from the TOP_WITH_LOGGING procedure. The first call passes the results of FORMAT_ERROR_STACK and the second procedure passes the results of FORMAT_ERROR_BACKTRACE .

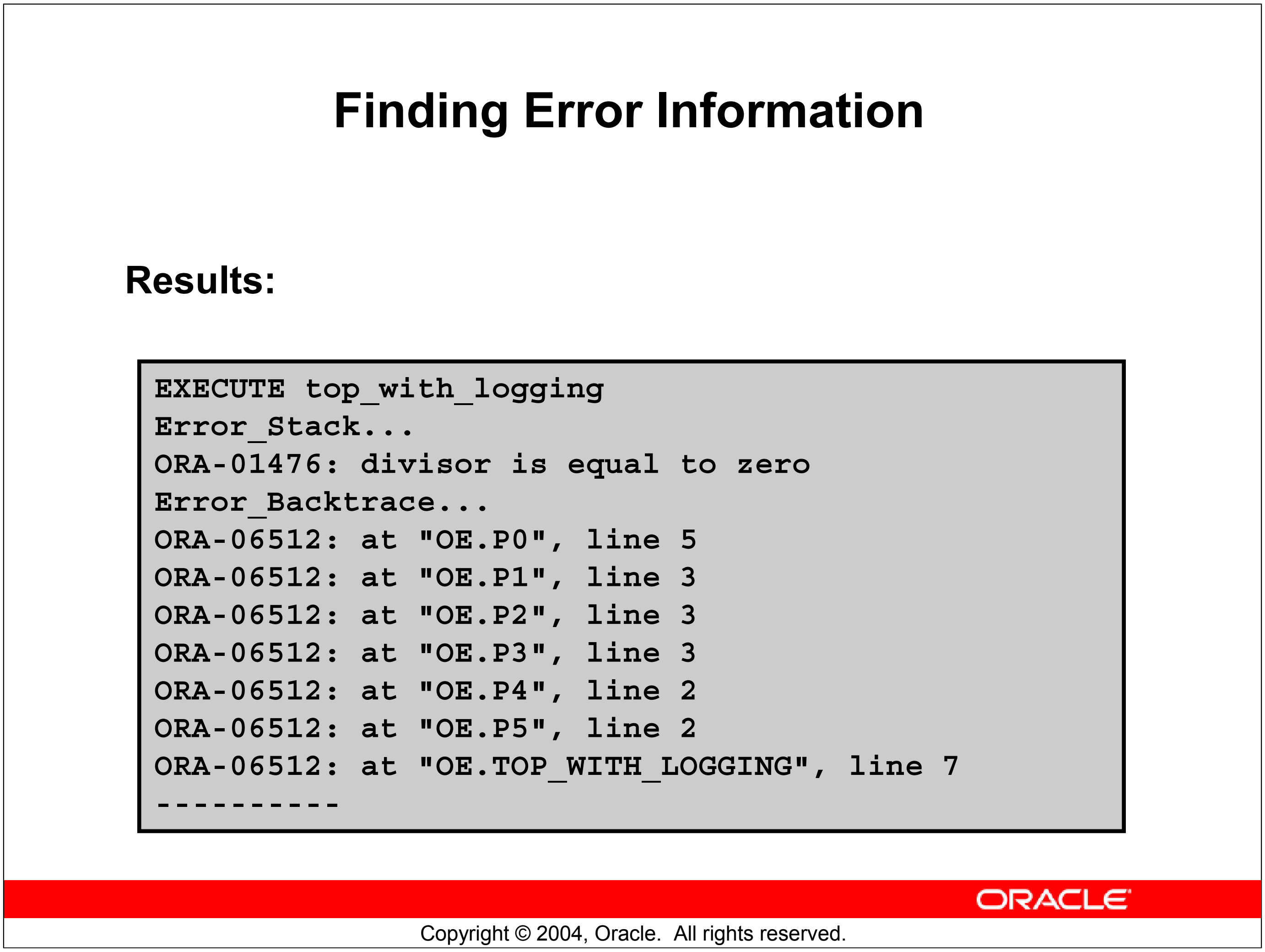
Note: You could use UTL_FILE instead of DBMS_OUTPUT to write and format the results to a file.

The LOG_ERRORS Example (continued)

Next, several procedures are created and one procedure calls another, so that a stack of procedures is built. The P0 procedure raises a zero divide exception when it is invoked. The call stack is:

```
TOP_WITH_LOGGING > P5 > P4 > P3 > P2 > P1 > P0
```

```
SET DOC OFF
SET FEEDBACK OFF
SET ECHO OFF

CREATE OR REPLACE PROCEDURE P0 IS
  e_01476 EXCEPTION;
  pragma exception_init ( e_01476, -1476 );
BEGIN
  RAISE e_01476;  -- this is a zero divide error
END P0;
/
CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
  P0();
END P1;
/
CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
  P1();
END P2;
/
CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
  P2();
END P3;
/
CREATE OR REPLACE PROCEDURE P4 IS
  BEGIN P3();
END P4;
/
CREATE OR REPLACE PROCEDURE P5 IS
  BEGIN P4();
END P5;
/
CREATE OR REPLACE PROCEDURE top IS
BEGIN
  P5(); -- this procedure is used to show the results
        -- without using the TOP_WITH_LOGGING routine.
END top;
/
SET FEEDBACK ON
```



Finding Error Information

Results:

```
EXECUTE top_with_logging
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP_WITH_LOGGING", line 7
----------
```

Finding Error Information Results

The results from executing the TOP_WITH_LOGGING procedure is shown. Note that the error stack displays the exception encountered. The backtrace information traces the flow of the exception to its origin.

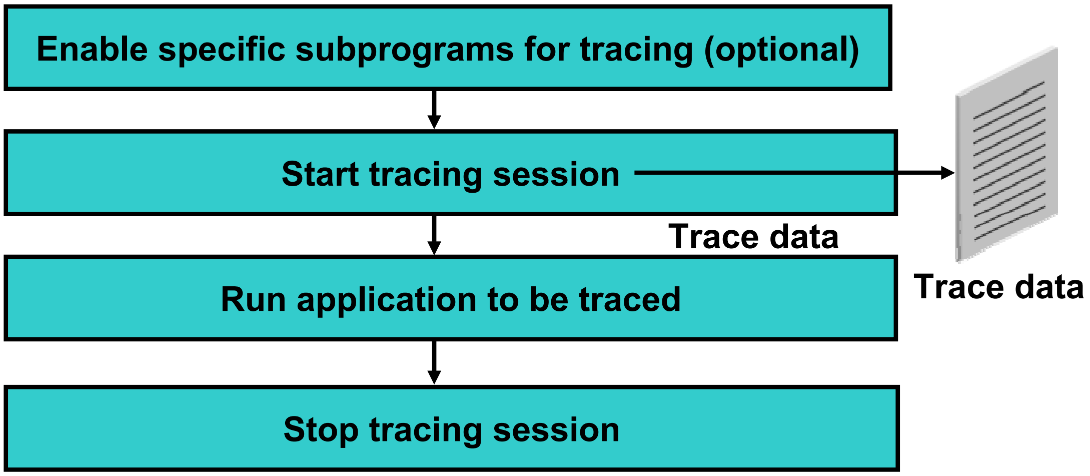
If you execute the TOP procedure without using the TOP_WITH_LOGGING procedure, these are the results:

```
EXECUTE top
BEGIN top; END;
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP", line 3
ORA-06512: at line 1
```

Note that the line number reported is misleading.

Tracing PL/SQL Execution

Tracing PL/SQL execution provides you with a better understanding of the program execution path, and is possible by using the `dbms_trace` package.



Enable specific subprograms for tracing (optional)

Start tracing session

Trace data

Run application to be traced

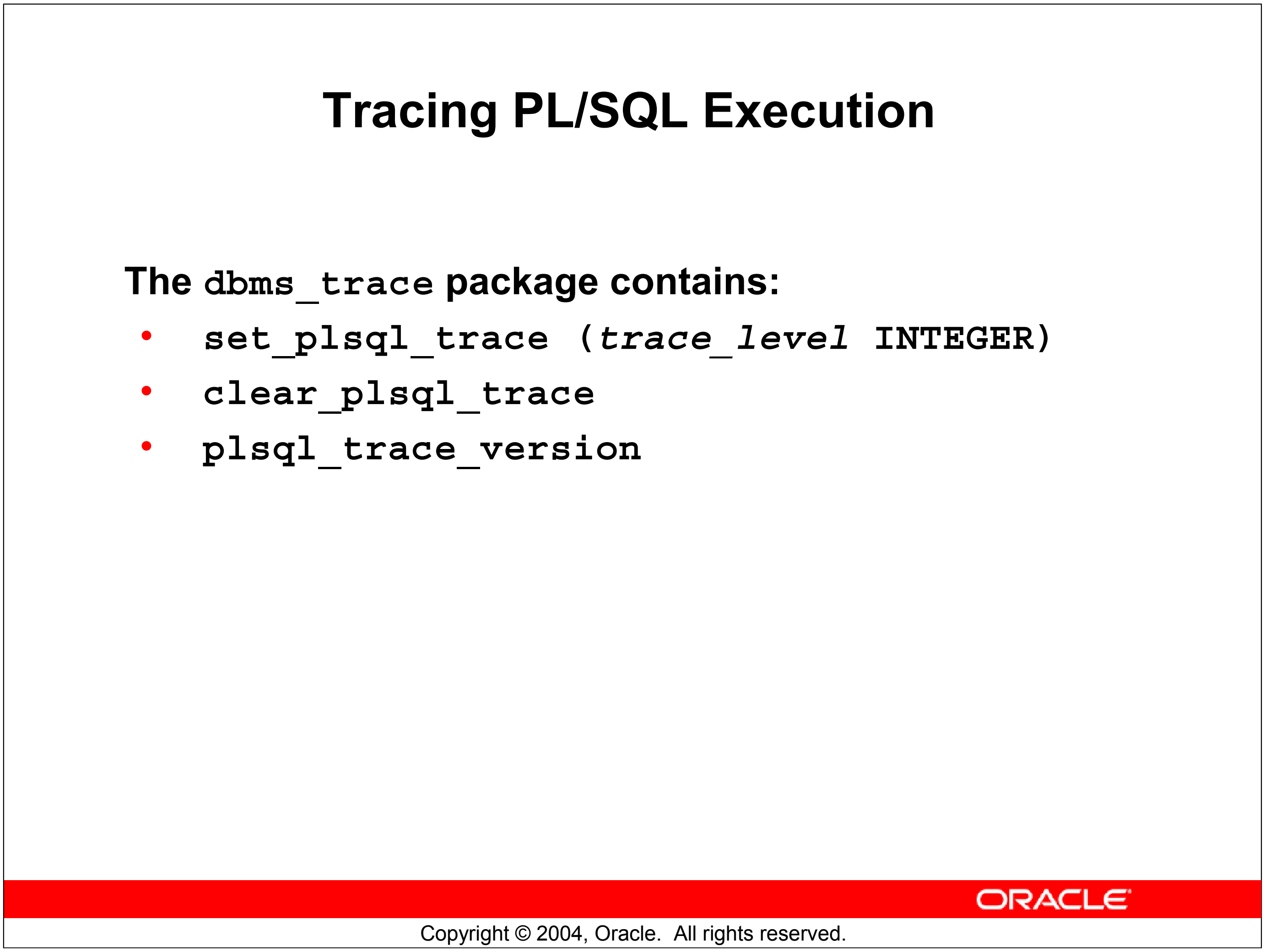
Stop tracing session

Tracing PL/SQL Execution

In large and complex PL/SQL applications, it can sometimes become difficult to keep track of subprogram calls when a number of them call each other. By tracing your PL/SQL code, you can get a clearer idea of the paths and order in which your programs execute.

While a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the `dbms_trace` package, to trace PL/SQL subprogram code.

Note: You cannot use PL/SQL tracing with the multithreaded server (MTS).



Tracing PL/SQL Execution

The `dbms_trace` package contains:

- `set_plsql_trace (`*`trace_level`* `INTEGER)`
- `clear_plsql_trace`
- `plsql_trace_version`

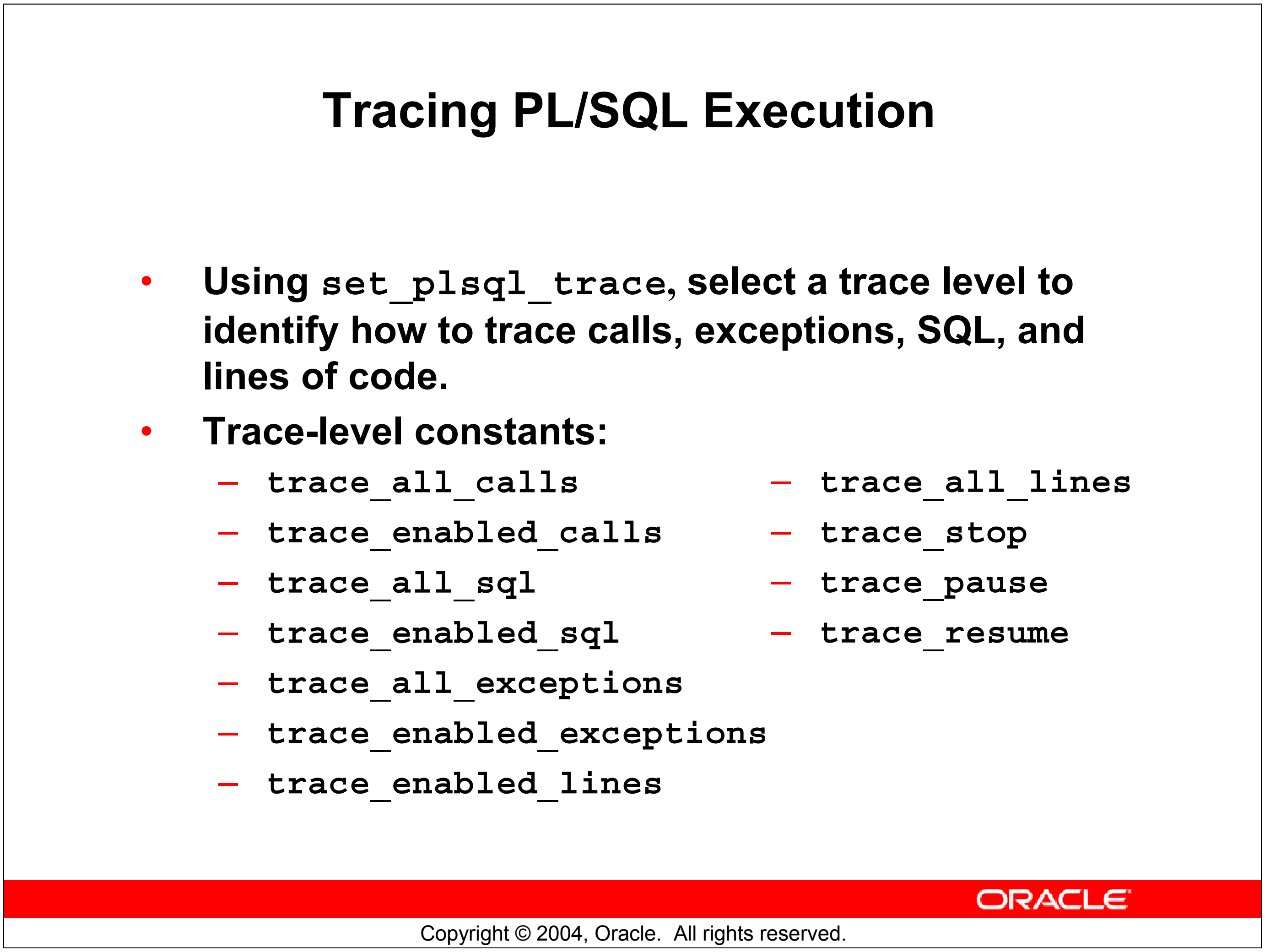
The `dbms_trace` Programs

`dbms_trace` provides subprograms to start and stop PL/SQL tracing in a session. The trace data is collected as the program executes, and it is written out to data dictionary tables.

Procedure	Description
`set_plsql_trace`	Start tracing data dumping in a session (You provide the trace level at which you want your PL/SQL code traced as an `IN` parameter.)
`clear_plsql_trace`	Stops trace data dumping in a session
`plsql_trace_version`	Returns the version number of the trace package as an out parameter

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session (`dbms_trace.set_plsql_trace`)
- Running the application that is to be traced
- Stopping the PL/SQL tracing session (`dbms_trace.clear_plsql_trace`)



Tracing PL/SQL Execution

- Using `set_plsql_trace`, select a trace level to identify how to trace calls, exceptions, SQL, and lines of code.
- Trace-level constants:
  - `trace_all_calls`
  - `trace_enabled_calls`
  - `trace_all_sql`
  - `trace_enabled_sql`
  - `trace_all_exceptions`
  - `trace_enabled_exceptions`
  - `trace_enabled_lines`
  - `trace_all_lines`
  - `trace_stop`
  - `trace_pause`
  - `trace_resume`

ORACLE

Specifying a Trace Level

During the trace session, there are two levels that you can specify to trace calls, exceptions, SQL, and lines of code.

Trace Calls

- **Level 1:** Trace all calls. This corresponds to the constant `trace_all_calls`.
- **Level 2:** Trace calls to enabled program units only. This corresponds to the constant `trace_enabled_calls`.

Trace Exceptions

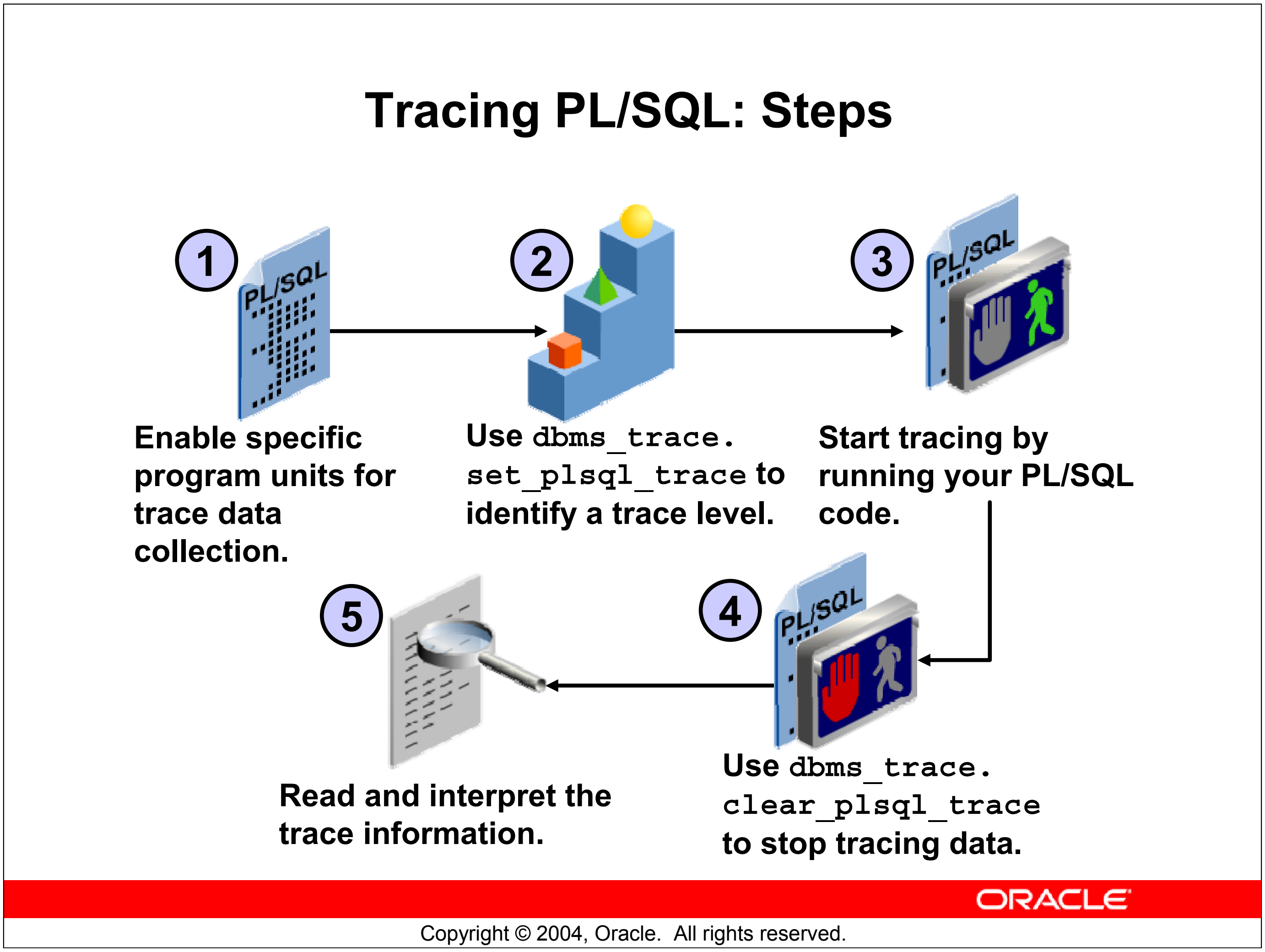
- **Level 1:** Trace all exceptions. This corresponds to `trace_all_exceptions`.
- **Level 2:** Trace exceptions raised in enabled program units only. This corresponds to `trace_enabled_exceptions`.

Trace SQL

- **Level 1:** Trace all SQL. This corresponds to the constant `trace_all_sql`.
- **Level 2:** Trace SQL in enabled program units only. This corresponds to the constant `trace_enabled_sql`.

Trace Lines

- **Level 1:** Trace all lines. This corresponds to the constant `trace_all_lines`.
- **Level 2:** Trace lines in enabled program units only. This corresponds to the constant `trace_enabled_lines`.



Tracing PL/SQL: Steps

1. Enable specific program units for trace data collection.

2. Use `dbms_trace.set_plsql_trace` to identify a trace level.

3. Start tracing by running your PL/SQL code.

4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.

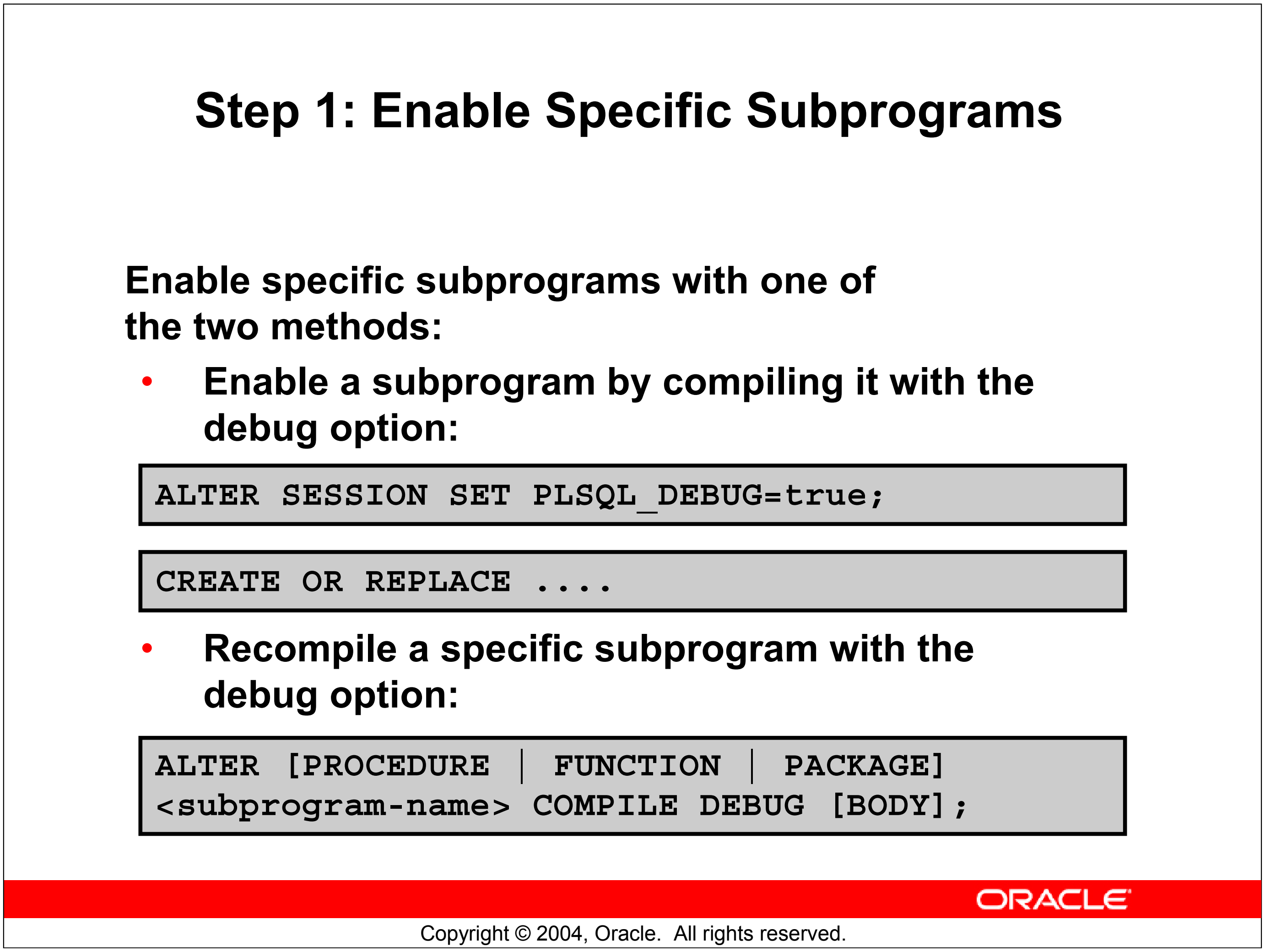
5. Read and interpret the trace information.

Steps to Trace PL/SQL Code

There are five steps to trace PL/SQL code using the `dbms_trace` package:

1. Enable specific program units for trace data collection.
2. Use `dbms_trace.set_plsql_trace` to identify a trace level.
3. Run your PL/SQL code.
4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

The next few pages demonstrate the steps to accomplish PL/SQL tracing.



Step 1: Enable Specific Subprograms

Enable specific subprograms with one of the two methods:

- **Enable a subprogram by compiling it with the debug option:**

```
ALTER SESSION SET PLSQL_DEBUG=true;
```

```
CREATE OR REPLACE ....
```

- **Recompile a specific subprogram with the debug option:**

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
<subprogram-name> COMPILE DEBUG [BODY];
```

Step 1: Enable Specific Subprograms

Profiling large applications may produce a huge volume of data that can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling a specific subprogram for trace data collection. You can enable a subprogram by compiling it with the debug option. You can do this in one of two ways:

- Enable a subprogram by compiling it with the ALTER SESSION debug option, then compile the program unit by using CREATE OR REPLACE syntax:

```
    ALTER SESSION SET PLSQL_DEBUG = true;
    CREATE OR REPLACE ...
```

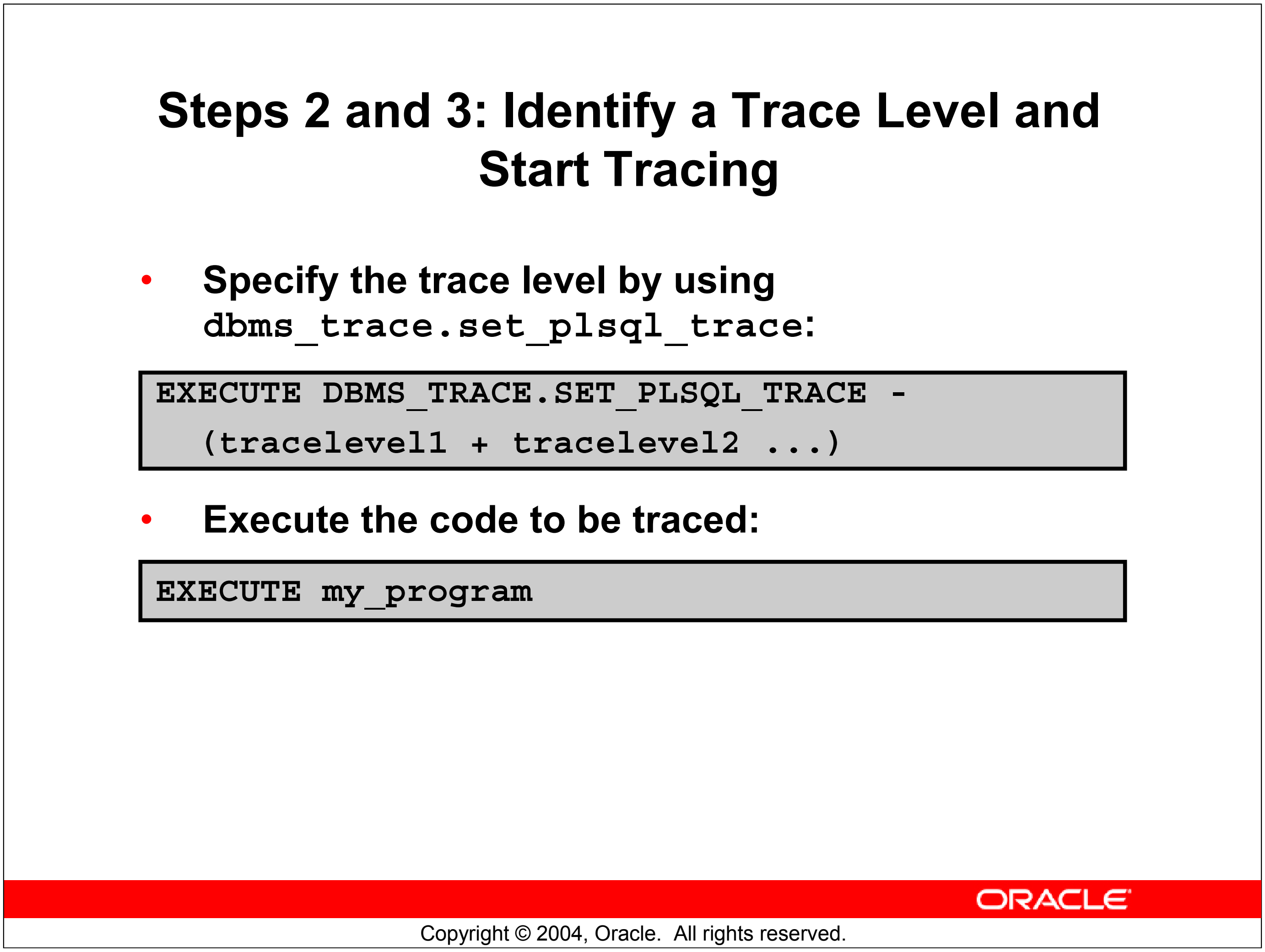
- Alternatively, recompile a specific subprogram with the debug option:

```
    ALTER [PROCEDURE | FUNCTION | PACKAGE]
                 <subprogram-name> COMPILE DEBUG [BODY];
```

Note: The second method cannot be used for anonymous blocks.

Enabling specific subprograms allows you to:

- Limit and control the amount of trace data, especially in large applications.
- Obtain additional trace information that is otherwise not available. For example, during the tracing session, if a subprogram calls another subprogram, the name of the called subprogram gets included in the trace data if the calling subprogram was enabled by compiling it in debug mode.



Steps 2 and 3: Identify a Trace Level and Start Tracing

- Specify the trace level by using `dbms_trace.set_plsql_trace`:

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE -
   (tracelevel1 + tracelevel2 ...)
```

- Execute the code to be traced:

```
EXECUTE my_program
```

Steps 2 and 3: Specify a Trace Level and Start Tracing

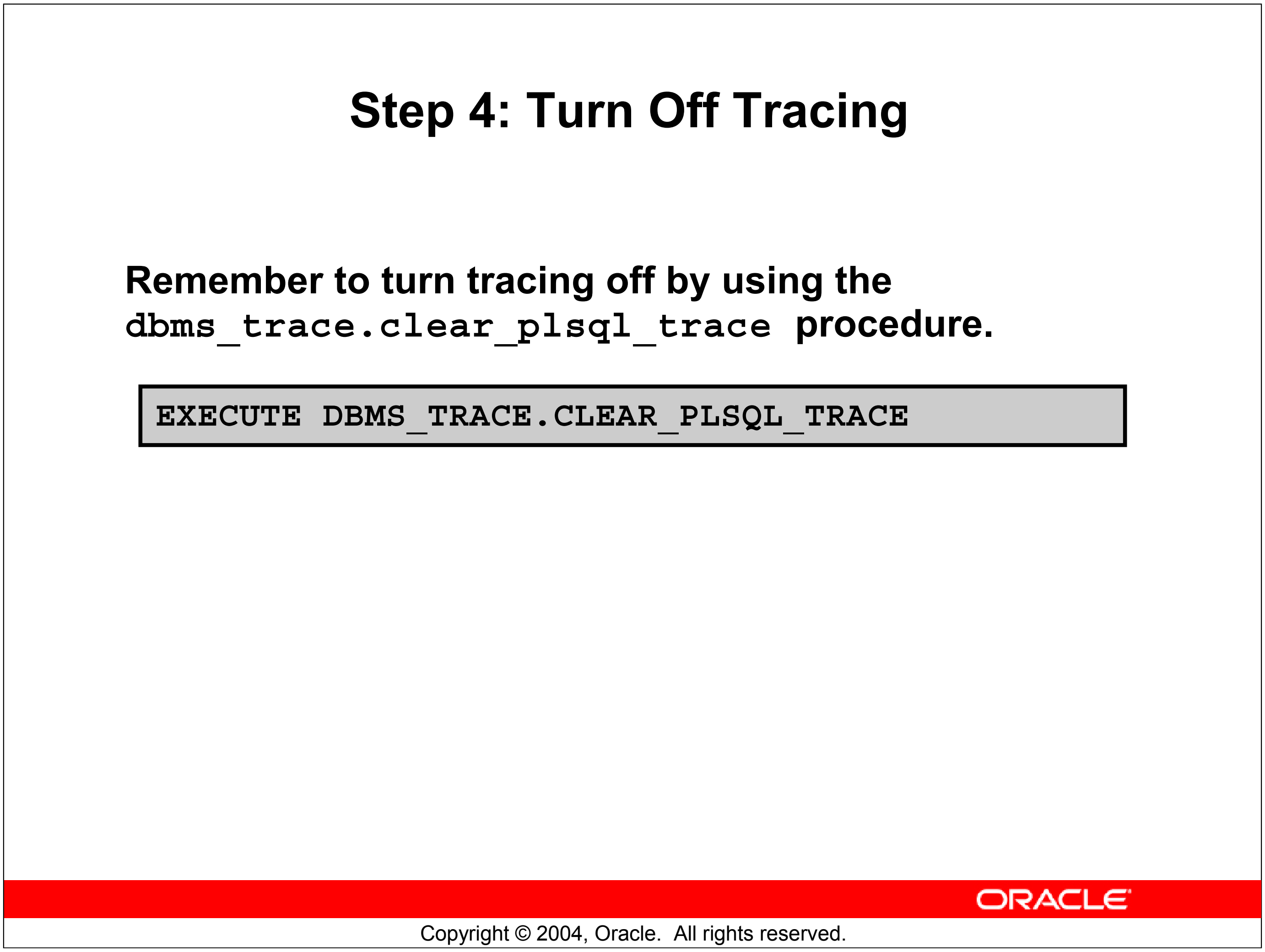
To trace PL/SQL code execution by using `dbms_trace`, follow these steps:

- Start the trace session using the syntax in the slide. For example:

```
EXECUTE -
DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```

Note:

- To specify additional trace levels in the argument, use the "+" sign between each trace level value.
- Execute the PL/SQL code. The trace data gets written to either the Oracle server trace file or to the data dictionary views.



Step 4: Turn Off Tracing

Remember to turn tracing off by using the `dbms_trace.clear_plsql_trace` procedure.

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```

Step 4: Turn Off Tracing

When you have completed tracing the PL/SQL program unit, turn tracing off by executing `dbms_trace.clear_plsql_trace`. This stops any further writing to the trace file.

To avoid the overhead of writing the trace information, it is recommended that you turn off the tracing when you are not using it.

Step 5: Examine the Trace Information

Examine the trace information:

- Call tracing writes out the program unit type, name, and stack depth.
- Exception tracing writes out the line number.

ORACLE

Step 5: Examine the Trace Information

- Lower trace levels supersede higher levels when tracing is activated for multiple tracing levels.
- If tracing is requested only for enabled subprograms, and if the current subprogram is not enabled, then no trace data is written.
- If the current subprogram is enabled, then call tracing writes out the subprogram type, name, and stack depth.
- If the current subprogram is not enabled, then call tracing writes out the subprogram type, line number, and stack depth.
- Exception tracing writes out the line number. Raising the exception shows information about whether the exception is user-defined or predefined and, in the case of predefined exceptions, the exception number.

Note: An enabled subprogram is compiled with the debug option.

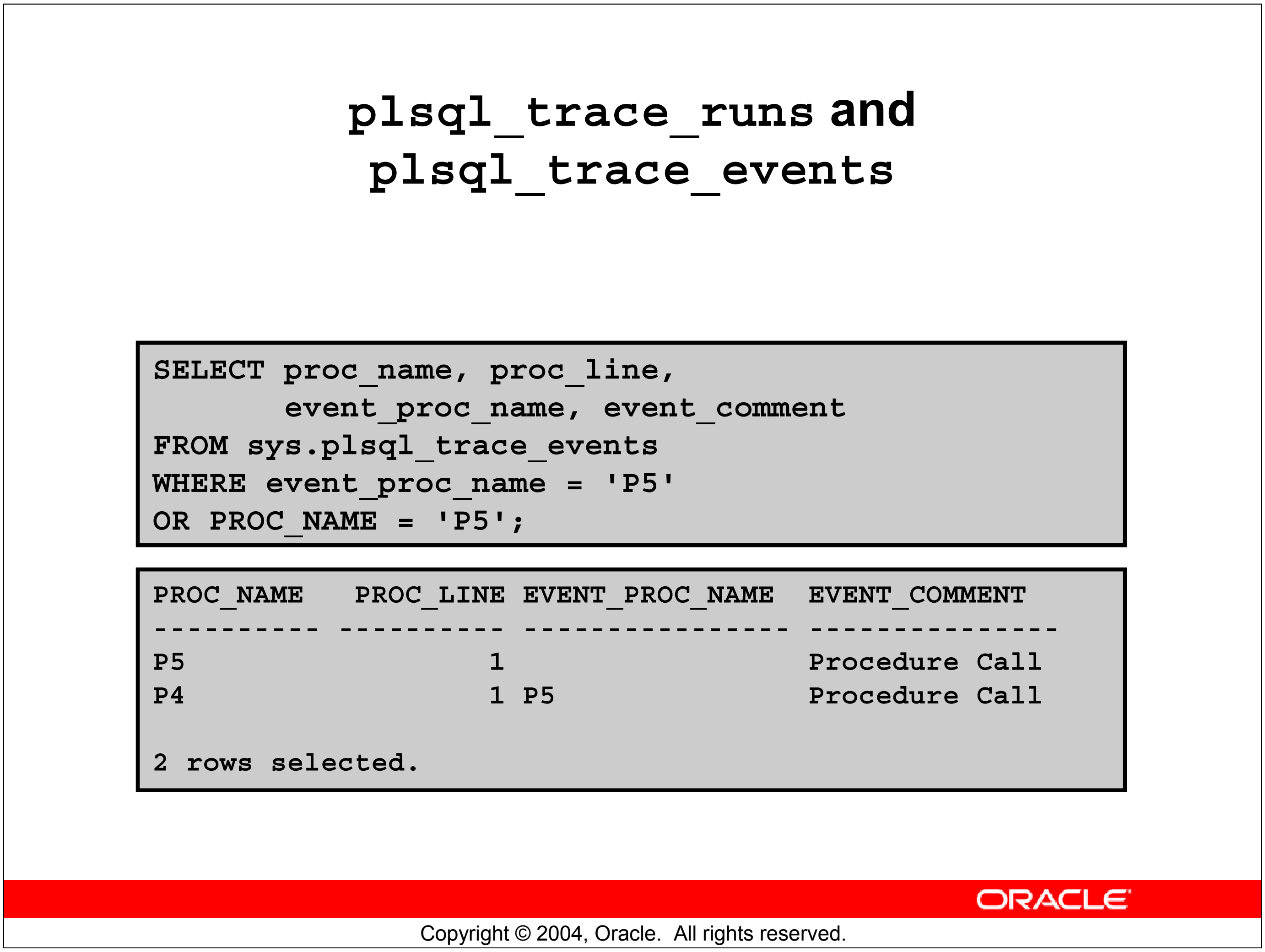
plsql_trace_runs and plsql_trace_events

- Trace information is written to the following dictionary views:
  - plsql_trace_runs dictionary view
  - plsql_trace_events dictionary view
- Run the tracetab.sql script to create the dictionary views.
- You need privileges to view the trace information in the dictionary views.

The plsql_trace_runs and plsql_trace_events Dictionary Views

All trace information is written to the dictionary views plsql_trace_runs and plsql_trace_events. These views are created (typically by a DBA) by running the tracetab.sql script. After the script is run, you need the SELECT privilege to view information from these dictionary views.

Note: With the Oracle release 8.1.6 and later, the trace information is written to the dictionary views. Prior to release 8.1.6, trace files were generated and trace information was written to the file. The location of this file is determined by the USER_DUMP_DEST initialization parameter. A file with a .trc extension is generated during the tracing.



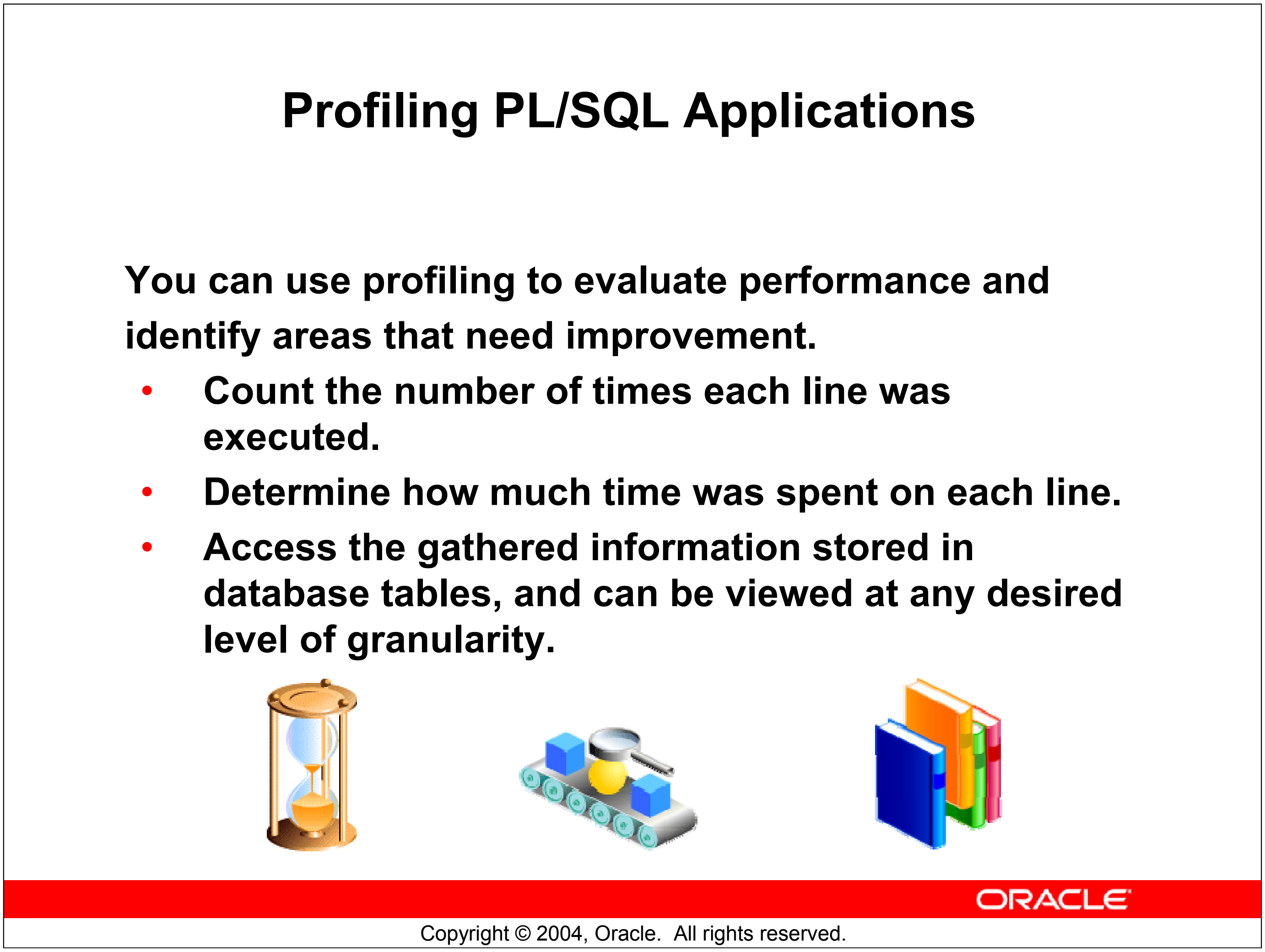
plsql_trace_runs and plsql_trace_events

```
SELECT proc_name, proc_line,
       event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_proc_name = 'P5'
OR PROC_NAME = 'P5';
```

```
PROC_NAME   PROC_LINE EVENT_PROC_NAME  EVENT_COMMENT
---------- ---------- ---------------- ---------------
P5                  1                  Procedure Call
P4                  1 P5               Procedure Call

2 rows selected.
```

Query the plsql_trace_runs and plsql_trace_events Views

Use the dictionary views plsql_trace_runs and plsql_trace_events to view the trace information generated by using the dbms_trace facility. plsql_trace_runs holds generic information about traced programs such as the date, time, owner, and name of the traced stored program. dbms_trace_events holds more specific information about the traced subprograms.



Profiling PL/SQL Applications

You can use profiling to evaluate performance and identify areas that need improvement.

- **Count the number of times each line was executed.**
- **Determine how much time was spent on each line.**
- **Access the gathered information stored in database tables, and can be viewed at any desired level of granularity.**

Profiling PL/SQL Applications

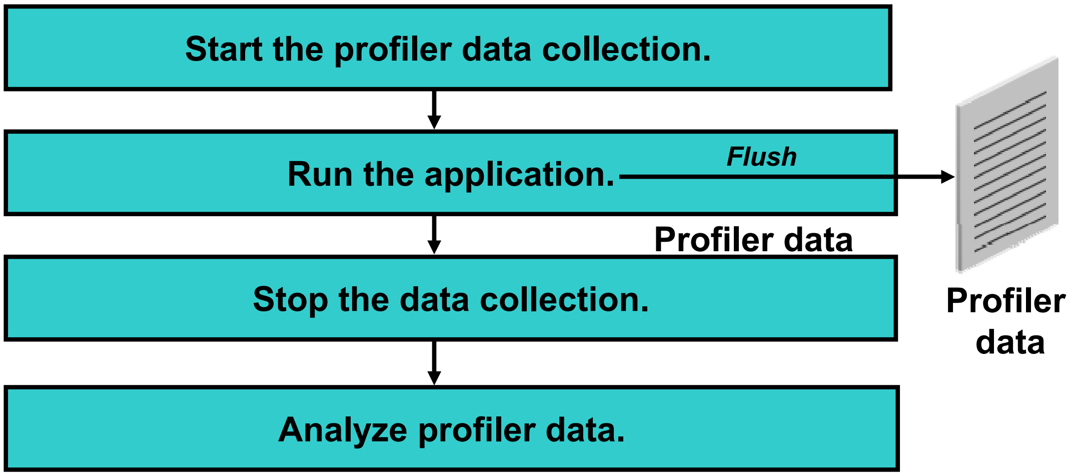
PL/SQL provides a tool called the Profiler that can be used to determine the execution time profile (or run-time behavior) of applications. The Profiler can be used to figure out which part of a particular application is running slowly. Such a tool is crucial in identifying performance bottlenecks. It can help you focus your efforts on improving the performance of only the relevant PL/SQL components, or, even better, the particular program segments where a lot of execution time is being spent.

The Profiler provides functions for gathering “profile” statistics, such as the total number of times each line was executed; time spent executing each line; and minimum and maximum duration spent on execution of a given line of code. For example, you can generate profiling information for all named library units used in a single session. This information is stored in database tables that can be queried later.

Third-party vendors can use the profiling API to build graphical, customizable tools. You can use Oracle 10*g*’s sample (demo) text-based report writer to gather meaningful data about their applications. The script is called `profrep.sql` and you can find it in your `Oracle_home/PLSQL/demo` directory. You can use the profiling API to analyze the performance of your PL/SQL applications and to locate bottlenecks. You can then use the profile information to appropriately tune your application.

Profiling PL/SQL Applications

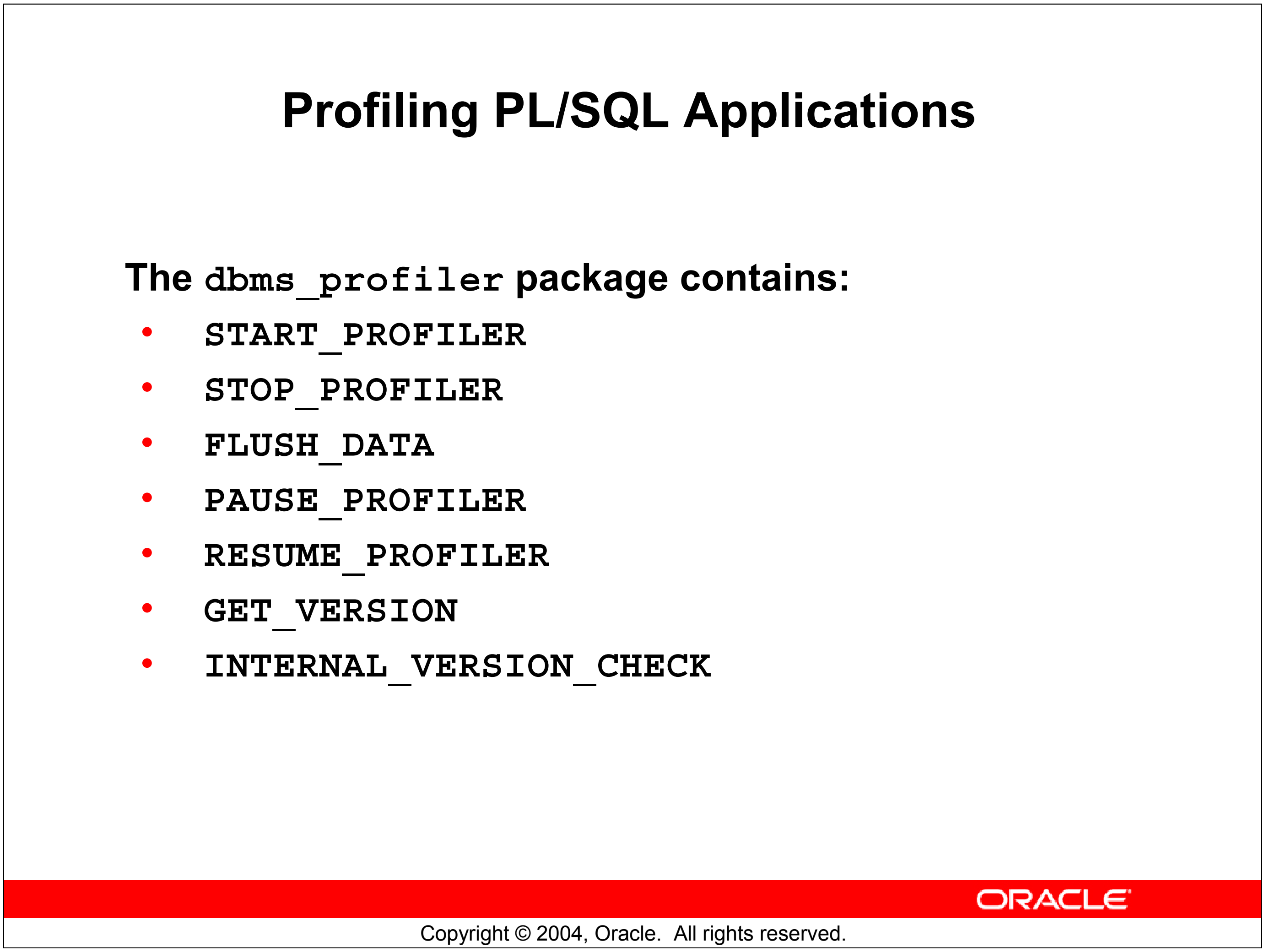
Use DBMS_PROFILER to profile existing PL/SQL applications and to identify performance bottlenecks.



Profiling PL/SQL Applications (continued)

The profiler API is implemented as a PL/SQL package, DBMS_PROFILER, which provides services for collecting and persistently storing PL/SQL profiler data.

Note: To set up profiling, two scripts need to be run. The profload.sql script is run under SYS. The proftab.sql script creates the profile dictionary tables. Run this script in the schema under which you want to collect profiling statistics.



Profiling PL/SQL Applications

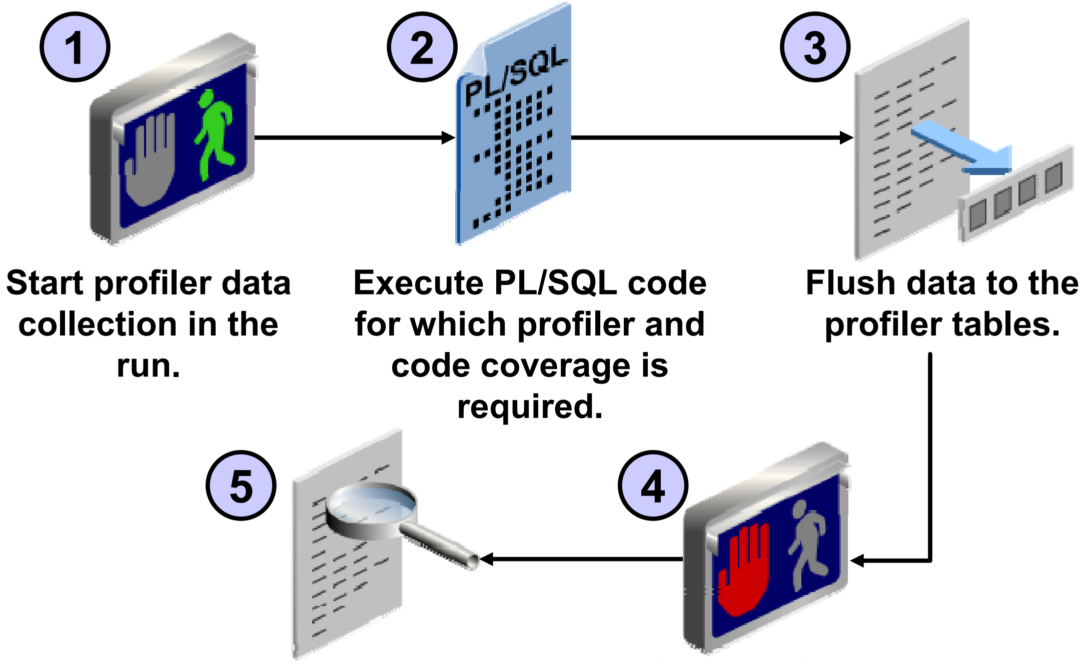
The `dbms_profiler` package contains:

- `START_PROFILER`
- `STOP_PROFILER`
- `FLUSH_DATA`
- `PAUSE_PROFILER`
- `RESUME_PROFILER`
- `GET_VERSION`
- `INTERNAL_VERSION_CHECK`

Profiling PL/SQL Applications (continued)

Routine	**Description**
`START_PROFILER` function	Starts profiler data collection in the user's session
`STOP_PROFILER` function	Stops profiler data collection in the user's session
`FLUSH_DATA` function	Flushes profiler data collected in the user's session
`PAUSE_PROFILER` function	Pauses profiler data collection
`RESUME_PROFILER` function	Resumes profiler data collection
`GET_VERSION` procedure	Gets the version of this API
`INTERNAL_VERSION_CHECK` function	Verifies that this version of the `DBMS_PROFILER` package can work with the implementation in the database

Profiling PL/SQL: Steps



Start profiler data collection in the run.

Execute PL/SQL code for which profiler and code coverage is required.

Flush data to the profiler tables.

Stop profiler data collection.

Analyze the data collected.

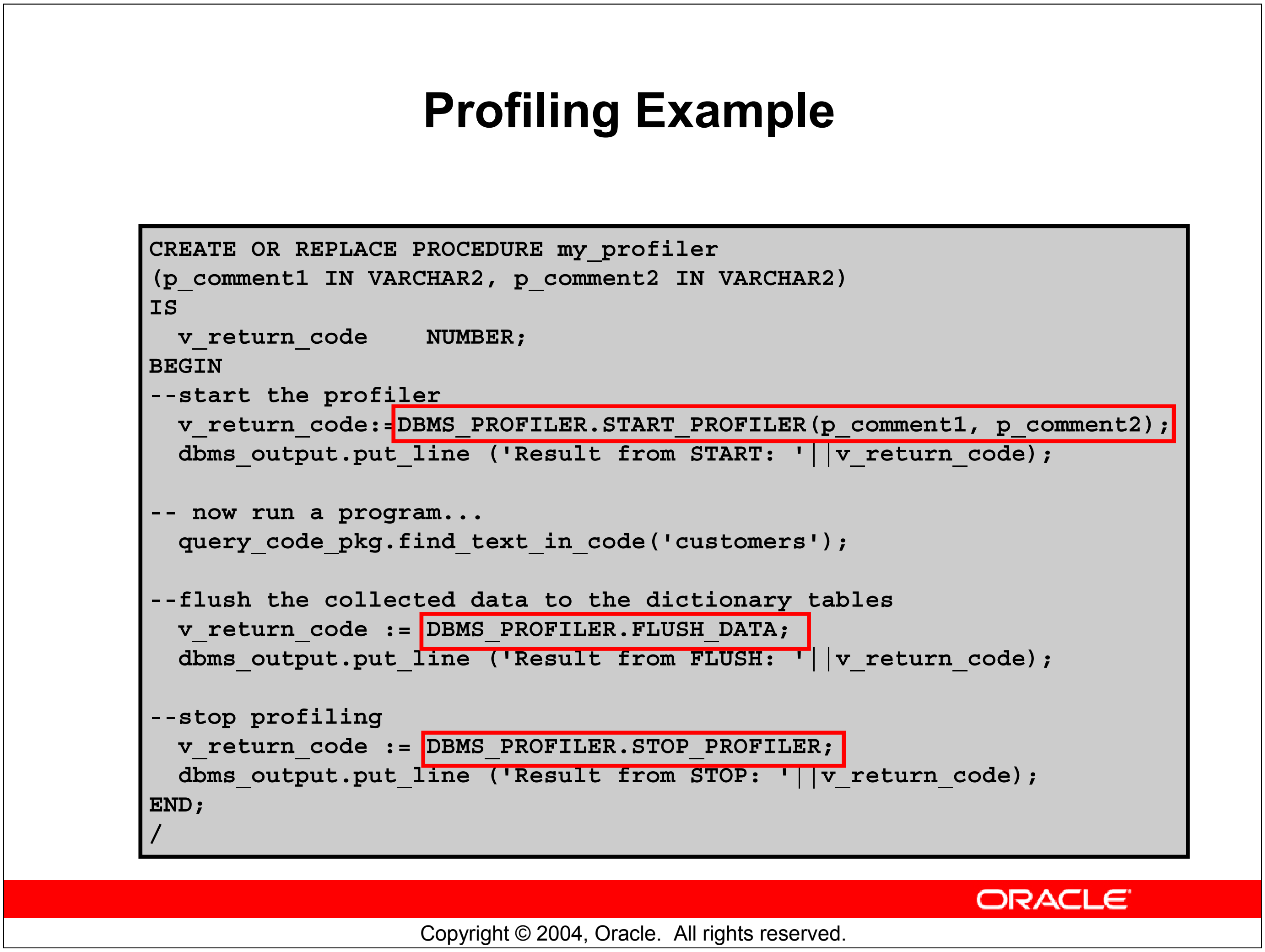
Steps to Profile PL/SQL Code

To profile PL/SQL code by using the `dbms_profiler` package, perform the following steps:

1. Start the profiler data collection by using `dbms_profiler.start_run`.
2. Execute the application that you are benchmarking.
3. Flush the data collected to the profiler tables by using `dbms_profiler.flush_data`.
4. Stop the profiler data collection by using `dbms_profiler.stop_run`.

Read and interpret the profiler information in the profiler tables:

- `PLSQL_PROFILER_RUNS`
- `PLSQL_PROFILER_UNITS`
- `PLSQL_PROFILER_DATA`



Profiling Example

```
CREATE OR REPLACE PROCEDURE my_profiler
(p_comment1 IN VARCHAR2, p_comment2 IN VARCHAR2)
IS
  v_return_code    NUMBER;
BEGIN
--start the profiler
  v_return_code:=DBMS_PROFILER.START_PROFILER(p_comment1, p_comment2);
  dbms_output.put_line ('Result from START: '||v_return_code);

-- now run a program...
  query_code_pkg.find_text_in_code('customers');

--flush the collected data to the dictionary tables
  v_return_code := DBMS_PROFILER.FLUSH_DATA;
  dbms_output.put_line ('Result from FLUSH: '||v_return_code);

--stop profiling
  v_return_code := DBMS_PROFILER.STOP_PROFILER;
  dbms_output.put_line ('Result from STOP: '||v_return_code);
END;
/
```

Running the Profiler

The `my_profiler` sample procedure shown starts the profiler, runs an application, flushes the data collected from the profiler to the dictionary tables, and stops the profiler. The functions `start_profiler`, `flush_data`, and `stop_profiler` return a numeric value indicating whether the function ran successfully. A return value of 0 indicates success.

Return Code	Meaning
0	Function ran successfully.
1	A subprogram was called with an incorrect parameter.
2	Data flush operation failed. Check whether the profiler tables have been created, are accessible, and that there is adequate space.
-1	There is a mismatch between package and database implementation.

`start_profiler` accepts two run comments as parameters. These two run comments default to the `sysdate` and null if they are not specified.

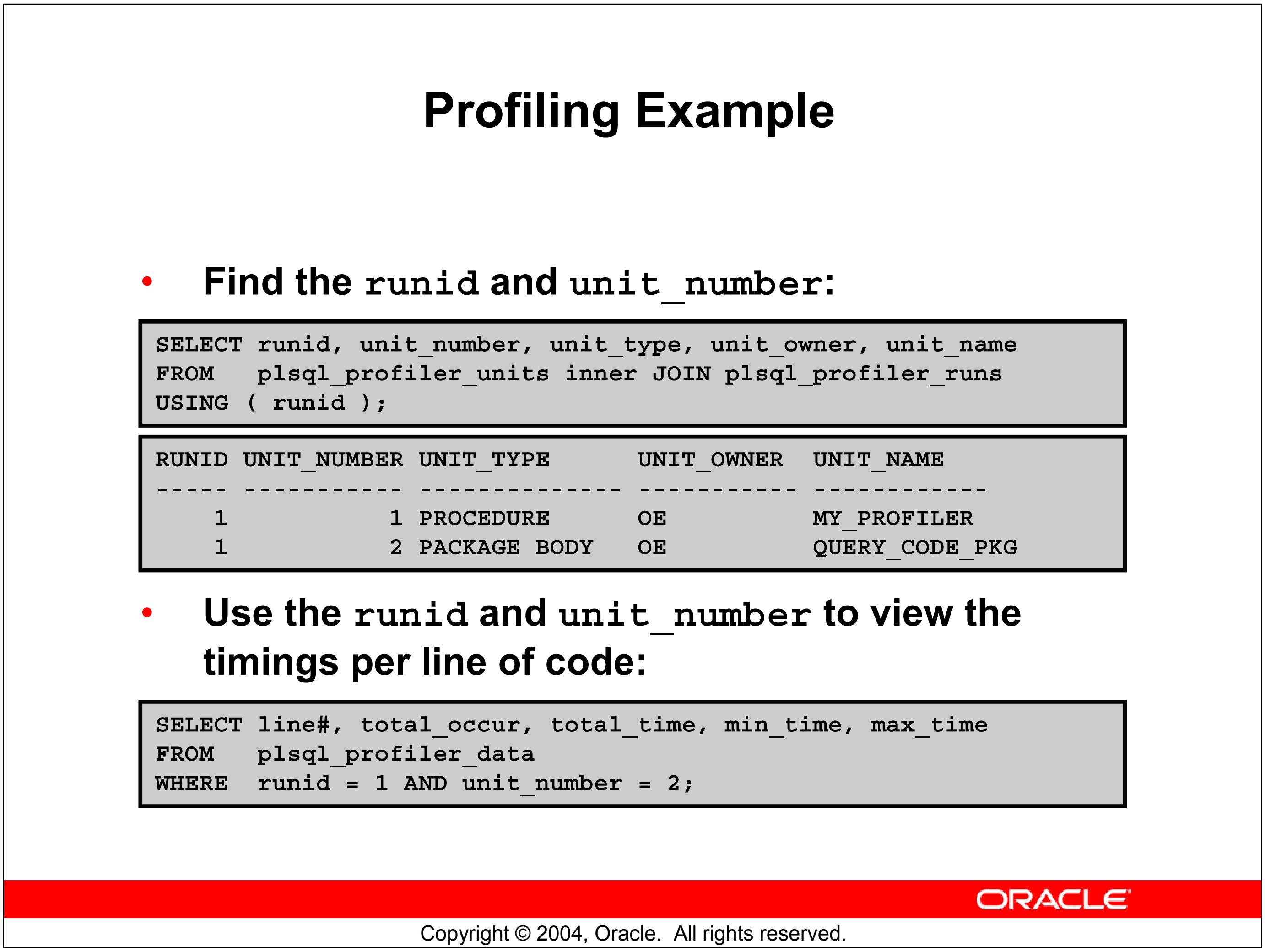
Profiling Example

```
EXECUTE my_profiler('Benchmark: 1', 'This is the first run!')
Result from START: 0
...
Result from FLUSH: 0
Result from STOP: 0

PL/SQL procedure successfully completed.
```

```
SELECT runid, run_owner, run_date, run_comment,
       run_comment1, run_total_time
FROM   plsql_profiler_runs;

  RUNID    RUN_OWNER  RUN_DATE  RUN_COMMENT  RUN_COMMEN RUN_TOTAL_TIME
---------- ---------- --------- ------------ ---------- --------------
         1 OE         23-MAY-04 Benchmark: 1 This is th     7.2632E+10
                                             e first ru
                                             n!
```

Examining the Results

The code shown in the slide shows some basic statistics. The query retrieves the `RUNID`, which can be used to find more information.



Profiling Example

- Find the `runid` and `unit_number`:

```
SELECT runid, unit_number, unit_type, unit_owner, unit_name
FROM   plsql_profiler_units inner JOIN plsql_profiler_runs
USING ( runid );
```

```
RUNID UNIT_NUMBER UNIT_TYPE      UNIT_OWNER  UNIT_NAME
----- ----------- -------------- ----------- ------------
    1           1 PROCEDURE      OE          MY_PROFILER
    1           2 PACKAGE BODY   OE          QUERY_CODE_PKG
```

- Use the `runid` and `unit_number` to view the timings per line of code:

```
SELECT line#, total_occur, total_time, min_time, max_time
FROM   plsql_profiler_data
WHERE  runid = 1 AND unit_number = 2;
```

Profiling Example

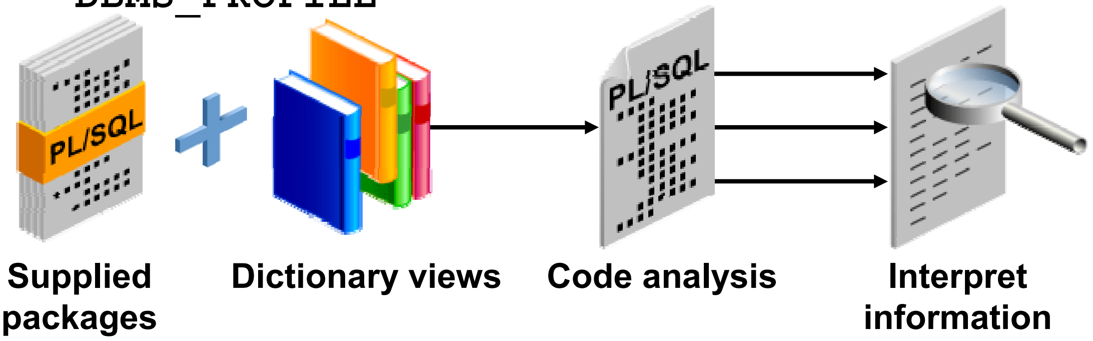
Query from the `PLSQL_PROFILER_DATA` table to view the timings per line of code executed.

```
     LINE# TOTAL_OCCUR TOTAL_TIME   MIN_TIME   MAX_TIME
---------- ----------- ---------- ---------- ----------
         8           1  225494518  225494518  225494518
        12           1    2948418    2948418    2948418
        13           0          0          0          0
        14           1     553980     553980     553980
        15           0          0          0          0
        16           1     703999     703999     703999
        17           0          0          0          0
        19           1    1036723    1036723    1036723
        21           1  844140290  844140290  844140290
        24           1    2911542    2911542    2911542
        25           1     317638     317638     317638

     LINE# TOTAL_OCCUR TOTAL_TIME   MIN_TIME   MAX_TIME
---------- ----------- ---------- ---------- ----------
        30           1    3710247    3710247    3710247
12 rows selected.
```

Summary

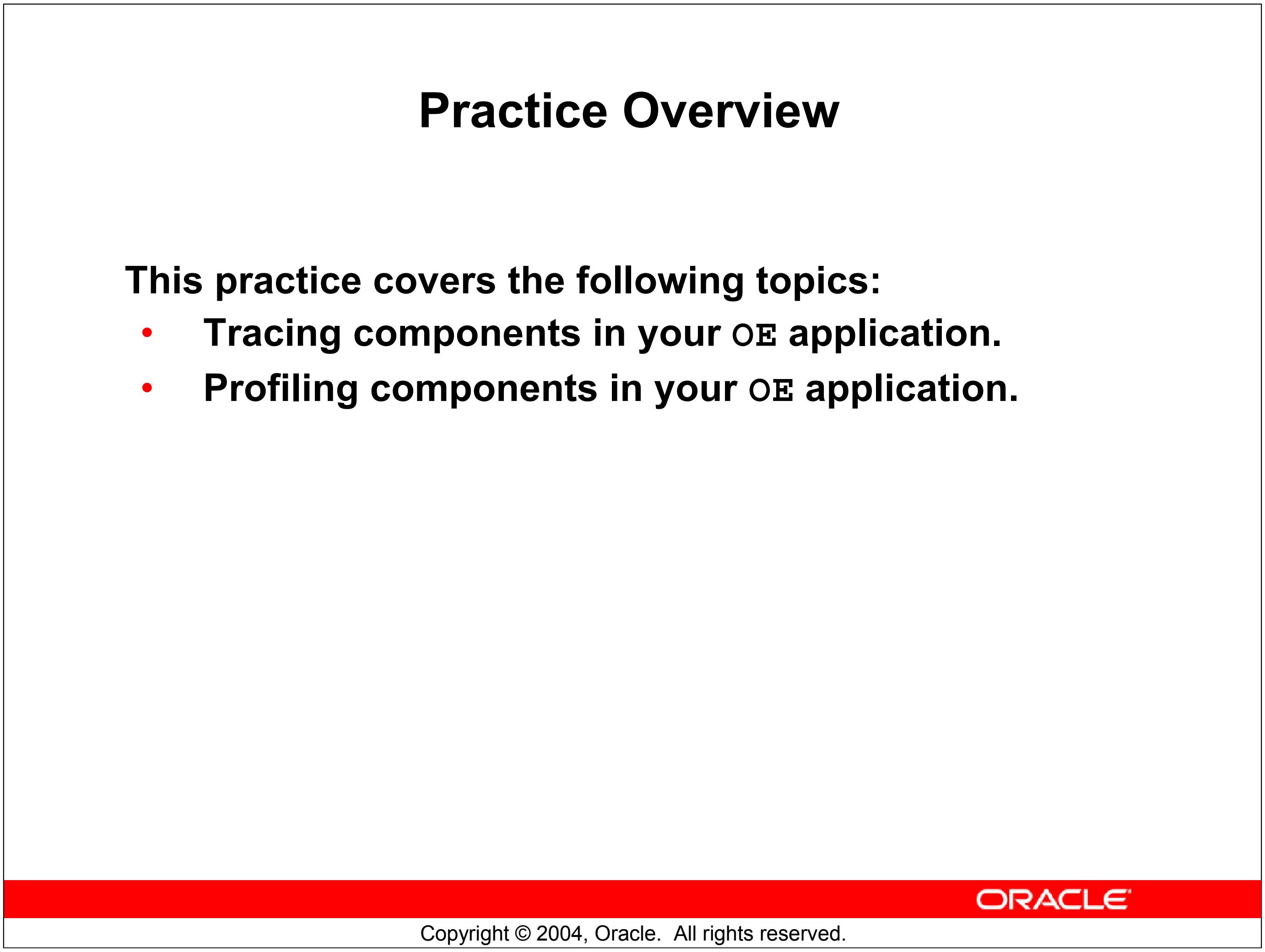
In this lesson, you should have learned how to:

- Use the dictionary views and supplied packages to get information about your PL/SQL application code
- Trace a PL/SQL application by using `DBMS_TRACE`
- Profile a PL/SQL application by using `DBMS_PROFILE`



Summary

In this lesson, you learned how to use the dictionary views and supplied PL/SQL packages to analyze your PL/SQL applications.



Practice Overview

This practice covers the following topics:

- Tracing components in your `OE` application.
- Profiling components in your `OE` application.

Copyright © 2004, Oracle. All rights reserved.

Practice Overview

Using the `OE` application that you have created, write code to analyze your application.

- Trace components in your `OE` application
- Profile components in your `OE` application

For detailed instructions on performing this practice, see Appendix A, “Practice Solutions.”

Practice 8

In this exercise, you will profile the CREDIT_CARD_PKG package created in an earlier lesson.

1. Run the lab_08_01.sql script to create the CREDIT_CARD_PKG package.

2. Run the proftab.sql script to create the profile tables under your schema.

3. Create a MY_PROFILER procedure to:
   - Start the profiler
   - Run the application
     ```
     EXECUTE credit_card_pkg.update_card_info -
               (130, 'AM EX', 121212121212)
     ```
   - Flush the profiler data
   - Stop the profiler

4. Execute the MY_PROFILER procedure.

5. Analyze the results of profiling in the PLSQL_PROFILER tables.

   In this exercise, you will trace the CREDIT_CARD_PKG package.

6. Enable the CREDIT_CARD_PKG for tracing by using the ALTER statement with the COMPILE DEBUG option.

7. Start the trace session and trace all calls.

8. Run the credit_card_pkg.update_card_info procedure with the following data:
   ```
    EXECUTE credit_card_pkg.update_card_info -
              (135, 'DC', 987654321)
   ```
9. Disable tracing.

10. Examine the trace information by querying the trace tables.

```
PROC_NAME          PROC_LINE EVENT_PROC_NAME    EVENT_COMMENT
----------------- ---------- ------------------ ---------------------
CUST_CARD_INFO             4 UPDATE_CARD_INFO   Procedure Call
                             CUST_CARD_INFO     PL/SQL Internal Call
UPDATE_CARD_INFO          31 CUST_CARD_INFO     Return from procedure
                                                 call
                           1 UPDATE_CARD_INFO   Return from procedure
                                                 call
```