12

Understanding and Influencing the PL/SQL Compiler

ORACLE

Objectives

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

- **Describe native and interpreted compilations**
- **List the features of native compilation**
- **Switch between native and interpreted compilations**
- **Set parameters that influence PL/SQL compilation**
- **Query data dictionary views on how PL/SQL code is compiled**
- **Use the compiler warning mechanism and the `DBMS_WARNING` package to implement compiler warnings**

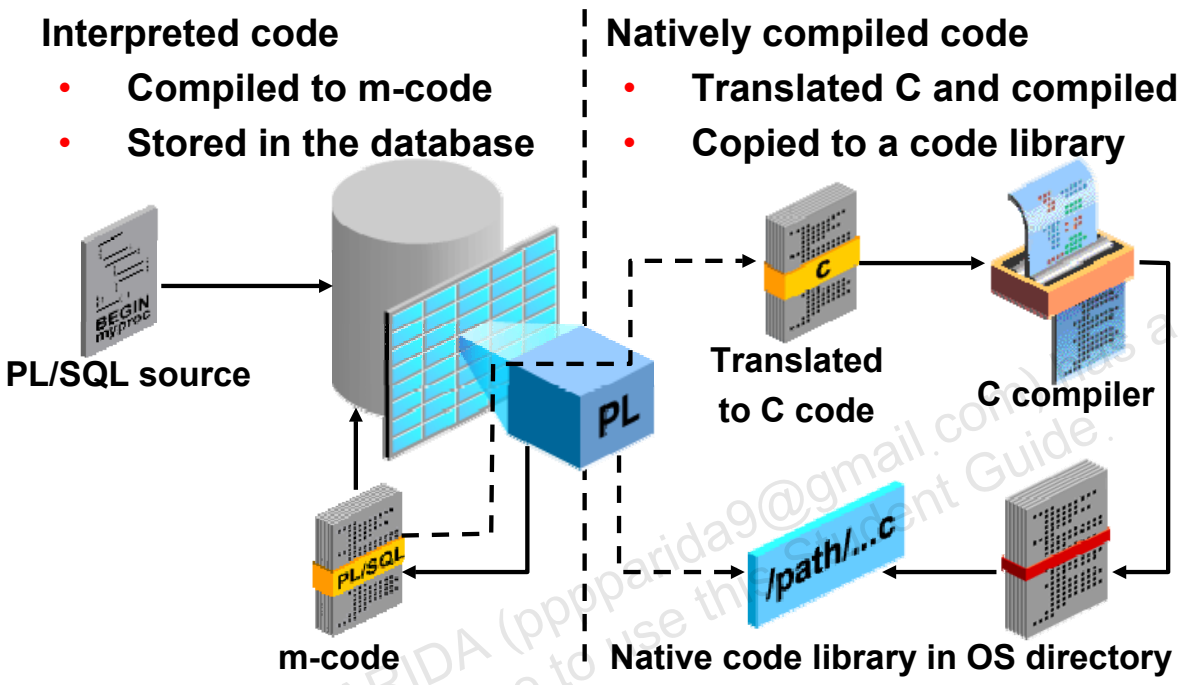
ORACLE

Lesson Aim

In this lesson, you learn to distinguish between native and interpreted compilation of PL/SQL code. The lesson discusses how to use native compilation, which is the default, for Oracle Database 10*g* with the benefit of having faster execution time for your PL/SQL code.

You also learn how to influence the compiler settings by setting variable session parameters, or using the programmatic interface provided by the `DBMS_WARNING` package. The lesson covers query compilation settings using the `USER_STORED_SETTINGS` and `USER_PLSQL_OBJECTS` data dictionary views.

Native and Interpreted Compilation



Native and Interpreted Compilation

As depicted in the slide, on the left of the vertical dotted line, a program unit processed as interpreted PL/SQL is compiled into machine-readable code (m-code), which is stored in the database and interpreted at run time.

On the right of the vertical dotted line, the PL/SQL source is subjected to native compilation, where the PL/SQL statements are compiled to m-code that is translated into C code. The m-code is not retained. The C code is compiled with the usual C compiler and linked to the Oracle process using native machine code library. The code library is stored in the database but copied to a specified directory path in the operating system, from which it is loaded at run time. Native code bypasses the typical run-time interpretation of code.

Note: Native compilation cannot do much to speed up SQL statements called from PL/SQL, but it is most effective for computation-intensive PL/SQL procedures that do not spend most of their time executing SQL.

You can natively compile both the supplied Oracle packages and your own PL/SQL code. Compiling all PL/SQL code in the database means that you see the speedup in your own code and all the built-in PL/SQL packages. If you decide that you will have significant performance gains in database operations using PL/SQL native compilation, Oracle recommends that you compile the whole database using the `NATIVE` setting.

Features and Benefits of Native Compilation

Native compilation:

- **Uses a generic `makefile` that uses the following operating system software:**
  - **C compiler**
  - **Linker**
  - **Make utility**
- **Generates shared libraries that are copied to the file system and loaded at run time**
- **Provides better performance (up to 30% faster than interpreted code) for computation-intensive procedural operations**

ORACLE

Features and Benefits of Native Compilation

The PL/SQL native compilation process makes use of a `makefile`, called `spnc_makefile.mk`, located in the `$ORACLE_HOME/plsql` directory. The `makefile` is processed by the Make utility that invokes the C compiler, which is the linker on the supported operating system, to compile and link the resulting C code into shared libraries. The shared libraries are stored inside the database and are copied to the file system. At run time, the shared libraries are loaded and run when the PL/SQL subprogram is invoked.

In accordance with Optimal Flexible Architecture (OFA) recommendations, the shared libraries should be stored near the data files. C code runs faster than PL/SQL, but it takes longer to compile than m-code. PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations.

Examples of such operations are data warehouse applications and applications with extensive server-side transformations of data for display. In such cases, expect speed increases of up to 30%.

Considerations When Using Native Compilation

Consider the following:

- **Debugging tools for PL/SQL cannot debug natively compiled code.**
- **Natively compiled code is slower to compile than interpreted code.**
- **Large amounts of natively compiled subprograms can affect performance due to operating system–imposed limitations when handling shared libraries. OS directory limitations can be managed by setting database initialization parameters:**
  - **`PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` and**
  - **`PLSQL_NATIVE_LIBRARY_DIR`**

ORACLE

Limitations of Native Compilation

As stated, the key benefit of natively compiled code is faster execution, particularly for computationally intensive PL/SQL code, as much as 30% more. Consider that:

- Debugging tools for PL/SQL do not handle procedures compiled for native execution. Therefore, use interpreted compilation in development environments, and natively compile the code in a production environment.
- The compilation time increases when using native compilation because of the requirement to translate the PL/SQL statement to its C equivalent and execute the Make utility to invoke the C compiler and linker for generating the resulting compiled code library.
- If many procedures and packages (more than 5,000) are compiled for native execution, a large number of shared objects in a single directory may affect performance. The operating system directory limitations can be managed by automatically distributing libraries across several subdirectories. To do this, perform the following tasks before natively compiling the PL/SQL code:
  - Set the `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` database initialization parameter to a large value, such as 1,000, before creating the database or compiling the PL/SQL packages or procedures.
  - Create `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` subdirectories in the path specified in the `PLSQL_NATIVE_LIBRARY_DIR` initialization parameter.

Parameters Influencing Compilation

System parameters are set in the `init`*`SID`*`.ora` file or by using the `SPFILE`:

```
PLSQL_NATIVE_LIBRARY_DIR = full-directory-path-name
PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT = count
```

System or session parameters

```
PLSQL_COMPILER_FLAGS = 'NATIVE' or 'INTERPRETED'
```


ORACLE

Parameters Influencing Compilation

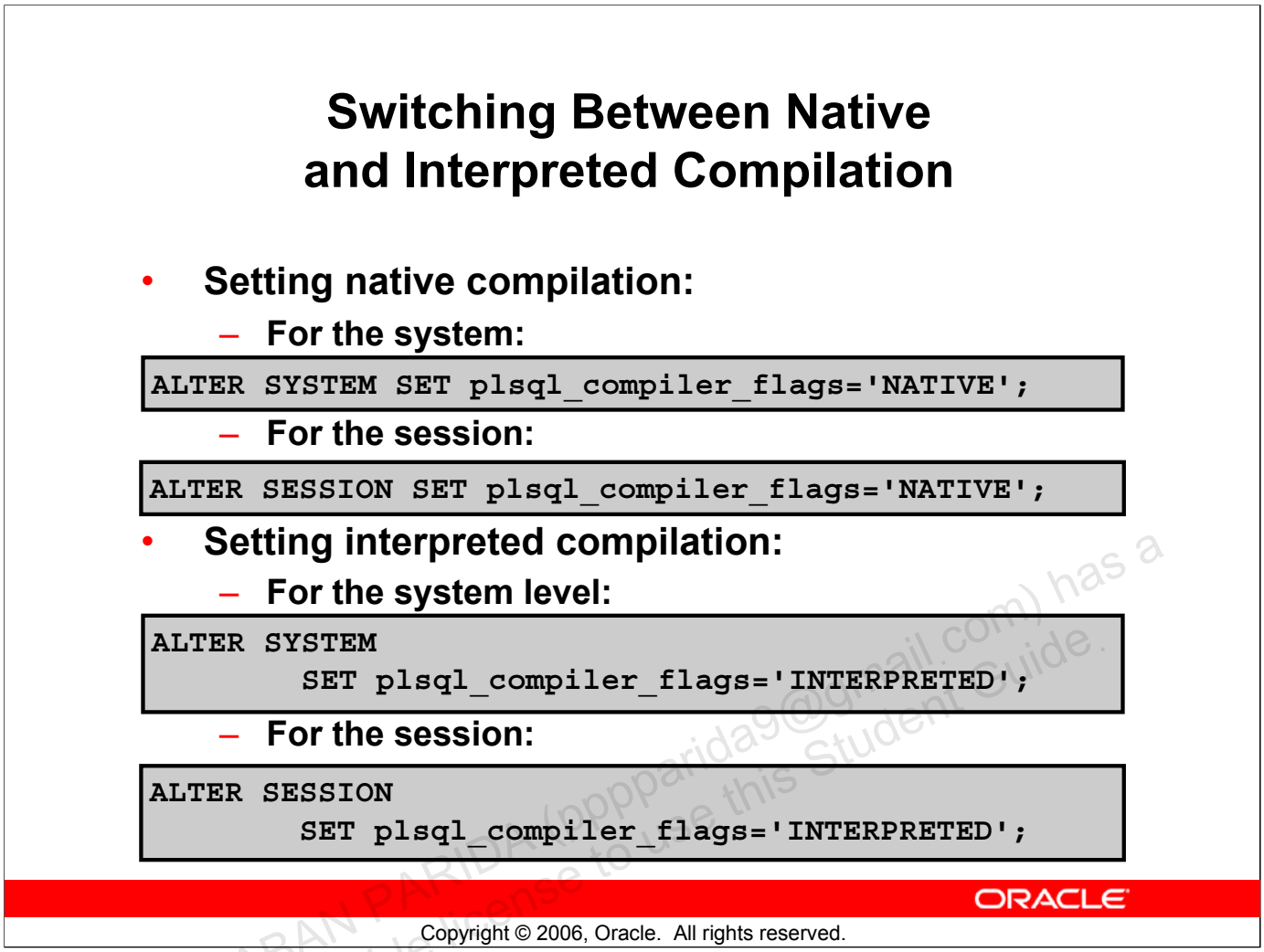
In all circumstances, whether you intend to compile a database as `NATIVE` or you intend to compile individual PL/SQL units at the session level, you must set all required parameters.

The system parameters are set in the `init`*`SID`*`.ora` file by using the `SPFILE` mechanism. Two parameters that are set as system-level parameters are the following:

- The `PLSQL_NATIVE_LIBRARY_DIR` value, which specifies the full path and directory name used to store the shared libraries that contain natively compiled PL/SQL code
- The `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` value, which specifies the number of subdirectories in the directory specified by the `PLSQL_NATIVE_LIBRARY_DIR` parameter. Use a script to create directories with consistent names (for example, `d0`, `d1`, `d2`, and so on), and then the libraries are automatically distributed among these subdirectories by the PL/SQL compiler.

By default, PL/SQL program units are kept in one directory.

The `PLSQL_COMPILER_FLAGS` parameter can be set to a value of `NATIVE` or `INTERPRETED`, either as a database initialization for a systemwide default or for each session using an `ALTER SESSION` statement.



Switching Between Native and Interpreted Compilation

The `PLSQL_COMPILER_FLAGS` parameter determines whether PL/SQL code is natively compiled or interpreted, and determines whether debug information is included. The default setting is `INTERPRETED,NON_DEBUG`. To enable PL/SQL native compilation, you must set the value of `PLSQL_COMPILER_FLAGS` to `NATIVE`.

If you compile the whole database as `NATIVE`, then Oracle recommends that you set `PLSQL_COMPILER_FLAGS` at the system level.

To set compilation type at the system level (usually done by a DBA), execute the following statements:

```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE'
ALTER SYSTEM SET plsql_compiler_flags='INTERPRETED'
```

To set compilation type at the session level, execute one of the following statements:

```
ALTER SESSION SET plsql_compiler_flags='NATIVE'
ALTER SESSION SET plsql_compiler_flags='INTERPRETED'
```

Viewing Compilation Information in the Data Dictionary

Query information in the following views:

- `USER_STORED_SETTINGS`
- `USER_PLSQL_OBJECTS`

Example:

```
SELECT param_value
FROM   user_stored_settings
WHERE  param_name = 'plsql_compiler_flags'
  AND  object_name = 'GET_EMPLOYEES';
```

Note: The `PARAM_VALUE` column has a value of `NATIVE` for procedures that are compiled for native execution; otherwise, it has a value of `INTERPRETED`.

ORACLE

Viewing Compilation Information in the Data Dictionary

To check whether an existing procedure is compiled for native execution or not, you can query the following data dictionary views:

```
[USER | ALL | DBA]_STORED_SETTINGS
[USER | ALL | DBA ]_PLSQL_OBJECTS
```

The example in the slide shows how you can check the status of the procedure called `GET_EMPLOYEES`. The `PARAM_VALUE` column has a value of `NATIVE` for procedures that are compiled for native execution; otherwise, it has a value of `INTERPRETED`.

After procedures are natively compiled and turned into shared libraries, they are automatically linked into the Oracle process. You do not need to restart the database, or move the shared libraries to a different location. You can call back and forth between stored procedures, whether they are all compiled interpreted (the default), all compiled for native execution, or a mixture of both.

Because the `PLSQL_COMPILER_FLAGS` setting is stored inside the library unit for each procedure, the procedures compiled for native execution are compiled the same way when the procedure is recompiled automatically after being invalidated, such as when a table that it depends on is re-created.

Using Native Compilation

To enable native compilation, perform the following steps:

1. **Edit the supplied `makefile` and enter appropriate paths and other values for your system.**
2. **Set the `PLSQL_COMPILER_FLAGS` parameter (at system or session level) to the value `NATIVE`. The default is `INTERPRETED`.**
3. **Compile the procedures, functions, and packages.**
4. **Query the data dictionary to see that a procedure is compiled for native execution.**

ORACLE

Using Native Compilation

To enable native compilation, perform the following steps:

1. Check and edit the compiler, linker, utility paths, and other values, if required.
2. Set the `PLSQL_COMPILER_FLAGS` to `NATIVE`.
3. Compile the procedures, functions, and packages. Compiling can be done by:
   - Using the appropriate `ALTER PROCEDURE`, `ALTER FUNCTION`, or `ALTER PACKAGE` statements with the `COMPILE` option
   - Dropping the procedure and re-creating it
   - Running one of the SQL*Plus scripts that sets up a set of Oracle-supplied packages
   - Creating a database using a preconfigured initialization file with its `PLSQL_COMPILER_FLAGS` set to `NATIVE`
4. Confirm the compilation type using the appropriate data dictionary tables.

Note: Dependencies between database objects are handled in the same manner as in previous Oracle database versions. If an object on which a natively compiled PL/SQL program unit depends changes, then the PL/SQL module is invalidated. The next time the same program unit is executed, the RDBMS attempts to revalidate the module. When a module is recompiled as part of revalidation, it is compiled using the setting that was used the last time the module was compiled, and it is saved in the `*_STORED_SETTINGS` view.

Compiler Warning Infrastructure

The PL/SQL compiler in Oracle Database 10*g* has been enhanced to produce warnings for subprograms. Warning levels:

- **Can be set:**
  - **Declaratively with the `PLSQL_WARNINGS` initialization parameter**
  - **Programmatically using the `DBMS_WARNINGS` package**
- **Are arranged in three categories: severe, performance, and informational**
- **Can be enabled and disabled by category or a specific message**

Examples of warning messages:

SP2-0804: Procedure created with compilation warnings

PLW-07203: Parameter `'IO_TBL'` may benefit from use of the `NOCOPY` compiler hint.

ORACLE

Compiler Warning Infrastructure

The Oracle PL/SQL compiler can issue warnings when you compile subprograms that produce ambiguous results or use inefficient constructs. You can selectively enable and disable these warnings:

- Declaratively by setting the `PLSQL_WARNINGS` initialization parameter
- Programmatically using the `DBMS_WARNINGS` package

The warning level is arranged in the following categories: severe, performance, and informational. Warnings levels can be enabled or disabled by category or by a specific warning message number.

Benefits of Compiler Warnings

Using compiler warnings can help to:

- Make your programs more robust and avoid problems at run time
- Identify potential performance problems
- Indicate factors that produce undefined results

Note: You can enable checking for certain warning conditions when these conditions are not serious enough to produce an error and keep you from compiling a subprogram.

Setting Compiler Warning Levels

Set the PLSQL_WARNINGS initialization parameter to enable the database to issue warning messages.

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:SEVERE',
                  'DISABLE:INFORMATIONAL';
```

- **The PLSQL_WARNINGS combine a qualifier value (ENABLE, DISABLE, or ERROR) with a comma-separated list of message numbers, or with one of the following modifier values:**
  - **ALL, SEVERE, INFORMATIONAL, or PERFORMANCE**
- **Warning messages use a PLW prefix.**

  PLW-07203: Parameter 'IO_TBL' may benefit from use of the NOCOPY compiler hint.

Setting Compiler Warning Levels

The PLSQL_WARNINGS setting enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors. The PLSQL_WARNINGS parameter can be set for the system using the initialization file or the ALTER SYSTEM statement, or for the session using the ALTER SESSION statement as shown in the example in the slide. By default, the value is set to DISABLE:ALL.

The parameter value comprises a comma-separated list of quoted qualifier and modifier keywords, where the keywords are separated by colons. The qualifier values are:

- **ENABLE:** To enable a specific warning or a set of warnings
- **DISABLE:** To disable a specific warning or a set of warnings
- **ERROR:** To treat a specific warning or a set of warnings as errors

The modifier value ALL applies to all warning messages. SEVERE, INFORMATIONAL, and PERFORMANCE apply to messages in their own category, and an integer list for specific warning messages. For example:

```
PLSQL_WARNINGS='ENABLE:SEVERE','DISABLE:INFORMATIONAL';
PLSQL_WARNINGS='DISABLE:ALL';
PLSQL_WARNINGS='DISABLE:5000','ENABLE:5001','ERROR:5002';
PLSQL_WARNINGS='ENABLE:(5000,5001)','DISABLE:(6000)';
```

Guidelines for Using PLSQL_WARNINGS

The PLSQL_WARNINGS setting:

- **Can be set to DEFERRED at the system level**
- **Is stored with each compiled subprogram**
- **That is current for the session is used, by default, when recompiling with:**
  - **A CREATE OR REPLACE statement**
  - **An ALTER...COMPILE statement**
- **That is stored with the compiled subprogram is used when REUSE SETTINGS is specified when recompiling with an ALTER...COMPILE statement**

ORACLE

Guidelines for Using PLSQL_WARNINGS

As already stated, the PLSQL_WARNINGS parameter can be set at the session level or the system level. When setting it at the system level, you can include the value DEFERRED so that it applies to future sessions but not the current one.

The settings for the PLSQL_WARNINGS parameter are stored along with each compiled subprogram. If you recompile the subprogram with a CREATE OR REPLACE statement, the current settings for that session are used. If you recompile the subprogram with an ALTER...COMPILE statement, then the current session setting is used unless you specify the REUSE SETTINGS clause in the statement, which uses the original setting that is stored with the subprogram.

`DBMS_WARNING` Package

The `DBMS_WARNING` package provides a way to programmatically manipulate the behavior of current system or session PL/SQL warning settings. Using `DBMS_WARNING` subprograms, you can:

- **Query existing settings**
- **Modify the settings for specific requirements or restore original settings**
- **Delete the settings**

Example: Saving and restoring warning settings for a development environment that calls your code that compiles PL/SQL subprograms, and suppresses warnings due to business requirements

`DBMS_WARNING` Package

The `DBMS_WARNING` package provides a way to manipulate the behavior of PL/SQL warning messages, in particular, by reading and changing the setting of the `PLSQL_WARNINGS` initialization parameter to control what kinds of warnings are suppressed, displayed, or treated as errors. This package provides the interface to query, modify, and delete current system or session settings.

The `DBMS_WARNINGS` package is valuable if you are writing a development environment that compiles PL/SQL subprograms. Using the package interface routines, you can control PL/SQL warning messages programmatically to suit your requirements.

Here is an example: Suppose you write some code to compile PL/SQL code. You know that the compiler will issue performance warnings when passing collection variables as `OUT` or `IN OUT` parameters without specifying the `NOCOPY` hint. The general environment that calls your compilation utility may or may not have appropriate warning level settings. In any case, your business rules indicate that the calling environment set must be preserved and that your compilation process should suppress the warnings. By calling subprograms in the `DBMS_WARNINGS` package, you can detect the current warning settings, change the setting to suit your business requirements, and restore the original settings when your processing has completed.

Using DBMS_WARNING Procedures

- **Package procedures change PL/SQL warnings:**

```
ADD_WARNING_SETTING_CAT(w_category,w_value,scope)
ADD_WARNING_SETTING_NUM(w_number,w_value,scope)
SET_WARNING_SETTING_STRING(w_value, scope)
```

  - **All parameters are IN parameters and have the VARCHAR2 data type. However, the w_number parameter is a NUMBER data type.**
  - **Parameter string values are not case sensitive.**
  - **The w_value parameters values are ENABLE, DISABLE, and ERROR.**
  - **The w_category values are ALL, INFORMATIONAL, SEVERE, and PERFORMANCE.**
  - **The scope value is either SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.**

ORACLE

Using DBMS_WARNING Procedures

The package procedures are the following:

- **ADD_WARNING_SETTING_CAT:** Modifies the current session or system warning settings of the warning_category previously supplied
- **ADD_WARNING_SETTING_NUM:** Modifies the current session or system warning settings of the warning_number previously supplied
- **SET_WARNING_SETTING_STRING:** Replaces previous settings with the new value

Using the SET_WARNING_SETTING_STRING, you can set one warning setting. If you have multiple warning settings, you should perform the following steps:

1. Call SET_WARNING_SETTING_STRING to set the initial warning setting string.
2. Call ADD_WARNING_SETTING_CAT (or ADD_WARNING_SETTING_NUM) repeatedly to add additional settings to the initial string.

Here is an example to establish the following warning setting string in the current session:

```
ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE
```

Execute the following two lines of code:

```
dbms_warning.set_warning_setting_string('ENABLE:ALL','session');
dbms_warning.add_warning_setting_cat('PERFORMANCE','disable',
                                     'session');
```

Using DBMS_WARNING Functions

- **Package functions read PL/SQL warnings:**

```
GET_CATEGORY(w_number) RETURN VARCHAR2
GET_WARNING_SETTING_CAT(w_category)RETURN VARCHAR2
GET_WARNING_SETTING_NUM(w_number) RETURN VARCHAR2
GET_WARNING_SETTING_STRING RETURN VARCHAR2
```

  - **GET_CATEGORY returns a value of ALL, INFORMATIONAL, SEVERE, or PERFORMANCE for a given message number.**
  - **GET_WARNING_SETTING_CAT returns ENABLE, DISABLE, or ERROR as the current warning value for a category name, and GET_WARNING_SETTING_NUM returns the value for a specific message number.**
  - **GET_WARNING_SETTING_STRING returns the entire warning string for the current session.**

ORACLE

Using DBMS_WARNING Functions

The following is a list of package functions:

- GET_CATEGORY returns the category name for the given message number.
- GET_WARNING_SETTING_CAT returns the current session warning setting for the specified category.
- GET_WARNING_SETTING_NUM returns the current session warning setting for the specified message number.
- GET_WARNING_SETTING_STRING returns the entire warning string for the current session.

To determine the current session warning settings, enter:

```
EXECUTE DBMS_OUTPUT.PUT_LINE( -
     DBMS_WARNING.GET_WARNING_SETTING_STRING);
```

To determine the category for warning message number PLW-07203, use:

```
EXECUTE DBMS_OUTPUT.PUT_LINE( -
     DBMS_WARNING.GET_CATEGORY(7203))
```

The result string should be PERFORMANCE.

Note: The message numbers must be specified as positive integers because the data type for the GET_CATEGORY parameter is PLS_INTEGER (allowing positive integer values).

Using DBMS_WARNING: Example

Consider the following scenario:
Save current warning settings, disable warnings for the PERFORMANCE category, compile a PL/SQL package, and restore the original warning setting.

```
CREATE PROCEDURE compile(pkg_name VARCHAR2) IS
  warn_value VARCHAR2(200);
  compile_stmt VARCHAR2(200) :=
    'ALTER PACKAGE '|| pkg_name ||' COMPILE';
BEGIN
  warn_value :=   -- Save current settings
      DBMS_WARNING.GET_WARNING_SETTING_STRING;
  DBMS_WARNING.ADD_WARNING_SETTING_CAT( -- change
      'PERFORMANCE', 'DISABLE', 'SESSION');
  EXECUTE IMMEDIATE compile_stmt;
  DBMS_WARNING.SET_WARNING_SETTING_STRING(--restore
      warn_value, 'SESSION');
END;
```

ORACLE

Using DBMS_WARNING: Example

In the slide, the example of the `compile` procedure is designed to compile a named PL/SQL package. The business rules require the following:

- Warnings in the performance category are suppressed.
- The calling environment's warning settings must be restored after the compilation is performed.

The code does not know or care about what the calling environment warning settings are; it simply uses the `DBMS_WARNING.GET_WARNING_SETTING_STRING` function to save the current setting.

This value is used to restore the calling environment setting using the `DBMS_WARNING.SET_WARNING_SETTING_STRING` procedure in the last line of the example code. Before compiling the package using Native Dynamic SQL, the `compile` procedure alters the current session warning level by disabling warnings for the `PERFORMANCE` category.

For example, the compiler will suppress warnings about PL/SQL parameters passed using `OUT` or `IN OUT` modes that do not specify the `NOCOPY` hint to gain better performance.

Using DBMS_WARNING: Example

To test the `compile` procedure, you can use the following script sequence in *i*SQL*Plus:

```
DECLARE
  PROCEDURE print(s VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(s);
  END;
BEGIN
  print('Warning settings before: '||
        DBMS_WARNING.GET_WARNING_SETTING_STRING);
  compile('my_package');
  print('Warning settings after: '||
        DBMS_WARNING.GET_WARNING_SETTING_STRING);
END;
/
SHOW ERRORS PACKAGE MY_PACKAGE
```

ORACLE

Using DBMS_WARNING: Example (continued)

The slide shows an anonymous block that is used to display the current warning settings for the session before compilation takes place, executes the compile procedure, and prints the current warning settings for the session again. The before and after values for the warning settings should be identical.

The last line containing the `SHOW ERRORS PACKAGE MY_PACKAGE` is used to verify whether the warning messages in the performance category are suppressed (that is, no performance-related warning messages are displayed).

To adequately test the `compile` procedure behavior, the `MY_PACKAGE` package should contain a subprogram with a collection (PL/SQL table) specified as an `OUT` or `IN OUT` argument without using the `NOCOPY` hint. Normally, with the `PERFORMANCE` category enabled, a compiler warning will be issued. Using the code examples shown in the last two slides, the warnings related to the `NOCOPY` hint are suppressed.

Summary

In this lesson, you should have learned how to:

- **Switch between native and interpreted compilations**
- **Set parameters that influence native compilation of PL/SQL programs**
- **Query data dictionary views that provide information on PL/SQL compilation settings**
- **Use the PL/SQL compiler warning mechanism:**
  - **Declaratively by setting the `PLSQL_WARNINGS` parameter**
  - **Programmatically using the `DBMS_WARNING` package**

ORACLE

Summary

The lesson covers details about how native and interpreted compilations work and how to use parameters that influence the way PL/SQL code is compiled.

The key recommendation is to enable native compilation by default, resulting in 30% faster performance (in some cases) for your PL/SQL logic. Benchmarks have shown that enabling native compilation in Oracle Database 10*g* results in twice the performance when compared to Oracle8*i* and Oracle9*i* databases, and as much as three times the performance of PL/SQL code executing in an Oracle8 database environment. For more information, refer to the Oracle white paper titled "PL/SQL Just Got Faster," by Bryn Llewellyn and Charles Wetherell, from the Oracle Technology Network (OTN) Web site at http://otn.oracle.com.

The lesson also covers the following two ways of influencing the new compiler warning system that was added to Oracle Database 10*g*:

- Setting the `PLSQL_WARNINGS` parameter
- Using the `DBMS_WARNING` package programmatic interface

Practice 12: Overview

This practice covers the following topics:

- **Enabling native compilation for your session and compiling a procedure**
- **Creating a subprogram to compile a PL/SQL procedure, function, or a package; suppressing warnings for the `PERFORMANCE` compiler warning category; and restoring the original session warning settings**
- **Executing the procedure to compile a PL/SQL package containing a procedure that uses a PL/SQL table as an `IN OUT` parameter without specifying the `NOCOPY` hint**

Practice 12: Overview

In this practice, you enable native compilation for your session and compile a procedure. You then create a subprogram to compile a PL/SQL procedure, function, or a package, and you suppress warnings for the `PERFORMANCE` compiler warning category. The procedure must restore the original session warning settings. You then execute the procedure to compile a PL/SQL package that you create, where the package contains a procedure with an `IN OUT` parameter without specifying the `NOCOPY` hint.

Practice 12

1. Alter the PLSQL_COMPILER_FLAGS parameter to enable native compilation for your session, and compile any subprogram that you have written.
   a. Execute the ALTER SESSION command to enable native compilation.
   b. Compile the EMPLOYEE_REPORT procedure. What occurs during compilation?
   c. Execute the EMPLOYEE_REPORT with the value 'UTL_FILE' as the first parameter, and 'native_salrepXX.txt' where XX is your student number.
   d. Switch compilation to use interpreted compilation.
2. In the COMPILE_PKG (from Practice 6), add an overloaded version of the procedure called MAKE, which will compile a named procedure, function, or package.
   a. In the specification, declare a MAKE procedure that accepts two string arguments, one for the name of the PL/SQL construct and the other for the type of PL/SQL program, such as PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY.
   b. In the body, write the MAKE procedure to call the DBMS_WARNINGS package to suppress the PERFORMANCE category. However, save the current compiler warning settings before you alter them. Then write an EXECUTE IMMEDIATE statement to compile the PL/SQL object using an appropriate ALTER...COMPILE statement with the supplied parameter values. Finally, restore the compiler warning settings that were in place for the calling environment before the procedure is invoked.
3. Write a new PL/SQL package called TEST_PKG containing a procedure called GET_EMPLOYEES that uses an IN OUT argument.
   a. In the specification, declare the GET_EMPLOYEES procedure with two parameters: an input parameter specifying a department ID, and an IN OUT parameter specifying a PL/SQL table of employee rows.
   Hint: You must declare a TYPE in the package specification for the PL/SQL table parameter's data type.
   b. In the package body, implement the GET_EMPLOYEES procedure to retrieve all the employee rows for a specified department into the PL/SQL table IN OUT parameter.
   Hint: Use the SELECT ... BULK COLLECT INTO syntax to simplify the code.
4. Use the ALTER SESSION statement to set the PLSQL_WARNINGS so that all compiler warning categories are enabled.
5. Recompile the TEST_PKG that you created two steps earlier (in Exercise 3). What compiler warnings are displayed, if any?
6. Write a PL/SQL anonymous block to compile the TEST_PKG package by using the overloaded COMPILE_PKG.MAKE procedure with two parameters. The anonymous block should display the current session warning string value before and after it invokes the COMPILE_PKG.MAKE procedure. Do you see any warning messages? Confirm your observations by executing the SHOW ERRORS PACKAGE command for the TEST_PKG.