



Oracle PL/SQL 19c

PL/SQL Fundamentals



Oracle PL/SQL 19c



Module 00: Introduction

- Day 1

- Module 01: Introduction to PL/SQL
- Module 02: Declaring Variables
- Module 03: Writing Executable Statements
- Module 04: Oracle DB – SQL Statements in PL/SQL
- Module 05: Control Structures
- Module 06: Composite Data Types

- Day 2

- Module 07: Explicit Cursors
- Module 08: Handling Exceptions
- Module 09: Creating Procedures
- Module 10: Creating Functions
- Module 11: Creating Packages
- Module 12: Working with Packages
- Module 13: Oracle Supplied Packages

- Day 3

- Module 14: Dynamic SQL
- Module 15: PL/SQL Design Considerations
- Module 16: Creating Triggers
- Module 17: Compound DDL & Event Triggers
- Module 18: PL/SQL Compiler
- Module 19: Managing PL/SQL Code
- Module 20: Managing Dependencies



Course Agenda

Your instructor will provide you with your credentials to Log-in to the virtual environment.

If you are taking this class remotely, you will be emailed your credentials and a how to data sheet on how to access the virtual environment.

This lesson introduces PL/SQL and the PL/SQL programming constructs. You also learn about the benefits of PL/SQL.

Oracle PL/SQL 19c

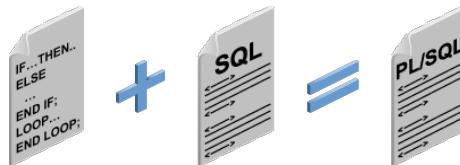
Module 01: Introduction to PL/SQL

Topics covered in this module:

- Explain the need for PL/SQL
- Explain the benefits of PL/SQL
- Identify the different types of PL/SQL blocks
- Output messages in PL/SQL

PL/SQL:

- Stands for “Procedural Language extension to SQL”
- Is Oracle Corporation’s standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL



9

Structured Query Language (SQL) is the primary language used to access and modify data in relational databases. There are only a few SQL commands, so you can easily learn and use them.

Consider an example:

```
SELECT first_name, department_id, salary FROM  
employees;
```

The preceding SQL statement is simple and straightforward. However, if you want to alter any data that is retrieved in a conditional manner, you soon encounter the limitations of SQL.

Consider a slightly modified problem statement: For every employee retrieved, check the department ID and salary. Depending on the department's performance and also the employee's salary, you may want to provide varying bonuses to the employees.

Looking at the problem, you know that you have to execute the preceding SQL statement, collect the data, and apply logic to the data.

One solution is to write a SQL statement for each department to give bonuses to the employees in that department. Remember that you also have to check the salary component before deciding the bonus amount. This makes it a little complicated.

A more effective solution might include conditional statements.
PL/SQL is designed to meet such requirements. It provides a
programming extension to the already-existing SQL.

PL/SQL:

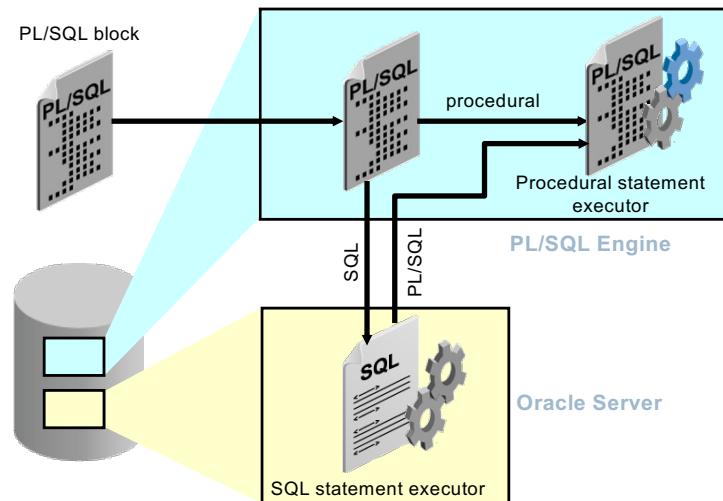
- Provides a block structure for executable units of code. Maintenance of code is made easier with such a well-defined structure.
- Provides procedural constructs such as:
 - Variables, constants, and data types
 - Control structures such as conditional statements and loops
 - Reusable program units that are written once and executed many times

10

PL/SQL defines a block structure for writing code. Maintaining and debugging code is made easier with such a structure because you can easily understand the flow and execution of the program unit.

PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, and object orientation. It brings state-of-the-art programming to the Oracle Server and toolset. PL/SQL provides all the procedural constructs that are available in any third-generation language (3GL).

PL/SQL Run-Time Architecture



11

The diagram in the slide shows a PL/SQL block being executed by the PL/SQL engine. The PL/SQL engine resides in:

The Oracle database for executing stored subprograms

The Oracle Forms client when you run client/server applications, or in the Oracle Application Server when you use Oracle Forms Services to run Forms on the Web

Irrespective of the PL/SQL run-time environment, the basic architecture remains the same. Therefore, all PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle Server processes. The SQL environment may also invoke the PL/SQL environment. For example, the PL/SQL environment is invoked when a PL/SQL function is used in a SELECT statement.

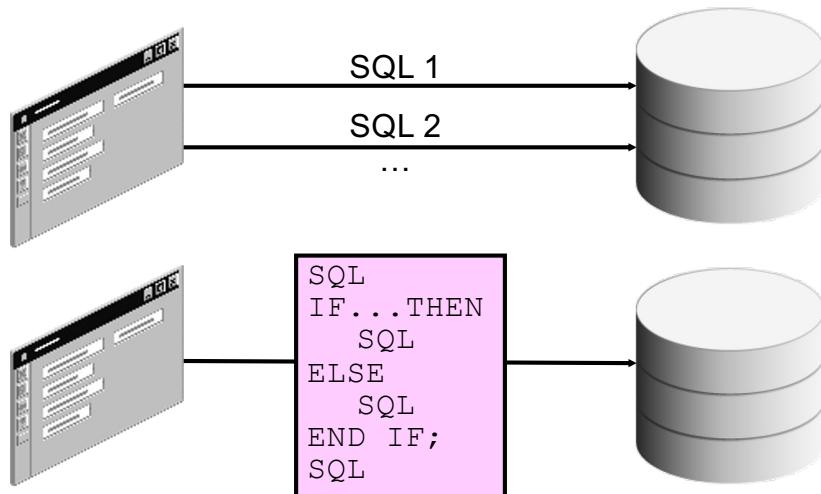
The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL

m-code instructions. When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the Oracle Server processes. The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent

statements in the PL/SQL block. The Oracle Forms PL/SQL engine runs in the client for the client/server implementation, and in the application server for the Forms Services implementation. In either case, SQL statements are typically sent over a network to an Oracle Server for processing.

Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance



12

tells the database server *what* to do. However, you cannot specify *how* to do it. PL/SQL integrates control statements and conditional statements with SQL, giving you better control of your SQL statements and their execution. Earlier in

Improved performance: Without PL/SQL, you would not be able to logically combine SQL statements as one unit. If you have designed an application that contains forms, you may have many different forms with fields in each form. When a form submits data, you may have to execute a number of SQL statements. SQL statements are sent to the database one at a time. This results in many network trips and one call to the database for each SQL statement, thereby increasing network traffic and reducing performance (especially in a client/server model).

With PL/SQL, you can combine all these SQL statements into a single program unit. The application can send the entire block to the database instead of sending the SQL statements one at a time. This significantly reduces the number of database calls. As the slide illustrates, if the application is SQL

intensive, you can use PL/SQL blocks to group SQL statements before sending them to the Oracle database server for execution.

Benefits of PL/SQL

- Modularized program development
- Integration with Oracle tools
- Portability
- Exception handling

13

You can group logically related statements within blocks.

You can nest blocks inside larger blocks to build powerful programs.

~~You can break your application into smaller modules. If you are~~

application into smaller, manageable, and logically related modules.

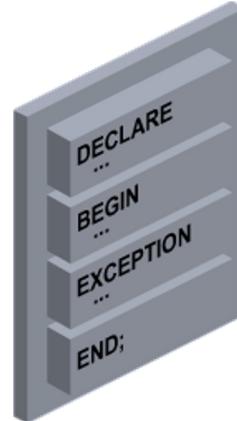
You can easily maintain and debug code.

In PL/SQL, modularization is implemented using procedures, functions, and packages, which are discussed in the lesson titled “Introducing Stored Procedures and Functions.”

Integration with tools: The PL/SQL engine is integrated in Oracle tools such as Oracle Forms and Oracle Reports. When you use these tools, the locally available PL/SQL engine processes the procedural statements; only the SQL statements are passed to the database.

PL/SQL Block Structure

- DECLARE (optional)
 - Variables, cursors, user-defined exceptions
- BEGIN (mandatory)
 - SQL statements
 - PL/SQL statements
- EXCEPTION (optional)
 - Actions to perform
when exceptions occur
- END; (mandatory)



14

Begin (required): The executable section begins with the keyword BEGIN. This section needs to have at least one statement. However,
the executable section of a PL/SQL block can include any number of

Exception handling (optional): The exception section is nested within the executable section. This section begins with the keyword EXCEPTION.

End (required): All PL/SQL blocks must conclude with an END statement. Observe that END is terminated with a semicolon.

Block Types

```
PROCEDURE name
IS
BEGIN
  --statements
[EXCEPTION]
END;
```

```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]
END;
```

```
[DECLARE]
BEGIN
  --statements
[EXCEPTION]
END;
```

1

Procedures

Functions

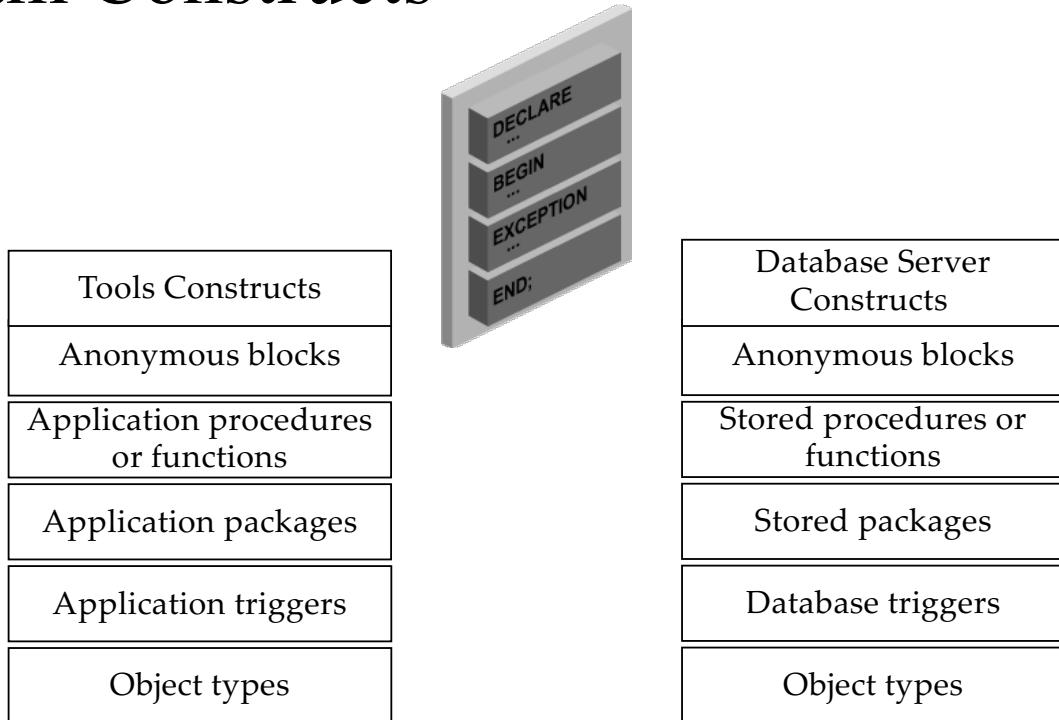
Anonymous blocks

Procedures: Procedures are named objects that contain SQL and/or PL/SQL statements.

Functions: Functions are named objects that contain SQL and/or PL/SQL statements. Unlike a procedure, a function returns a value of a specified data type.

Anonymous blocks: Anonymous blocks are unnamed blocks. They are declared inline at the point in an application where they are to be executed and are compiled each time the application is executed. These blocks are not stored in the database. They are passed to the PL/SQL engine for execution at run time. Triggers in Oracle Developer components consist of such blocks. If you want to execute the same block again, you have to rewrite the block. You cannot invoke or call the block that you wrote earlier because blocks are anonymous and do not exist after they are executed.

Program Constructs

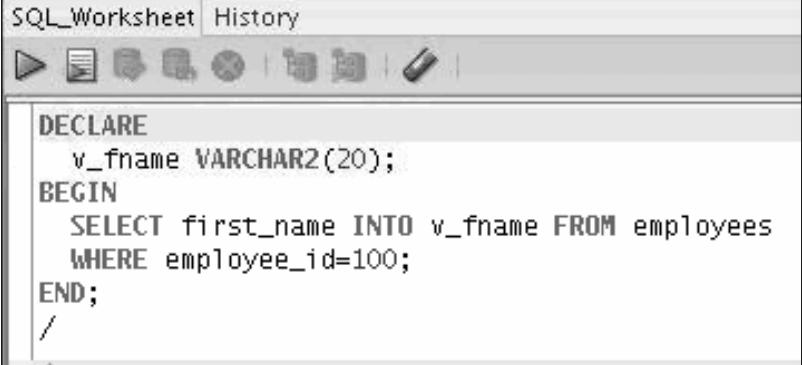


1

Program Construct	Description	Availability
Anonymous blocks	Unnamed PL/SQL blocks that are embedded within an application or are issued interactively	All PL/SQL environments
Application procedures or functions	Named PL/SQL blocks that are stored in an Oracle Forms Developer application or a shared library; can accept parameters and can be invoked repeatedly by name	Oracle Developer tools components (for example, Oracle Forms Developer, Oracle Reports)
Stored procedures or functions	Named PL/SQL blocks that are stored in the Oracle server; can accept parameters and can be invoked repeatedly by name	Oracle server or Oracle Developer tools
Packages (application or stored)	Named PL/SQL modules that group related procedures, functions, and identifiers	Oracle server and Oracle Developer tools components (for example, Oracle Forms Developer)

Examining an Anonymous Block

An anonymous block in the SQL Developer workspace:



The screenshot shows the SQL Worksheet window of Oracle SQL Developer. The title bar says "SQL Worksheet History". Below the title bar is a toolbar with various icons. The main area contains the following PL/SQL code:

```
DECLARE
    v_fname VARCHAR2(20);
BEGIN
    SELECT first_name INTO v_fname FROM employees
    WHERE employee_id=100;
END;
/
```

17

The example block has the declarative section and the executable section. You need not pay attention to the syntax of statements in the block; you learn the [syntax later in the course](#).

employee_id is 100, and stores it in a variable called v_fname.

Executing an Anonymous Block

Click the Run Script button to execute the anonymous block:

The screenshot shows the Oracle SQL Worksheet interface. At the top, there's a toolbar with various icons. A callout arrow points from the text "Click the Run Script button to execute the anonymous block:" to the "Run Script" icon (a green triangle with a white circle) in the toolbar. Below the toolbar is a status bar showing "SQL Worksheet History" and "0.46330699 seconds". The main area contains an anonymous block of PL/SQL code:

```
DECLARE
    v_fname VARCHAR2(20);
BEGIN
    SELECT first_name INTO v_fname FROM employees
    WHERE employee_id=100;
END;
/
```

Below the code, the results pane shows the message "anonymous block completed".

Viewing the Output of a PL/SQL Block

The screenshot shows the Oracle SQL Worksheet interface. In the main pane, an anonymous PL/SQL block is displayed:

```
SET SERVEROUTPUT ON
DECLARE
    v_fname VARCHAR(20);
BEGIN
    SELECT first_name
    INTO v_fname
    FROM employees
    WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE('The First Name of the Employee is ' || v_fname);
END;
/
anonymous block completed
The First Name of the Employee is Steven
```

A callout box points to the command `DBMS_OUTPUT.PUT_LINE` with the text: "Press F5 to execute the command and PL/SQL block."

In the bottom navigation bar, the "Script Output" tab is highlighted with an orange box.

The output pane shows the results of the execution:

```
anonymous block completed
The First Name of the Employee is Steven
```

1

2. Runs the anonymous PL/SQL block
The output appears on the Script Output tab.

Topics covered in this module:

- Integrate SQL statements with PL/SQL program constructs
- Describe the benefits of PL/SQL
- Differentiate between PL/SQL block types
- Output messages in PL/SQL

PL/SQL is a language that has programming features that serve as extensions to SQL. SQL, which is a nonprocedural language, is made procedural with PL/SQL programming constructs. PL/SQL applications can run on any platform or operating system on which an Oracle Server runs. In this lesson, you learned how to build basic PL/SQL blocks.

Oracle PL/SQL 19c

Module 02: Declaring Variables

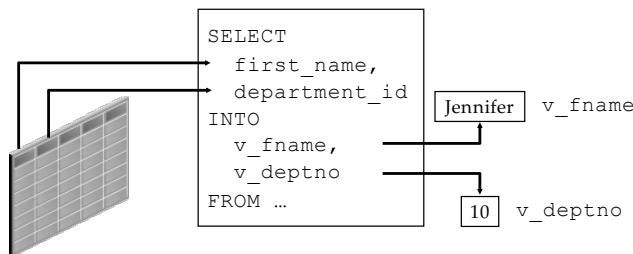
Topics covered in this module:

- Recognize valid and invalid identifiers
- List the uses of variables
- Declare and initialize variables
- List and describe various data types
- Identify the benefits of using the %TYPE attribute
- Declare, use, and print bind variables

You have already learned about basic PL/SQL blocks and their sections. In this lesson, you learn about valid and invalid identifiers. You learn how to declare and initialize variables in the declarative section of a PL/SQL block. The lesson describes the various data types. You also learn about the %TYPE attribute and its benefits.

Variables can be used for:

- Temporary storage of data
- Manipulation of stored values
- Reusability



23

With PL/SQL, you can declare variables and then use them in SQL and procedural statements.

Variables are mainly used for storage of data and manipulation of stored values. Consider the PL/SQL statement in the slide. The statement retrieves `first_name` and `department_id` from the table. If you have to manipulate `first_name` or `department_id`, you have to store the retrieved value. Variables are used to temporarily store the value. You can use the value stored in these variables for processing and manipulating data. Variables can store any PL/SQL object such as variables, types, cursors, and subprograms.

Reusability is another advantage of declaring variables. After the variables are declared, you can use them repeatedly in an application by referring to them multiple times in various statements.

Requirements for Variable Names

A variable name:

- Must start with a letter
- Can include letters or numbers
- Can include special characters (such as \$, _, and #)
- Must contain no more than 30 characters
- Must not include reserved words



24

The rules for naming a variable are listed in the slide.

Variables are:

- Declared and (optionally) initialized in the declarative section
- Used and assigned new values in the executable section
- Passed as parameters to PL/SQL subprograms
- Used to hold the output of a PL/SQL subprogram

25

You can use variables in the following ways:

Declare and initialize them in the declaration section: You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the NOT NULL constraint on the variable. Forward references are not allowed. You must declare a variable before referencing it in other statements, including other declarative statements.

Use them and assign new values to them in the executable section: In the executable section, the existing value of the variable can be replaced with a new value.

Pass them as parameters to PL/SQL subprograms: Subprograms can take parameters. You can pass variables as parameters to subprograms.

Use them to hold the output of a PL/SQL subprogram: Variables can be used to hold the value that is returned by a function.



Declaring and Initializing PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
[ := | DEFAULT expr];
```

Examples:

```
DECLARE
    v_hiredate      DATE;
    v_deptno        NUMBER(2) NOT NULL := 10;
    v_location       VARCHAR2(13) := 'Atlanta';
    c_comm           CONSTANT NUMBER := 1400;
```

26

You must declare all PL/SQL identifiers in the declaration section before referencing them in the PL/SQL block. You have the option of assigning an initial value to a variable (as shown in the slide). You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, be sure that they are already declared separately in a previous statement.

In the syntax:

identifier Is the name of the variable
CONSTANT Constrains the variable so that its value cannot change
 (Constants must be initialized.)
data type Is a scalar, composite, reference, or LOB data type (This course
 covers only scalar, composite, and LOB data types.)
NOT NULL Constrains the variable so that it contains a value (NOT NULL
 variables must be initialized.)
expr Is any PL/SQL expression that can be a literal expression,
 another variable, or an expression involving operators and functions
Note: In addition to variables, you can also declare cursors and exceptions in
the declarative section. You learn about declaring cursors in the lesson titled
“Using Explicit Cursors” and about exceptions in the lesson titled “Handling

Exceptions."

Declaring and Initializing PL/SQL Variables

```
① DECLARE
    v_myName VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
    v_myName := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```



```
② DECLARE
    v_myName VARCHAR2(20) := 'John';
BEGIN
    v_myName := 'Steven';
    DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

27

Examine the two code blocks in the slide.

1. In the first block, the `v_myName` variable is declared but not initialized. A value `John` is assigned to the variable in the executable section.

String literals must be enclosed in single quotation marks. If your string has a quotation mark as in "Today's Date," the string would be '`Today''s Date`'.

The assignment operator is "`:=`".

The `PUT_LINE` procedure is invoked by passing the `v_myName` variable. The value of the variable is concatenated with the string '`My name is:`'.

Output of this anonymous block is:

2. In the second block, the `v_myName` variable is declared and initialized in the executable section. The value `John` is assigned to the variable in the executable section of the block. The output of this anonymous block



holds
the

```
anonymous block completed
My name is: Steven
```

is:

Delimiters in String Literals

```
DECLARE
    v_event VARCHAR2(15);
BEGIN
    v_event := q'!Father's day!';
    DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
    '|| v_event );
    v_event := q'[Mother's day]';
    DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
    '|| v_event );
END;
/
```

Resulting output

anonymous block completed
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day

28

If your string contains an apostrophe (identical to a single quotation mark), you must double the quotation mark, as in the following example:

```
v_event VARCHAR2(15):='Father''s day';
```

The first quotation mark acts as the escape character. This makes your string complicated, especially if you have SQL statements as strings. You can specify any character that is not present in the string as a delimiter. The slide shows how to use the `q'` notation to specify the delimiter. The example uses `!` and `[` as delimiters. Consider the following example:

```
v_event := q'!Father's day!';
```

You can compare this with the first example on this page. You start the string with `q'` if you want to use a delimiter. The character following the notation is the delimiter used. Enter your string after specifying the delimiter, close the delimiter, and close the notation with a single quotation mark. The following example shows how to use `[` as a delimiter:

```
v_event := q'[Mother's day]';
```

- PL/SQL variables:
 - Scalar
 - Reference
 - Large object (LOB)
 - Composite
- Non-PL/SQL variables: Bind variables

29

Every PL/SQL variable has a data type, which specifies a storage format.

constraints, and a value range of values. PL/SQL supports several data type categories, including scalar, reference, large object (LOB), and composite.

Scalar data types: Scalar data types hold a single value. The value depends on the data type of the variable. For example, the `v_myName` variable in the example in the section “Declaring and Initializing PL/SQL Variables” (in this lesson) is of type `VARCHAR2`. Therefore, `v_myName` can hold a string value. PL/SQL also supports Boolean variables.

Reference data types: Reference data types hold values, called *pointers*, which point to a storage location.

LOB data types: LOB data types hold values, called *locators*, which specify the location of large objects (such as graphic images) that are stored outside the table.

Composite data types: Composite data types are available by using PL/SQL *collection* and *record* variables. PL/SQL collections and records contain internal elements that you can treat as individual variables.

Non-PL/SQL variables include host language variables declared in precompiler

programs, screen fields in Forms applications, and host variables. You learn about host variables later in this lesson.

For more information about LOBs, see the *PL/SQL User's Guide and Reference*.

Guidelines for Declaring Variables

- Follow consistent naming conventions.
- Use meaningful identifiers for variables.
- Initialize variables that are designated as NOT NULL and CONSTANT.
- Initialize variables with the assignment operator (`:=`) or the DEFAULT keyword:

```
v_myName VARCHAR2(20) := 'John';
```

```
v_myName VARCHAR2(20) DEFAULT 'John';
```

- Declare one identifier per line for better readability and code maintenance.

30

Here are some guidelines to follow when you declare PL/SQL variables.

Follow consistent naming conventions—for example, you might use `name` to represent a variable and `c_name` to represent a constant. Similarly, to name a variable, you can use `v_fname`. The key is to apply your naming convention consistently for easier identification. Use meaningful and appropriate identifiers for variables. For example, consider using `salary` and `sal_with_commission` instead of `salary1` and `salary2`.

If you use the NOT NULL constraint, you must assign a value when you declare the variable.

In constant declarations, the CONSTANT keyword must precede the type specifier. The following declaration names a constant of NUMBER type and assigns the value of 50,000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error. After initializing a constant, you cannot change its value.

```
sal CONSTANT NUMBER := 50000.00;
```

Guidelines for Declaring PL/SQL Variables

- Avoid using column names as identifiers.

```
DECLARE
    employee_id NUMBER(6);
BEGIN
    SELECT employee_id
    INTO   employee_id
    FROM   employees
    WHERE  last_name = 'Kochhar';
END;
/
```

- Use the NOT NULL constraint when the variable must hold a value.

31

Initialize the variable to an expression with the assignment operator (=) or with the DEFAULT reserved word. If you do not assign an initial value, the new variable contains NULL by default until you assign a value. To assign or reassign a value to a variable, you write a PL/SQL assignment statement. However, it is good programming practice to initialize all variables.

Two objects can have the same name only if they are defined in different blocks. Where they coexist, you can qualify them with labels and use them.

Avoid using column names as identifiers. If PL/SQL variables occur in SQL statements and have the same name as a column, the Oracle Server assumes that it is the column that is being referenced. Although the code example in the slide works, code that is written using the same name for a database table and a variable is not easy to read or maintain.

Impose the NOT NULL constraint when the variable must contain a value. You cannot assign nulls to a variable that is defined as NOT NULL. The NOT NULL constraint must be

followed by an initialization clause.

```
pincode VARCHAR2(15) NOT NULL := 'Oxford';
```

Naming Conventions of PL/SQL Structures Used in This Course

PL/SQL Structure	Convention	Example
Variable	v_variable_name	v_rate
Constant	c_constant_name	c_rate
Subprogram parameter	p_parameter_name	p_id
Bind (host) variable	b_bind_name	b_salary
Cursor	cur_cursor_name	cur_emp
Record	rec_record_name	rec_emp
Type	type_name_type	ename_table_type
Exception	e_exception_name	e_products_invalid
File handle	f_file_handle_name	f_file

32

The table in the slide displays some examples of the naming conventions for PL/SQL structures that are used in this course.

Base Scalar Data Types

- o CHAR [(maximum_length)]
- o VARCHAR2 (maximum_length)
- o NUMBER [(precision, scale)]
- o BINARY_INTEGER
- o PLS_INTEGER
- o BOOLEAN
- o BINARY_FLOAT
- o BINARY_DOUBLE

33

Data Type	Description
CHAR [(maximum_length)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
VARCHAR2 (maximum_length)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(precision, scale)]	Number having precision p and scale s . The precision p can range from 1 through 38. The scale s can range from -84 through 127.
BINARY_INTEGER	Base type for integers between -2,147,483,647 and 2,147,483,647

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Data Type	Description
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 4712 B.C. and A.D. 9999.
TIMESTAMP	The TIMESTAMP data type, which extends the DATE data type, stores the year, month, day, hour, minute, second, and fraction of second. The syntax is <code>TIMESTAMP[(precision)]</code> , where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.
TIMESTAMP WITH TIME ZONE	The TIMESTAMP WITH TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is <code>TIMESTAMP[(precision)] WITH TIME ZONE</code> , where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.

Examples:

```
DECLARE
    v_emp_job          VARCHAR2(9);
    v_count_loop       BINARY_INTEGER := 0;
    v_dept_total_sal  NUMBER(9,2)  := 0;
    v_orderdate        DATE := SYSDATE + 7;
    c_tax_rate         CONSTANT NUMBER(3,2) := 8.25;
    v_valid            BOOLEAN NOT NULL := TRUE;
    ...
```

35

The examples of variable declaration shown in the slide are defined as follows.

- **v_emp_job**: Variable to store an employee job title
- **v_count_loop**: Variable to count the iterations of a loop; initialized to 0
- **v_dept_total_sal**: Variable to accumulate the total salary for a department; initialized to 0
- **v_orderdate**: Variable to store the ship date of an order; initialized to one week from today
- **c_tax_rate**: Constant variable for the tax rate (which never changes throughout the PL/SQL block); set to 8 . 25
- **v_valid**: Flag to indicate whether a piece of data is valid or invalid; initialized to TRUE

- Is used to declare a variable according to:
 - A database column definition
 - Another declared variable
- Is prefixed with:
 - The database table and column name
 - The name of the declared variable

PL/SQL variables are usually declared to hold and manipulate data stored in a database. When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error occurs during execution. If you have to design large subprograms, this can be time consuming and error prone.

Rather than hard-coding the data type and precision of a variable, you can use the %TYPE attribute to declare a variable according to another previously declared variable or database column. The %TYPE attribute is most often used when the value stored in the variable is derived from a table in the database. When you use the %TYPE attribute to declare a variable, you should prefix it with the database table and column name. If you refer to a previously declared variable, prefix the variable name of the previously declared variable to the variable being declared.

o Syntax

```
identifier      table.column_name%TYPE;
```

o Examples

```
...
  v_emp_lname      employees.last_name%TYPE;
...
```

```
...
  v_balance        NUMBER(7,2);
  v_min_balance   v_balance%TYPE := 1000;
...
```

37

Declare variables to store the last name of an employee. The `v_emp_lname` variable is defined to be of the same data type as the `v_last_name` column in the `employees` table. The `%TYPE` attribute provides the data type of a database column.

Declare variables to store the balance of a bank account, as well as the minimum balance, which is 1,000. The `v_min_balance` variable is defined to be of the same data type as the `v_balance` variable. The `%TYPE` attribute provides the data type of a variable.

A NOT NULL database column constraint does not apply to variables that are declared using `%TYPE`. Therefore, if you declare a variable using the `%TYPE` attribute that uses a database column defined as NOT NULL, you can assign the NULL value to the variable.

Declaring Boolean Variables

- Only the TRUE, FALSE, and NULL values can be assigned to a Boolean variable.
- Conditional expressions use the logical operators AND and OR, and the unary operator NOT to check the variable values.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

38

With PL/SQL, you can compare variables in both SQL and procedural statements. These comparisons, called Boolean expressions, consist of simple or complex expressions separated by relational operators. In a SQL statement, you can use Boolean expressions to specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control. NULL stands for a missing, inapplicable, or unknown value.

Examples

```
emp_sal1 := 50000;  
emp_sal2 := 60000;
```

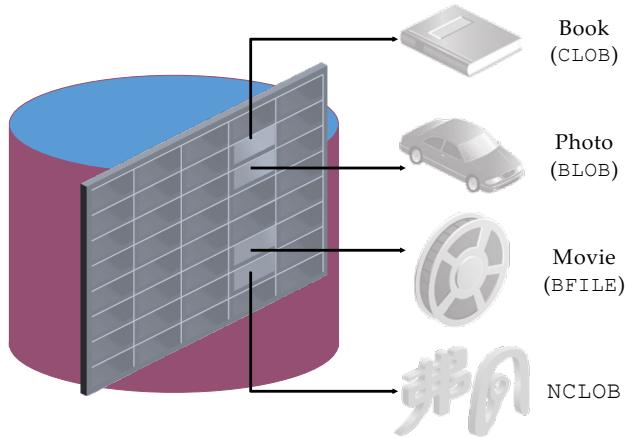
The following expression yields TRUE:

```
emp_sal1 < emp_sal2
```

Declare and initialize a Boolean variable:

```
DECLARE  
    flag BOOLEAN := FALSE;  
BEGIN  
    flag := TRUE;  
END;  
/
```

LOB Data Type Variables



39

Large objects (LOBs) are meant to store a large amount of data. A database column can be of the LOB category. With the LOB category of data types (BLOB, CLOB, and so on), you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) of up to 128 terabytes depending on the database block size. LOB data types allow efficient, random, piecewise access to data and can be attributes of an object type.

The character large object (CLOB) data type is used to store large blocks of character data in the database.

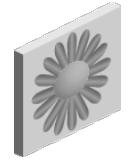
The binary large object (BLOB) data type is used to store large unstructured or structured binary objects in the database. When you insert or retrieve such data into or from the database, the database does not interpret the data. External applications that use this data must interpret the data.

The binary file (BFILE) data type is used to store large binary files. Unlike other LOBs, BFILES are stored outside the database and not in the database. They could be operating system files. Only a pointer to the BFILE is stored in the database.

The national language character large object (NCLOB) data type is used to store large blocks of single-byte or fixed-width multibyte NCHAR unicode data in the database.

Composite Data Types: Records and Collections

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	
------	-----------	---------	--

PL/SQL Collections:

1	SMITH	1	5000
2	JONES	2	2345
3	NANCY	3	12
4	TIM	4	3456

↓ ↓ ↓ ↓
 VARCHAR2 PLS_INTEGER NUMBER PLS_INTEGER

40

As mentioned previously, a scalar data type holds a single value and has no internal components. Composite data types—called PL/SQL Records and PL/SQL Collections—have internal components that you can treat as individual variables.

In a PL/SQL record, the internal components can be of different data types, and are called fields. You access each field with this syntax: `record_name.field_name`. A record variable can hold a table row, or some columns from a table row. Each record field corresponds to a table column.

In a PL/SQL collection, the internal components are always of the same data type, and are called elements. You access each element by its unique subscript. Lists and arrays are classic examples of collections. There are three types of PL/SQL collections: Associative Arrays, Nested Tables, and VARRAY types.

Note

PL/SQL Records and Associative Arrays are covered in the lesson titled: “Working with Composite Data Types.”

NESTED TABLE and VARRAY data types are covered in the course

titled *Oracle Database 10g: Advanced PL/SQL* or *Oracle Database 19c: Advanced PL/SQL*.

Bind variables are:

- Created in the environment
- Also called *host* variables
- Created with the VARIABLE keyword*
- Used in SQL statements and PL/SQL blocks
- Accessed even after the PL/SQL block is executed
- Referenced with a preceding colon

Values can be output using the PRINT command.

* Required when using SQL*Plus and SQL Developer

41

Bind variables are variables that you create in a host environment. For this reason, they are sometimes called *host* variables.

Uses of Bind Variables

Bind variables are created in the environment and not in the declarative section of a PL/SQL block. Therefore, bind variables are accessible even after the block is executed. When created, bind variables can be used and manipulated by multiple subprograms. They can be used in SQL statements and PL/SQL blocks just like any other variable. These variables can be passed as run-time values into or out of PL/SQL subprograms.

Note: A bind variable is an environment variable, but is not a global variable.

Creating Bind Variables

To create a bind variable in SQL Developer, use the VARIABLE command. For example, you declare a variable of type NUMBER and VARCHAR2 as follows:

```
VARIABLE return_code NUMBER  
VARIABLE return_msg VARCHAR2(30)
```

Viewing Values in Bind Variables

You can reference the bind variable using SQL Developer and view its value using the PRINT command.

Referencing Bind Variables

Example:

```
VARIABLE b_emp_salary NUMBER
BEGIN
    SELECT salary INTO :b_emp_salary
    FROM employees WHERE employee_id = 178;
END;
/
PRINT b_emp_salary
SELECT first_name, last_name
FROM employees
WHERE salary=:b_emp_salary;
```

Output

Script Output x	
anonymous block completed	
B_EMP_SALARY	
7000	
FIRST_NAME	LAST_NAME
Oliver	Tuvault
Sarath	Sewall
Kimberely	Grant

42

As stated previously, after you create a bind variable, you can reference that variable in any other SQL statement or PL/SQL program.

In the example, `b_emp_salary` is created as a bind variable in the PL/SQL block. Then, it is used in the `SELECT` statement that follows.

When you execute the PL/SQL block shown in the slide, you see the following output:

The `PRINT` command executes:

```
b_emp_salary
-----
7000
```

Then, the output of the SQL statement follows:

FIRST_NAME	LAST_NAME
Oliver	Tuvault
Sarath	Sewall
Kimberely	Grant

Note: To display all bind variables, use the `PRINT` command without a variable.

Using AUTOPRINT with Bind Variables

The screenshot shows the Oracle SQL Worksheet interface. In the main area, there is a code editor window titled "Worksheet" containing the following PL/SQL code:

```
1 VARIABLE b_emp_salary NUMBER
2 SET AUTOPRINT ON
3 DECLARE
4   v_empno NUMBER(6):=&empno;
5 BEGIN
6   SELECT salary INTO :b_emp_salary
7   FROM employees WHERE employee_id = v_empno;
8 END;
9 /
10
```

An orange box highlights the line "SET AUTOPRINT ON". A modal dialog box titled "Enter Substitution Variable" is displayed, prompting for an "EMPNO:" value. The value "178" is entered in the input field. Two buttons, "OK" and "Cancel", are at the bottom of the dialog. An orange arrow points from the "OK" button to the "Script Output" window on the right.

The "Script Output" window shows the output of the anonymous block. It displays the message "Task completed in", followed by "b_emp_salary", and then the value "7000".

43

Use the `SET AUTOPRINT ON` command to automatically display the bind variables used in a successful PL/SQL block.

Example

In the code example:

A bind variable named `b_emp_salary` is created and `AUTOPRINT` is turned on.

A variable named `v_empno` is declared, and a substitution variable is used to receive user input.

Finally, the bind variable and temporary variables are used in the executable section of the PL/SQL block.

When a valid employee number is entered—in this case 178—the output of the bind variable is automatically printed. The bind variable contains the salary for the employee number that is provided by the user.

Topics covered in this module:

- Recognize valid and invalid identifiers
- Declare variables in the declarative section of a PL/SQL block
- Initialize variables and use them in the executable section
- Differentiate between scalar and composite data types
- Use the %TYPE attribute
- Use bind variables

44

An anonymous PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements to perform a logical function.

The declarative part is the first part of a PL/SQL block and is used for declaring objects such as variables, constants, cursors, and definitions of error situations called *exceptions*.

In this lesson, you learned how to declare variables in the declarative section. You saw some of the guidelines for declaring variables. You learned how to initialize variables when you declare them.

The executable part of a PL/SQL block is the mandatory part and contains SQL and PL/SQL statements for querying and manipulating data. You learned how to initialize variables in the executable section and also how to use them and manipulate the values of variables.

Oracle PL/SQL 19c

Module 03: Writing Executable Statements

Topics included in this module:

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Describe when implicit conversions take place and when explicit conversions have to be dealt with
- Write nested blocks and qualify variables with labels
- Write readable code with appropriate indentation
- Use sequences in PL/SQL expressions

46

You learned how to declare variables and write executable statements in a PL/SQL block. In this lesson, you learn how lexical units make up a PL/SQL block. You learn to write nested blocks. You also learn about the scope and visibility of variables in nested blocks and about qualifying variables with labels.

Lexical units:

- Are building blocks of any PL/SQL block
- Are sequences of characters including letters, numerals, tabs, spaces, returns, and symbols
- Can be classified as:
 - Identifiers: v_fname, c_percent
 - Delimiters: ; , +, -
 - Literals: John, 428, True
 - Comments: --, /* */

47

Lexical units include letters, numerals, special characters, tabs, spaces, returns, and symbols.

Identifiers: Identifiers are the names given to PL/SQL objects. You learned to identify valid and invalid identifiers. Recall that keywords cannot be used as identifiers.

Quoted identifiers:

Make identifiers case-sensitive.

Include characters such as spaces.

Use reserved words.

Examples:

"begin date" DATE;

"end date" DATE;

"exception thrown" BOOLEAN DEFAULT TRUE;

All subsequent usage of these variables should have double quotation marks. However, use of quoted identifiers is not recommended.

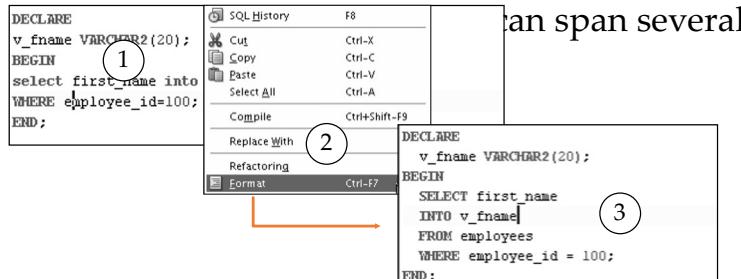
Delimiters: Delimiters are symbols that have special meaning. You already learned that the semicolon (;) is used to terminate a SQL or PL/SQL statement. Therefore, ; is an example of a delimiter.

For more information, refer to the *PL/SQL User's Guide and Reference*.

- Using Literals

- Character and date literals must be enclosed in single quotation marks.
- Numbers can be simple values or in scientific

```
v_name := 'Henderson';
```



48

Using Literals

A literal is an explicit numeric, character string, date, or Boolean value that is not represented by an identifier.

Character literals include all printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.

Numeric literals can be represented either by a simple value (for example, -32.5) or in scientific notation (for example, 2E5 means $2 * 10^5 = 200,000$).

Formatting Code

In a PL/SQL block, a SQL statement can span several lines (as shown in example 3 in the slide).

You can format an unformatted SQL statement (as shown in example 1 in the slide) by using the SQL Worksheet shortcut menu. Right-click the active SQL Worksheet and, in the shortcut menu that appears, select the Format option (as shown in example 2).

Note: You can also use the shortcut key combination of Ctrl + F7 to format your code.



Commenting Code

- o Prefix single-line comments with two hyphens (--) .
- o Place a block comment between the symbols /* and */.

Example:

```
DECLARE
  ...
  v_annual_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
   monthly salary input from the user */
  v_annual_sal := monthly_sal * 12;
  --The following line displays the annual salary
  DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

49

You should comment code to document each phase and to assist debugging
in PL/SQL code.

A single-line comment is commonly prefixed with two hyphens (--) .
You can also enclose a comment between the symbols /* and */ .

Note: For multiline comments, you can either precede each comment line with two hyphens, or use the block comment format.
Comments are strictly informational and do not enforce any conditions or behavior on the logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance.

- Available in procedural statements:
 - Single-row functions
- Not available in procedural statements:
 - DECODE
 - Group functions

SQL provides several predefined functions that can be used in SQL statements.

Most of these functions (such as single-row number and character functions, data type conversion functions, and date and time-stamp functions) are valid in PL/SQL expressions.

The following functions are not available in procedural statements:

• DECODE

Group functions: AVG, MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE

Group functions apply to groups of rows in a table and are, therefore, available only in SQL statements in a PL/SQL block. The functions mentioned here are only a subset of the complete list.

- Get the length of a string:

```
v_desc_size INTEGER(5);
v_prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod description
v_desc_size:= LENGTH(v_prod_description);
```

- Get the number of months an employee has

```
v_tenure:= MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

51

You can use SQL functions to manipulate data. These functions are grouped into the following categories.

- Number
- Character
- Conversion
- Date
- Miscellaneous

Starting in 19c:

```
DECLARE
    v_new_id NUMBER;
BEGIN
    v_new_id := my_seq.NEXTVAL;
END;
/
```

Before 19c:

```
DECLARE
    v_new_id NUMBER;
BEGIN
    SELECT my_seq.NEXTVAL INTO v_new_id FROM Dual;
END;
/
```

In Oracle Database 19c, you can use the NEXTVAL and CURRVAL

pseudocolumns in any PL/SQL context, where an expression of the NUMBER data type may legally appear. Although the old style of using a SELECT statement to query a sequence is still valid, it is recommended that you do not use it.

Before Oracle Database 19c, you were forced to write a SQL statement in order to use a sequence object value in a PL/SQL subroutine. Typically, you would write a SELECT statement to reference the pseudocolumns of NEXTVAL and CURRVAL to obtain a sequence number. This method created a usability problem.

In Oracle Database 19c, the limitation of forcing you to write a SQL statement to retrieve a sequence value is eliminated. With the sequence enhancement feature:

- Sequence usability is improved
- The developer has to type less
- The resulting code is clearer

- Converts data to comparable data types
- Is of two types:
 - Implicit conversion
 - Explicit conversion
- Functions:
 - TO_CHAR
 - TO_DATE
 - TO_NUMBER
 - TO_TIMESTAMP

53

In any programming language, converting one data type to another is a common requirement. PL/SQL can handle such conversions with scalar data types. Data type conversions can be of two types:

Implicit conversions: PL/SQL attempts to convert data types dynamically if they are mixed in a statement. Consider the following example:

```
DECLARE
    v_salary NUMBER(6):=6000;
    v_sal_hike VARCHAR2(5):='1000';
    v_total_salary v_salary%TYPE;
BEGIN
    v_total_salary:=v_salary + v_sal_hike;
END;
/
```

In this example, the `sal_hike` variable is of the `VARCHAR2` type. When calculating the total salary, PL/SQL first converts `sal_hike` to `NUMBER`, and then performs the operation. The result is of the `NUMBER` type.

Implicit conversions can be between:

- Characters and numbers
- Characters and dates

Data Type Conversion

① -- implicit data type conversion
v_date_of_joining DATE:= '02-Feb-2000';

② -- error in data type conversion
v_date_of_joining DATE:= 'February 02,2000';

③ -- explicit data type conversion
v_date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');

54

Note the three examples of implicit and explicit conversions of the DATE data type in the slide:

1. Because the string literal being assigned to date_of_joining is in the default format, this example performs implicit conversion and assigns the specified date to date_of_joining.
2. The PL/SQL returns an error because the date that is being assigned is not in the default format.
3. The TO_DATE function is used to explicitly convert the given date in a particular format and assign it to the DATE data type variable date_of_joining.

- Writing executable statements in a PL/SQL block
- Writing nested blocks
- Using operators and developing readable code

PL/SQL blocks can be nested.

- o An executable section (BEGIN ... END) can contain nested blocks.
- o An exception section can contain nested blocks.



56

Being procedural gives PL/SQL the ability to nest statements. You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. If your executable section has code for many logically related functionalities to support multiple business requirements, you can divide the executable section into smaller blocks. The exception section can also contain nested blocks.

Nested Blocks: Example

```
DECLARE
  v_outer_variable VARCHAR2(20) := 'GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) := 'LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

```
anonymous block completed
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE
```

57

The example shown in the slide has an outer (parent) block and a nested (child) block. The `v_outer_variable` variable is declared in the outer block and the `v_inner_variable` variable is declared in the inner block. `v_outer_variable` is local to the outer block but global to the inner block. When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. There is no variable with the same name in the inner block, so PL/SQL looks for the variable in the outer block. Therefore, `v_outer_variable` is considered to be the global variable for all the enclosing blocks. You can access this variable in the inner block as shown in the slide. Variables declared in a PL/SQL block are considered local to that block and global to all its subblocks.

`v_inner_variable` is local to the inner block and is not global because the inner block does not have any nested blocks. This variable can be accessed only within the inner block. If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks. PL/SQL does not look downward in the child blocks.

Variable Scope and Visibility

```
DECLARE
  v_father_name VARCHAR2(20) := 'Patrick';
  v_date_of_birth DATE := '20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20) := 'Mike';
    v_date_of_birth DATE := '12-Dec-2002';
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
      DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
      DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
    END;
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
  END;
/

```

58

The output of the block shown in the slide is as follows:

anonymous block completed

Father's Name: Patrick

Date of Birth: 12-DEC-02

Child's Name: Mike

Date of Birth: 20-APR-72

Examine the date of birth that is printed for father and child. The output does not provide the correct information, because the scope and visibility of the variables are not applied correctly.

The *scope* of a variable is the portion of the program in which the variable is declared and is accessible.

The *visibility* of a variable is the portion of the program where the variable can be accessed without using a qualifier.

Scope

The `v_father_name` variable and the first occurrence of the `v_date_of_birth` variable are declared in the outer block. These variables have the scope of the block in which they are declared. Therefore, the scope of these variables is limited to the outer block.

Using a Qualifier with Nested Blocks

```
BEGIN <<outer>>
DECLARE
  v_father_name VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                         ||outer.v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
  END;
END;
END outer;
```

59

A *qualifier* is a label given to a block. You can use a qualifier to access the variables that have scope but are not visible.

Example

In the code example:

The outer block is labeled `outer`

Within the inner block, the `outer` qualifier is used to access the `v_date_of_birth` variable that is declared in the outer block.

Therefore, the father's date of birth and the child's date of birth can both be printed from within the inner block.

The output of the code in the slide shows the correct information:

Note: Labeling a block allows you to access its variables from other blocks. You can label any block.

anonymous block completed
Father's Name: Patrick
Date of Birth: 20-APR-72
Child's Name: Mike
Date of Birth: 12-DEC-02

Challenge: Determining Variable Scope

```
BEGIN <<outer>>
DECLARE
    v_sal      NUMBER(7,2) := 60000;
    v_comm     NUMBER(7,2) := v_sal * 0.20;
    v_message  VARCHAR2(255) := ' eligible for commission';
BEGIN
    DECLARE
        v_sal      NUMBER(7,2) := 50000;
        v_comm     NUMBER(7,2) := 0;
        v_total_comp NUMBER(7,2) := v_sal + v_comm;
    BEGIN
        1--> v_message := 'CLERK not'||v_message;
        outer.v_comm := v_sal * 0.30;
    END;
    2--> v_message := 'SALESMAN'||v_message;
END;
END outer;
/
```

60

Evaluate the PL/SQL block in the slide. Determine each of the following values according to the rules of scoping:

1. Value of `v_message` at position 1
2. Value of `v_total_comp` at position 2
3. Value of `v_comm` at position 1
4. Value of `outer.v_comm` at position 1
5. Value of `v_comm` at position 2
6. Value of `v_message` at position 2

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations
- Exponential operator (**)

61

The operations in an expression are performed in a particular order depending on their precedence (priority). The following table shows the default order of operations from high priority to low priority:

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -,	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion



Operators in PL/SQL: Examples

- Increment the counter for a loop.

```
loop_count := loop_count + 1;
```

- Set the value of a Boolean flag.

```
good_sal := sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number

```
valid := (empno IS NOT NULL);
```

62

When you are working with nulls, you can avoid some common mistakes by keeping in mind the following rules.

Comparisons involving nulls always yield NULL.

Applying the logical operator NOT to a null yields NULL.

In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed.

Make code maintenance easier by:

- Documenting code with comments
- Developing a case convention for the code
- Developing naming conventions for identifiers and other objects
- Enhancing readability by indenting

63

Follow programming guidelines shown in the slide to produce clear code and reduce maintenance when developing a PL/SQL block.

Code Conventions

The following table provides guidelines for writing code in uppercase or lowercase characters to help distinguish keywords from named objects.

Category	Case Convention	Examples
SQL statements	Uppercase	SELECT, INSERT
PL/SQL keywords	Uppercase	DECLARE, BEGIN, IF
Data types	Uppercase	VARCHAR2, BOOLEAN
Identifiers and parameters	Lowercase	v_sal, emp_cursor, g_sal, p_empno
Database tables	Lowercase, plural	employees, departments
Database columns	Lowercase, singular	employee_id, department_id

For clarity, indent each level of code.

```
BEGIN
    IF x=0 THEN
        y:=1;
    END IF;
END;
/
```

```
DECLARE
    deptno      NUMBER(4);
    location_id NUMBER(4);
BEGIN
    SELECT department_id,
           location_id
    INTO   deptno,
           location_id
    FROM   departments
    WHERE  department_name
           = 'Sales';
...
END;
/
```

64

For clarity and enhanced readability, indent each level of code. To show structure, you can divide lines by using carriage returns and you can indent lines by using spaces and tabs. Compare the following IF statements for readability:

```
IF x>y THEN max:=x;ELSE max:=y;END IF;
```

```
IF x > y THEN
    max := x;
ELSE
    max := y;
END IF;
```

Topics covered in this module:

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Write nested blocks to break logically related functionalities
- Decide when to perform explicit conversions
- Qualify variables in nested blocks
- Use sequences in PL/SQL expressions

Because PL/SQL is an extension of SQL, the general syntax rules that apply to SQL also apply to PL/SQL.

A block can have any number of nested blocks defined within its executable part. Blocks defined within a block are called subblocks. You can nest blocks only in the executable part of a block. Because the exception section is also a part of the executable section, it can also contain nested blocks. Ensure correct scope and visibility of the variables when you have nested blocks.

Avoid using the same identifiers in the parent and child blocks.

Most of the functions available in SQL are also valid in PL/SQL expressions.

Conversion functions convert a value from one data type to another.

Comparison operators compare one expression with another. The result is always TRUE, FALSE, or NULL. Typically, you use comparison operators in conditional control statements and in the WHERE clause of SQL data manipulation statements. The relational operators enable you to compare arbitrarily complex expressions.

Oracle PL/SQL 19c

Module 04: Oracle Database Server:
SQL Statements in PL/SQL

Topics covered in this module:

- Determine the SQL statements that can be directly included in a PL/SQL executable block
- Manipulate data with DML statements in PL/SQL
- Use transaction control statements in PL/SQL
- Make use of the INTO clause to hold the values returned by a SQL statement
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes

67

In this lesson, you learn to embed standard SQL SELECT, INSERT, UPDATE, DELETE, and MERGE statements in PL/SQL blocks. You learn how to include data definition language (DDL) and transaction control statements in PL/SQL. You learn the need for cursors and differentiate between the two types of cursors. The lesson also presents the various SQL cursor attributes that can be used with implicit cursors.

- Retrieve a row from the database by using the SELECT command.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the COMMIT, ROLLBACK, or SAVEPOINT command.

In a PL/SQL block, you use SQL statements to retrieve and modify data from the database table. PL/SQL supports data manipulation language (DML) and transaction control commands. You can use DML commands to modify the data in a database table. However, remember the following points while using DML statements and transaction control commands in PL/SQL blocks:

The END keyword signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.

PL/SQL does not directly support data definition language (DDL) statements such as CREATE TABLE, ALTER TABLE, or DROP TABLE. PL/SQL supports early binding, which cannot happen if applications have to create database objects at run time by passing values. DDL statements cannot be directly executed. These statements are dynamic SQL statements. Dynamic SQL statements are built as character strings at run time and can contain placeholders for parameters. Therefore, you can use dynamic SQL to execute your DDL statements in PL/SQL. The details of working with dynamic SQL are covered in the course titled *Oracle Database: Develop PL/SQL Program Units*.

PL/SQL does not directly support data control language (DCL) statements such as GRANT or REVOKE. You can use dynamic SQL to execute them.

- Retrieve data from the database with a SELECT statement.
- Syntax:

```
SELECT  select_list
INTO    {variable_name[, variable_name]...
        | record_name}
FROM    table
[WHERE  condition];
```

69

Use the SELECT statement to retrieve data from the database

<i>select_list</i>	List of at least one column; can include SQL expressions, row functions, or group functions
<i>variable_name</i>	Scalar variable that holds the retrieved value
Guidelines for Retrieving Data in PL/SQL	
<i>record_name</i>	PL/SQL record that holds the retrieved values
<i>table</i>	Terminate each SQL statement with a semicolon (;).
<i>condition</i>	Specifies the database table to store in a variable by using the INTO clause. Is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants variables, constants, literals, and PL/SQL expressions. However, when you use the INTO clause, you should fetch only one row; using the WHERE clause is required in such cases.

- The INTO clause is required.
- Queries must return only one row.

```
DECLARE
  v_fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO v_fname
  FROM employees WHERE employee_id=200;
  DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
END;
/
```

```
anonymous block completed
First Name is : Jennifer
```

70

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables that hold the values that SQL returns from the SELECT clause. You must specify one variable for each item selected, and the order of the variables must correspond with the items selected.

Use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: Queries must return only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages these errors by raising standard exceptions, which you can handle in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Include a WHERE condition in the SQL statement so that the statement returns a single row. You learn about exception handling in the lesson titled “Handling Exceptions.”

Note: In all cases where DBMS_OUTPUT.PUT_LINE is used in the code examples, the SET SERVEROUTPUT ON statement precedes the block.

Retrieving Data in PL/SQL: Example

Retrieve hire_date and salary for the specified employee.

```
DECLARE
    v_emp_hiredate    employees.hire_date%TYPE;
    v_emp_salary      employees.salary%TYPE;
BEGIN
    SELECT    hire_date, salary
    INTO      v_emp_hiredate, v_emp_salary
    FROM     employees
    WHERE    employee_id = 100;
    DBMS_OUTPUT.PUT_LINE ('Hire date is :'|| v_emp_hiredate);
    DBMS_OUTPUT.PUT_LINE ('Salary is :'|| v_emp_salary);
END;
/
```

```
anonymous block completed
Hire date is : 17-JUN-87
Salary is : 24000
```

71

In the example in the slide, the `v_emp_hiredate` and `v_emp_salary` variables are declared in the declarative section of the PL/SQL block. In the executable section, the values of the `hire_date` and `salary` columns for the employee with the `employee_id` 100 are retrieved from the `employees` table. Next, they are stored in the `v_emp_hiredate` and `v_emp_salary` variables, respectively. Observe how the `INTO` clause, along with the `SELECT` statement, retrieves the database column values and stores them in the PL/SQL variables.

Note: The `SELECT` statement retrieves `hire_date`, and then `salary`. The variables in the `INTO` clause must thus be in the same order. For example, if you exchange `v_emp_hiredate` and `v_emp_salary` in the statement in the slide, the statement results in an error.

- Return the sum of salaries for all the employees in the specified department.

- Example:

```
DECLARE
    v_sum_sal    NUMBER(10,2);
    v_deptno     NUMBER NOT NULL := 60;
BEGIN
    SELECT  SUM(salary)  -- group function
    INTO v_sum_sal
    FROM employees
    WHERE   department_id = v_deptno;
    DBMS_OUTPUT.PUT_LINE ('The sum of salary is ' || v_sum_sal);
END;
```

```
anonymous block completed
The sum of salary is 28800
```

72

In the example in the slide, the `v_sum_sal` and `v_deptno` variables are declared in the declarative section of the PL/SQL block. In the executable section, the total salary for the employees in the department with `department_id` 60 is computed using the SQL aggregate function `SUM`. The calculated total salary is assigned to the `v_sum_sal` variable.

Note: Group functions cannot be used in PL/SQL syntax. They must be used in SQL statements within a PL/SQL block as shown in the example in the slide. For instance, you *cannot* use group functions using the following syntax:

```
V_sum_sal := SUM(employees.salary);
```

Naming Ambiguities

```
• DECLARE
•     hire_date      employees.hire_date%TYPE;
•     sysdate        hire_date%TYPE;
•     employee_id    employees.employee_id%TYPE := 176;
• BEGIN
•     SELECT  hire_date, sysdate
•     INTO    hire_date, sysdate
•     FROM   employees
•     WHERE   employee_id = employee_id;
• END;
• /
```

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
01422. 00000 - "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

73

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables.

The example shown in the slide is defined as follows: Retrieve the hire date and today's date from the `employees` table for `employee_id` 176. This example raises an unhandled run-time exception because, in the `WHERE` clause, the PL/SQL variable names are the same as the database column names in the `employees` table.

The following `DELETE` statement removes all employees from the `employees` table, where the last name is not null (not just "King"), because the Oracle Server assumes that both occurrences of `last_name` in the `WHERE` clause refer to the database column:

```
DECLARE
    last_name VARCHAR2(25) := 'King';
BEGIN
    DELETE FROM employees WHERE last_name =
    last_name;
    ...

```

- Use a naming convention to avoid ambiguity in the WHERE clause.
- Avoid using database column names as identifiers.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.
- The names of local variables and formal parameters take precedence over the names of database *tables*.
- The names of database table *columns* take precedence over the names of local variables.
- The names of variables take precedence over the function names.

74

Avoid ambiguity in the WHERE clause by adhering to a naming convention

that distinguishes database column names from PL/SQL variable names.

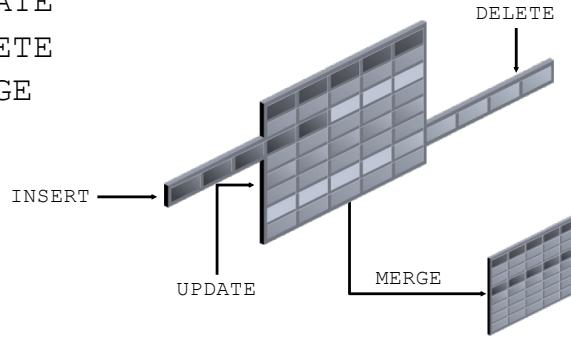
Database columns and identifiers should have distinct names.

Syntax errors can arise because PL/SQL checks the database first for a column in the table.

Note: There is no possibility of ambiguity in the SELECT clause because any identifier in the SELECT clause must be a database column name. There is no possibility of ambiguity in the INTO clause because identifiers in the INTO clause must be PL/SQL variables. The possibility of confusion is present only in the WHERE clause.

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



75

You manipulate data in the database by using DML commands. You can issue DML commands such as INSERT, UPDATE, DELETE, and MERGE without restriction in PL/SQL. Row locks (and table locks) are released by including the COMMIT or ROLLBACK statements in the PL/SQL code.

The INSERT statement adds new rows to the table.

The UPDATE statement modifies existing rows in the table.

The DELETE statement removes rows from the table.

The MERGE statement selects rows from one table to update or insert into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause.

Note: MERGE is a deterministic statement. That is, you cannot update the same row of the target table multiple times in the same MERGE statement. You must have INSERT and UPDATE object privileges on the target table and SELECT privilege on the source table.

Add new employee information to the EMPLOYEES table.

```
BEGIN
  INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
  VALUES (employees_seq.NEXTVAL, 'Ruth', 'Cores',
  'RCORES', CURRENT_DATE, 'AD_ASST', 4000);
END;
/
```

76

In the example in the slide, an `INSERT` statement is used within a PL/SQL block to insert a record into the `EMPLOYEES` table. While using the `INSERT` command in a PL/SQL block, you can:

Use SQL functions such as `USER` and `CURRENT_DATE`

Generate primary key values by using existing database sequences

Derive values in the PL/SQL block

Note: The data in the `EMPLOYEES` table needs to remain unchanged. Even though the `EMPLOYEES` table is not read-only, inserting, updating, and deleting are not allowed on this table to ensure consistency of output.

Therefore, the command `ROLLBACK` is used as shown in the code example:

`code_05_15_s.sql`.

Increase the salary of all employees who

```
DECLARE
    sal_increase    employees.salary%TYPE := 800;
BEGIN
    UPDATE      employees
    SET          salary = salary + sal_increase
    WHERE        job_id = 'ST_CLERK';
END;
/
```

```
anonymous block completed
FIRST_NAME      SALARY
-----
Julia           4000
Irene           3500
James           3200
Steven          3000
...

```

```
Curtis          3900
Randall         3400
Peter           3300
20 rows selected
```

77

There may be ambiguity in the `SET` clause of the `UPDATE` statement

because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable. Recall that if column names and identifier names are identical in the `WHERE` clause, the Oracle Server looks to the database first for the name.

Remember that the `WHERE` clause is used to determine the rows that are affected. If no rows are modified, no error occurs (unlike the `SELECT` statement in PL/SQL).

Note: PL/SQL variable assignments always use `:=`, and SQL column assignments always use `=`.

Deleting Data: Example

Delete rows that belong to department 10 from the employees table.

```
DECLARE
    deptno    employees.department_id%TYPE := 10;
BEGIN
    DELETE FROM    employees
    WHERE   department_id = deptno;
END;
/
```

78

The `DELETE` statement removes unwanted rows from a table. If the `WHERE` clause is not used, all the rows in a table can be removed if there are no integrity constraints.

Merging Rows

Insert or update rows in the `copy_emp` table to match the `employees` table.

```
BEGIN
MERGE INTO copy_emp c
  USING employees e
    ON (e.employee_id = c.empno)
  WHEN MATCHED THEN
    UPDATE SET
      c.first_name      = e.first_name,
      c.last_name       = e.last_name,
      c.email           = e.email,
      . . .
  WHEN NOT MATCHED THEN
    INSERT VALUES(e.employee_id, e.first_name, e.last_name,
                  . . ., e.department_id);
END;
/
```

79

The `MERGE` statement inserts or updates rows in one table by using data from another table. Each row is inserted or updated in the target table depending on an equijoin condition.

The example shown matches the `empno` column in the `copy_emp` table to the `employee_id` column in the `employees` table. If a match is found, the row is updated to match the row in the `employees` table. If the row is not found, it is inserted into the `copy_emp` table.

The complete example of using `MERGE` in a PL/SQL block is shown on the next page.

- A cursor is a pointer to the private memory area allocated by the Oracle Server. It is used to handle the result set of a SELECT statement.
- There are two types of cursors: implicit and explicit.
 - **Implicit:** Created and managed internally by the Oracle Server to process SQL statements
 - **Explicit:** Declared explicitly by the programmer

You have already learned that you can include SQL statements that return a single row in a PL/SQL block. The data retrieved by the SQL statement should be held in variables using the INTO clause.

Where Does the Oracle Server Process SQL Statements?

The Oracle Server allocates a private memory area called the *context area* for processing SQL statements. The SQL statement is parsed and processed in this area. The information required for processing and the information retrieved after processing are all stored in this area. You have no control over this area because it is internally managed by the Oracle Server.

A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle Server. When the executable block issues a SQL statement, PL/SQL creates an implicit cursor.

Types of Cursors

There are two types of cursors:

Implicit: An *implicit cursor* is created and managed by the Oracle Server. You do not have access to it. The Oracle Server creates such a cursor when it has to execute a SQL statement.

SQL Cursor Attributes for Implicit Cursors

Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affected at least one row
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement did not affect even one row
SQL%ROWCOUNT	An integer value that represents the number of rows affected by the most recent SQL statement

SQL cursor attributes enable you to evaluate what happened when an implicit cursor was last used. Use these attributes in PL/SQL statements but not in SQL statements.

You can test the SQL%ROWCOUNT, SQL%FOUND, and SQL%NOTFOUND attributes in the executable section of a block to gather information after the appropriate DML command executes. PL/SQL does not return an error if a DML statement does not affect rows in the underlying table. However, if a SELECT statement does not retrieve any rows, PL/SQL returns an exception. Observe that the attributes are prefixed with SQL. These cursor attributes are used with implicit cursors that are automatically created by PL/SQL and for which you do not know the names. Therefore, you use SQL instead of the cursor name.

The SQL%NOTFOUND attribute is the opposite of SQL%FOUND. This attribute may be used as the exit condition in a loop. It is useful in UPDATE and DELETE statements when no rows are changed because exceptions are not returned in these cases.

You learn about explicit cursor attributes in the lesson titled “Using Explicit Cursors.”

SQL Cursor Attributes for Implicit Cursors

Delete rows that have the specified employee ID from the employees table.
Print the number of rows deleted.

Example:

```
DECLARE
    v_rows_deleted VARCHAR2(30)
    v_empno employees.employee_id%TYPE := 176;
BEGIN
    DELETE FROM employees
    WHERE employee_id = v_empno;
    v_rows_deleted := (SQL%ROWCOUNT ||
                       ' row deleted.');
    DBMS_OUTPUT.PUT_LINE (v_rows_deleted);
END;
```

82

The example in the slide deletes a row with employee_id 176 from the employees table. Using the SQL%ROWCOUNT attribute, you can print the number of rows deleted.

Topics covered in this module:

- Embed DML statements, transaction control statements, and DDL statements in PL/SQL
- Use the INTO clause, which is mandatory for all SELECT statements in PL/SQL
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes to determine the outcome of SQL statements

83

DML commands and transaction control statements can be used in PL/SQL programs without restriction. However, the DDL commands cannot be used directly.

A SELECT statement in a PL/SQL block can return only one row. It is mandatory to use the INTO clause to hold the values retrieved by the SELECT statement.

A cursor is a pointer to the memory area. There are two types of cursors. Implicit cursors are created and managed internally by the Oracle Server to execute SQL statements. You can use SQL cursor attributes with these cursors to determine the outcome of the SQL statement. Explicit cursors are declared by programmers.

Oracle PL/SQL 19c

Module 05: Control Structures

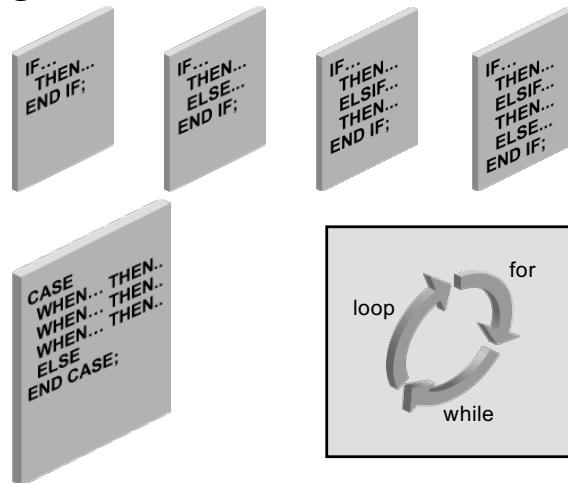
Topics covered in this module:

- Identify the uses and types of control structures
- Construct an IF statement
- Use CASE statements and CASE expressions
- Construct and identify loop statements
- Use guidelines when using conditional control structures

You have learned to write PL/SQL blocks containing declarative and executable sections. You have also learned to include expressions and SQL statements in the executable block.

In this lesson, you learn how to use control structures such as IF statements, CASE expressions, and LOOP structures in a PL/SQL block.

Controlling Flow of Execution



86

You can change the logical flow of statements within the PL/SQL block with a number of control structures. This lesson addresses four types of PL/SQL control structures: conditional constructs with the `IF` statement, `CASE` expressions, `LOOP` control structures, and the `CONTINUE` statement.

Syntax:

```
IF condition THEN  
    statements;  
[ELSIF condition THEN  
    statements;]  
[ELSE  
    statements;]  
END IF;
```

87

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

In the syntax:

<i>condition</i>	Is a Boolean variable or expression that returns TRUE, FALSE, or NULL
THEN	Introduces a clause that associates the Boolean expression with the sequence of statements that follows it
<i>statements</i>	Can be one or more PL/SQL or SQL statements. (They may include additional IF statements containing several nested IF, ELSE, and ELSIF statements.) The statements in the THEN clause are executed only if the condition in the associated IF clause evaluates to TRUE.

Simple IF Statement

```
DECLARE
    v_myage  number:=31;
BEGIN
    IF v_myage  < 11
    THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    END IF;
END;
/
```

anonymous block completed

88

Simple IF Example

The slide shows an example of a simple IF statement with the THEN clause.

The v_myage variable is initialized to 31.

The condition for the IF statement returns FALSE because v_myage is not less than 11.

Therefore, the control never reaches the THEN clause.

Adding Conditional Expressions

An IF statement can have multiple conditional expressions related with logical operators such as AND, OR, and NOT.

For example:

```
IF (myfirstname='Christopher' AND v_myage <11)
...
...
```

The condition uses the AND operator and therefore, evaluates to TRUE only if both conditions are evaluated as TRUE. There is no limitation on the number of conditional expressions. However, these statements must be related with appropriate logical operators.

IF THEN ELSE Statement

```
DECLARE
    v_myage  number:=31;
BEGIN
    IF
        v_myage < 11
    THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' I am not a child ');
    END IF;
END;
/
```

```
anonymous block completed
I am not a child
```

89

An ELSE clause is added to the code in the previous slide. The condition has not changed and, therefore, still evaluates to FALSE. Recall that the statements in the THEN clause are executed only if the condition returns TRUE. In this case, the condition returns FALSE and the control moves to the ELSE statement.

The output of the block is shown below the code.

IF ELSIF ELSE Clause

```
DECLARE
    v_myage number:=31;
BEGIN
    IF v_myage < 11 THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSIF v_myage < 20 THEN
        DBMS_OUTPUT.PUT_LINE(' I am young ');
    ELSIF v_myage < 30 THEN
        DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
    ELSIF v_myage < 40 THEN
        DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' I am always young ');
    END IF;
END;
/
```

```
anonymous block completed
I am in my thirties
```

90

The IF clause may contain multiple ELSIF clauses and an ELSE clause. The example illustrates the following characteristics of these clauses:

The ELSIF clauses can have conditions, unlike the ELSE clause.

The condition for ELSIF should be followed by the THEN clause, which is executed if the condition for ELSIF returns TRUE.

When you have multiple ELSIF clauses, if the first condition is FALSE or NULL, the control shifts to the next ELSIF clause.

Conditions are evaluated one by one from the top.

If all conditions are FALSE or NULL, the statements in the ELSE clause are executed.

The final ELSE clause is optional.

In the example, the output of the block is shown below the code.

NULL Value in IF Statement

```
DECLARE
    v_myage  number;
BEGIN
    IF v_myage  < 11 THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' I am not a child ');
    END IF;
END;
/
```

```
anonymous block completed
I am not a child
```

91

In the example shown in the slide, the variable `v_myage` is declared but not initialized. The condition in the `IF` statement returns `NULL` rather than `TRUE` or `FALSE`. In such a case, the control goes to the `ELSE` statement.

Guidelines

You can perform actions selectively based on conditions that are being met.

When you write code, remember the spelling of the keywords:

—`ELSIF` is one word.

—`END IF` is two words.

If the controlling Boolean condition is `TRUE`, the associated sequence of statements is executed; if the controlling Boolean condition is `FALSE` or `NULL`, the associated sequence of statements is passed over. Any number of `ELSIF` clauses is permitted.

Indent the conditionally executed statements for clarity.

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses expressions. The value returned by these expressions is used to select one of several alternatives.

```
CASE selector
    WHEN expression1 THEN result1
    [WHEN expression2 THEN result2
    ...
    WHEN expressionN THEN resultN]
    [ELSE resultN+1]
END;
```

92

A CASE expression returns a result based on one or more alternatives. To return the result, the CASE expression uses a selector, which is an expression whose value is used to return one of several alternatives. The selector is followed by one or more WHEN clauses that are checked sequentially. The value of the selector determines which result is returned. If the value of the selector equals the value of a WHEN clause expression, that WHEN clause is executed and that result is returned.

PL/SQL also provides a searched CASE expression, which has the form:

```
CASE
    WHEN search_condition1 THEN result1
    [WHEN search_condition2 THEN result2
    ...
    WHEN search_conditionN THEN resultN]
    [ELSE resultN+1]
END;
```

A searched CASE expression has no selector. Furthermore, the WHEN clauses in CASE expressions contain search conditions that yield a Boolean value rather than expressions that can yield a value of any type.

CASE Expressions: Example

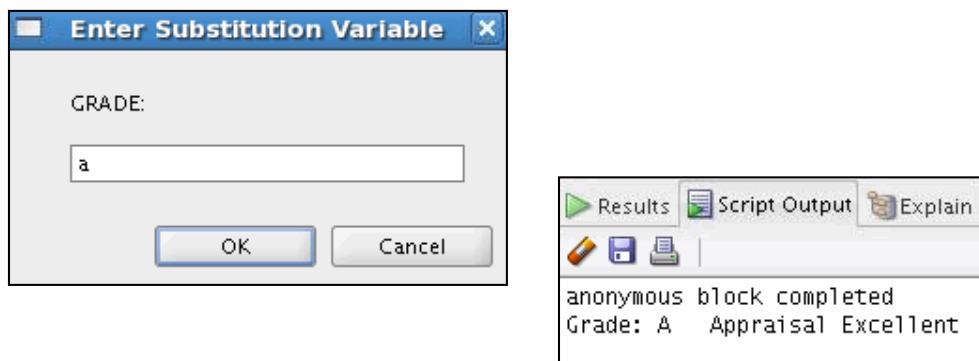
```
SET VERIFY OFF
DECLARE
    v_grade  CHAR(1) := UPPER('&grade');
    v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal := CASE v_grade
        WHEN 'A' THEN 'Excellent'
        WHEN 'B' THEN 'Very Good'
        WHEN 'C' THEN 'Good'
        ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || 
                          ' Appraisal ' || v_appraisal);
END;
/
```

93

In the example in the slide, the CASE expression uses the value in the v_grade variable as the expression. This value is accepted from the user by using a substitution variable. Based on the value entered by the user, the CASE expression returns the value of the v_appraisal variable based on the value of the v_grade value.

Result

When you enter a or A for v_grade, as shown in the Substitution Variable window, the output of the example is as follows:



Searched CASE Expressions

```
DECLARE
    v_grade  CHAR(1) := UPPER('&grade');
    v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal := CASE
        WHEN v_grade = 'A' THEN 'Excellent'
        WHEN v_grade IN ('B','C') THEN 'Good'
        ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || 
                          ' Appraisal ' || v_appraisal);
END;
/
```

94

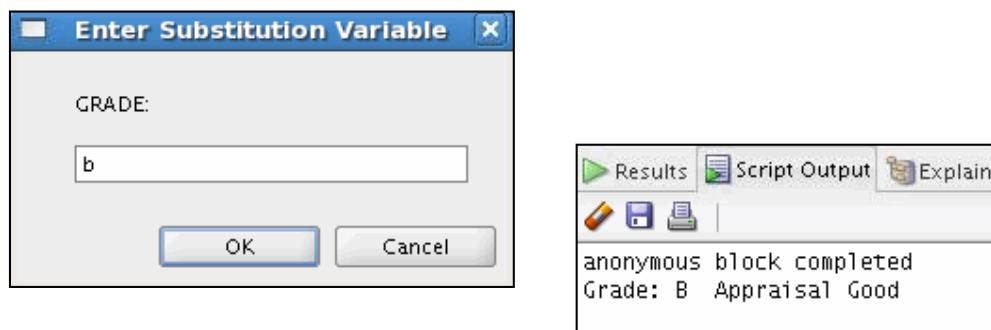
In the previous example, you saw a single test expression, the `v_grade` variable.

The `WHEN` clause compared a value against this test expression.

In searched `CASE` statements, you do not have a test expression. Instead, the `WHEN` clause contains an expression that results in a Boolean value. The same example is rewritten in this slide to show searched `CASE` statements.

Result

The output of the example is as follows when you enter `b` or `B` for `v_grade`:



CASE Statement

```
DECLARE
    v_deptid NUMBER;
    v_deptname VARCHAR2(20);
    v_emps NUMBER;
    v_mngid NUMBER:= 108;
BEGIN
    CASE  v_mngid
    WHEN  108 THEN
        SELECT department_id, department_name
        INTO v_deptid, v_deptname FROM departments
        WHERE manager_id=108;
        SELECT count(*) INTO v_emps FROM employees
        WHERE department_id=v_deptid;
    WHEN  200 THEN
        ...
    END CASE;
    DBMS_OUTPUT.PUT_LINE ('You are working in the'|| v_deptname ||
    ' department. There are'||v_emps ||' employees in this
    department');
END;
/
```

95

Recall the use of the `IF` statement. You may include *n* number of PL/SQL statements in the `THEN` clause and also in the `ELSE` clause. Similarly, you can include statements in the `CASE` statement, which is more readable compared to multiple `IF` and `ELSIF` statements.

How a `CASE Expression` Differs from a `CASE Statement`

A `CASE expression` evaluates the condition and returns a value, whereas a `CASE statement` evaluates the condition and performs an action. A `CASE statement` can be a complete PL/SQL block.

`CASE statements end with END CASE;`

`CASE expressions end with END;`

The output of the slide code example is as follows:

The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The output window displays the following text:

```
anonymous block completed
You are working in the Finance department. There are 6 employees in this department
```

• When you are working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield NULL.
- Applying the logical operator NOT to a null yields NULL.
- If the condition yields NULL in conditional control statements, its associated sequence of statements is not executed.

Consider the following example:

```
x := 5,  
y := NULL;  
...  
IF x != y THEN -- yields NULL, not TRUE  
    -- sequence_of_statements that are not executed  
END IF;
```

You may expect the sequence of statements to execute because `x` and `y` seem unequal. But nulls are indeterminate. Whether or not `x` is equal to `y` is unknown. Therefore, the `IF` condition yields NULL and the sequence of statements is bypassed.

```
a := NULL;  
b := NULL;  
...  
IF a = b THEN -- yields NULL, not TRUE  
    -- sequence_of_statements that are not executed  
END IF;
```

In the second example, you may expect the sequence of statements to execute because `a` and `b` seem equal. But, again, equality is unknown, so the

`IF` condition yields `NULL` and the sequence of statements is bypassed.

Build a simple Boolean condition with a comparison operator.

AND	TRUE	FALSE	NULL	OR	TRUE	FALSE	NULL	NOT	
TRUE	TRUE	FALSE	NULL	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	NULL	FALSE	TRUE
NULL	NULL	FALSE	NULL	NULL	TRUE	NULL	NULL	NULL	NULL

You can build a simple Boolean condition by combining number, character, and date expressions with comparison operators.

You can build a complex Boolean condition by combining simple Boolean conditions with the logical operators AND, OR, and NOT. The logical operators are used to check the Boolean variable values and return TRUE, FALSE, or NULL. In the logic tables shown in the slide:

- FALSE takes precedence in an AND condition, and TRUE takes precedence in an OR condition
- AND returns TRUE only if both of its operands are TRUE
- OR returns FALSE only if both of its operands are FALSE
- NULL AND TRUE always evaluates to NULL because it is not known whether the second operand evaluates to TRUE

Note: The negation of NULL (NOT NULL) results in a null value because null values are indeterminate.

Boolean Expressions or Logical Expression?

What is the value of flag in each case?

```
flag := reorder_flag AND available_flag;
```

REORDER_FLAG	AVAILABLE_FLAG	FLAG
TRUE	TRUE	? (1)
TRUE	FALSE	? (2)
NULL	TRUE	? (3)
NULL	FALSE	? (4)

98

The AND logic table can help you to evaluate the possibilities for the Boolean condition in the statement.

Answers

1. TRUE
2. FALSE
3. NULL
4. FALSE

Iterative Control: *LOOP* Statements

- Loops repeat a statement (or a sequence of statements) multiple times.
- There are three loop types:
 - Basic loop
 - FOR loop
 - WHILE loop



99

PL/SQL provides several facilities to structure loops to repeat a statement or sequence of statements multiple times. Loops are mainly used to execute statements repeatedly until an exit condition is reached. It is mandatory to have an exit condition in a loop; otherwise, the loop is infinite.

Looping constructs are the third type of control structures. PL/SQL provides the following types of loops:

- Basic loop that performs repetitive actions without overall conditions
 - FOR loops that perform iterative actions based on a count
 - WHILE loops that perform iterative actions based on a condition

Note: An EXIT statement can be used to terminate loops. A basic loop must have an EXIT. The cursor FOR loop (which is another type of FOR loop) is discussed in the lesson titled “Using Explicit Cursors.”

Syntax:

```
LOOP  
    statement1;  
    . . .  
    EXIT [WHEN condition];  
END LOOP;
```

100

The simplest form of a `LOOP` statement is the basic loop, which encloses a sequence of statements between the `LOOP` and `END LOOP` keywords. Each time the flow of execution reaches the `END LOOP` statement, control is returned to the corresponding `LOOP` statement above it. A basic loop allows execution of its statements at least once, even if the `EXIT` condition is already met upon entering the loop. Without the `EXIT` statement, the loop would be infinite.

EXIT Statement

You can use the `EXIT` statement to terminate a loop. Control passes to the next statement after the `END LOOP` statement. You can issue `EXIT` either as an action within an `IF` statement or as a stand-alone statement within the loop. The `EXIT` statement must be placed inside a loop. In the latter case, you can attach a `WHEN` clause to enable conditional termination of the loop. When the `EXIT` statement is encountered, the condition in the `WHEN` clause is evaluated. If the condition yields `TRUE`, the loop ends and control passes to the next statement after the loop.

A basic loop can contain multiple `EXIT` statements, but it is recommended that you have only one `EXIT` point.

Basic Loop: Example

```
DECLARE
    v_countryid      locations.country_id%TYPE := 'CA';
    v_loc_id         locations.location_id%TYPE;
    v_counter        NUMBER(2) := 1;
    v_new_city       locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES(v_loc_id + v_counter, v_new_city, v_countryid);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
END LOOP;
END;
/
```

101

The basic loop example shown in the slide is defined as follows: “Insert three new location IDs for the CA country code and the city of Montreal.”

Note

A basic loop allows execution of its statements until the EXIT WHEN condition is met.

If the condition is placed in the loop such that it is not checked until after the loop statements execute, the loop executes at least once.

However, if the exit condition is placed at the top of the loop (before any of the other executable statements) and if that condition is true, the loop exits and the statements never execute.

Results

To view the output, run the code example: `code_06_22_s.sql`.

Syntax:

```
WHILE condition LOOP
    statement1;
    statement2;
    .
    .
    END LOOP;
```

Use the WHILE loop to repeat statements while a condition is TRUE.

102

You can use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE. The condition is evaluated at the start of each iteration. The loop terminates when the condition is FALSE or NULL. If the condition is FALSE or NULL at the start of the loop, no further iterations are performed. Thus, it is possible that none of the statements inside the loop are executed.

In the syntax:

condition Is a Boolean variable or expression (TRUE, FALSE, or NULL)
statement Can be one or more PL/SQL or SQL statements

If the variables involved in the conditions do not change during the body of the loop, the condition remains TRUE and the loop does not terminate.

Note: If the condition yields NULL, the loop is bypassed and control passes to the next statement.

WHILE Loops: Example

```
DECLARE
  v_countryid  locations.country_id%TYPE := 'CA';
  v_loc_id     locations.location_id%TYPE;
  v_new_city   locations.city%TYPE := 'Montreal';
  v_counter    NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

103

In the example in the slide, three new location IDs for the CA country code and the city of Montreal are added.

With each iteration through the WHILE loop, a counter (`v_counter`) is incremented.

If the number of iterations is less than or equal to the number 3, the code within the loop is executed and a row is inserted into the `locations` table.

After `v_counter` exceeds the number of new locations for this city and country, the condition that controls the loop evaluates to FALSE and the loop terminates.

Results

To view the output, run the code example: `code_06_24_s.sql`.

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared **implicitly**.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
        statement1;
        statement2;
        . . .
    END LOOP;
```

104

FOR loops have the same general structure as the basic loop. In addition, they have a control statement before the LOOP keyword to set the number of iterations that the PL/SQL performs.

In the syntax:

<i>counter</i>	Is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached
REVERSE	Causes the counter to decrement with each iteration from the upper bound to the lower bound. Do not declare the counter; it is declared implicitly as an integer.
<i>lower_bound</i>	Note: The lower bound is still referenced first.
<i>upper_bound</i>	Specifies the lower bound for the range of counter values
	Specifies the upper bound for the range of counter values

FOR Loops: Example

```
DECLARE
  v_countryid    locations.country_id%TYPE := 'CA';
  v_loc_id       locations.location_id%TYPE;
  v_new_city     locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id
  FROM locations
  WHERE country_id = v_countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + i), v_new_city, v_countryid );
  END LOOP;
END;
/
```

105

You have already learned how to insert three new locations for the CA country code and the city of Montreal by using the basic loop and the WHILE loop. The example in this slide shows how to achieve the same by using the FOR loop.

Results

To view the output, run the code example `code_06_27_s.sql`.

- Reference the counter only within the loop; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be NULL.

The slide lists the guidelines to follow when writing a FOR loop.

Note: The lower and upper bounds of a LOOP statement do not need to be numeric literals. They can be expressions that convert to numeric values.

Example:

```
DECLARE
    v_lower NUMBER := 1;
    v_upper NUMBER := 100;
BEGIN
    FOR i IN v_lower..v_upper LOOP
        ...
    END LOOP;
END;
/
```

Suggested Use of Loops

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the WHILE loop if the condition must be evaluated at the start of each iteration.
- Use a FOR loop if the number of iterations is known.

107

A basic loop allows the execution of its statement at least once, even if the condition is already met upon entering the loop. Without the EXIT statement, the loop would be infinite.

You can use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE. The condition is evaluated at the start of each iteration. The loop terminates when the condition is FALSE. If the condition is FALSE at the start of the loop, no further iterations are performed.

FOR loops have a control statement before the LOOP keyword to determine the number of iterations that the PL/SQL performs. Use a FOR loop if the number of iterations is predetermined.

- You can nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the EXIT statement that references the label.

You can nest the FOR, WHILE, and basic loops within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception is raised. However, you can label loops and exit the outer loop with the EXIT statement.

Label names follow the same rules as the other identifiers. A label is placed before a statement, either on the same line or on a separate line. White space is insignificant in all PL/SQL parsing except inside literals. Label basic loops by placing the label before the word LOOP within label delimiters (<<label>>). In FOR and WHILE loops, place the label before FOR or WHILE.

If the loop is labeled, the label name can be included (optionally) after the END LOOP statement for clarity.

Nested Loops and Labels: Example

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
    EXIT WHEN v_counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
/
```

109

In the example in the slide, there are two loops. The outer loop is identified by the label <<Outer_Loop>> and the inner loop is identified by the label <<Inner_Loop>>.

The identifiers are placed before the word LOOP within label delimiters (<<label>>). The inner loop is nested within the outer loop. The label names are included after the END LOOP statements for clarity.

- Definition

- Adds the functionality to begin the next loop iteration
- Provides programmers with the ability to transfer control to the next iteration of a loop
- Uses parallel structure and semantics to the EXIT statement

- Benefits

- Eases the programming process
- May provide a small performance improvement over the previous programming workarounds to simulate the CONTINUE statement

110

The CONTINUE statement enables you to transfer control within a loop back to a new iteration or to leave the loop. Many other programming languages have this functionality. With the Oracle Database 19c release, PL/SQL also offers this functionality. Before the Oracle Database 19c release, you could code a workaround by using Boolean variables and conditional statements to simulate the CONTINUE programmatic functionality. In some cases, the workarounds are less efficient.

The CONTINUE statement offers you a simplified means to control loop iterations. It may be more efficient than the previous coding workarounds. The CONTINUE statement is commonly used to filter data within a loop body before the main processing begins.

PL/SQL CONTINUE Statement: Example 1

The screenshot shows the Oracle SQL Developer interface. On the left, there is a code editor window containing PL/SQL code. On the right, there is a results window showing the output of the executed code.

Code (Left):

```
DECLARE
    v_total SIMPLE_INTEGER := 0;
BEGIN
    FOR i IN 1..10 LOOP
        ① v_total := v_total + i;
        dbms_output.put_line
            ('Total is: '|| v_total);
        CONTINUE WHEN i > 5;
        v_total := v_total + i;
        ② dbms_output.put_line
            ('Out of Loop Total is:
             '|| v_total);
    END LOOP;
END;
/
```

Results (Right):

```
anonymous block completed
Total is: 1
Out of Loop Total is:
2
Total is: 4
Out of Loop Total is:
6
Total is: 9
Out of Loop Total is:
12
Total is: 16
Out of Loop Total is:
20
Total is: 25
Out of Loop Total is:
30
Total is: 36
Total is: 43
Total is: 51
Total is: 60
Total is: 70
```

An orange arrow points from the line labeled ① in the code to the first 'Out of Loop' output in the results window.

111

This is a 19c only code. Continue statement was introduced only in 19c.

In the example, there are two assignments using the `v_total` variable:

1. The first assignment is executed for each of the 10 iterations of the loop.
2. The second assignment is executed for the first five iterations of the loop. The `CONTINUE` statement transfers control within a loop back to a new iteration, so for the last five iterations of the loop, the second `TOTAL` assignment is not executed.

The end result of the `TOTAL` variable is 70.

PL/SQL CONTINUE Statement: Example 2

```
DECLARE
  v_total NUMBER := 0;
BEGIN
  <<BeforeTopLoop>>
  FOR i IN 1..10 LOOP
    v_total := v_total + 1;
    dbms_output.put_line
      ('Total is: ' || v_total);
    FOR j IN 1..10 LOOP
      CONTINUE BeforeTopLoop WHEN i + j > 10;
      v_total := v_total + 1;
    END LOOP;
  END LOOP;
END two_loop;
```

The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The output pane displays the following text:
anonymous block completed
Total is: 1
Total is: 6
Total is: 10
Total is: 13
Total is: 15
Total is: 16
Total is: 17
Total is: 18
Total is: 19
Total is: 20

112

You can use the `CONTINUE` statement to jump to the next iteration of an outer loop. This is a 19c only code.

To do this, provide the outer loop a label to identify where the `CONTINUE` statement should go.

The `CONTINUE` statement in the innermost loop terminates that loop whenever the `WHEN` condition is true (just like the `EXIT` keyword). After the innermost loop is terminated by the `CONTINUE` statement, control transfers to the next iteration of the outermost loop labeled `BeforeTopLoop` in this example.

When this pair of loops completes, the value of the `TOTAL` variable is 20.

You can also use the `CONTINUE` statement within an inner block of code, which does not contain a loop as long as the block is nested inside an appropriate outer loop.

Restrictions

The `CONTINUE` statement cannot appear outside a loop at all—this generates a compiler error.

You cannot use the `CONTINUE` statement to pass through a procedure, function, or method boundary—this generates a compiler error.

Topics covered in this module:

- Conditional (IF statement)
- CASE expressions and CASE statements
- Loops:
 - Basic loop
 - FOR loop
 - WHILE loop
- EXIT statement
- CONTINUE statement

113

A language can be called a programming language only if it provides control structures for the implementation of business logic. These control structures are also used to control the flow of the program. PL/SQL is a programming language that integrates programming constructs with SQL.

A conditional control construct checks for the validity of a condition and performs an action accordingly. You use the IF construct to perform a conditional execution of statements.

An iterative control construct executes a sequence of statements repeatedly, as long as a specified condition holds TRUE. You use the various loop constructs to perform iterative operations.



Oracle PL/SQL 19c



Module 06: Composite Data Types

Topics covered in this module:

- Describe PL/SQL collections and records
- Create user-defined PL/SQL records
- Create a PL/SQL record with the %ROWTYPE attribute
- Create associative arrays
 - INDEX BY table
 - INDEX BY table of records

115

You have already been introduced to composite data types. In this lesson, you learn more about composite data types and their uses.

- Can hold multiple values (unlike scalar types)
- Are of two types:
 - PL/SQL records
 - PL/SQL collections
 - Associative array (INDEX BY table)
 - Nested table
 - VARRAY

116

You learned that variables of the scalar data type can hold only one value whereas variables of the composite data type can hold multiple values of the scalar data type or the composite data type. There are two types of composite data types:

PL/SQL records: Records are used to treat related but dissimilar data as a logical unit. A PL/SQL record can have variables of different types. For example, you can define a record to hold employee details. This involves storing an employee number as NUMBER, a first name and last name as VARCHAR2, and so on. By creating a record to store employee details, you create a logical collective unit. This makes data access and manipulation easier.

PL/SQL collections: Collections are used to treat data as a single unit. Collections are of three types:

- Associative array
- Nested table
- VARRAY

Why Use Composite Data Types?

You have all the related data as a single unit. You can easily access and modify data. Data is easier to manage, relate, and transport if it is composite. An

analogy is having a single bag for all your laptop components rather than a separate bag for each component.

PL/SQL Records or Collections?

- Use PL/SQL records when you want to store values of different data types but only one occurrence at a time.
- Use PL/SQL collections when you want to store values of the same data type.

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	
------	-----------	---------	---

PL/SQL collection:

1	SMITH
2	JONES
3	BENNETT
4	KRAMER

↓ ↓
PLS_INTEGER VARCHAR2

117

If both PL/SQL records and PL/SQL collections are composite types, how do you choose which one to use?

Use PL/SQL records when you want to store values of different data types that are logically related. For example, you can create a PL/SQL record to hold employee details and indicate that all the values stored are related because they provide information about a particular employee.

Use PL/SQL collections when you want to store values of the same data type. Note that this data type can also be of the composite type (such as records). You can define a collection to hold the first names of all employees. You may have stored n names in the collection; however, name 1 is not related to name 2. The relation between these names is only that they are employee names. These collections are similar to arrays in programming languages such as C, C++, and Java.

- Must contain one or more components (called *fields*) of any scalar, RECORD, or INDEX BY table data type
- Are similar to structures in most third-generation languages (including C and C++)
- Are user-defined and can be a subset of a row in a table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing

118

A record is a group of related data items stored in fields, each with its own name and data type.

Each record defined can have as many fields as necessary.

Records can be assigned initial values and can be defined as NOT NULL.

Fields without initial values are initialized to NULL.

The DEFAULT keyword as well as := can be used in initializing fields. You can define RECORD types and declare user-defined records in the declarative part of any block, subprogram, or package.

You can declare and reference nested records. One record can be the component of another record.

Syntax:

1 `TYPE type_name IS RECORD
 (field_declaration[, field_declaration]...);`

2 `identifier type_name;`

field_declaration:

`field_name {field_type | variable%TYPE
 | table.column%TYPE | table%ROWTYPE}
 [NOT NULL] {:= | DEFAULT} expr`

119

PL/SQL records are user-defined composite types. To use them, perform the following steps.

1. Define the record in the declarative section of a PL/SQL block. The syntax for defining the record is shown in the slide.
2. Declare (and optionally initialize) the internal components of this record type.

In the syntax:

`type_name` Is the name of the RECORD type (This identifier is used to declare records.)

`field_name` Is the name of a field within the record

`field_type` Is the data type of the field (It represents any PL/SQL data type except REF CURSOR. You can

use the %TYPE and %ROWTYPE attributes.)

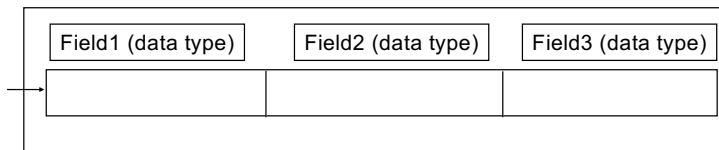
`expr` Is the initial value

The NOT NULL constraint prevents assigning of nulls to the specified fields.

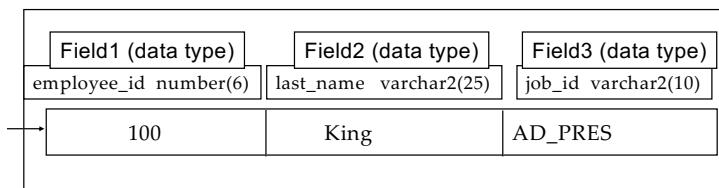
Be sure to initialize the NOT NULL fields.

PL/SQL Record Structure

Field declarations:



Example:



120

Fields in a record are accessed with the name of the record. To reference or initialize an individual field, use the dot notation:

record_name.field_name

For example, you reference the `job_id` field in the `emp_record` record as follows:

`emp_record.job_id`

You can then assign a value to the record field:

`emp_record.job_id := 'ST_CLERK';`

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram. They cease to exist when you exit the block or subprogram.

%ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE  
    identifier reference%ROWTYPE;
```

121

You learned that %TYPE is used to declare a variable of the column type. The variable has the same data type and size as the table column. The benefit of %TYPE is that you do not have to change the variable if the column is altered. Also, if the variable is a number and is used in any calculations, you need not worry about its precision.

The %ROWTYPE attribute is used to declare a record that can hold an entire row of a table or view. The fields in the record take their names and data types from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

The slide shows the syntax for declaring a record. In the syntax:

identifier Is the name chosen for the record as a whole

reference Is the name of the table, view, cursor, or cursor variable on which the record is to be based (The table or view must exist for

In the following example a record is declared using %ROWTYPE as a data type specifier:

```
DECLARE
```

```
emp_record employees%ROWTYPE;
```

```
...
```

Creating a PL/SQL Record: Example

```
DECLARE
  TYPE t_rec IS RECORD
    (v_sal number(8),
     v_minsal number(8) default 1000,
     v_hire_date employees.hire_date%type,
     v_rec1 employees%rowtype);
  v_myrec t_rec;
BEGIN
  v_myrec.v_sal := v_myrec.v_minsal + 500;
  v_myrec.v_hire_date := sysdate;
  SELECT * INTO v_myrec.v_rec1
    FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name || ' ' ||
    to_char(v_myrec.v_hire_date) || ' ' || to_char(v_myrec.v_sal));
END;
```

```
anonymous block completed
King 16-FEB-09 1500
```

122

The field declarations used in defining a record are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the record type first, and then declare an identifier using that type.

In the example in the slide, a PL/SQL record is created using the required two-step process:

1. A record type (`t_rec`) is defined
2. A record (`v_myrec`) of the `t_rec` type is declared

Note

The record contains four fields: `v_sal`, `v_minsal`, `v_hire_date`, and `v_rec1`.

`v_rec1` is defined using the `%ROWTYPE` attribute, which is similar to the `%TYPE` attribute. With `%TYPE`, a field inherits the data type of a specified column. With `%ROWTYPE`, a field inherits the column names and data types of all columns in the referenced table.

PL/SQL record fields are referenced using the `<record>. <field>` notation, or the `<record>. <field>. <column>` notation for fields that are defined with the `%ROWTYPE` attribute.

You can add the NOT NULL constraint to any field declaration to prevent assigning nulls to that field. Remember that fields that are declared as NOT NULL must be initialized.

Advantages of Using the %ROWTYPE Attribute

- The number and data types of the underlying database columns need not be known—and, in fact, might change at run time.
- The %ROWTYPE attribute is useful when you want to retrieve a row with:
 - The SELECT * statement
 - Row-level INSERT and UPDATE statements

123

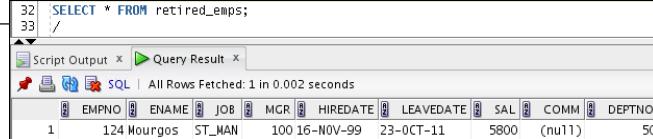
The advantages of using the %ROWTYPE attribute are listed in the slide. Use the %ROWTYPE attribute when you are not sure about the structure of the underlying database table.

The main advantage of using %ROWTYPE is that it simplifies maintenance. Using %ROWTYPE ensures that the data types of the variables declared with this attribute change dynamically when the underlying table is altered. If a DDL statement changes the columns in a table, the PL/SQL program unit is invalidated. When the program is recompiled, it automatically reflects the new table format.

The %ROWTYPE attribute is particularly useful when you want to retrieve an entire row from a table. In the absence of this attribute, you would be forced to declare a variable for each of the columns retrieved by the SELECT statement.

Another %ROWTYPE Attribute Example

```
DECLARE
    v_employee_number number:= 124;
    v_emp_rec    employees%ROWTYPE;
BEGIN
    SELECT * INTO v_emp_rec FROM employees
    WHERE employee_id = v_employee_number;
    INSERT INTO retired_emps(empno, ename, job, mgr,
                           hiredate, leavedate, sal, comm, deptno)
        VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
                v_emp_rec.job_id, v_emp_rec.manager_id,
                v_emp_rec.hire_date, SYSDATE,
                v_emp_rec.salary, v_emp_rec.commission_pct,
                v_emp_rec.department_id);
END;
/
32: SELECT * FROM retired_emps;
33: /
```



124

Another example of the %ROWTYPE attribute is shown in the slide. If an employee is retiring, information about that employee is added to a table that holds information about retired employees. The user supplies the employee number. The record of the employee specified by the user is retrieved from the employees table and stored in the emp_rec variable, which is declared using the %ROWTYPE attribute.

The CREATE statement that creates the retired_emps table is:

```
CREATE TABLE retired_emps
(EMPNO    NUMBER(4), ENAME   VARCHAR2(10),
 JOB      VARCHAR2(9), MGR     NUMBER(4),
 HIREDATE DATE, LEAVEDATE DATE,
 SAL      NUMBER(7,2), COMM    NUMBER(7,2),
 DEPTNO   NUMBER(2))
```

Note

The record that is inserted into the retired_emps table is shown in the slide.

To see the output shown in the slide, place your cursor on the SELECT statement at the bottom of the code example in SQL Developer and press F9.

The complete code example is found in `code_07_14_n-s.sql`.

Inserting a Record by Using %ROWTYPE

```
...
DECLARE
    v_employee_number number:= 124;
    v_emp_rec retired_emps%ROWTYPE;
BEGIN
    SELECT employee_id, last_name, job_id, manager_id,
    hire_date, hire_date, salary, commission_pct,
    department_id INTO v_emp_rec FROM employees
    WHERE employee_id = v_employee_number;
    INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
SELECT * FROM retired_emps;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124 Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

125

Compare the `INSERT` statement in the previous slide with the `INSERT` statement in this slide. The `emp_rec` record is of type `retired_emps`. The number of fields in the record must be equal to the number of field names in the `INTO` clause. You can use this record to insert values into a table. This makes the code more readable.

Examine the `SELECT` statement in the slide. You select `hire_date` twice and insert the `hire_date` value in the `leavedate` field of `retired_emps`. No employee retires on the hire date. The inserted record is shown in the slide. (You will see how to update this in the next slide.)

Note: To see the output shown in the slide, place your cursor on the `SELECT` statement at the bottom of the code example in SQL Developer and press F9.

Updating a Row in a Table by Using a Record

```
SET VERIFY OFF
DECLARE
    v_employee_number number:= 124;
    v_emp_rec retired_emps%ROWTYPE;
BEGIN
    SELECT * INTO v_emp_rec FROM retired_emps where
    v_emp_rec.leavedate:=CURRENT_DATE;
    UPDATE retired_emps SET ROW = v_emp_rec WHERE
    empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124 Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

126

You learned to insert a row by using a record. This slide shows you how to update a row by using a record.

The ROW keyword is used to represent the entire row.

The code shown in the slide updates the leavedate of the employee.

The record is updated as shown in the slide.

Note: To see the output shown in the slide, place your cursor on the SELECT statement at the bottom of the code example in SQL Developer and press F9.

Associative Arrays (*INDEX BY Tables*)

An associative array is a PL/SQL collection with two columns:

- Primary key of integer or string data type
- Column of scalar or record data type

Key	Values
1	JONES
2	HARDEY
3	MADURO
4	KRAMER

127

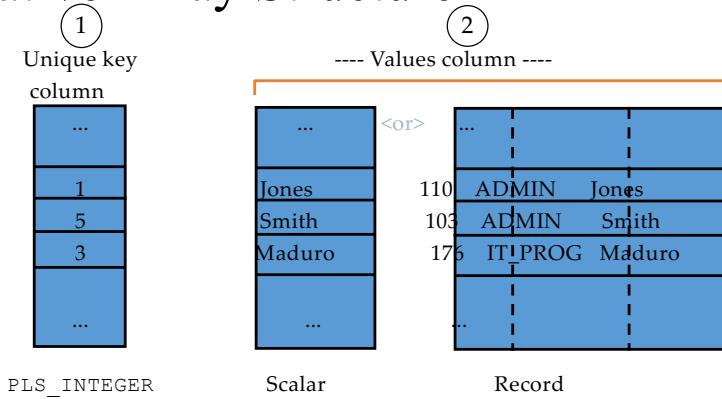
An associative array is a type of PL/SQL collection. It is a composite data type and is user-defined. Associative arrays are sets of key-value pairs. They can store data using a primary key value as the index, where the key values are not necessarily sequential. Associative arrays are also known as *INDEX BY* tables.

Associative arrays have only two columns, neither of which can be named:

The first column, of integer or string type, acts as the primary key.

The second column, of scalar or record data type, holds values.

Associative Array Structure



128

As previously mentioned, associative arrays have two columns. The second column either holds one value per row, or multiple values.

Unique Key Column: The data type of the key column can be:

Numeric, either BINARY_INTEGER or PLS_INTEGER. These two numeric data types require less storage than NUMBER, and arithmetic operations on these data types are faster than the NUMBER arithmetic.

VARCHAR2 or one of its subtypes

“Value” Column: The value column can be either a scalar data type or a record data type. A column with scalar data type can hold only one value per row, whereas a column with record data type can hold multiple values per row.

Other Characteristics

An associative array is not populated at the time of declaration. It contains no keys or values, and you cannot initialize an associative array in its declaration.

An explicit executable statement is required to populate the associative array.

Like the size of a database table, the size of an associative array is

unconstrained. That is, the number of rows can increase dynamically so that your associative array grows as new rows are added. Note that the keys do not have to be sequential, and can be both positive and negative.

Syntax:

```

1   TYPE type_name IS TABLE OF
    { column_type [NOT NULL] | variable%TYPE [NOT NULL]
    | table.column%TYPE [NOT NULL]
    | table%ROWTYPE }
    INDEX BY { PLS_INTEGER | BINARY_INTEGER
    | VARCHAR2(<size>) } ;
2   identifier type_name;

```

Ex

```

...
TYPE ename_table_type IS TABLE OF
employees.last_name%TYPE
INDEX BY PLS_INTEGER;
...
ename_table ename_table_type;

```

129

There are two steps involved in creating an associative array:

1. Declare a TABLE data type using the INDEX BY option.
2. Declare a variable of that data type.

Syntax

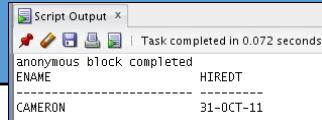
type_name Is the name of the TABLE type (This name is used in the subsequent declaration of the array identifier.)

column_type Is any scalar or composite data type such as VARCHAR2, DATE,
Note: Then NOT NULL (You can use the NOT NULL attribute to provide constraints on the column data type.)

identifier Example Is the name of the identifier that represents an entire associative array. In the example, an associative array with the variable name *ename_table* is declared to store the last names of employees.

Creating and Accessing Associative Arrays

```
...
•DECLARE
•  TYPE ename_table_type IS TABLE OF
•    employees.last_name%TYPE
•    INDEX BY PLS_INTEGER;
•  TYPE hiredate_table_type IS TABLE OF DATE
•    INDEX BY PLS_INTEGER;
•  ename_table    ename_table_type;
•  hiredate_table    hiredate_table_type;
•BEGIN
•  ename_table(1)    := 'CAMERON';
•  hiredate_table(8) := SYSDATE + 7;
•  IF ename_table.EXISTS(1) THEN
•    INSERT INTO ...
•  ...
•END;
•/
•...
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the message 'Task completed in 0.072 seconds' and the output of an anonymous block. The output table has columns 'ENAME' and 'HIREDT'. A single row is shown: CAMERON with a hire date of 31-OCT-11.

ENAME	HIREDT
CAMERON	31-OCT-11

130

The example in the slide creates two associative arrays, with the identifiers `ename_table` and `hiredate_table`.

The key of each associative array is used to access an element in the array, by using the following syntax:

identifier(index)

In both arrays, the index value belongs to the `PLS_INTEGER` type.

To reference the first row in the `ename_table` associative array, specify: `ename_table(1)`

To reference the eighth row in the `hiredate_table` associative array, specify: `hiredate_table(8)`

Note

The magnitude range of a `PLS_INTEGER` is $-2,147,483,647$ through $2,147,483,647$, so the primary key value can be negative. Indexing does not need to start with 1.

The `exists (i)` method returns `TRUE` if a row with index i is returned. Use the `exists` method to prevent an error that is raised in reference to a nonexistent table element.

The complete code example is found in `code_07_21_s.sql`.

Using INDEX BY Table Methods

The following methods make associative arrays easier to use:

- EXISTS
- COUNT
- FIRST
- LAST
- PRIOR
- NEXT
- DELETE

131

An INDEX BY table method is a built-in procedure or function that operates

on an associative array and is called by using the dot notation.

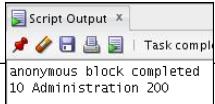
Syntax: `table_name.method_name[(parameters)]`

Method	Description
EXISTS (<i>n</i>)	Returns TRUE if the index <i>n</i> in an associative array exists
COUNT	Returns the number of elements that an associative array currently contains
FIRST	<ul style="list-style-type: none"> • Returns the first (smallest) index number in an associative array • Returns NULL if the associative array is empty
LAST	<ul style="list-style-type: none"> • Returns the last (largest) index number in an associative array • Returns NULL if the associative array is empty
PRIOR (<i>n</i>)	Returns the index number that precedes index <i>n</i> in an associative array
NEXT (<i>n</i>)	Returns the index number that succeeds index <i>n</i> in an associative array
DELETE	<ul style="list-style-type: none"> • DELETE removes all elements from an associative array. • DELETE (<i>n</i>) removes the index <i>n</i> from an associative array. • DELETE (<i>m</i>, <i>n</i>) removes all elements in the range <i>m</i> ... <i>n</i> from an associative array.

INDEX BY Table of Records Option

Define an associative array to hold an entire row from a table.

```
DECLARE
    TYPE dept_table_type IS TABLE OF
        departments%ROWTYPE INDEX PLS_INTEGER;
    dept_table dept_table_type;
    -- Each element of dept_table is a record
Begin
    SELECT * INTO dept_table(1) FROM departments
        WHERE department_id = 10;
    DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id || ||
        dept_table(1).department_name || ||
        dept_table(1).manager_id);
END;
/
```



132

As previously discussed, an associative array that is declared as a table of scalar data type can store the details of only one column in a database table. However, there is often a need to store all the columns retrieved by a query. The INDEX BY table of records option enables one array definition to hold information about all the fields of a database table.

Creating and Referencing a Table of Records

As shown in the associative array example in the slide, you can:

- Use the %ROWTYPE attribute to declare a record that represents a row in a database table
- Refer to fields within the dept_table array because each element of the array is a record

The differences between the %ROWTYPE attribute and the composite data type PL/SQL record are as follows:

- PL/SQL record types can be user-defined, whereas %ROWTYPE implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use %ROWTYPE, you cannot specify the fields. The %ROWTYPE attribute represents a table row with all the

fields based on the definition of that table.

User-defined records are static, but `%ROWTYPE` records are dynamic—they are based on a table structure. If the table structure changes, the record structure also picks up the change.

INDEX BY Table of Records Option: Example 2

```
DECLARE
    TYPE emp_table_type IS TABLE OF
        employees%ROWTYPE INDEX BY PLS_INTEGER;
    my_emp_table emp_table_type;
    max_count      NUMBER(3):= 104;
BEGIN
    FOR i IN 100..max_count
    LOOP
        SELECT * INTO my_emp_table(i) FROM employees
        WHERE employee_id = i;
    END LOOP;
    FOR i IN my_emp_table.FIRST..my_emp_table.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
    END LOOP;
END;
/
```

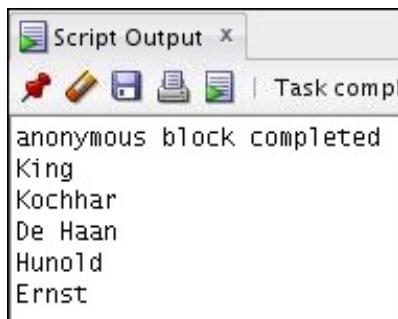
133

The example in the slide declares an associative array, using the INDEX BY table of records option, to temporarily store the details of employees whose employee IDs are between 100 and 104. The variable name for the array is `emp_table_type`.

Using a loop, the information of the employees from the `EMPLOYEES` table is retrieved and stored in the array. Another loop is used to print the last names from the array. Note the use of the `first` and `last` methods in the example.

Note: The slide demonstrates one way to work with an associative array that uses the INDEX BY table of records method. However, you can do the same more efficiently using cursors. Cursors are explained in the lesson titled “Using Explicit Cursors.”

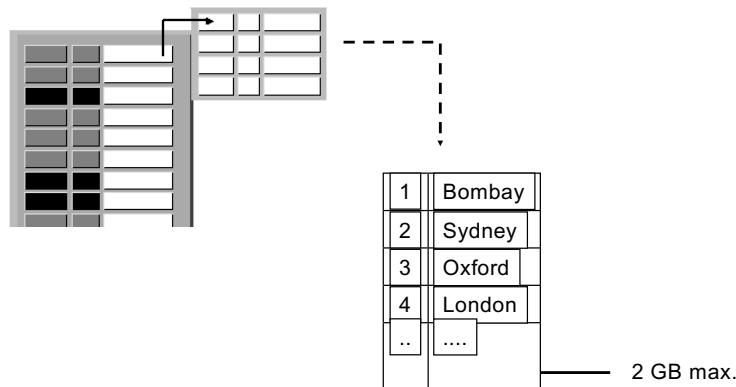
The results of the code example are as follows:



The screenshot shows the 'Script Output' window in Oracle SQL Developer. The window title is 'Script Output'. It contains the message 'anonymous block completed' followed by a list of last names: King, Kochhar, De Haan, Hunold, and Ernst.

```
anonymous block completed
King
Kochhar
De Haan
Hunold
Ernst
```

Nested Tables



134

The functionality of nested tables is similar to that of associative arrays; however, there are differences in the nested table implementation.

The nested table is a valid data type in a schema-level table, but an associative array is not. Therefore, unlike associative arrays, nested tables can be stored in the database.

The size of a nested table can increase dynamically, although the maximum size is 2 GB.

The “key” cannot be a negative value (unlike in the associative array). Though reference is made to the first column as key, there is no key in a nested table. There is a column with numbers.

Elements can be deleted from anywhere in a nested table, leaving a sparse table with nonsequential “keys.” The rows of a nested table are not in any particular order.

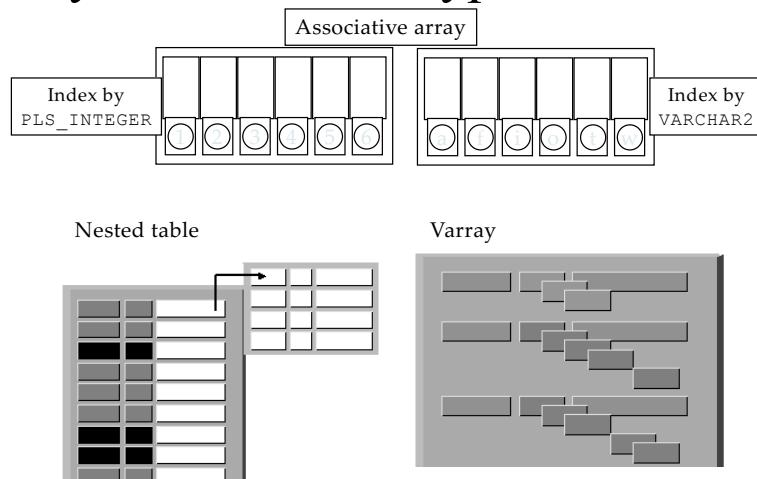
When you retrieve values from a nested table, the rows are given consecutive subscripts starting from 1.

Syntax

```
TYPE type_name IS TABLE OF  
{column_type | variable%TYPE}
```

| table.column%TYPE} [NOT NULL]
| table.%ROWTYPE

Summary of Collection Types



135

Associative Arrays

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be either integer- or character-based. The array value may be of the scalar data type (single value) or the record data type (multiple values).

Because associative arrays are intended for storing temporary data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`.

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, thereby allowing varrays to be stored in database tables.

Topics covered in this module:

- Define and reference PL/SQL variables of composite data types
 - PL/SQL record
 - Associative array
 - INDEX BY table
 - INDEX BY table of records
- Define a PL/SQL record by using the %ROWTYPE attribute
- Compare and contrast the three PL/SQL collection types:
 - Associative array
 - Nested table
 - VARRAY

136

A PL/SQL record is a collection of individual fields that represent a row in a table. By using records, you can group the data into one structure, and then manipulate this structure as one entity or logical unit. This helps reduce coding and keeps the code easy to maintain and understand.

Like PL/SQL records, a PL/SQL collection is another composite data type.

PL/SQL collections include:

Associative arrays (also known as INDEX BY tables). They are objects of TABLE type and look similar to database tables, but with a slight difference. The so-called INDEX BY tables use a primary key to give you array-like access to rows. The size of an associative array is unconstrained.

Nested tables. The key for nested tables cannot have a negative value, unlike INDEX BY tables. The key must also be in a sequence.

Variable-size arrays (VARRAY). A VARRAY is similar to associative arrays, except that a VARRAY is constrained in size.



Oracle PL/SQL 19c



Module 07: Explicit Cursors

Topics covered in this module:

- Distinguish between implicit and explicit cursors
- Discuss the reasons for using explicit cursors
- Declare and control explicit cursors
- Use simple loops and cursor FOR loops to fetch data
- Declare and use cursors with parameters
- Lock rows with the FOR UPDATE clause
- Reference the current row with the WHERE CURRENT OF clause

138

You have learned about implicit cursors that are automatically created by PL/SQL when you execute a SQL SELECT or DML statement. In this lesson, you learn about explicit cursors. You learn to differentiate between implicit and explicit cursors. You also learn to declare and control simple cursors, as well as cursors with parameters.

Every SQL statement that is executed by the Oracle Server has an associated individual cursor:

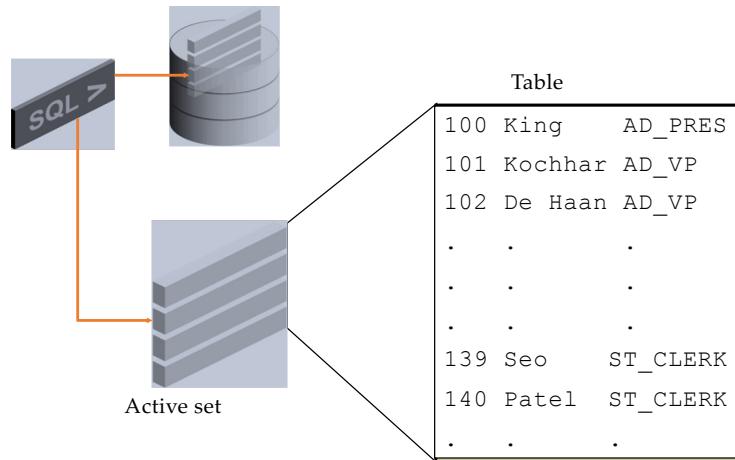
- Implicit cursors: declared and managed by PL/SQL for all DML and PL/SQL SELECT statements
- Explicit cursors: declared and managed by the programmer

139

The Oracle Server uses work areas (called private SQL areas) to execute SQL statements and to store processing information. You can use explicit cursors to name a private SQL area and to access its stored information.

Cursor Type	Description
Implicit	The Oracle Server implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor. Using PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor.
Explicit	For queries that return multiple rows, explicit cursors are declared and managed by the programmer, and manipulated through specific statements in the block's executable actions.

Explicit Cursor Operations



140

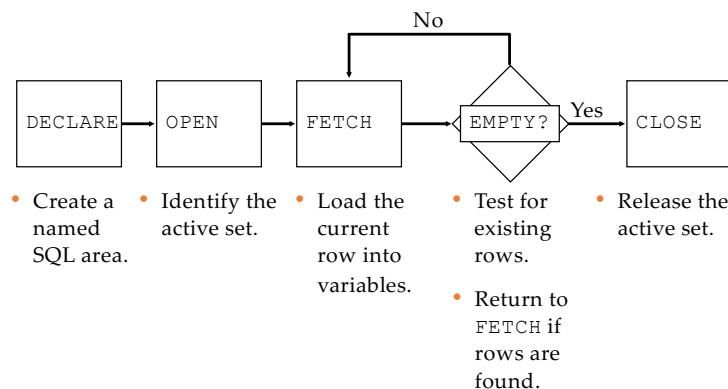
You declare explicit cursors in PL/SQL when you have a `SELECT` statement that returns multiple rows. You can process each row returned by the `SELECT` statement.

The set of rows returned by a multiple-row query is called the *active set*. Its size is the number of rows that meet your search criteria. The diagram in the slide shows how an explicit cursor “points” to the current row in the active set. This enables your program to process the rows one at a time.

Explicit cursor functions:

- Can perform row-by-row processing beyond the first row returned by a query
- Keep track of the row that is currently being processed
- Enable the programmer to manually control explicit cursors in the PL/SQL block

Controlling Explicit Cursors



141

Now that you have a conceptual understanding of cursors, review the steps to use them.

1. In the declarative section of a PL/SQL block, declare the cursor by naming it and defining the structure of the query to be associated with it.

2. Open the cursor.

The OPEN statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.

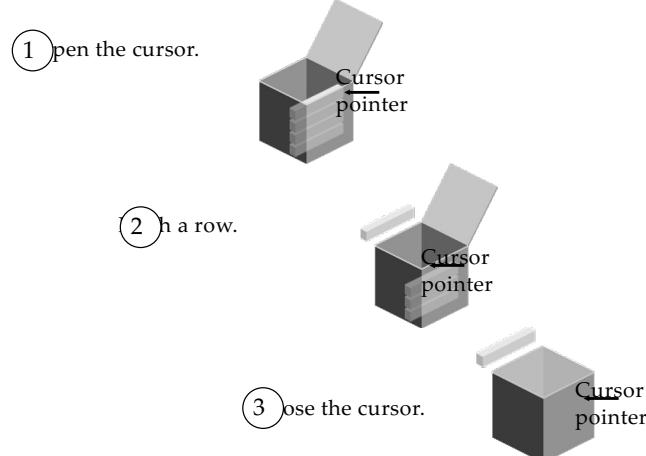
3. Fetch data from the cursor.

In the flow diagram shown in the slide, after each fetch, you test the cursor for any existing row. If there are no more rows to process, you must close the cursor.

4. Close the cursor.

The CLOSE statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Controlling Explicit Cursors



142

A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in the active set.

1. The `OPEN` statement executes the query associated with the cursor, identifies the active set, and positions the cursor at the first row.
2. The `FETCH` statement retrieves the current row and advances the cursor to the next row until there are no more rows or a specified condition is met.
3. The `CLOSE` statement releases the cursor.

Syntax:

```
CURSOR cursor_name IS  
    select_statement;
```

```
DECLARE  
    CURSOR c_emp_cursor IS  
        SELECT employee_id, last_name FROM employees  
        WHERE department_id = 30;
```

```
DECLARE  
    v_locid NUMBER:= 1700;  
    CURSOR c_dept_cursor IS  
        SELECT * FROM departments  
        WHERE location_id = v_locid;  
    ...
```

143

The syntax to declare a cursor is shown in the slide. In the syntax:

cursor_name is a PL/SQL identifier

select_statement Is a SELECT statement without an INTO clause

The active set of a cursor is determined by the SELECT statement in the cursor declaration. It is mandatory to have an INTO clause for a SELECT statement in PL/SQL. However, note that the SELECT statement in the cursor declaration cannot have an INTO clause. That is because you are only defining a cursor in the declarative section and not retrieving any rows into the cursor.

Note

Do not include the INTO clause in the cursor declaration because it appears later in the FETCH statement.

If you want the rows to be processed in a specific sequence, use the ORDER BY clause in the query.

The cursor can be any valid SELECT statement, including joins, subqueries, and so on.

Opening the Cursor

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
    ...
BEGIN
    OPEN c_emp_cursor;
```

144

The OPEN statement executes the query associated with the cursor, identifies the active set, and positions the cursor pointer at the first row. The OPEN statement is included in the executable section of the PL/SQL block.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area
2. Parses the SELECT statement
3. Binds the input variables (sets the values for the input variables by obtaining their memory addresses)
4. Identifies the active set (the set of rows that satisfy the search criteria). Rows in the active set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows from the cursor to the variables.
5. Positions the pointer to the first row in the active set

Note: If a query returns no rows when the cursor is opened, PL/SQL does not raise an exception. You can find out the number of rows returned with an explicit cursor by using the <cursor_name>%ROWCOUNT attribute.

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =30;
    v_empno employees.employee_id%TYPE;
    v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  FETCH c_emp_cursor INTO v_empno, v_lname;
  DBMS_OUTPUT.PUT_LINE( v_empno || ' '||v_lname);
END;
/
```

```
anonymous block completed
114 Raphaely
```

145

The `FETCH` statement retrieves the rows from the cursor one at a time. After each fetch, the cursor advances to the next row in the active set. You can use the `%NOTFOUND` attribute to determine whether the entire active set has been retrieved.

Consider the example shown in the slide. Two variables, `empno` and `lname`, are declared to hold the fetched values from the cursor. Examine the `FETCH` statement.

You have successfully fetched the values from the cursor to the variables. However, there are six employees in department 30, but only one row was fetched. To fetch all rows, you must use loops. In the next slide, you see how a loop is used to fetch all the rows.

The `FETCH` statement performs the following operations:

1. Reads the data for the current row into the output PL/SQL variables
2. Advances the pointer to the next row in the active set

Fetching Data from the Cursor

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
        v_empno employees.employee_id%TYPE;
        v_lname employees.last_name%TYPE;
BEGIN
    OPEN c_emp_cursor;
    LOOP
        FETCH c_emp_cursor INTO v_empno, v_lname;
        EXIT WHEN c_emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( v_empno ||' '||v_lname);
    END LOOP;
END;
/
```

146

Observe that a simple LOOP is used to fetch all the rows. Also, the cursor attribute %NOTFOUND is used to test for the exit condition. The output of the PL/SQL block is:

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

Closing the Cursor

```
...
LOOP
    FETCH c_emp_cursor INTO empno, lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno || ' ' || v_lname);
END LOOP;
CLOSE c_emp_cursor;
END;
/
```

147

The CLOSE statement disables the cursor, releases the context area, and “undefines” the active set. Close the cursor after completing the processing of the FETCH statement. You can reopen the cursor if required. A cursor can be reopened only if it is closed. If you attempt to fetch data from a cursor after it is closed, an INVALID_CURSOR exception is raised.

Note: Although it is possible to terminate the PL/SQL block without closing cursors, you should make it a habit to close any cursor that you declare explicitly to free resources.

There is a maximum limit on the number of open cursors per session, which is determined by the OPEN_CURSORS parameter in the database parameter file. (OPEN_CURSORS = 50 by default.)

Process the rows of the active set by fetching values into a PL/SQL record.

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
        v_emp_record  c_emp_cursor%ROWTYPE;
BEGIN
    OPEN c_emp_cursor;
    LOOP
        FETCH c_emp_cursor INTO v_emp_record;
        EXIT WHEN c_emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                            ||' '||v_emp_record.last_name);
    END LOOP;
    CLOSE c_emp_cursor;
END;
```

148

You have already seen that you can define records that have the structure of columns in a table. You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set, because you can simply fetch into the record. Therefore, the values of the rows are loaded directly into the corresponding fields of the record.

```
anonymous block completed
114 Raphaelly
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

Syntax:

```
FOR record_name IN cursor_name LOOP  
    statement1;  
    statement2;  
    . . .  
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

149

You learned to fetch data from cursors by using simple loops. You now learn to use a cursor FOR loop, which processes rows in an explicit cursor. It is a shortcut because the cursor is opened, a row is fetched once for each iteration in the loop, the loop exits when the last row is processed, and the cursor is closed automatically. The loop itself is terminated automatically at the end of the iteration where the last row is fetched.

In the syntax:

<i>record_name</i>	Is the name of the implicitly declared record
<i>cursor_name</i>	Is a PL/SQL identifier for the previously declared cursor

Guidelines

Do not declare the record that controls the loop; it is declared implicitly.

Test the cursor attributes during the loop if required.

Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement.

Cursor FOR Loops

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
    BEGIN
        FOR emp_record IN c_emp_cursor
        LOOP
            DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
            ||' '||emp_record.last_name);
        END LOOP;
    END;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

150

The example that was used to demonstrate the usage of a simple loop to fetch data from cursors is rewritten to use the cursor FOR loop.

`emp_record` is the record that is implicitly declared. You can access the fetched data with this implicit record (as shown in the slide). Observe that no variables are declared to hold the fetched data using the `INTO` clause. The code does not have the `OPEN` and `CLOSE` statements to open and close the cursor, respectively.

Explicit Cursor Attributes

Use explicit cursor attributes to obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows that's been fetched.

151

As with implicit cursors, there are four attributes for obtaining the status information of a cursor. When appended to the cursor variable name, these attributes return useful information about the execution of a cursor manipulation statement.

Note: You cannot reference cursor attributes directly in a SQL statement.

- You can fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

```
IF NOT c_emp_cursor%ISOPEN THEN
    OPEN c_emp_cursor;
END IF;
LOOP
    FETCH c_emp_cursor...
```

152

You can fetch rows only when the cursor is open. Use the %ISOPEN cursor attribute to determine whether the cursor is open.

Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.

Use the %ROWCOUNT cursor attribute to do the following:

Process an exact number of rows.

Fetch the rows in a loop and determine when to exit the loop.

Note: %ISOPEN returns the status of the cursor: TRUE if open and FALSE if not.

%ROWCOUNT and %NOTFOUND: Example

```
DECLARE
  CURSOR c_emp_cursor IS SELECT employee_id,
    last_name FROM employees;
  v_emp_record  c_emp_cursor%ROWTYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_emp_record;
    EXIT WHEN c_emp_cursor%ROWCOUNT > 10 OR
      c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
      || '-' || v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END ; /
```

anonymous block completed
174 Abel
166 Ande
130 Atkinson
105 Austin
204 Baer
116 Baida
167 Banda
172 Bates
192 Bell
151 Bernstein

153

The example in the slide retrieves the first 10 employees one by one. This example shows how the %ROWCOUNT and %NOTFOUND attributes can be used for exit conditions in a loop.

Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

```
BEGIN
    FOR emp_record IN (SELECT employee_id, last_name
                        FROM employees WHERE department_id =30)
    LOOP
        DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
                           || ' ' || emp_record.last_name);
    END LOOP;
END;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

154

Note that there is no declarative section in this PL/SQL block. The difference between the cursor FOR loops using subqueries and the cursor FOR loops lies in the cursor declaration. If you are writing cursor FOR loops using subqueries, you need not declare the cursor in the declarative section. You have to provide the SELECT statement that determines the active set in the loop itself.

The example that was used to illustrate a cursor FOR loop is rewritten to illustrate a cursor FOR loop using subqueries.

Note: You cannot reference explicit cursor attributes if you use a subquery in a cursor FOR loop because you cannot give the cursor an explicit name.

Syntax:

```
CURSOR cursor_name  
[(parameter_name datatype, ...)]  
IS  
    select_statement;
```

- Pass parameter values to a cursor when the cursor is opened and the query is executed.

- Open an explicit cursor several times with a

```
OPEN cursor_name(parameter_value,.....) ;
```

155

You can pass parameters to a cursor. This means that you can open and close an explicit cursor several times in a block, returning a different active set on each occasion. For each execution, the previous cursor is closed and reopened with a new set of parameters.

Each formal parameter in the cursor declaration must have a corresponding actual parameter

in the OPEN statement. Parameter data types are the same as those for scalar variables, but

you do not give them sizes. The parameter names are for reference in the query expression of the cursor.

In the syntax:

cursor_name Is a PL/SQL identifier for the declared cursor

parameter_name Is the name of a parameter

datatype Is the scalar data type of the parameter

The parameter notation does not offer greater functionality; it simply allows *select_statement*, Is a SELECT statement without the INTO clause you to specify input values easily and clearly. This is particularly useful when the same cursor is referenced repeatedly.

Cursors with Parameters

```
DECLARE
  CURSOR c_emp_cursor (deptno NUMBER) IS
    SELECT employee_id, last_name
    FROM employees
    WHERE department_id = deptno;
  ...
BEGIN
  OPEN c_emp_cursor (10);
  ...
  CLOSE c_emp_cursor;
  OPEN c_emp_cursor (20);
  ...
```

```
anonymous block completed
200 Whalen
201 Hartstein
202 Fay
```

156

Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for reference in the cursor's query. In the following example, a cursor is declared and is defined with one parameter:

```
DECLARE
  CURSOR c_emp_cursor(deptno NUMBER) IS SELECT ...
```

The following statements open the cursor and return different active sets:

```
OPEN c_emp_cursor(10);
OPEN c_emp_cursor(20);
```

You can pass parameters to the cursor that is used in a cursor FOR loop:

```
DECLARE
  CURSOR c_emp_cursor(p_deptno NUMBER, p_job
  VARCHAR2)IS
    SELECT ...
BEGIN
  FOR emp_record IN c_emp_cursor(10, 'Sales') LOOP ...
```

Syntax:

```
SELECT ...
  FROM ...
  FOR UPDATE [OF column_reference] [NOWAIT | WAIT n];
```

- Use explicit locking to deny access to other sessions for the duration of a transaction.
- Lock the rows *before* the update or delete.

157

If there are multiple sessions for a single database, there is the possibility that the rows of a particular table were updated after you opened your cursor. You see the updated data only when you reopen the cursor. Therefore, it is better to have locks on the rows before you update or delete rows. You can lock the rows with the FOR UPDATE clause in the cursor query.

In the syntax:

The FOR UPDATE clause is the last clause in a SELECT statement, even after column_reference (if exists). It specifies the column in the table against which the query is performed (A list of columns may also be used). FOR UPDATE OF col_name (s) performs row locking to particular tables. FOR NOWAIT UPDATE OF col_name (s) returns an Oracle Server error if the rows are locked by another session.

Syntax:

```
WHERE CURRENT OF cursor ;
```

- Use cursors to update or delete the current row.
- Include the FOR UPDATE clause in the cursor query to first lock the rows.

```
UPDATE employees  
SET salary = ...  
WHERE CURRENT OF c_emp_cursor;
```

158

The WHERE CURRENT OF clause is used in conjunction with the FOR UPDATE clause to refer to the current row in an explicit cursor. The WHERE CURRENT OF clause is used in the UPDATE or DELETE statement, whereas the FOR UPDATE clause is specified in the cursor declaration. You can use the combination for updating and deleting the current row from the corresponding database table. This enables you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the row ID. You must include the FOR UPDATE clause in the cursor query so that the rows are locked on OPEN.

In the syntax:

cursor Is the name of a declared cursor (The cursor must have been declared with the FOR UPDATE clause.)

Topics covered in this module:

- Distinguish cursor types:
 - Implicit cursors are used for all DML statements and single-row queries.
 - Explicit cursors are used for queries of zero, one, or more rows.
- Create and handle explicit cursors
- Use simple loops and cursor FOR loops to handle multiple rows in the cursors
- Evaluate cursor status by using cursor attributes
- Use the FOR UPDATE and WHERE CURRENT OF clauses to update or delete the current fetched row

159

The Oracle Server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return multiple rows, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor FOR loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor FOR loops do this implicitly. If you are updating or deleting rows, lock the rows by using a FOR UPDATE clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a WHERE CURRENT OF clause in conjunction with the FOR UPDATE clause to reference the current row fetched by the cursor.

Oracle PL/SQL 19c

Module 08: Handling Exceptions

Topics covered in this module:

- Define PL/SQL exceptions
- Recognize unhandled exceptions
- List and use different types of PL/SQL exception handlers
- Trap unanticipated errors
- Describe the effect of exception propagation in nested blocks
- Customize PL/SQL exception messages

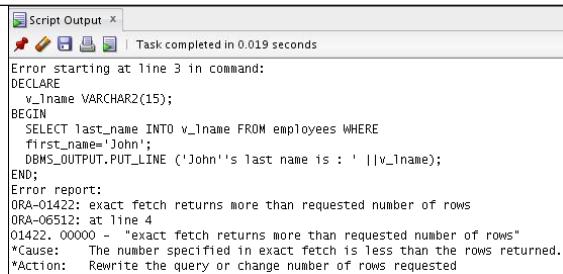
161

You learned to write PL/SQL blocks with a declarative section and an executable section. All the SQL and PL/SQL code that must be executed is written in the executable block.

So far it has been assumed that the code works satisfactorily if you take care of compile-time errors. However, the code may cause some unanticipated errors at run time. In this lesson, you learn how to deal with such errors in the PL/SQL block.

What Is an Exception?

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees
    WHERE first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
END;
```



```
Script Output x | Task completed in 0.019 seconds
Error starting at line 3 in command:
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname FROM employees WHERE
        first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
END;
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 - "exact fetch returns more than requested number of rows"
*Cause: The number specified in exact fetch is less than the rows returned.
*Action: Rewrite the query or change number of rows requested
```

162

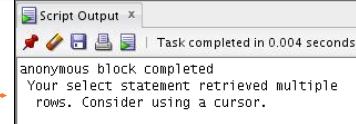
Consider the example shown in the slide. There are no syntax errors in the code, which means that you must be able to successfully execute the anonymous block. The `SELECT` statement in the block retrieves the last name of John.

However, you see the following error report when you execute the code:

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
The code does not work as expected. You expected the SELECT statement to
01422. retrieve only one row; however, it retrieves multiple rows. Such errors that
*Cause: occur at runtime are called exceptions. When an exception occurs, the PL/SQL
*Action: block is terminated. You can handle such exceptions in your PL/SQL block.
```

Handling the Exception: An Example

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees
    WHERE first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE (' Your select statement retrieved
multiple rows. Consider using a cursor.');
END;
/
```



163

You have previously learned how to write PL/SQL blocks with a declarative section (beginning with the `DECLARE` keyword) and an executable section (beginning and ending with the `BEGIN` and `END` keywords, respectively).

For exception handling, you include another optional section called the *exception section*.

This section begins with the `EXCEPTION` keyword.

If present, this must be the last section in a PL/SQL block.

Example

In the example in the slide, the code from the previous slide is rewritten to handle the exception that occurred. The output of the code is shown in the slide as well.

By adding the `EXCEPTION` section of the code, the PL/SQL program does not terminate abruptly. When the exception is raised, the control shifts to the exception section and all the statements in the exception section are executed. The PL/SQL block terminates with normal, successful completion

- An exception is a PL/SQL error that is raised during program execution.
- An exception can be raised:
 - Implicitly by the Oracle Server
 - Explicitly by the program
- An exception can be handled:
 - By trapping it with a handler
 - By propagating it to the calling environment

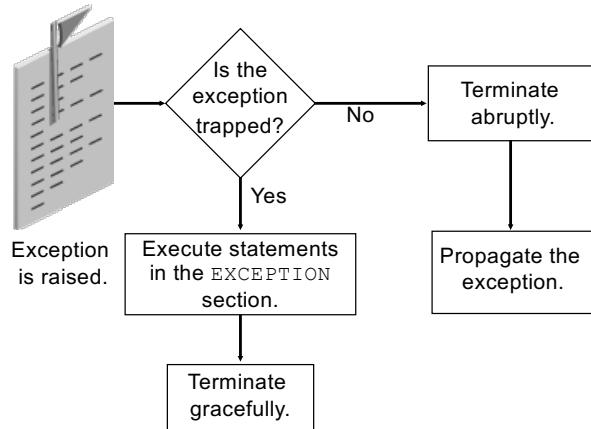
An exception is an error in PL/SQL that is raised during the execution of a block. A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends.

Two Methods for Raising an Exception

An Oracle error occurs and the associated exception is raised automatically. For example, if the ORA-01403 error occurs when no rows are retrieved from the database in a SELECT statement, PL/SQL raises the NO_DATA_FOUND exception. These errors are converted into predefined exceptions.

Depending on the business functionality your program implements, you may have to explicitly raise an exception. You raise an exception explicitly by issuing the RAISE statement in the block. The raised exception may be either user-defined or predefined. There are also some non-predefined Oracle errors. These errors are any standard Oracle errors that are not predefined. You can explicitly declare exceptions and associate them with the non-predefined Oracle errors.

Handling Exceptions



165

Trapping an Exception

Include an EXCEPTION section in your PL/SQL program to trap exceptions. If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, the exception does not propagate to the enclosing block or to the calling environment. The PL/SQL block terminates successfully.

Propagating an Exception

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to an enclosing block or to the calling environment. The calling environment can be any application (such as SQL*Plus that invokes the PL/SQL program).

- Predefined Oracle Server
- Non-predefined Oracle Server
- User-defined

There are three types of exceptions

Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code	You need not declare these exceptions. They are predefined by the Oracle server and are raised implicitly.
Non-predefined Oracle Server error Note: Some application tools with client-side PL/SQL (such as Oracle Developer Forms) have their own exceptions.	Any other standard Oracle Server error	You need to declare these within the declarative section; the Oracle server raises the error implicitly, and you can catch the error in the exception handler.
User-defined error	A condition that the developer determines is abnormal	You need to declare in the declarative section and raise explicitly.

Syntax to Trap Exceptions

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

167

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block. Each handler consists of a WHEN clause, which specifies an exception name, followed by a sequence of statements to be executed when that exception is raised.

You can include any number of handlers within an EXCEPTION section to handle specific exceptions. However, you cannot have multiple handlers for a single exception.

Exception trapping syntax includes the following elements:

<i>exception</i>	Is the standard name of a predefined exception or the name of a user-defined exception declared within the declarative section
<i>statement</i>	Is one or more PL/SQL or SQL statements
OTHERS	Is an optional exception-handling clause that traps any exceptions that have not been explicitly handled

Guidelines for Trapping Exceptions

- The EXCEPTION keyword starts the exception-handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.
- WHEN OTHERS is the last clause.

168

Begin the exception-handling section of the block with the EXCEPTION keyword.

Define several exception handlers, each with its own set of actions, for the block.

When an exception occurs, PL/SQL processes only one handler before leaving the block.

Place the OTHERS clause after all other exception-handling clauses.

You can have only one OTHERS clause.

Exceptions cannot appear in assignment statements or SQL statements.



Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX

169

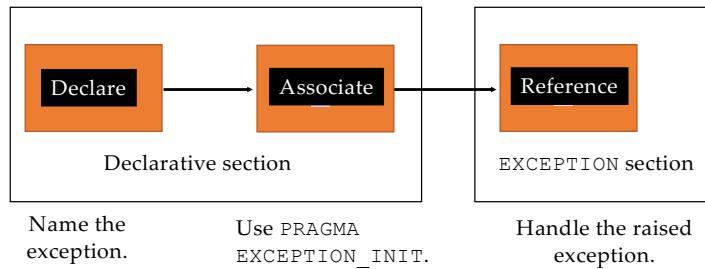
Trap a predefined Oracle Server error by referencing its predefined name

within the corresponding exception-handling routine.

For a complete list of predefined exceptions, see the *PL/SQL User's Guide and Reference*.

Note: PL/SQL declares predefined exceptions in the STANDARD package.

Trapping Non-Predefined Oracle Server Errors



170

Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle Server. They are standard Oracle errors. You create exceptions with standard Oracle errors by using the PRAGMA EXCEPTION_INIT function. Such exceptions are called non-predefined exceptions.

You can trap a non-predefined Oracle Server error by declaring it first. The declared exception is raised implicitly. In PL/SQL, PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. This enables you to refer to any internal exception by name and to write a specific handler for it.

Note: PRAGMA (also called *pseudoinstructions*) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle Server error number.

Non-Predefined Error Trapping: Example

To trap Oracle Server error 01400 ("cannot insert NULL"):

```

DECLARE
    e_insert_excep EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);
BEGIN
    INSERT INTO departments
        (department_id, department_name) VALUES (280, NULL);
EXCEPTION
    WHEN e_insert_excep THEN
        DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/

```

171

The example illustrates the three steps associated with trapping a non-predefined error.

1. Declare the name of the exception in the declarative section, using the syntax:

exception EXCEPTION;

In the syntax, *exception* is the name of the exception.

2. Associate the declared exception with the standard Oracle Server error number by using the PRAGMA EXCEPTION_INIT function. Use the following syntax:

PRAGMA EXCEPTION_INIT(exception, error_number);

In the syntax, *exception* is the previously declared exception and *error_number* is a standard Oracle Server error number.

3. Reference the declared exception within the corresponding exception-handling routine.

Example

The example in the slide tries to insert the NULL value for the department_name column of the departments table. However, the

operation is not successful because department_name is a NOT NULL column. Note the following line in the example:

```
DBMS_OUTPUT.PUT_LINE(SQLERRM);
```

The SQLERRM function is used to retrieve the error message. You learn more about SQLERRM in the next few slides.

Functions for Trapping Exceptions

- SQLCODE: Returns the numeric value for the error code
- SQLERRM: Returns the message associated with the error number

172

When an exception occurs, you can identify the associated error code or error message by using two functions. Based on the values of the code or the message, you can decide which subsequent actions to take. SQLCODE returns the Oracle error number for internal exceptions. SQLERRM returns the message associated with the error number.

Function	Description
SQLCODE Values: Examples	
SQLCODE	Returns the numeric value for the error code (You can assign it to a NUMBER variable.)
SQLERRM	Returns character data containing the message associated with the error number

SQLCODE Value	Description
0	No exception encountered
1	User-defined exception
+100	NO_DATA_FOUND exception
<i>negative number</i>	Another Oracle server error number

Functions for Trapping Exceptions

```
DECLARE
    error_code      NUMBER;
    error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
    WHEN OTHERS THEN
        ROLLBACK;
        error_code := SQLCODE ;
        error_message := SQLERRM ;
        INSERT INTO errors (e_user, e_date, error_code,
                           error_message) VALUES (USER,SYSDATE,error_code,
                           error_message);
    END;
/

```

173

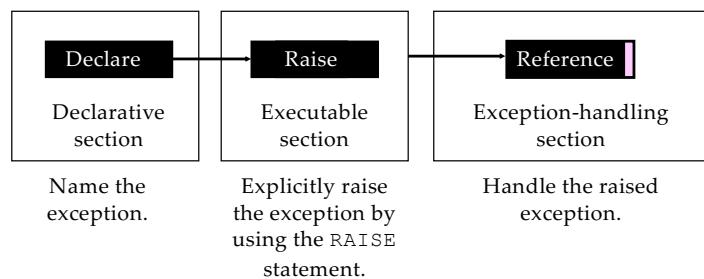
When an exception is trapped in the WHEN OTHERS exception handler, you can use a set of generic functions to identify those errors. The example in the slide illustrates the values of SQLCODE and SQLERRM assigned to variables, and then those variables being used in a SQL statement. You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, and then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
    err_num NUMBER;
    err_msg VARCHAR2(100);
BEGIN
...
EXCEPTION
...
    WHEN OTHERS THEN
        err_num := SQLCODE;
        err_msg := SUBSTR(SQLERRM, 1, 100);
        INSERT INTO errors VALUES (err_num, err_msg);
    END;

```

/

Trapping User-Defined Exceptions



174

PL/SQL enables you to define your own exceptions depending on the requirements of your application. For example, you may prompt the user to enter a department number.

Define an exception to deal with error conditions in the input data. Check whether the department number exists. If it does not, you may have to raise the user-defined exception.

PL/SQL exceptions must be:

Declared in the declarative section of a PL/SQL block

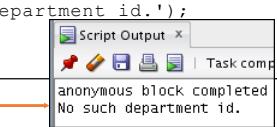
Raised explicitly with RAISE statements

Handled in the EXCEPTION section

Trapping User-Defined Exceptions

```
DECLARE
    v_deptno NUMBER := 500;
    v_name VARCHAR2(20) := 'Testing';
    e_invalid_department EXCEPTION; ── 1
BEGIN
    UPDATE departments
    SET department_name = v_name
    WHERE department_id = v_deptno;
    IF SQL%NOTFOUND THEN
        RAISE e_invalid_department; ── 2
    END IF;
    COMMIT;
EXCEPTION
    WHEN e_invalid_department THEN
        DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```

175



You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name of the user-defined exception within the declarative section.

Syntax:

exception EXCEPTION;

In the syntax, *exception* is the name of the exception.

2. Use the RAISE statement to raise the exception explicitly within the executable section.

Syntax:

RAISE *exception*;

In the syntax, *exception* is the previously declared exception.

3. Reference the declared exception within the corresponding exception-handling routine.

Example

The block shown in the slide updates the `department_name` of a department. The user supplies the department number and the new name. If the supplied department number does not exist, no rows are updated in the `departments` table. An exception is raised and a message is printed for the user that an invalid department number was entered.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception again and propagate it back to the calling environment.

Propagating Exceptions in a Subblock

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  ...
  e_no_rows      exception;
  e_integrity    exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp cursor LOOP
    BEGIN
      SELECT ...
      UPDATE ...
      IF SQL%NOTFOUND THEN
        RAISE e_no_rows;
      END IF;
    END;
  END LOOP;
EXCEPTION
  WHEN e_integrity THEN ...
  WHEN e_no_rows THEN ...
END;
/
```

176

When a subblock handles an exception, it terminates normally. Control resumes in the enclosing block immediately after the subblock's `END` statement.

However, if a PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates to successive enclosing blocks until it finds a handler. If none of these blocks handles the exception, an unhandled exception in the host environment results.

When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

Note in the example that the exceptions (`no_rows` and `integrity`) are declared in the outer block. In the inner block, when the `no_rows` exception is raised, PL/SQL looks for the exception to be handled in the subblock.

Because the exception is not handled in the subblock, the exception propagates to the outer block, where PL/SQL finds the handler.



RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,  
                      message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

177

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax:

<i>error_number</i>	Is a user-specified number for the exception between –20,000 and –20,999
<i>message</i>	Is the user-specified message for the exception; is a character string up to 2,048 bytes long
TRUE FALSE	Is an optional Boolean parameter (If TRUE, the error is placed on the stack of previous errors. If FALSE, which is the default, the error replaces all previous errors.)



RAISE_APPLICATION_ERROR Procedure

- Is used in two different places:
 - Executable section
 - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle Server errors

178

The RAISE_APPLICATION_ERROR procedure can be used in either the executable section or the exception section of a PL/SQL program, or both. The returned error is consistent with how the Oracle Server produces a predefined, non-predefined, or user-defined error. The error number and message are displayed to the user.

Executable section:

```
BEGIN  
...  
    DELETE FROM employees  
        WHERE manager_id = v_mgr;  
    IF SQL%NOTFOUND THEN  
        RAISE_APPLICATION_ERROR(-20202,  
            'This is not a valid manager');  
    END IF;  
    ...
```

```
...  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RAISE_APPLICATION_ERROR (-20201,  
            'Manager is not a valid employee.');
```

179

The slide shows that the RAISE_APPLICATION_ERROR procedure can be used in both the executable and the exception sections of a PL/SQL program. Here is another example of using the RAISE_APPLICATION_ERROR procedure:

```
DECLARE  
    e_name EXCEPTION;  
BEGIN  
...  
    DELETE FROM employees  
        WHERE last_name = 'Higgins';  
    IF SQL%NOTFOUND THEN RAISE e_name;  
    END IF;  
EXCEPTION  
    WHEN e_name THEN  
        RAISE_APPLICATION_ERROR (-20999, 'This is not a  
valid last name'); ...  
END;  
/
```

Topics covered in this module:

- Define PL/SQL exceptions
- Add an EXCEPTION section to the PL/SQL block to deal with exceptions at run time
- Handle different types of exceptions:
 - Predefined exceptions
 - Non-predefined exceptions
 - User-defined exceptions
- Propagate exceptions in nested blocks and call applications

180

In this lesson, you learned how to deal with different types of exceptions. In

PL/SQL, a warning or error condition at run time is called an exception.

Predefined exceptions are error conditions that are defined by the Oracle Server. Non-predefined exceptions can be any standard Oracle Server errors. User-defined exceptions are exceptions specific to your application. The PRAGMA EXCEPTION_INIT function can be used to associate a declared exception name with an Oracle Server error.

You can define exceptions of your own in the declarative section of any PL/SQL block. For example, you can define an exception named INSUFFICIENT_FUNDS to flag overdrawn bank accounts.

When an error occurs, an exception is raised. Normal execution stops and transfers control to the exception-handling section of your PL/SQL block. Internal exceptions are raised implicitly (automatically) by the run-time system; however, user-defined exceptions must be raised explicitly. To handle raised exceptions, you write separate routines called exception handlers.

Introduction To Oracle PL/SQL

Module 09: Creating Procedures

Topics covered in this module:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure
- Display the procedures' information

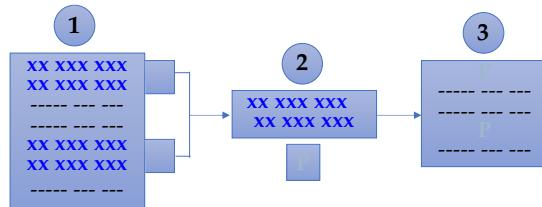
182

Lesson Aim

In this lesson, you learn to create, execute, and remove procedures with or without parameters. Procedures are the foundation of modular programming in PL/SQL. To make procedures more flexible, it is important that varying data is either calculated or passed into a procedure by using input parameters. Calculated results can be returned to the caller of a procedure by using OUT parameters.

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

Creating a Modularized Subprogram Design



- o Modularize code into subprograms.

1. Locate code sequences repeated more than once.
2. Create subprogram P containing the repeated code
3. Modify original code to invoke the new subprogram.

183

Creating a Modularized and Layered Subprogram Design

The diagram illustrates the principle of modularization with subprograms, the creation of smaller manageable pieces of flexible and reusable code.

Flexibility is achieved by using subprograms with parameters, which in turn makes the same code reusable for different input values. To modularize existing code, perform the following steps:

1. Locate and identify repetitive sequences of code.
2. Move the repetitive code into a PL/SQL subprogram.
3. Replace the original repetitive code with calls to the new PL/SQL subprogram.

Following this modular and layered approach can help you create code that is easier to maintain, particularly when the business rules change. In addition, keeping the SQL logic simple and free of complex business logic can benefit from the work of Oracle Database Optimizer, which can reuse parsed SQL statements for better use of server-side resources.

Create subprogram layers for your application.

- Data access subprogram layer with SQL logic
- Business logic subprogram layer, which may or may not use the data access layer

Because PL/SQL allows SQL statements to be seamlessly embedded into the logic, it is too easy to have SQL statement spread all over the code. However, it is recommended that you keep the SQL logic separate from the business logic—that is, create a layered application design with a minimum of two layers:

Data access layer: For subroutines to access the data by using SQL statements

Business logic layer: For subprograms to implement the business processing rules, which may or may not call on the data access layer routines

- PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:
 - Anonymous blocks
 - Procedures and functions
 - Packages
 - Database triggers
- The benefits of using modular program constructs are:
 - Easy maintenance
 - Improved data security and integrity
 - Improved performance
 - Improved code clarity

185

A subprogram is based on standard PL/SQL structures. It contains a declarative section, an executable section, and an optional exception-handling section (for example, anonymous blocks, procedures, functions, packages, and triggers). Subprograms can be compiled and stored in the database, providing modularity, extensibility, reusability, and maintainability.

Modularization converts large blocks of code into smaller groups of code called modules. After modularization, the modules can be reused by the same program or shared with other programs. It is easier to maintain and debug code that comprises smaller modules than it is to maintain code in a single large program. Modules can be easily extended for customization by incorporating more functionality, if required, without affecting the remaining modules of the program.

Subprograms provide easy maintenance because the code is located in one place and any modifications required to the subprogram can, therefore, be performed in this single location. Subprograms provide improved data integrity and security. The data objects are accessed through the subprogram, and a user can invoke the subprogram only if the appropriate access privilege is granted to the user.

Note: Knowing how to develop anonymous blocks is a prerequisite for this

course. For detailed information about anonymous blocks, see the course titled *Oracle: PL/SQL Fundamentals*.

Anonymous blocks:

- Form the basic PL/SQL block structure
- Initiate PL/SQL processing tasks from applications
- Can be nested within the executable section of any PL/SQL block

```
[DECLARE      -- Declaration Section (Optional)
     variable declarations; ... ]
BEGIN        -- Executable Section (Mandatory)
    SQL or PL/SQL statements;
[EXCEPTION   -- Exception Section (Optional)
    WHEN exception THEN statements; ]
END;         -- End of Block (Mandatory)
```

186

Anonymous blocks are typically used for:

writing trigger code for Oracle Forms components

Initiating calls to procedures, functions, and package constructs

Isolating exception handling within a block of code

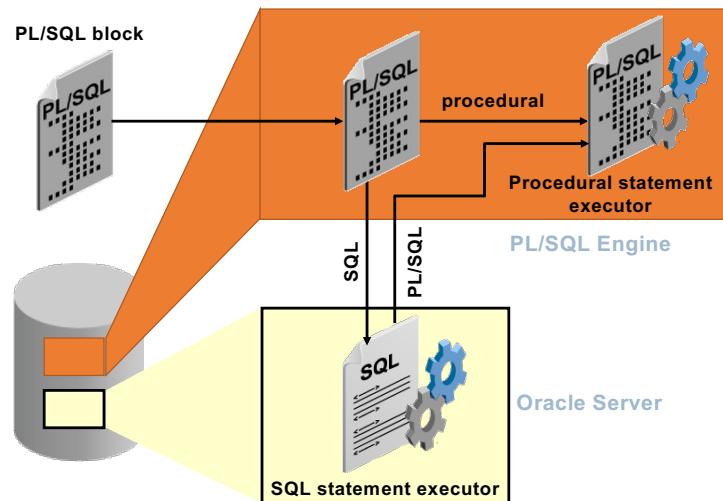
Nesting inside other PL/SQL blocks for managing code flow control

The `DECLARE` keyword is optional, but it is required if you declare variables, constants, and exceptions to be used within the PL/SQL block.

`BEGIN` and `END` are mandatory and require at least one statement between them, either SQL, PL/SQL, or both.

The exception section is optional and is used to handle errors that occur within the scope of the PL/SQL block. Exceptions can be propagated to the caller of the anonymous block by excluding an exception handler for the specific exception, thus creating what is known as an *unhandled* exception.

PL/SQL Run-time Architecture



187

The diagram shows a PL/SQL block being executed by the PL/SQL engine. The PL/SQL engine resides in:

The Oracle database for executing stored subprograms

The Oracle Forms client when running client/server applications, or in the Oracle Application Server when using Oracle Forms Services to run Forms on the Web

Irrespective of the PL/SQL run-time environment, the basic architecture remains the same. Therefore, all PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle server processes. The SQL environment may also invoke the PL/SQL environment for example when a function is used in a select statement.

The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL m-code instructions. When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the Oracle server processes. The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent statements in the PL/SQL block. The Oracle Forms PL/SQL engine runs in the client for the client/server implementation, and in the application server for

the Forms Services implementation. In either case, SQL statements are typically sent over a network to an Oracle server for processing.

What Are PL/SQL Subprograms?

- A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters.
- You can declare and define a subprogram within either a PL/SQL block or another subprogram.
- A subprogram consists of a specification and a body.
- A subprogram can be a procedure or a function.
- Typically, you use a procedure to perform an action and a function to compute and return a value.
- Subprograms can be grouped into PL/SQL packages.

188

A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters. You can declare and define a subprogram within either a PL/SQL block or another subprogram.

Subprogram Parts: A subprogram consists of a specification (spec) and a body. To declare a subprogram, you must provide the spec, which includes descriptions of any parameters. To define a subprogram, you must provide both the spec and the body. You can either declare a subprogram first and define it later in the same block or subprogram, or declare and define it at the same time.

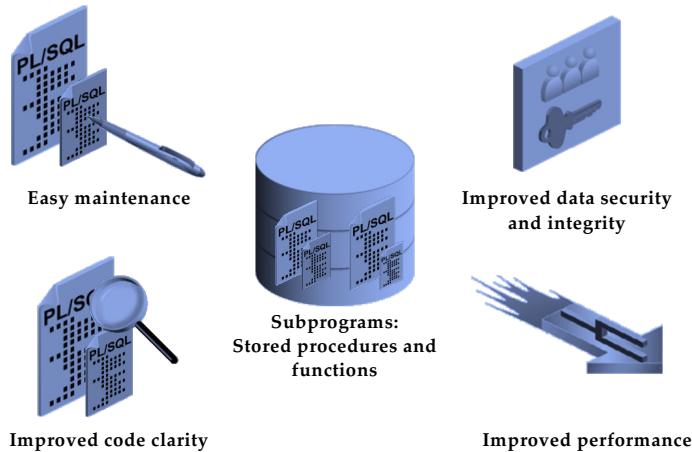
Subprogram Types: PL/SQL has two types of subprograms: procedures and functions. Typically, you use a procedure to perform an action and a function to compute and return a value.

A procedure and a function have the same structure, except that only a function has some additional items such as the RETURN clause or the RETURN statement.

The RETURN clause specifies the data type of the return value (required). A RETURN statement specifies the return value (required). Functions are covered in more detail in the next lesson titled “Creating Functions and Debugging Subprograms.”

Subprograms can be grouped into PL/SQL packages, which make code even more reusable and maintainable. For details, see the lessons about packages (Lessons 4 and 5).

The Benefits of Using PL/SQL Subprograms



189

Benefits of Subprograms

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

Easy maintenance is realized because subprograms are located in one place. Modifications need to be done in only one place to affect multiple applications and minimize excessive testing.

Improved data security can be achieved by controlling indirect access to database objects from nonprivileged users with security privileges. The subprograms are by default executed with definer's right. The execute privilege does not allow a calling user direct access to objects that are accessible to the subprogram.

Data integrity is managed by having related actions performed together or not at all.

Improved performance can be realized from reuse of parsed PL/SQL code that becomes available in the shared SQL area of the server. Subsequent calls to the subprogram avoid parsing the code again. Because PL/SQL code is parsed at compile time, the parsing overhead of SQL statements is avoided at run time. Code can be written to reduce the number of network calls to the database, and therefore, decrease network traffic.

Improved code clarity can be attained by using appropriate names and conventions to describe the action of the routines, thereby reducing the need for comments and enhancing the clarity of the code.

Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	Subprograms called functions must return values.
Cannot take parameters	Can take parameters

190

The table in the slide not only shows the differences between anonymous blocks and subprograms, but also highlights the general benefits of subprograms.

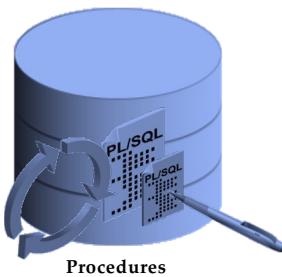
Anonymous blocks are not persistent database objects. They are compiled and executed only once. They are not stored in the database for reuse. If you want to reuse, you must rerun the script that creates the anonymous block, which causes recompilation and execution.

Procedures and functions are compiled and stored in the database in a compiled form.

They are recompiled only when they are modified. Because they are stored in the database, any application can make use of these subprograms based on appropriate permissions. The calling application can pass parameters to the procedures if the procedure is designed to accept parameters. Similarly, a calling application can retrieve a value if it invokes a function or a procedure.

- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information

- Are a type of subprogram that perform an action
- Can be stored in the database as a schema object
- Promote reusability and maintainability



192

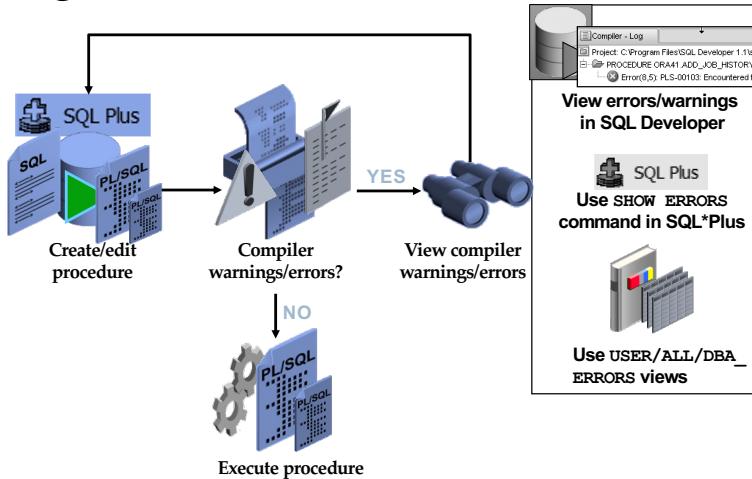
Definition of a Procedure

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

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

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

Creating Procedures: Overview



193

To develop a procedure using a tool such as SQL Developer, perform the following steps:

1. Create the procedure using SQL Developer's Object Navigator tree or the SQL Worksheet area.
2. Compile the procedure. The procedure is created in the database and gets compiled. The `CREATE PROCEDURE` statement creates and stores the source code and the compiled *m-code* in the database. To compile the procedure, right-click the procedure's name in the Object Navigator tree, and then click Compile.
3. If compilation errors exist, then the *m-code* is not stored and you must edit the source code to make corrections. You cannot invoke a procedure that contains compilation errors. You can view the compilation errors in SQL Developer, SQL*Plus, or the appropriate data dictionary views as shown in the slide.
4. After successful compilation, execute the procedure to perform the desired action. You can run the procedure using SQL Developer or use the `EXECUTE` command in SQL*Plus.

Note: If compilation errors occur, use a `CREATE OR REPLACE PROCEDURE` statement to overwrite the existing code if you previously used

a CREATE PROCEDURE statement. Otherwise, drop the procedure first (using DROP) and then execute the CREATE PROCEDURE statement.

Procedures with *CREATE OR REPLACE*

- Use the CREATE clause to create a stand-alone procedure that is stored in the Oracle database.
- Use the OR REPLACE option to overwrite an existing procedure.

```

CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter1 [mode] datatype1,
    parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; ...]
BEGIN
  -- actions;
END [procedure_name];
  
```

PL/SQL block

194

You can use the `CREATE PROCEDURE` SQL statement to create stand-alone procedures that are stored in an Oracle database. A procedure is similar to a miniature program: it performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the BEGIN-END block that contains its code and handles any exceptions.

PL/SQL blocks start with `BEGIN`, optionally preceded by the declaration of local variables. PL/SQL blocks end with either `END` or `END procedure_name`.

The `REPLACE` option indicates that if the procedure exists, it is dropped and replaced with the new version created by the statement. The `REPLACE` option does not drop any of the privileges associated with the procedure.

Other Syntactic Elements

`parameter1` represents the name of a parameter.

The `mode` option defines how a parameter is used: `IN` (default), `OUT`, or `IN OUT`.

`datatype1` specifies the parameter data type, without any precision.

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

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

Naming Conventions of PL/SQL Structures Used in This Course

PL/SQL Structure	Convention	Example
Variable	v_variable_name	v_rate
Constant	c_constant_name	c_rate
Subprogram parameter	p_parameter_name	p_id
Bind (host) variable	b_bind_name	b_salary
Cursor	cur_cursor_name	cur_emp
Record	rec_record_name	rec_emp
Type	type_name_type	ename_table_type
Exception	e_exception_name	e_products_invalid
File handle	f_file_handle_name	f_file

195

The slide table displays some examples of the naming conventions for PL/SQL structures that are used in this course.

What Are Parameters and Parameter Modes?

- Are declared after the subprogram name in the PL/SQL header
- Pass or communicate data between the calling environment and the subprogram
- Are used like local variables but are dependent on their parameter-passing mode:
 - An **IN** parameter mode (the default) provides values for a subprogram to process
 - An **OUT** parameter mode returns a value to the caller
 - An **IN OUT** parameter mode supplies an input value, which may be returned (output) as a modified value

196

What Are Parameters?

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

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

An **IN** parameter passes a constant value from the calling environment into the procedure.

An **OUT** parameter passes a value from the procedure to the calling environment.

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

Parameters can be thought of as a special form of local variable, whose input values are initialized by the calling environment when the subprogram is called, and whose output values are returned to the calling environment when the subprogram returns control to the caller.

- Formal parameters: Local variables declared in the parameter list of a subprogram specification
- Actual parameters (or arguments): Literal values, variables, and expressions used in the parameter list of the calling subprogram

```
-- Procedure definition, Formal parameters
CREATE PROCEDURE raise_sal(p_id NUMBER, p_sal NUMBER) IS
BEGIN
...
END raise_sal;

-- Procedure calling, Actual parameters (arguments)
v_emp_id := 100;
raise_sal(v_emp_id, 2000)
```

The diagram illustrates the mapping between formal and actual parameters. Two orange arrows originate from the identifiers 'p_id' and 'p_sal' in the first code block (Procedure Definition). These arrows point to the identifiers 'v_emp_id' and '2000' respectively in the second code block (Procedure Calling).

197

Formal parameters are local variables that are declared in the parameter list

of a subprogram specification. In the first example, in the `raise_sal`

procedure, the variable `p_id` and

`p_sal` identifiers represent the formal parameters.

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

Actual parameters:

Are associated with formal parameters during the subprogram call

Can also be expressions, as in the following example:

```
raise_sal(v_emp_id, raise+100);
```

The formal and actual parameters should be of compatible data types. If necessary, before assigning the value, PL/SQL converts the data type of the actual parameter value to that of the formal parameter.

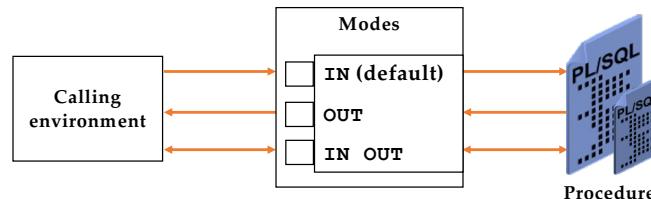
Note: Actual parameters are also referred to as *actual arguments*.

Procedural Parameter Modes

- Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.
- The `IN` mode is the default if no mode is specified.

```
CREATE PROCEDURE proc_name(param_name [mode] datatype)
...

```



198

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

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

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

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

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

Comparing the Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Value is returned to the calling environment	Value passed into sub-program; value returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

199

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

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

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

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

Using the IN Parameter Mode: Example

The screenshot shows the Oracle SQL Developer interface. At the top, there is a code editor window containing the following PL/SQL code:

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

Below the code editor is a results window showing the output of the compilation command:

```
PROCEDURE raise_salary Compiled.
```

At the bottom, there is another code editor window containing the execution command:

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

200

Using IN Parameters: Example

The example in the slide shows a procedure with two IN parameters. Running the first slide example creates the `raise_salary` procedure in the database. The second slide example invokes `raise_salary` and provides the first parameter value of 176 for the employee ID, and a salary increase of 10 percent for the second parameter value.

To invoke a procedure by using the SQL Worksheet of SQL Developer or by using SQL*Plus, use the following EXECUTE command shown in the second code example in the slide.

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

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

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

Using the OUT Parameter Mode: Example

```
CREATE OR REPLACE PROCEDURE query_emp
(p_id      IN employees.employee_id%TYPE,
 p_name    OUT employees.last_name%TYPE,
 p_salary  OUT employees.salary%TYPE) IS
BEGIN
  SELECT last_name, salary INTO p_name, p_salary
  FROM   employees
  WHERE  employee_id = p_id;
END query_emp;
/

SET SERVEROUTPUT ON
DECLARE
  v_emp_name employees.last_name%TYPE;
  v_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, v_emp_name, v_emp_sal);
  DBMS_OUTPUT.PUT_LINE(v_emp_name||' earns ' ||
    to_char(v_emp_sal, '$999,999.00'));
END;
/
```

201

Using the OUT Parameters: Example

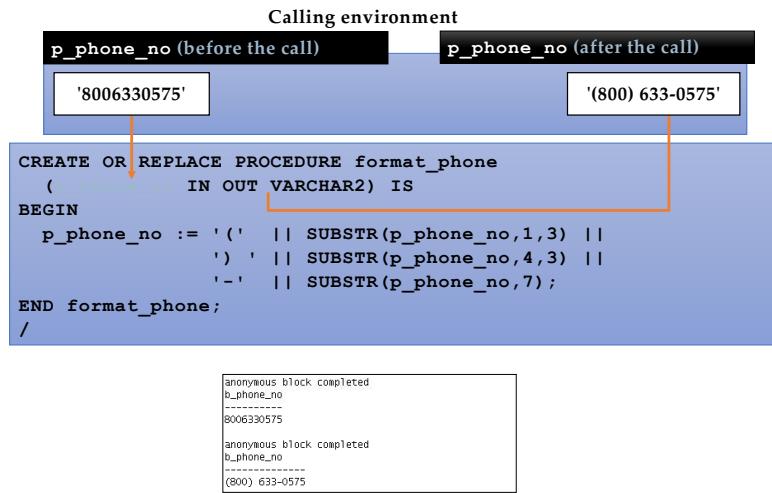
In the slide example, you create a procedure with OUT parameters to retrieve information about an employee. The procedure accepts the value 171 for employee ID and retrieves the name and salary of the employee with ID 171 into the two OUT parameters. The query_emp procedure has three formal parameters. Two of them are OUT parameters that return values to the calling environment, shown in the second code box in the slide. The procedure accepts an employee ID value through the p_id parameter. The v_emp_name and v_emp_salary variables are populated with the information retrieved from the query into their two corresponding OUT parameters. The following is the result of running the code in the second code example in the slide. v_emp_name holds the value Smith and v_emp_salary holds the value 7400.

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

The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The output window displays the following text:

```
anonymous block completed
Smith earns $7,400.00
```

Using the IN OUT Parameter Mode: Example



202

Using IN OUT Parameters: Example

Using an IN OUT parameter, you can pass a value into a procedure that can be updated. The actual parameter value supplied from the calling environment can return either the original unchanged value or a new value that is set within the procedure.

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

The slide example creates a procedure with an IN OUT parameter to accept a 10-character string containing digits for a phone number. The procedure returns the phone number formatted with parentheses around the first three characters and a hyphen after the sixth digit—for example, the phone string 8006330575 is returned as (800) 633-0575.

The following code uses the b_phone_no host variable of SQL*Plus to provide the input value passed to the FORMAT_PHONE procedure. The procedure is executed and returns an updated string in the b_phone_no host variable. The output of the following code is displayed in the slide above:

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

Viewing the OUT Parameters:

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

```
SET SERVEROUTPUT ON

DECLARE
    v_emp_name employees.last_name%TYPE;
    v_emp_sal  employees.salary%TYPE;
BEGIN
    query_emp(171, v_emp_name, v_emp_sal);
    DBMS_OUTPUT.PUT_LINE('Name: ' || v_emp_name);
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_emp_sal);
END;
```

```
anonymous block completed
Name: Smith
Salary: 7400
```

203

Viewing the OUT Parameters: Using the DBMS_OUTPUT Subroutine

The slide example illustrates how to view the values returned from the OUT parameters in SQL*Plus or the SQL Developer Worksheet.

You can use PL/SQL variables in an anonymous block to retrieve the OUT parameter values. The DBMS_OUTPUT.PUT_LINE procedure is called to print the values held in the PL/SQL variables. The SET SERVEROUTPUT must be ON.

1. Use SQL*Plus host variables.
2. Execute QUERY_EMP using host variables.
3. Print the host variables.

```
VARIABLE b_name  VARCHAR2(25)
VARIABLE b_sal   NUMBER
EXECUTE query_emp(171, :b_name, :b_sal)
PRINT b_name b_sal
```



```
anonymous block completed
b_name
-----
Smith

b_sal
-----
7400
```

204

The example in the slide demonstrates how to use SQL*Plus host variables that are created using the VARIABLE command. The SQL*Plus variables are external to the PL/SQL block and are known as host or bind variables. To reference host variables from a PL/SQL block, you must prefix their names with a colon (:). To display the values stored in the host variables, you must use the SQL*Plus PRINT command followed by the name of the SQL*Plus variable (without the colon because this is not a PL/SQL command or context).

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

- When calling a subprogram, you can write the actual parameters using the following notations:
 - Positional: Lists the actual parameters in the same order as the formal parameters
 - Named: Lists the actual parameters in arbitrary order and uses the association operator (`=>`) to associate a named formal parameter with its actual parameter
 - Mixed: Lists some of the actual parameters as positional and some as named
- Prior to Oracle Database 19c, only the positional notation is supported in calls from SQL
- Starting in Oracle Database 19c, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements

205

Syntax for Passing Parameters

When calling a subprogram, you can write the actual parameters using the following notations:

Positional: You list the actual parameter values in the same order in which the formal parameters are declared. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the error can be hard to detect. You must change your code if the procedure's parameter list changes.

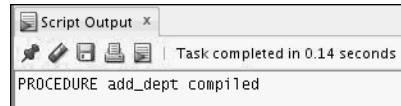
Named: You list the actual values in arbitrary order and use the association operator to associate each actual parameter with its formal parameter by name. The PL/SQL **association operator** is an “equal” sign followed by an “is greater than” sign, without spaces: `=>`. The order of the parameters is not significant. This notation is more verbose, but makes your code easier to read and maintain. You can sometimes avoid changing your code if the procedure's parameter list changes, for example, if the parameters are reordered or a new optional parameter is added.

Mixed: You list the first parameter values by their position and the remainder by using the special syntax of the named method. You can

use this notation to call procedures that have some required parameters, followed by some optional parameters.

Passing Actual Parameters: Creating the add_dept Procedure

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



206

Passing Parameters: Examples

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

Passing Actual Parameters: Examples

```
-- Passing parameters using the positional notation.  
EXECUTE add_dept ('TRAINING', 2500)
```

anonymous block completed			
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	TRAINING		2500
1 rows selected			

```
-- Passing parameters using the named notation.  
EXECUTE add_dept (p_loc=>2400, p_name=>'EDUCATION')
```

anonymous block completed			
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
290	EDUCATION		2400
1 rows selected			

207

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

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

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

Using the **DEFAULT** Option for the Parameters

- Defines default values for parameters
- Provides flexibility by combining the positional and named parameter-passing syntax

```

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

EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', p_loc => 1200)
EXECUTE add_dept (p_loc => 1200)

```

208

You can assign a default value to an `IN` parameter as follows:

The assignment operator (`:=`), as shown for the `name` parameter in the slide

The `DEFAULT` option, as shown for the `p_loc` parameter in the slide

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

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

The second code box in the slide shows three ways of invoking the `add_dept` procedure:

The first example assigns the default values for each parameter.

The second example illustrates a combination of position and named notation to assign values. In this case, using named notation is presented as an example.

The last example uses the default value for the `name` parameter,

Unknown, and the supplied value for the `p_loc` parameter

Calling Procedures

- You can call procedures using anonymous blocks, another procedure, or packages.
- You must own the procedure or have the EXECUTE privilege.

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

PROCEDURE process_employees Compiled.

209

You can invoke procedures by using:

ANONYMOUS BLOCKS

Another procedure or PL/SQL subprogram

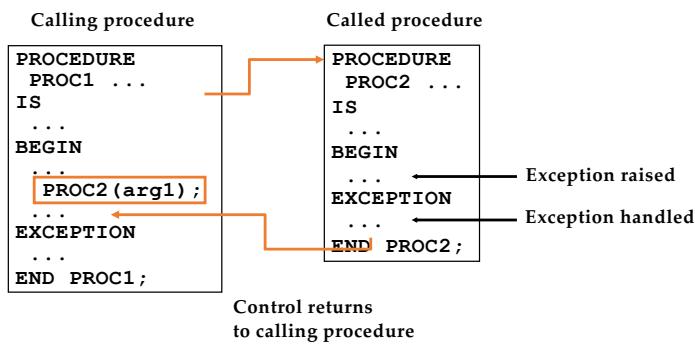
Examples on the preceding pages have illustrated how to use anonymous blocks (or the EXECUTE command in SQL Developer or SQL*Plus).

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

Note: You must own the procedure or have the EXECUTE privilege.

- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information

Handled Exceptions



211

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

When an exception is raised in a called procedure, the control immediately goes to the exception section of that block. An exception is considered handled if the exception section provides a handler for the exception raised. When an exception occurs and is handled, the following code flow takes place:

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

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

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

Handled Exceptions: Example

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

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

212

The two procedures in the slide are the following:

The `add_department` procedure creates a new department record by allocating a new department number from an Oracle sequence, and sets the `department_name`, `manager_id`, and `location_id` column values using the `p_name`, `p_mgr`, and `p_loc` parameters, respectively.

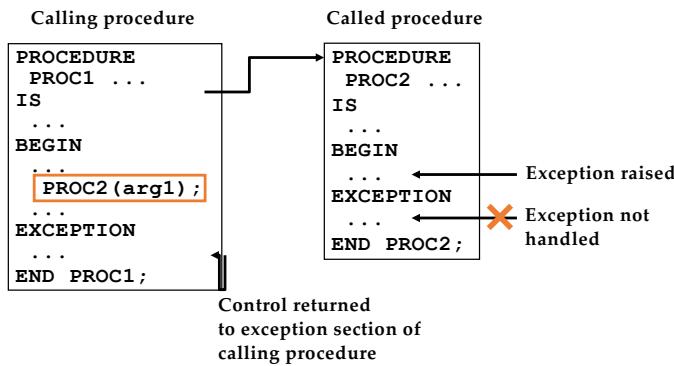
The `create_departments` procedure creates more than one department by using calls to the `add_department` procedure.

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

The Editing department was added because a foreign key constraint violation occurred. This indicates that no manager has an ID of 99. Because the exception was handled in the `add_department` procedure, the `create_department` procedure continues to execute. A query on the `DEPARTMENTS` table where the

`location_id` is 1800 shows that Media and Advertising are added but the Editing record is not.

Exceptions Not Handled



213

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

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

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

Exceptions Not Handled: Example

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

```
CREATE PROCEDURE create_departments_noex IS
BEGIN
    add_department_noex('Media', 100, 1800);
    add_department_noex('Editing', 99, 1800);
    add_department_noex('Advertising', 101, 1800); X
END;
```

214

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

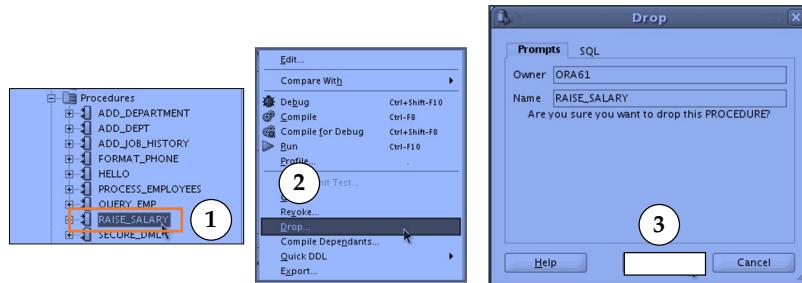
```
Error starting at line 8 in command:
EXECUTE create_departments_noex
Error report:
ORA-02291: integrity constraint (ORA62.DEPT_MGR_FK) violated - parent key not found
ORA-06512: at "ORA62.ADD_DEPARTMENT_NOEX", line 4
ORA-06512: at "ORA62.CREATE_DEPARTMENTS_NOEX", line 4
ORA-06512: at line 1
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause: A foreign key value has no matching primary key value.
*Action: Delete the foreign key or add a matching primary key.
```

Removing Procedures:

- Using the DROP statement:

```
DROP PROCEDURE raise_salary;
```

- Using SQL Developer:



215

When a stored procedure is no longer required, you can use the `DROP PROCEDURE` SQL statement followed by the procedure's name to remove it as follows:

```
DROP PROCEDURE procedure_name
```

You can also use SQL Developer to drop a stored procedure as follows:

- Right-click the procedure name in the **Procedures** node, and then click **Drop**. The **Drop** dialog box is displayed.
- Click **Apply** to drop the procedure.

Note

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

You might have to refresh the **Procedures** node before you can see the results of the drop operation. To refresh the **Procedures** node, right-click the procedure name in the **Procedures** node, and then click **Refresh**.

Viewing Procedure Information Using the Data Dictionary Views

```
DESCRIBE user_source
```

Name	Null	Type
NAME		VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)

4 rows selected

```
SELECT text
  FROM user_source
 WHERE name = 'ADD_DEPT' AND type = 'PROCEDURE'
 ORDER BY line;
```

```
|| TEXT
1 PROCEDURE add_dept(
2 p_name IN departments.department_name%TYPE,
3 p_loc IN departments.location_id%TYPE) IS
4
5 BEGIN
6 INSERT INTO departments(department_id,department_name,location_id)
7 VALUES(departments_seq.NEXTVAL,p_name,p_loc);
8 END add_dept;
```

216

Viewing Procedure in the Data Dictionary

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

The `USER_SOURCE` table to display PL/SQL code that you own

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

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

The `TEXT` column holds a line of PL/SQL source code.

The `NAME` column holds the name of the subprogram in uppercase text.

The `TYPE` column holds the subprogram type, such as `PROCEDURE` or `FUNCTION`.

The `LINE` column stores the line number for each source code line.

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

preceding columns.

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

Topics covered in this module:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure
- Display the procedures' information

Introduction to Oracle PL/SQL

Module 10: Creating Functions &
Debugging Subprograms

Topics covered in this module:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Understand the basic functionality of the SQL Developer debugger

219

In this lesson, you learn how to create, invoke, and maintain functions.

- Working with functions:
 - Differentiating between a procedure and a function
 - Describing the uses of functions
 - Creating, invoking, and removing stored functions
- Introducing the SQL Developer debugger

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value for another subprogram
- Can be grouped into PL/SQL packages

221

A function is a named PL/SQL block that can accept parameters, be invoked,

and return a value. In general, you use a function to compute a value.

Functions and procedures are structured alike. A function must return a value to the calling environment, whereas a procedure returns zero or more values to its calling environment. Like a procedure, a function has a header, a declarative section, an executable section, and an optional exception-handling section. A function must have a RETURN clause in the header and at least one RETURN statement in the executable section.

Functions can be stored in the database as schema objects for repeated execution. A function that is stored in the database is referred to as a stored function. Functions can also be created on client-side applications.

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

A function may also be called as part of a SQL expression or as part of a PL/SQL expression. In the context of a SQL expression, a function must obey specific rules to control side effects. In a PL/SQL expression, the function identifier acts like a variable whose value depends on the parameters passed to it.

Functions (and procedures) can be grouped into PL/SQL packages. Packages make code even more reusable and maintainable. Packages are covered in the lessons titled “Creating Packages” and “Working with Packages.”

The PL/SQL block must have at least one RETURN statement.

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter1 [mode1] datatype1, . . .)]
RETURN datatype IS|AS
[local_variable_declarations;
. . .]
BEGIN
-- actions;
RETURN expression;
END [function_name];
```

PL/SQL Block

222

Syntax for Creating Functions

A function is a PL/SQL block that returns a value. A RETURN statement must be provided to return a value with a data type that is consistent with the function declaration.

You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

You should consider the following points about the CREATE FUNCTION statement:

The REPLACE option indicates that if the function exists, it is dropped and replaced with the new version that is created by the statement.

The RETURN data type must not include a size specification.

The PL/SQL block starts with a BEGIN after the declaration of any local variables and ends with an END, optionally followed by the *function_name*.

There must be at least one RETURN *expression* statement.

You cannot reference host or bind variables in the PL/SQL block of a stored function.

Note: Although the OUT and IN OUT parameter modes can be used with

functions, it is not good programming practice to use them with functions. However, if you need to return more than one value from a function, consider returning the values in a composite data structure such as a PL/SQL record or a PL/SQL table.

The Difference Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a RETURN statement without a value	Must contain at least one RETURN statement

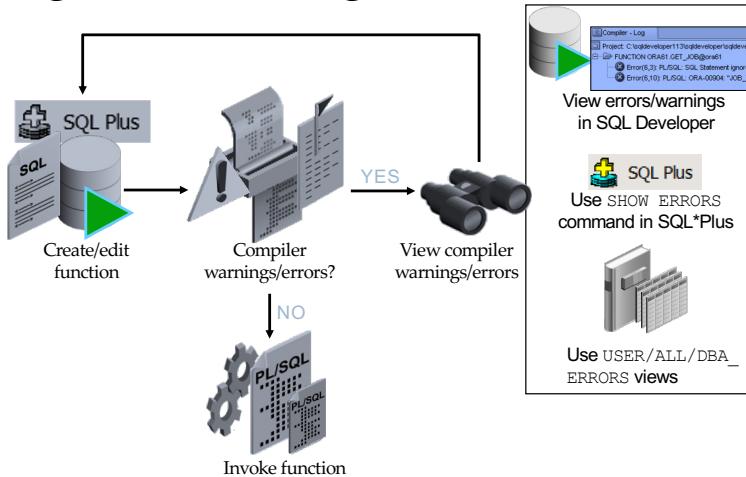
223

You create a procedure to store a series of actions for later execution. A procedure can contain zero or more parameters that can be transferred to and from the calling environment, but a procedure does not have to return a value. A procedure can call a function to assist with its actions.

Note: A procedure containing a single OUT parameter would be better rewritten as a function returning the value.

You create a function when you want to compute a value that must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions typically return only a single value, and the value is returned through a RETURN statement. Functions used in SQL statements should not use OUT or IN OUT mode parameters. Although a function using output parameters can be used in a PL/SQL procedure or block, it cannot be used in SQL statements.

Creating and Running Functions: Overview



224

The diagram in the slide illustrates the basic steps involved in creating and running a function:

1. Create the function using SQL Developer's Object Navigator tree or the SQL Worksheet area.
2. Compile the function. The function is created in the database. The CREATE FUNCTION statement creates and stores source code and the compiled *m-code* in the database. To compile the function, right-click the function's name in the Object Navigator tree, and then click Compile.
3. If there are compilation warning or errors, you can view (and then correct) the warnings or errors using one of the following methods:
 - a. Using the SQL Developer interface (the Compiler – Log tab)
 - b. Using the SHOW ERRORS SQL*Plus command
 - c. Using the USER/ALL/DBA_ERRORS views
4. After successful compilation, invoke the function to return the desired value.

Creating and Invoking a Stored Function Using the CREATE FUNCTION Statement: Example

```
CREATE OR REPLACE FUNCTION get_sal
(p_id employees.employee_id%TYPE) RETURN NUMBER IS
v_sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO v_sal
  FROM employees
  WHERE employee_id = p_id;
  RETURN v_sal;
END get_sal;
/
```

```
FUNCTION get_sal Compiled.
```

```
-- Invoke the function as an expression or as
-- a parameter value.
```

```
EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed
Added Dept: Media
24000
```

225

Stored Function: Example

The `get_sal` function is created with a single input parameter and returns the salary as a number. Execute the command as shown, or save it in a script file and run the script to create the `get_sal` function.

The `get_sal` function follows a common programming practice of using a single `RETURN` statement that returns a value assigned to a local variable. If your function has an exception section, then it may also contain a `RETURN` statement.

Invoke a function as part of a PL/SQL expression because the function will return a value to the calling environment. The second code box uses the SQL*Plus `EXECUTE` command to call the `DBMS_OUTPUT.PUT_LINE` procedure whose argument is the return value from the function `get_sal`. In this case, `get_sal` is invoked first to calculate the salary of the employee with ID 100. The salary value returned is supplied as the value of the `DBMS_OUTPUT.PUT_LINE` parameter, which displays the result (if you have executed a `SET SERVEROUTPUT ON`).

Note: A function must always return a value. The example does not return a value if a row is not found for a given `id`. Ideally, create an exception handler to return a value as well.

Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables  
VARIABLE b_salary NUMBER  
EXECUTE :b_salary := get_sal(100)
```

```
anonymous block completed  
b_salary  
-----  
24000
```

```
-- As a PL/SQL expression, get the results using a local  
-- variable  
SET SERVEROUTPUT ON  
DECLARE  
    sal employees.salary%type;  
BEGIN  
    sal := get_sal(100);  
    DBMS_OUTPUT.PUT_LINE('The salary is: '|| sal);  
END;  
/
```

```
anonymous block completed  
The salary is: 24000
```

226

If functions are well designed, they can be powerful constructs. Functions can be invoked in the following ways:

As part of PL/SQL expressions: You can use host or local variables to hold the returned value from a function. The first example in the slide uses a host variable and the second example uses a local variable in an anonymous block.

Note: The benefits and restrictions that apply to functions when used in a SQL statement are discussed on the next few pages.

Using Different Methods for Executing Functions

```
-- Use as a parameter to another subprogram  
EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed  
24000
```

```
-- Use in a SQL statement (subject to restrictions)  
SELECT job_id, get_sal(employee_id)  
FROM employees;
```

JOB_ID	GET_SAL(EMPLOYEE_ID)
SH_CLERK	2600
SH_CLERK	2600
AD_ASST	4400
MK_MAN	13000
...	
SH_CLERK	3100
SH_CLERK	3000

107 rows selected

227

As a parameter to another subprogram: The first example in the slide demonstrates this usage. The `get_sal` function with all its arguments is nested in the parameter required by the `DBMS_OUTPUT.PUT_LINE` procedure. This comes from the concept of nesting functions as discussed in the course titled *Oracle Database: SQL Fundamentals I*.

As an expression in a SQL statement: The second example in the slide shows how a function can be used as a single-row function in a SQL statement.

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the WHERE clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

SQL statements can reference PL/SQL user-defined functions anywhere a SQL expression is allowed. For example, a user-defined function can be used anywhere that a built-in SQL function, such as `UPPER()`, can be placed.

Advantages

Permits calculations that are too complex, awkward, or unavailable with SQL. Functions increase data independence by processing complex data analysis within the Oracle server, rather than by retrieving the data into an application

Increases efficiency of queries by performing functions in the query rather than in the application

Manipulates new types of data (for example, latitude and longitude) by encoding character strings and using functions to operate on the strings

Using a Function in a SQL Expression: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

FUNCTION tax(value Compiled.			
EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarrino	7700	616
112	Uman	7800	624
113	Popp	6900	552

229

Function in SQL Expressions: Example

The example in the slide shows how to create a `tax` function to calculate income tax. The function accepts a `NUMBER` parameter and returns the calculated income tax based on a simple flat tax rate of 8%.

To execute the code shown in the slide example in SQL Developer, enter the code in the SQL Worksheet, and then click the **Run Script** icon. The `tax` function is invoked as an expression in the `SELECT` clause along with the employee ID, last name, and salary for employees in a department with ID 100. The return result from the `tax` function is displayed with the regular output from the query.

User-defined functions act like built-in single-row functions and can be used in:

- The SELECT list or clause of a query
- Conditional expressions of the WHERE and HAVING clauses
- The CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses of a query
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

230

A PL/SQL user-defined function can be called from any SQL expression where a built-in single-row function can be called as shown in the following example.

```
SELECT employee_id, tax(salary)
  FROM employees
 WHERE tax(salary) > (SELECT MAX(tax(salary)))
   FROM employees
 WHERE department_id = 30)
 ORDER BY tax(salary) DESC;
```

EMPLOYEE_ID	TAX(SALARY)
100	1920
101	1360
102	1360
145	1120
146	1080
201	1040
205	960
147	960
108	960
168	920

10 rows selected

- User-defined functions that are callable from SQL expressions must:
 - Be stored in the database
 - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
 - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
 - You must own the function or have the `EXECUTE` privilege
 - You may need to enable the `PARALLEL_ENABLE` keyword to allow a parallel execution of the SQL statement

231

The user-defined PL/SQL functions that are callable from SQL expressions must meet the following requirements.

The function must be stored in the database.

The function parameters must be `IN` and of valid SQL data types.

The functions must return data types that are valid SQL data types.

They cannot be PL/SQL-specific data types such as `BOOLEAN`, `RECORD`, or `TABLE`. The same restriction applies to the parameters of the function.

The following restrictions apply when calling a function in a SQL statement:

Parameters must use positional notation. Named notation is not supported.

You must own or have the `EXECUTE` privilege on the function.

You may need to enable the `PARALLEL_ENABLE` keyword to allow a parallel execution of the SQL statement using the function. Each parallel slave will have private copies of the function's local variables.

Other restrictions on a user-defined function include the following: It cannot be called from the `CHECK` constraint clause of a `CREATE TABLE` or `ALTER TABLE` statement. In addition, it cannot be used to specify a default value for a column. Only stored functions are callable from SQL statements. Stored

procedures cannot be called unless invoked from a function that meets the preceding requirements.

Functions called from:

- o A SELECT statement cannot contain DML statements
- o An UPDATE or DELETE statement on a table T cannot query or contain DML on the same table T
- o SQL statements cannot end transactions (that is, cannot execute COMMIT or ROLLBACK operations)

Note: Calls to subprograms that break these restrictions are also not allowed in the function.

232

To execute a SQL statement that calls a stored function, the Oracle server

must know whether the function is free of specific side effects. The side effects are unacceptable changes to database tables.

Additional restrictions apply when a function is called in expressions of SQL statements:

When a function is called from a SELECT statement or a parallel UPDATE or DELETE statement, the function cannot modify database tables.

When a function is called from an UPDATE or DELETE statement, the function cannot query or modify database tables modified by that statement.

When a function is called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute directly or indirectly through another subprogram or SQL transaction control statements such as:

A COMMIT or ROLLBACK statement

A session control statement (such as SET ROLE)

A system control statement (such as ALTER SYSTEM)

Any DDL statements (such as CREATE) because they are

followed by an automatic commit

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_sal NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                        email, hire_date, job_id, salary)
    VALUES(1, 'Frost', 'jfrost@company.com',
           SYSDATE, 'SA_MAN', p_sal);
  RETURN (p_sal + 100);
END;
```



```
UPDATE employees
  SET salary = dml_call_sql(2000)
 WHERE employee_id = 170;
```

```
FUNCTION dml_call_sql(p_sal Compiled.
Error starting at line 1 in command:
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170
Error report:
SQL Error: ORA-06501: table SPAR.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "SPAR.DML_CALL_SQL", line 2
04051. 00000 -  "table %s.%s is mutating, trigger/function may not see it"
*Cause:  A trigger (or a user defined plsql function that is referenced in
        this statement) attempted to look at (or modify) a table that was
        in the middle of being modified by the statement which fired it.
*Action: Rewrite the trigger (or function) so it does not read that table.
```

233

The `dml_call_sql` function in the slide contains an `INSERT` statement that inserts a new record into the `EMPLOYEES` table and returns the input salary value incremented by 100. This function is invoked in the `UPDATE` statement that modifies the salary of employee 170 to the amount returned from the function. The `UPDATE` statement fails with an error indicating that the table is mutating (that is, changes are already in progress in the same table). In the following example, the `query_call_sql` function queries the `SALARY` column of the `EMPLOYEES` table:

```
CREATE OR REPLACE FUNCTION query_call_sql(p_a
  NUMBER)
  RETURN NUMBER IS
  v_s NUMBER;
BEGIN
  SELECT salary INTO v_s FROM employees
  WHERE employee_id = 170;
  RETURN (v_s + p_a);
END;
```

When invoked from the following `UPDATE` statement, it returns the error message similar to the error message shown in the slide:

```
UPDATE employees SET salary = query_call_sql(100)
WHERE employee_id = 170;
```

- PL/SQL allows arguments in a subroutine call to be specified using positional, named, or mixed notation.
- Prior to Oracle Database 19c, only the positional notation is supported in calls from SQL.
- Starting in Oracle Database 19c, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements.
- For long parameter lists, with most having default values, you can omit values from the optional parameters.
- You can avoid duplicating the default value of the optional parameter at each call site.

Named and Mixed Notation from SQL:

```
CREATE OR REPLACE FUNCTION f(
    p_parameter_1 IN NUMBER DEFAULT 1,
    p_parameter_5 IN NUMBER DEFAULT 5)
RETURN NUMBER
IS
    v_var number;
BEGIN
    v_var := p_parameter_1 + (p_parameter_5 * 2);
    RETURN v_var;
END f;
/
```

FUNCTION f compiled

```
SELECT f(p_parameter_5 => 10) FROM DUAL;
```

```
F(P_PARAMETER_5=>10)
-----
21
1 rows selected
```

235

Example of Using Named and Mixed Notation from a SQL Statement

In the example in the slide, the call to the function `f` within the SQL SELECT statement uses the named notation. Before Oracle Database 19c, you could not use the named or mixed notation when passing parameters to a function from within a SQL statement. Before Oracle Database 19c, you received the following error:

```
SELECT f(p_parameter_5 => 10) FROM DUAL;
```

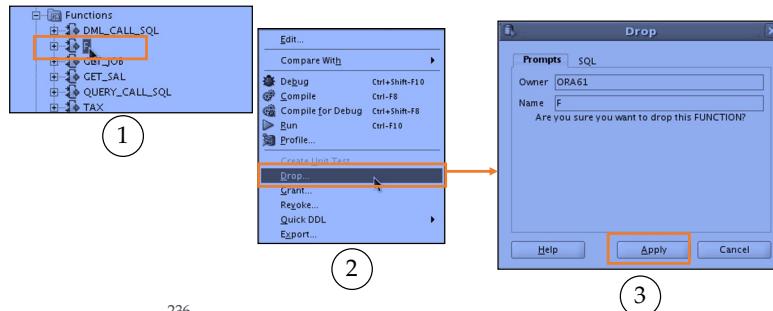
```
ORA-00907: missing right parenthesis
```

Removing Functions:

- Using the DROP statement:

```
DROP FUNCTION f;
```

- Using SQL Developer:



236

Removing Functions

Using the `DROP` Statement

When a stored function is no longer required, you can use a SQL statement in SQL*Plus to drop it. To remove a stored function by using SQL*Plus, execute the `DROP FUNCTION` SQL command.

Using `CREATE OR REPLACE` Versus `DROP` and `CREATE`

The `REPLACE` clause in the `CREATE OR REPLACE` syntax is equivalent to dropping a function and re-creating it. When you use the `CREATE OR REPLACE` syntax, the privileges granted on this object to other users remain the same. When you `DROP` a function and then re-create it, all the privileges granted on this function are automatically revoked.

Using SQL Developer

To drop a function in SQL Developer, right-click the function name in the **Functions** node, and then select **Drop**. The **Drop** dialog box is displayed. To drop the function, click **Apply**.

Viewing Functions Using Data Dictionary Views

```
DESCRIBE USER_SOURCE
```

Name	Null	Type
NAME		VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)

4 rows selected

```
SELECT text
FROM user_source
WHERE type = 'FUNCTION'
ORDER BY line;
```

Results:

TEXT
1 FUNCTION lxx(p_value IN NUMBER)
2 FUNCTION query_call_sq(p_a NUMBER) RETURN NUMBER IS
3 FUNCTION get_sal
4 FUNCTION dml_call_sq(p_sal NUMBER)
5 RETURN NUMBER IS
6 RETURN NUMBER IS
7 (p_id employees employee_id%TYPE) RETURN NUMBER IS
8 v_s NUMBER,

237

The source code for PL/SQL functions is stored in the data dictionary tables. The source code is accessible for PL/SQL functions that are successfully or unsuccessfully compiled. To view the PL/SQL function code stored in the data dictionary, execute a `SELECT` statement on the following tables where the `TYPE` column value is `FUNCTION`:

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

The second example in the slide uses the `USER_SOURCE` table to display the source code for all the functions in your schema.

You can also use the `USER_OBJECTS` data dictionary view to display a list of your function names.

Note: The output of the second code example in the slide was generated using the Execute Statement (F9) icon on the toolbar to provide better-formatted output.

- You can use the debugger to control the execution of your PL/SQL program.
- To debug a PL/SQL subprogram, a *security administrator* needs to grant the following privileges to the application developer:
 - DEBUG ANY PROCEDURE
 - DEBUG CONNECT SESSION

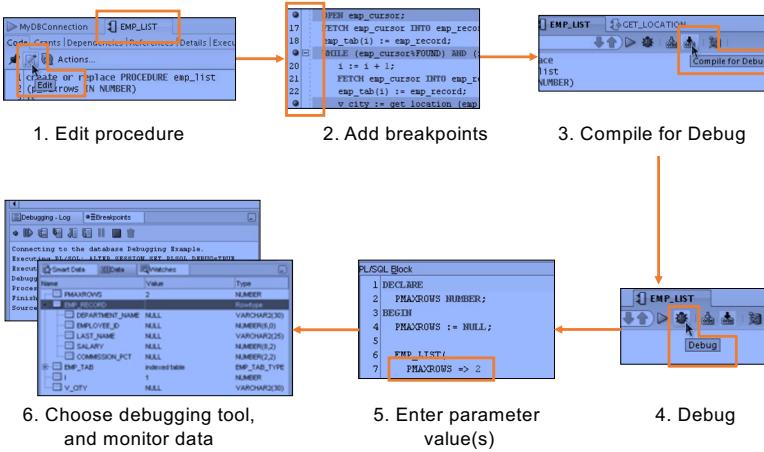
```
GRANT DEBUG ANY PROCEDURE TO ora61;  
GRANT DEBUG CONNECT SESSION TO ora61;
```

238

Moving Through Code While Debugging

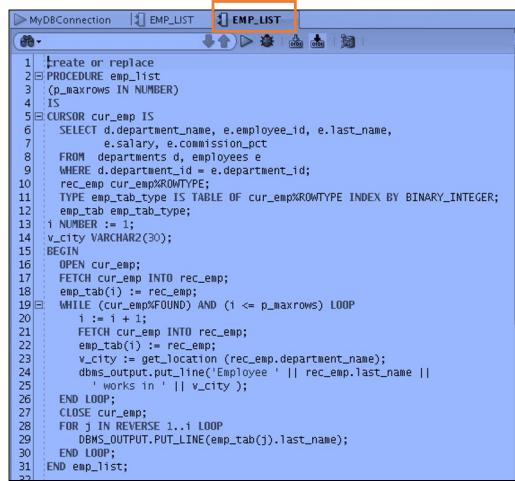
The SQL Developer debugger enables you to control the execution of your program. You can control whether your program executes a single line of code, an entire subprogram (procedure or function), or an entire program block. By manually controlling when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

Debugging a Subprogram: Overview



239

The Procedure or Function Code Editing Tab



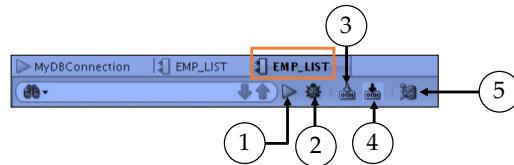
A screenshot of the Oracle SQL Developer interface. The title bar shows 'MyDBConnection' and two tabs: 'EMP_LIST' and 'EMP_LIST'. The 'EMP_LIST' tab is active and highlighted with a red box. The main area displays the PL/SQL code for the 'emp_list' procedure. The code uses cursor variables and a temporary table to list employees from different departments, printing their names and city.

```
1  CREATE OR REPLACE
2  PROCEDURE emp_list
3  (p_maxrows IN NUMBER)
4  IS
5  CURSOR cur_emp IS
6    SELECT d.department_name, e.employee_id, e.last_name,
7           e.salary, e.commission_pct
8    FROM departments d, employees e
9   WHERE d.department_id = e.department_id;
10  TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11  emp_tab emp_tab_type;
12  i NUMBER := 1;
13  v_city VARCHAR2(30);
14
15  BEGIN
16    OPEN cur_emp;
17    FETCH cur_emp INTO rec_emp;
18    emp_tab(i) := rec_emp;
19    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20      i := i + 1;
21      FETCH cur_emp INTO rec_emp;
22      emp_tab(i) := rec_emp;
23      v_city := get_location(rec_emp.department_name);
24      dbms_output.put_line('Employee ' || rec_emp.last_name || '
25      || works_in ' || v_city);
26    END LOOP;
27    CLOSE cur_emp;
28    FOR i IN REVERSE 1..1 LOOP
29      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30    END LOOP;
31  END emp_list;
32
```

240

This tab displays a toolbar and the text of the subprogram, which you can edit. You can set and unset breakpoints for debugging by clicking to the left of the thin vertical line beside each statement with which you want to associate a breakpoint. (When a breakpoint is set, a red circle is displayed.)

The Procedure or Function Tab Toolbar



Icon	Description
1. Run	Starts normal execution of the function or procedure, and displays the results in the Running - Log tab
2. Debug	Executes the subprogram in debug mode, and displays the Debugging - Log tab, which includes the debugging toolbar for controlling execution
3. Compile	Compiles the subprogram
4. Compile for Debug	Compiles the subprogram so that it can be debugged
5. Profile	Displays the Profile window that you use to specify parameter values for running, debugging, or profiling a PL/SQL function or procedure

241

The Debugging – Log Tab Toolbar



Icon	Description
1. Find Execution Point	Goes to the next execution point
2. Step Over	Bypasses the next subprogram and goes to the next statement after the subprogram
3. Step Into	Executes a single program statement at a time. If the execution point is located on a call to a subprogram, it steps into the first statement in that subprogram
4. Step Out	Leaves the current subprogram and goes to the next statement with a breakpoint

242

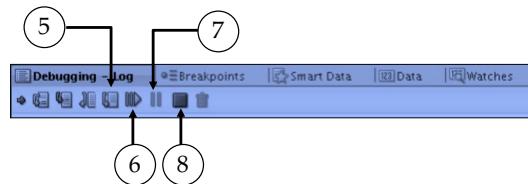
The Debugging - Log contains the debugging toolbar and informational messages.

1. **Find Execution Point:** Goes to the execution point (the next line of source code to be executed by the debugger)
2. **Step Over:** Bypasses the next subprogram (unless the subprogram has a breakpoint) and goes to the next statement after the subprogram. If the execution point is located on a subprogram call, it runs that subprogram without stopping (instead of stepping into it), and then positions the execution point on the statement that follows the call. If the execution point is located on the last statement of a subprogram, Step Over returns from the subprogram, placing the execution point on the line of code that follows the call to the subprogram from which you are returning.
3. **Step Into:** Executes a single program statement at a time. If the execution point is located on a call to a subprogram, Step Into steps into that subprogram and places the execution point on its first statement. If the execution point is located on the last statement of a subprogram, Step Into returns from the subprogram, placing the execution point on the line of code that follows the call to the

subprogram from which you are returning.

4. **Step Out:** Leaves the current subprogram and goes to the next statement

The Debugging – Log Tab Toolbar



Icon	Description
5. Step to End of Method	Goes to the last statement of the current subprogram
6. Resume	Continues execution
7. Pause	Halts execution but does not exit
8. Terminate	Halts and exits the execution

243

5. **Step to End of Method:** Goes to the last statement of the current subprogram
6. **Resume:** Continues execution
7. **Pause:** Halts execution but does not exit, thus allowing you to resume execution
8. **Terminate:** Halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the subprogram, click the Run or Debug icon in the Source tab toolbar.

Additional Tabs

Debugging - Log	Breakpoints	Smart Data	Data	Watches
Name	Value	Type		
P_MAXROWS	100	NUMBER		
REC_EMP		Rowtype		
EMP_TAB	indexed table	EMP_TAB_TYPE		
I	1	NUMBER		
V_CITY	NULL	VARCHAR2(30)		

Tab	Description
Breakpoints	Displays breakpoints, both system-defined and user-defined.
Smart Data	Displays information about variables. You can specify these preferences by right-clicking in the Smart Data window and selecting Preferences.
Data	Located under the code text area; displays information about all variables
Watches	Located under the code text area; displays information about watches

244

Setting Expression Watches

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watches window displays the current value of the expression. As your program runs, the value of the watch changes as your program updates the values of the variables in the watch expression.

A watch evaluates an expression according to the current context, which is controlled by the selection in the Stack window. If you move to a new context, the expression is reevaluated for the new context. If the execution point moves to a location where any of the variables in the watch expression are undefined, the entire watch expression becomes undefined. If the execution point returns to a location where the watch expression can be evaluated, the Watches window again displays the value of the watch expression.

To open the Watches window, click View, then Debugger, then Watches.

To add a watch, right-click in the Watches window and select Add Watch. To edit a watch, right-click in the Watches window and select Edit Watch.

Note: If you cannot see some of the debugging tabs described in this lesson, you can re-display such tabs by using the View > Debugger menu option.

Debugging a Procedure Example: Creating a New emp_list Procedure

```
1 CREATE OR REPLACE PROCEDURE emp_list(pmaxrows IN NUMBER) AS
2 CURSOR emp_cursor IS
3 SELECT d.department_name,
4       e.employee_id,
5       e.last_name,
6       e.salary,
7       e.commission_pct
8 FROM departments d,
9      employees e
10 WHERE d.department_id = e.department_id;
11 emp_record emp_cursor%rowtype;
12 type emp_tab_type is TABLE OF emp_cursor%rowtype INDEX BY binary_integer;
13 emp_tab emp_tab_type;
14 i NUMBER := 1;
15 v_city VARCHAR2(30);
16 BEGIN
17
18   OPEN emp_cursor;
19   FETCH emp_cursor
20   INTO emp_record;
21   emp_tab(i) := emp_record;
22   WHILE(emp_cursor%FOUND)
23     AND i < pmaxrows
24   LOOP
25     i := i + 1;
26     FETCH emp_cursor
27     INTO emp_record;
28     emp_tab(i) := emp_record;
29     v_city := get_location(emp_record.department_name);
30     DBMS_OUTPUT.PUT_LINE('Employee '||emp_record.last_name||' works in '||v_city);
31   END LOOP;
32
33   CLOSE emp_cursor;
34   FOR j IN REVERSE 1 .. i
35   LOOP
36     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
37   END LOOP;
38 END emp_list;
```

245

The emp_list procedure gathers employee information such as the employee's department name, ID, name, salary, and commission percentage. The procedure creates a record to store the employee's information. The procedure also creates a table that can hold multiple records of employees. "i" is a counter.

The code opens the cursor and fetches the employees' records. The code also checks whether or not there are more records to fetch or whether the number of records fetched so far is less than the number of records that you specify. The code eventually prints out the employees' information. The procedure also calls the get_location function that returns the name of the city in which an employee works.

Note: Make sure that you are displaying the procedure code in edit mode. To edit the procedure code, click the Edit icon on the procedure's toolbar.

Debugging a Procedure Example: Creating a New get_location Function

```
1 CREATE OR REPLACE FUNCTION get_location(p_deptname IN VARCHAR2) RETURN VARCHAR2 AS
2   v_loc_id NUMBER;
3   v_city VARCHAR2(30);
4 BEGIN
5   SELECT d.location_id,
6         l.city
7     INTO v_loc_id,
8          v_city
8    FROM departments d,
9         locations l
10   WHERE UPPER(department_name) = UPPER(p_deptname)
11   AND d.location_id = l.location_id;
12   RETURN v_city;
13 END get_location;
```

246

Debugging a Procedure Example: Creating a New Procedure

This function returns the city in which an employee works. It is called from the `emp_list` procedure.

Setting Breakpoints and Compiling emp_list for Debug Mode

The screenshot shows the Oracle SQL Developer interface with the 'MyDBConnection' connection selected. In the center, the code editor displays the PL/SQL procedure 'EMP_LIST'. Four breakpoints are highlighted with orange boxes: one at the beginning of the cursor declaration (line 5), one at the start of the loop (line 18), one in the loop body (line 23), and one at the end of the cursor declaration (line 31). The status bar at the bottom right shows the message '247'. The 'Messages - Log' tab at the bottom indicates that the procedure was successfully compiled.

```
1| create or replace
2| PROCEDURE emp_list
3| (p_maxrows IN NUMBER)
4| IS
5| CURSOR cur_emp IS
6|   SELECT d.department_name, e.employee_id, e.last_name,
7|   e.salary, e.commission_pct
8|   FROM department d, employees e
9|   WHERE d.department_id = e.department_id;
10| TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11| emp_tab emp_tab_type;
12| i NUMBER := 1;
13| l NUMBER := 1;
14| v_city VARCHAR2(30);
15|
16| OPEN cur_emp;
17| FETCH cur_emp INTO rec_emp;
18| WHILE (cur_emp%FOUND) AND (l <= p_maxrows) LOOP
19|   FETCH cur_emp INTO rec_emp;
20|   emp_tab(l) := rec_emp;
21|   v_city := get_location(rec_emp.department_name);
22|   dbms_output.put_line(rec_emp.last_name || ' works in ' || v_city );
23|   l := l + 1;
24| END LOOP;
25| CLOSE cur_emp;
26| emp_tab%ROWTYPE l;
27| DBMS_OUTPUT.PUT_LINE(emp_tab(l).last_name);
28| END LOOP;
29| END emp_list;
30|
31|
32|
```

Messages - Log
EMP_LIST Compiled

247

In the example in the slide, the `emp_list` procedure is displayed in edit mode, and four breakpoints are added to various locations in the code. To compile the procedure for debugging, right-click the code, and then select Compile for Debug from the shortcut menu. The Messages – Log tab displays the message that the procedure was compiled.

Compiling the get_location Function for Debug Mode

The screenshot shows the Oracle SQL Developer interface. A PL/SQL editor window is open with the title bar 'GET_LOCATION'. The code in the editor is:

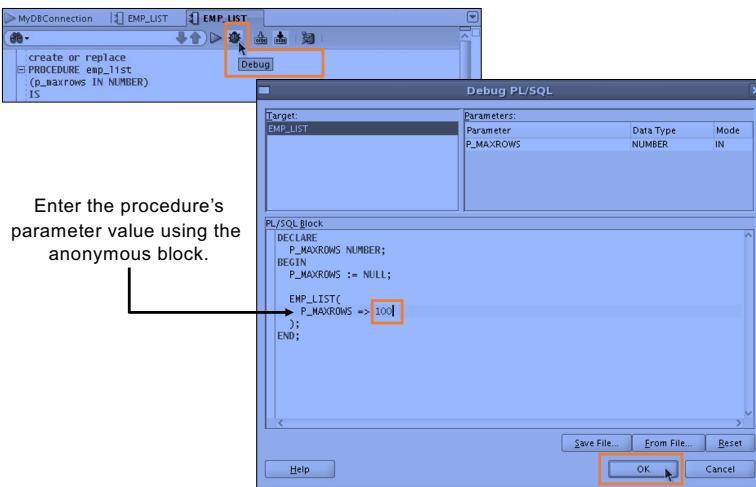
```
1 CREATE OR REPLACE
2 FUNCTION get_location
3 ( p_deptname IN VARCHAR2 ) RETURN VARCHAR2
4 AS
5   v_loc_id NUMBER;
6   v_city VARCHAR2(30);
7 BEGIN
8   SELECT d.location_id, l.city INTO v_loc_id, v_city
9   FROM departments d, locations l
10  WHERE upper(department_name) = upper(p_deptname)
11  AND d.location_id = l.location_id;
12  RETURN v_city;
13 END GET_LOCATION;
14
```

A right-click context menu is displayed over the code area, with the option 'Compile for Debug' highlighted. The status bar at the bottom shows the message 'GET_LOCATION Compiled'.

248

In the example in the slide, the `get_location` function is displayed in edit mode. To compile the function for debugging, right-click the code, and then select `Compile for Debug` from the shortcut menu. The `Messages – Log` tab displays the message that the function was compiled.

Debugging emp_list and Entering Values for the PMAXROWS Parameter



249

The next step in the debugging process is to debug the procedure using any of the several available methods mentioned earlier, such as clicking the Debug icon on the procedure's toolbar. An anonymous block is displayed, where you are prompted to enter the parameters for this procedure. `emp_list` has one parameter, `PMAXROWS`, which specifies the number of records to return. Replace the second `PMAXROWS` with a number such as 100, and then click OK.

Debugging emp_list : Step Into (F7) the Code

Program control stops at first breakpoint.

1

```
1 CREATE OR REPLACE PROCEDURE emp_list
2   (p_maxrows IN NUMBER)
3 IS
4   CURSOR cur_emp IS
5     SELECT d.department_name, e.employee_id, e.last_name,
6           e.salary, e.commission_pct
7     FROM employees d, employee e
8    WHERE d.department_id = e.department_id;
9
10  TYPE emp_tab%ROWTYPE IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11  emp_tab emp_tab%TYPE;
12  i NUMBER := 1;
13  v_city VARCHAR2(30);
14
15  BEGIN
16    OPEN cur_emp;
17    FETCH cur_emp INTO rec_emp;
18    emp_tab(i) := rec_emp;
19    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20      i := i + 1;
21      FETCH cur_emp INTO rec_emp;
22      emp_tab(i) := rec_emp;
23      v_city := get_location (rec_emp.department_name);
24      dbms_output.put_line('Employee ' || rec_emp.last_name ||
25                           ' works in ' || v_city );
26    END LOOP;
27    CLOSE cur_emp;
28    FOR j IN REVERSE 1..1 LOOP
29      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30    END LOOP;
31  END emp_list;
32
```

Debugging - Log
Debugger accepted connection from database on port 4000.
Processing 59 classes that have already been prepared...
Finished processing prepared classes.
Source breakpoint occurred at line 15 of EMP_LIST.pis.
Messages Debugging

250

The Step Into Debugging Tool

The Step Into command executes a single program statement at a time. If the execution point is located on a call to a subprogram, the Step Into command steps into that subprogram and places the execution point on the subprogram's first statement. If the execution point is located on the last statement of a subprogram, choosing Step Into causes the debugger to return from the subprogram, placing the execution point on the line of code that follows the call to the subprogram you are returning from. The term *single stepping* refers to using Step Into to run successively through the statements in your program code. You can step into a subprogram in any of the following ways: Select Debug > Step Into, press F7, or click the Step Into icon in the Debugging – Log toolbar.

In the example in the slide, the program control is stopped at the first breakpoint in the code. The arrow next to the breakpoint indicated that this is the line of code that will be executed next. Note the various tabs at the bottom of the window.

Note: The Step Into and Step Over commands offer the simplest way of moving through your program code. While the two commands are very similar, they each offer a different way to control code execution.

Debugging emp_list: Step Into (F7) the Code

The screenshot shows the Oracle SQL Developer interface with the following details:

- Step 1:** The code editor highlights the line `OPEN cur_emp;`. A callout arrow points from this line to the cursor definition in the code.
- Step 2:** The code editor highlights the line `OPEN cur_emp;`. A callout arrow points from this line to the cursor execution in the code.
- Step 3:** The code editor highlights the line `SELECT d.department_name, e.employee_id,`. A callout arrow points from this line to the select statement in the code.

```

1: 14: OPEN cur_emp;
2: 15:   FOR i IN REVERSE 1..1 LOOP
3: 16:     emp_tab(i) := rec_emp;
4: 17:   WHILE (cur_emp%FOUND) AND (1 <= p_maxrows) LOOP
5: 18:     emp_tab(i+1) := rec_emp;
6: 19:     emp_tab(i) := rec_emp;
7: 20:     emp_tab(i) := rec_emp;
8: 21:     emp_tab(i) := rec_emp;
9: 22:     emp_tab(i) := rec_emp;
10: 23:     emp_tab(i) := rec_emp;
11: 24:     emp_tab(i) := rec_emp;
12: 25:     emp_tab(i) := rec_emp;
13: 26:     emp_tab(i) := rec_emp;
14: 27:     emp_tab(i) := rec_emp;
15: 28:     emp_tab(i) := rec_emp;
16: 29:     emp_tab(i) := rec_emp;
17: 30:   END LOOP;
18: 31:   CLOSE cur_emp;
19: 32:   FOR j IN REVERSE 1..1 LOOP
20: 33:     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
21: 34:   END LOOP;
22: 35: END emp_list;
23: 36:
24: 37:
25: 38:
26: 39:
27: 40:
28: 41:
29: 42:
30: 43:
31: 44:
32: 45:
33: 46:
34: 47:
35: 48:
36: 49:
37: 50:
38: 51:
39: 52:
40: 53:
41: 54:
42: 55:
43: 56:
44: 57:
45: 58:
46: 59:
47: 60:
48: 61:
49: 62:
50: 63:
51: 64:
52: 65:
53: 66:
54: 67:
55: 68:
56: 69:
57: 70:
58: 71:
59: 72:
60: 73:
61: 74:
62: 75:
63: 76:
64: 77:
65: 78:
66: 79:
67: 80:
68: 81:
69: 82:
70: 83:
71: 84:
72: 85:
73: 86:
74: 87:
75: 88:
76: 89:
77: 90:
78: 91:
79: 92:
80: 93:
81: 94:
82: 95:
83: 96:
84: 97:
85: 98:
86: 99:
87: 100:
88: 101:
89: 102:
90: 103:
91: 104:
92: 105:
93: 106:
94: 107:
95: 108:
96: 109:
97: 110:
98: 111:
99: 112:
100: 113:
101: 114:
102: 115:
103: 116:
104: 117:
105: 118:
106: 119:
107: 120:
108: 121:
109: 122:
110: 123:
111: 124:
112: 125:
113: 126:
114: 127:
115: 128:
116: 129:
117: 130:
118: 131:
119: 132:
120: 133:
121: 134:
122: 135:
123: 136:
124: 137:
125: 138:
126: 139:
127: 140:
128: 141:
129: 142:
130: 143:
131: 144:
132: 145:
133: 146:
134: 147:
135: 148:
136: 149:
137: 150:
138: 151:
139: 152:
140: 153:
141: 154:
142: 155:
143: 156:
144: 157:
145: 158:
146: 159:
147: 160:
148: 161:
149: 162:
150: 163:
151: 164:
152: 165:
153: 166:
154: 167:
155: 168:
156: 169:
157: 170:
158: 171:
159: 172:
160: 173:
161: 174:
162: 175:
163: 176:
164: 177:
165: 178:
166: 179:
167: 180:
168: 181:
169: 182:
170: 183:
171: 184:
172: 185:
173: 186:
174: 187:
175: 188:
176: 189:
177: 190:
178: 191:
179: 192:
180: 193:
181: 194:
182: 195:
183: 196:
184: 197:
185: 198:
186: 199:
187: 200:
188: 201:
189: 202:
190: 203:
191: 204:
192: 205:
193: 206:
194: 207:
195: 208:
196: 209:
197: 210:
198: 211:
199: 212:
199: 213:
200: 214:
201: 215:
202: 216:
203: 217:
204: 218:
205: 219:
206: 220:
207: 221:
208: 222:
209: 223:
210: 224:
211: 225:
212: 226:
213: 227:
214: 228:
215: 229:
216: 230:
217: 231:
218: 232:
219: 233:
220: 234:
221: 235:
222: 236:
223: 237:
224: 238:
225: 239:
226: 240:
227: 241:
228: 242:
229: 243:
230: 244:
231: 245:
232: 246:
233: 247:
234: 248:
235: 249:
236: 250:
237: 251:
238: 252:
239: 253:
240: 254:
241: 255:
242: 256:
243: 257:
244: 258:
245: 259:
246: 260:
247: 261:
248: 262:
249: 263:
250: 264:
251: 265:
252: 266:
253: 267:
254: 268:
255: 269:
256: 270:
257: 271:
258: 272:
259: 273:
260: 274:
261: 275:
262: 276:
263: 277:
264: 278:
265: 279:
266: 280:
267: 281:
268: 282:
269: 283:
270: 284:
271: 285:
272: 286:
273: 287:
274: 288:
275: 289:
276: 290:
277: 291:
278: 292:
279: 293:
280: 294:
281: 295:
282: 296:
283: 297:
284: 298:
285: 299:
286: 300:
287: 301:
288: 302:
289: 303:
290: 304:
291: 305:
292: 306:
293: 307:
294: 308:
295: 309:
296: 310:
297: 311:
298: 312:
299: 313:
299: 314:
300: 315:
301: 316:
302: 317:
303: 318:
304: 319:
305: 320:
306: 321:
307: 322:
308: 323:
309: 324:
310: 325:
311: 326:
312: 327:
313: 328:
314: 329:
315: 330:
316: 331:
317: 332:
318: 333:
319: 334:
320: 335:
321: 336:
322: 337:
323: 338:
324: 339:
325: 340:
326: 341:
327: 342:
328: 343:
329: 344:
330: 345:
331: 346:
332: 347:
333: 348:
334: 349:
335: 350:
336: 351:
337: 352:
338: 353:
339: 354:
340: 355:
341: 356:
342: 357:
343: 358:
344: 359:
345: 360:
346: 361:
347: 362:
348: 363:
349: 364:
350: 365:
351: 366:
352: 367:
353: 368:
354: 369:
355: 370:
356: 371:
357: 372:
358: 373:
359: 374:
360: 375:
361: 376:
362: 377:
363: 378:
364: 379:
365: 380:
366: 381:
367: 382:
368: 383:
369: 384:
370: 385:
371: 386:
372: 387:
373: 388:
374: 389:
375: 390:
376: 391:
377: 392:
378: 393:
379: 394:
380: 395:
381: 396:
382: 397:
383: 398:
384: 399:
385: 400:
386: 401:
387: 402:
388: 403:
389: 404:
390: 405:
391: 406:
392: 407:
393: 408:
394: 409:
395: 410:
396: 411:
397: 412:
398: 413:
399: 414:
399: 415:
400: 416:
401: 417:
402: 418:
403: 419:
404: 420:
405: 421:
406: 422:
407: 423:
408: 424:
409: 425:
410: 426:
411: 427:
412: 428:
413: 429:
414: 430:
415: 431:
416: 432:
417: 433:
418: 434:
419: 435:
420: 436:
421: 437:
422: 438:
423: 439:
424: 440:
425: 441:
426: 442:
427: 443:
428: 444:
429: 445:
430: 446:
431: 447:
432: 448:
433: 449:
434: 450:
435: 451:
436: 452:
437: 453:
438: 454:
439: 455:
440: 456:
441: 457:
442: 458:
443: 459:
444: 460:
445: 461:
446: 462:
447: 463:
448: 464:
449: 465:
450: 466:
451: 467:
452: 468:
453: 469:
454: 470:
455: 471:
456: 472:
457: 473:
458: 474:
459: 475:
460: 476:
461: 477:
462: 478:
463: 479:
464: 480:
465: 481:
466: 482:
467: 483:
468: 484:
469: 485:
470: 486:
471: 487:
472: 488:
473: 489:
474: 490:
475: 491:
476: 492:
477: 493:
478: 494:
479: 495:
480: 496:
481: 497:
482: 498:
483: 499:
484: 500:
485: 501:
486: 502:
487: 503:
488: 504:
489: 505:
490: 506:
491: 507:
492: 508:
493: 509:
494: 510:
495: 511:
496: 512:
497: 513:
498: 514:
499: 515:
499: 516:
500: 517:
501: 518:
502: 519:
503: 520:
504: 521:
505: 522:
506: 523:
507: 524:
508: 525:
509: 526:
510: 527:
511: 528:
512: 529:
513: 530:
514: 531:
515: 532:
516: 533:
517: 534:
518: 535:
519: 536:
520: 537:
521: 538:
522: 539:
523: 540:
524: 541:
525: 542:
526: 543:
527: 544:
528: 545:
529: 546:
530: 547:
531: 548:
532: 549:
533: 550:
534: 551:
535: 552:
536: 553:
537: 554:
538: 555:
539: 556:
540: 557:
541: 558:
542: 559:
543: 560:
544: 561:
545: 562:
546: 563:
547: 564:
548: 565:
549: 566:
550: 567:
551: 568:
552: 569:
553: 570:
554: 571:
555: 572:
556: 573:
557: 574:
558: 575:
559: 576:
560: 577:
561: 578:
562: 579:
563: 580:
564: 581:
565: 582:
566: 583:
567: 584:
568: 585:
569: 586:
570: 587:
571: 588:
572: 589:
573: 590:
574: 591:
575: 592:
576: 593:
577: 594:
578: 595:
579: 596:
580: 597:
581: 598:
582: 599:
583: 600:
584: 601:
585: 602:
586: 603:
587: 604:
588: 605:
589: 606:
590: 607:
591: 608:
592: 609:
593: 610:
594: 611:
595: 612:
596: 613:
597: 614:
598: 615:
599: 616:
599: 617:
600: 618:
601: 619:
602: 620:
603: 621:
604: 622:
605: 623:
606: 624:
607: 625:
608: 626:
609: 627:
610: 628:
611: 629:
612: 630:
613: 631:
614: 632:
615: 633:
616: 634:
617: 635:
618: 636:
619: 637:
620: 638:
621: 639:
622: 640:
623: 641:
624: 642:
625: 643:
626: 644:
627: 645:
628: 646:
629: 647:
630: 648:
631: 649:
632: 650:
633: 651:
634: 652:
635: 653:
636: 654:
637: 655:
638: 656:
639: 657:
640: 658:
641: 659:
642: 660:
643: 661:
644: 662:
645: 663:
646: 664:
647: 665:
648: 666:
649: 667:
650: 668:
651: 669:
652: 670:
653: 671:
654: 672:
655: 673:
656: 674:
657: 675:
658: 676:
659: 677:
660: 678:
661: 679:
662: 680:
663: 681:
664: 682:
665: 683:
666: 684:
667: 685:
668: 686:
669: 687:
670: 688:
671: 689:
672: 690:
673: 691:
674: 692:
675: 693:
676: 694:
677: 695:
678: 696:
679: 697:
680: 698:
681: 699:
682: 700:
683: 701:
684: 702:
685: 703:
686: 704:
687: 705:
688: 706:
689: 707:
690: 708:
691: 709:
692: 710:
693: 711:
694: 712:
695: 713:
696: 714:
697: 715:
698: 716:
699: 717:
699: 718:
700: 719:
701: 720:
702: 721:
703: 722:
704: 723:
705: 724:
706: 725:
707: 726:
708: 727:
709: 728:
710: 729:
711: 730:
712: 731:
713: 732:
714: 733:
715: 734:
716: 735:
717: 736:
718: 737:
719: 738:
720: 739:
721: 740:
722: 741:
723: 742:
724: 743:
725: 744:
726: 745:
727: 746:
728: 747:
729: 748:
730: 749:
731: 750:
732: 751:
733: 752:
734: 753:
735: 754:
736: 755:
737: 756:
738: 757:
739: 758:
740: 759:
741: 760:
742: 761:
743: 762:
744: 763:
745: 764:
746: 765:
747: 766:
748: 767:
749: 768:
750: 769:
751: 770:
752: 771:
753: 772:
754: 773:
755: 774:
756: 775:
757: 776:
758: 777:
759: 778:
760: 779:
761: 780:
762: 781:
763: 782:
764: 783:
765: 784:
766: 785:
767: 786:
768: 787:
769: 788:
770: 789:
771: 790:
772: 791:
773: 792:
774: 793:
775: 794:
776: 795:
777: 796:
778: 797:
779: 798:
780: 799:
781: 800:
782: 801:
783: 802:
784: 803:
785: 804:
786: 805:
787: 806:
788: 807:
789: 808:
790: 809:
791: 810:
792: 811:
793: 812:
794: 813:
795: 814:
796: 815:
797: 816:
798: 817:
799: 818:
799: 819:
800: 820:
801: 821:
802: 822:
803: 823:
804: 824:
805: 825:
806: 826:
807: 827:
808: 828:
809: 829:
810: 830:
811: 831:
812: 832:
813: 833:
814: 834:
815: 835:
816: 836:
817: 837:
818: 838:
819: 839:
820: 840:
821: 841:
822: 842:
823: 843:
824: 844:
825: 845:
826: 846:
827: 847:
828: 848:
829: 849:
830: 850:
831: 851:
832: 852:
833: 853:
834: 854:
835: 855:
836: 856:
837: 857:
838: 858:
839: 859:
840: 860:
841: 861:
842: 862:
843: 863:
844: 864:
845: 865:
846: 866:
847: 867:
848: 868:
849: 869:
850: 870:
851: 871:
852: 872:
853: 873:
854: 874:
855: 875:
856: 876:
857: 877:
858: 878:
859: 879:
860: 880:
861: 881:
862: 882:
863: 883:
864: 884:
865: 885:
866: 886:
867: 887:
868: 888:
869: 889:
870: 890:
871: 891:
872: 892:
873: 893:
874: 894:
875: 895:
876: 896:
877: 897:
878: 898:
879: 899:
880: 900:
881: 901:
882: 902:
883: 903:
884: 904:
885: 905:
886: 906:
887: 907:
888: 908:
889: 909:
890: 910:
891: 911:
892: 912:
893: 913:
894: 914:
895: 915:
896: 916:
897: 917:
898: 918:
899: 919:
899: 920:
900: 921:
901: 922:
902: 923:
903: 924:
904: 925:
905: 926:
906: 927:
907: 928:
908: 929:
909: 930:
910: 931:
911: 932:
912: 933:
913: 934:
914: 935:
915: 936:
916: 937:
917: 938:
918: 939:
919: 940:
920: 941:
921: 942:
922: 943:
923: 944:
924: 945:
925: 946:
926: 947:
927: 948:
928: 949:
929: 950:
930: 951:
931: 952:
932: 953:
933: 954:
934: 955:
935: 956:
936: 957:
937: 958:
938: 959:
939: 960:
940: 961:
941: 962:
942: 963:
943: 964:
944: 965:
945: 966:
946: 967:
947: 968:
948: 969:
949: 970:
950: 971:
951: 972:
952: 973:
953: 974:
954: 975:
955: 976:
956: 977:
957: 978:
958: 979:
959: 980:
960: 981:
961: 982:
962: 983:
963: 984:
964: 985:
965: 986:
966: 987:
967: 988:
968: 989:
969: 990:
970: 991:
971: 992:
972: 993:
973: 994:
974: 995:
975: 996:
976: 997:
977: 998:
978: 999:
979: 1000:
980: 1001:
981: 1002:
982: 1003:
983: 1004:
984: 1005:
985: 1006:
986: 1007:
987: 1008:
988: 1009:
989: 1010:
990: 1011:
991: 1012:
992: 1013:
993: 1014:
994: 1015:
995: 1016:
996: 1017:
997: 1018:
998: 1019:
999: 1020:
999: 1021:
1000: 1022:
1001: 1023:
1002: 1024:
1003: 1025:
1004: 1026:
1005: 1027:
1006: 1028:
1007: 1029:
1008: 1030:
1009: 1031:
1010: 1032:
1011: 1033:
1012: 1034:
1013: 1035:
1014: 1036:
1015: 1037:
1016: 1038:
1017: 1039:
1018: 1040:
1019: 1041:
1020: 1042:
1021: 1043:
1022: 1044:
1023: 1045:
1024: 1046:
1025: 1047:
1026: 1048:
1027: 1049:
1028: 1050:
1029: 1051:
1030: 1052:
1031: 1053:
1032: 1054:
1033: 1055:
1034: 1056:
1035: 1057:
1036: 1058:
1037: 1059:
1038: 1060:
1039: 1061:
1040: 1062:
1041: 1063:
1042: 1064:
1043: 1065:
1044: 1066:
1045: 1067:
1046: 1068:
1047: 1069:
1048: 1070:
1049: 1071:
1050: 1072:
1051: 1073:
1052: 1074:
1053: 1075:
1054: 1076:
1055: 1077:
1056: 1078:
1057: 1079:
1058: 1080:
1059: 1081:
1060: 1082:
1061: 1083:
1062: 1084:
1063: 1085:
1064: 1086:
1065: 1087:
1066: 1088:
1067: 1089:
1068: 1090:
1069: 1091:
1070: 1092:
1071: 1093:
1072: 1094:
1073: 1095:
1074: 1096:
1075: 1097:
1076: 1098:
1077: 1099:
1078: 1100:
1079: 1101:
1080: 1102:
1081: 1103:
1082: 1104:
1083: 1105:
1084: 1106:
1085: 1107:
1086: 1108:
1087: 1109:
1088: 1110:
1089: 1111:
1090: 1112:
1091: 1113:
1092: 1114:
1093: 1115:
1094: 1116:
1095: 1117:
1096: 1118:
1097: 1119:
1098: 1120:
1099: 1121:
1100: 1122:
1101: 1123:
1102: 1124:
1103: 1125:
1104: 1126:
1105: 1127:
1106: 1128:
1107: 1129:
1108: 1130:
1109: 1131:
1110: 1132:
1111: 1133:
1112: 1134:
1113: 1135:
1114: 1136:
1115: 1137:
1116: 1138:
1117: 1139:
1118: 1140:
1119: 1141:
1120: 1142:
1121: 1143:
1122: 1144:
1123: 1145:
1124: 1146:
1125: 1147:
1126: 1148:
1127: 1149:
1128: 1150:
1129: 1151:
1130: 1152:
1131: 1153:
1132: 1154:
1133: 1155:
1134: 1156:
1135: 1157:
1136: 1158:
1137: 1159:
1138: 1160:
1139: 1161:
1140: 1162:
1141: 1163:
1142: 1164:
1143: 1165:
1144: 1166:
1145: 1167:
1146: 1168:
1147: 1169:
1148: 1170:
1149: 1171:
1150: 1172:
1151: 1173:
1152: 1174:
1153: 1175:
1154: 1176:
1155: 1177:
1156: 1178:
1157: 1179:
1158: 1180:
1159: 1181:
1160: 1182:
1161: 1183:
1162: 1184:
1163: 1185:
1164: 1186:
1165: 1187:
1166: 1188:
1167: 1189:
1168: 1190:
1169: 1191:
1170: 1192:
1171: 1193:
1172: 1194:
1173: 1195:
1174: 1196:
1175: 1197:
1176: 1198:
1177: 1199:
1178: 1200:
1179: 1201:
1180: 1202:
1181: 1203:
1182: 1
```

Viewing the Data

The screenshot shows two code snippets in the top pane and their corresponding Data tabs in the bottom pane.

Code Snippet 1:

```
19 OPEN emp_cursor;
20 FETCH emp_cursor;
```

Data Tab 1 (Line 20):

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP	Rowtype	EMPLOYEE%TYPE
DEPARTMENT_NAME	NULL	VARCHAR2(30)
EMPLOYEE_ID	NULL	NUMBER(6,0)
LAST_NAME	NULL	VARCHAR2(25)
SALARY	NULL	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB%TYPE	INDEXED TABLE	EMPLOYEE%TYPE element
VALUES	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

Code Snippet 2:

```
9 OPEN cur_emp;
10   FETCH cur_emp INTO rec_emp;
11   emp.tab() := rec_emp;
12   WHILE (cur_empFOUND) AND (i <= p_maxrows) LOOP
```

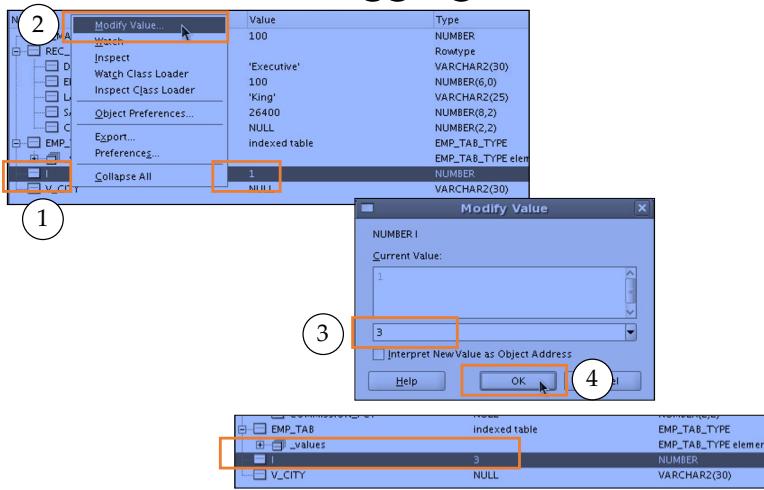
Data Tab 2 (Line 12):

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP	Rowtype	EMPLOYEE%TYPE
DEPARTMENT_NAME	'Executive'	VARCHAR2(30)
EMPLOYEE_ID	100	NUMBER(6,0)
LAST_NAME	'King'	VARCHAR2(25)
SALARY	26400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB%TYPE	INDEXED TABLE	EMPLOYEE%TYPE element
VALUES	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

252

While you are debugging your code, you can use the Data tab to display and modify the variables. You can also set watches to monitor a subset of the variables displayed in the Data tab. To display or hide the Data, Smart Data, and Watch tabs: Select View > Debugger, and then select the tabs that you want to display or hide.

Modifying the Variables While Debugging the Code



253

To modify the value of a variable in the Data tab, right-click the variable name, and then select Modify Value from the shortcut menu. The Modify Value window is displayed. The current value for the variable is displayed. You can enter a new value in the second text box, and then click OK.

Debugging emp_list: Step Over the Code

Step Over (F8):
Executes the Cursor
(same as F7),
but control is not transferred
to Open Cursor code

```

14 BEGIN
④ OPEN cur_emp;
⑤ FETCH cur_emp INTO rec_emp; 1 F8
⑥ emp_tab(i) := rec_emp;
⑦ WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
⑧   i := i + 1;
⑨   FETCH cur_emp INTO rec_emp;
⑩   emp_tab(i) := rec_emp;
⑪   v_city := get_location (rec_emp.department_name);
⑫   dbms_output.put_line('Employee ' || rec_emp.last_name ||
⑬   ' works in ' || v_city);
⑭

```

```

14 BEGIN
④ OPEN cur_emp;
⑤ FETCH cur_emp INTO rec_emp; 2 F8
⑥ emp_tab(i) := rec_emp;
⑦ WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
⑧   i := i + 1;
⑨   FETCH cur_emp INTO rec_emp;
⑩   emp_tab(i) := rec_emp;
⑪   v_city := get_location (rec_emp.department_name);
⑫   dbms_output.put_line('Employee ' || rec_emp.last_name ||
⑬   ' works in ' || v_city);
⑭

```

```

14 BEGIN
④ OPEN cur_emp;
⑤ FETCH cur_emp INTO rec_emp;
⑥ emp_tab(i) := rec_emp;
⑦ WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
⑧   i := i + 1;
⑨   FETCH cur_emp INTO rec_emp;
⑩   emp_tab(i) := rec_emp;
⑪   v_city := get_location (rec_emp.department_name);
⑫   dbms_output.put_line('Employee ' || rec_emp.last_name ||
⑬   ' works in ' || v_city);
⑭

```

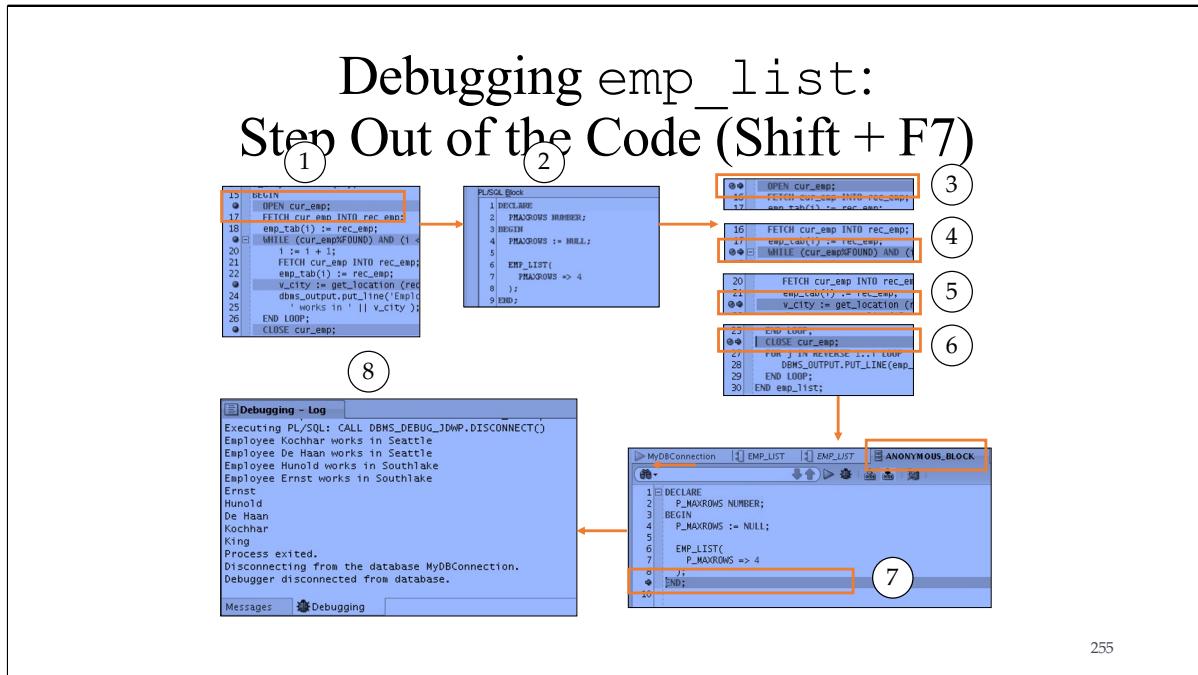
254

The Step Over Debugging Tool

The Step Over command, like Step Into, enables you to execute program statements one at a time. However, if you issue the Step Over command when the execution point is located on a subprogram call, the debugger runs that subprogram without stopping (instead of stepping into it), and then positions the execution point on the statement that follows the subprogram call. If the execution point is located on the last statement of a subprogram, choosing Step Over causes the debugger to return from the subprogram, placing the execution point on the line of code that follows the call to the subprogram you are returning from. You can step over a subprogram in any of the following ways: Select Debug > Step Over, press F8, or click the Step Over icon in the Debugging – Log toolbar.

In the example in the slide, stepping over will execute the open cursor line without transferring program control to the cursor definition as was the case with the Step Into option example.

Debugging emp_list: Step Out of the Code (Shift + F7)



The Step Out of the code option leaves the current subprogram and goes to the next statement subprogram. In the example in the slide, starting with step 3, if you press Shift + F7, program control leaves the procedure and goes to the next statement in the anonymous block. Continuing to press Shift + F7 will take you to the next anonymous block that prints the contents of the SQL buffer.

Debugging emp_list: Run to Cursor (F4)

```

10    emp_record.emp_CURSORROWTYPE;
11    TYPE emp_tab_type IS TABLE OF emp_CURSORROWTYPE INDEX BY BINARY_INTEGER;
12    emp_tab emp_tab_type;
13    i NUMBER := 1;
14    v_city VARCHAR2(30);
15
16    BEGIN
17        OPEN emp_cursor;
18        FETCH emp_cursor INTO emp_record;
19        emp_tab(1) := emp_record;
20        WHILE (emp_cursorFOUND) AND (i <= pMaxRows) LOOP
21            i := i + 1;
22            FETCH emp_cursor INTO emp_record;
23            emp_tab(i) := emp_record;
24            v_city := get_location(emp_record.department_name);
25            dbms_output.put_line('Employee ' || emp_record.last_name ||
26                ' works in ' || v_city );
27        END LOOP;
28        CLOSE emp_cursor;
29
30        FOR j IN reverse(1..i) LOOP
31            DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
32        END LOOP;
33    END emp_list;
34

```

Smart Data Watch

Name	Value	Type
EMP_ROWS	5	NUMBER
EMP_RECORD	1	NUMBER
EMP_TAB		INDEXED TABLE
V_CITY	NULL	VARCHAR2

Run to Cursor F4:

Run to your cursor location without having to single step or set a breakpoint.

256

Running to the Cursor Location

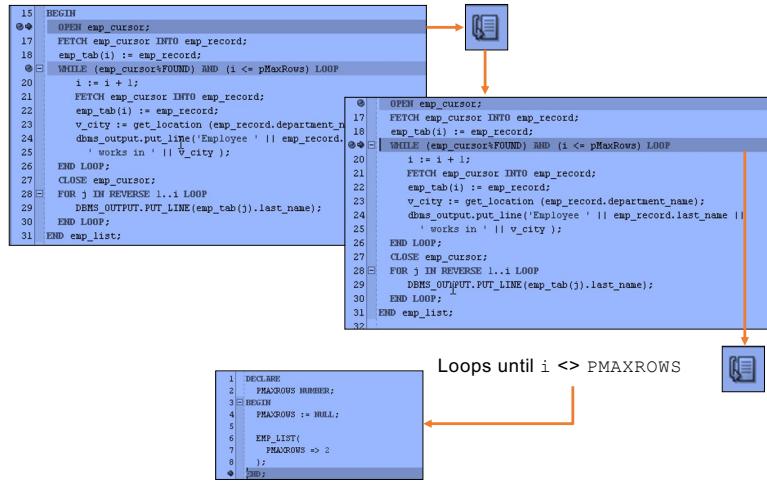
When stepping through your application code in the debugger, you may want to run to a particular location without having to single step or set a breakpoint. To run to a specific program location: In a subprogram editor, position your text cursor on the line of code where you want the debugger to stop. You can run to the cursor location using any of the following procedures: In the procedure editor, right-click and choose Run to Cursor, choose the Debug > Run to Cursor option from the main menu, or press F4.

Any of the following conditions may result:

When you run to the cursor, your program executes without stopping, until the execution reaches the location marked by the text cursor in the source editor.

If your program never actually executes the line of code where the text cursor is, the Run to Cursor command will cause your program to run until it encounters a breakpoint or until it finishes.

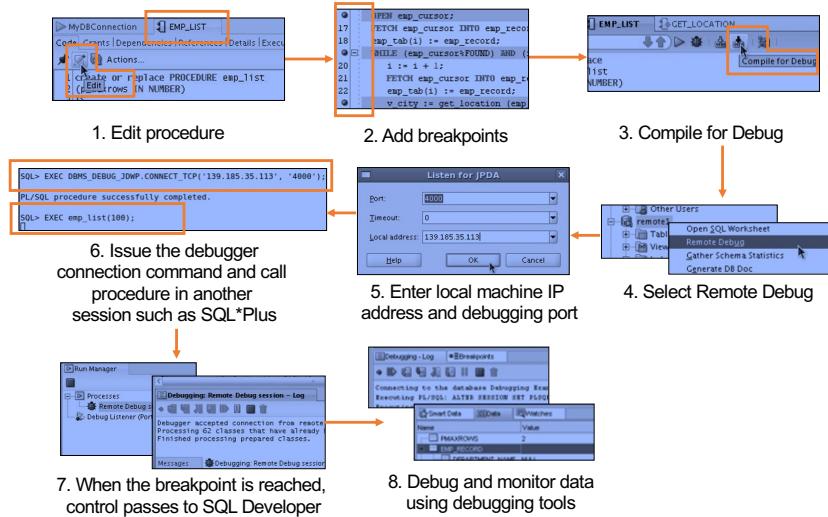
Debugging emp_list: Step to End of Method



257

Step to End of Method goes to the last statement in the current subprogram or to the next breakpoint if there are any in the current subprogram. In the example in the slide, the Step to End of Method debugging tool is used. Because there is a second breakpoint, selecting Step to End of Method transfers control to that breakpoint. Selecting Step to End of Method again goes through the iterations of the while loop first, and then transfers the program control to the next executable statement in the anonymous block.

Debugging a Subprogram Remotely: Overview



258

Remote Debugging

You can use remote debugging to connect to any PL/SQL code in a database and debug it, regardless of where the database is located as long as you have access to the database and have created a connection to it. Remote debugging is when something else, other than you as a developer, kicks off a procedure that you can then debug. Remote debugging and local debugging have many steps in common. To debug remotely, perform the following steps:

1. Edit the subprogram that you would like to debug.
2. Add the breakpoints in the subprogram.
3. Click the **Compile for Debug** icon in the toolbar.
4. Right-click the connection for the remote database, and select **Remote Debug**.

5. In the **Debugger - Attach to JPDA** dialog box, enter the local machine IP address and port number, such as 4000, and then click **OK**.
6. Issue the debugger connection command using a different session such as SQL*Plus, and then call the procedure in that session.
7. When a breakpoint is reached, control is passed back to the original SQL Developer session and the debugger toolbar is displayed.

8. Debug the subprogram and monitor the data using the debugging tools discussed earlier.

Topics covered in this module:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Understand the basic functionality of the SQL Developer debugger

259

A function is a named PL/SQL block that must return a value. Generally, you create a function to compute and return a value, and you create a procedure to perform an action.

A function can be created or dropped.

A function is invoked as a part of an expression.

Introduction to Oracle PL/SQL

Module 11: Creating Packages

Topics covered in this module:

- Describe packages and list their components
- Create a package to group together related variables, cursors, constants, exceptions, procedures, and functions
- Designate a package construct as either public or private
- Invoke a package construct
- Describe the use of a bodiless package

In this lesson, you learn what a package is and what its components are. You also learn how to create and use packages.

What Are PL/SQL Packages?

- A package is a schema object that groups logically related PL/SQL types, variables, and subprograms.
- Packages usually have two parts:
 - A specification (spec)
 - A body
- The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- The body defines the queries for the cursors and the code for the subprograms.
- Enable the Oracle server to read multiple objects into memory at once.

262

PL/SQL Packages: Overview

PL/SQL packages enable you to bundle related PL/SQL types, variables, data structures, exceptions, and subprograms into one container. For example, a Human Resources package can contain hiring and firing procedures, commission and bonus functions, and tax exemption variables.

A package usually consists of two parts stored separately in the database:

- A specification
- A body (optional)

The package itself cannot be called, parameterized, or nested. After writing and compiling, the contents can be shared with many applications.

When a PL/SQL-packaged construct is referenced for the first time, the whole package is loaded into memory. Subsequent access to constructs in the same package does not require disk input/output (I/O).

Advantages of Using Packages

- Modularity: Encapsulating related constructs
- Easier maintenance: Keeping logically related functionality together
- Easier application design: Coding and compiling the specification and body separately
- Hiding information:
 - Only the declarations in the package specification are visible and accessible to applications
 - Private constructs in the package body are hidden and inaccessible
 - All coding is hidden in the package body

263

Packages provide an alternative to creating procedures and functions as stand-alone schema objects, and they offer several benefits.

Modularity and easier maintenance: You encapsulate logically related programming structures in a named module. Each package is easy to understand, and the interface between packages is simple, clear, and well defined.

Easier application design: All you need initially is the interface information in the package specification. You can code and compile a specification without its body. Thereafter, stored subprograms that reference the package can compile as well. You need not define the package body fully until you are ready to complete the application.

Hiding information: You decide which constructs are public (visible and accessible) and which are private (hidden and inaccessible). Declarations in the package specification are visible and accessible to applications. The package body hides the definition of the private constructs, so that only the package is affected (not your application or any calling programs) if the definition changes. This enables you to change the implementation without having to recompile the calling programs. Also, by hiding implementation details from users, you

protect the integrity of the package.

Advantages of Using Packages

- Added functionality: Persistency of public variables and cursors
- Better performance:
 - The entire package is loaded into memory when the package is first referenced.
 - There is only one copy in memory for all users.
 - The dependency hierarchy is simplified.
- Overloading: Multiple subprograms of the same name

264

Added functionality: Packaged public variables and cursors persist for

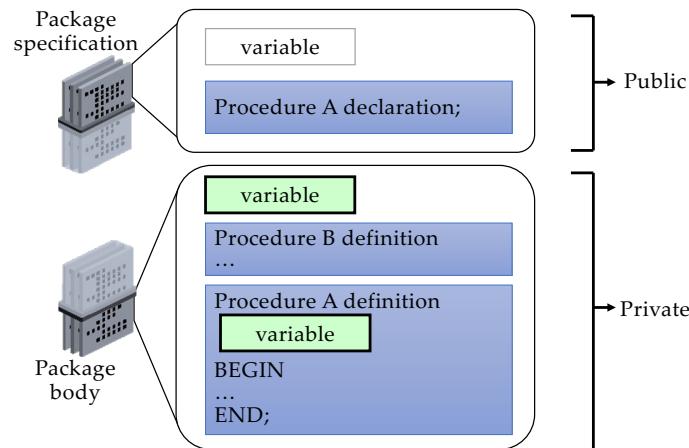
the duration of a session. Thus, they can be shared by all subprograms that execute in the environment. They also enable you to maintain data across transactions without having to store it in the database. Private constructs also persist for the duration of the session but can be accessed only within the package.

Better performance: When you call a packaged subprogram the first time, the entire package is loaded into memory. Later calls to related subprograms in the package, therefore, require no further disk I/O. Packaged subprograms also stop cascading dependencies and thus avoid unnecessary compilation.

Overloading: With packages, you can overload procedures and functions, which means you can create multiple subprograms with the same name in the same package, each taking parameters of different number or data type.

Note: Dependencies are covered in detail in the lesson titled “Managing Dependencies.”

Components of a PL/SQL Package



265

You create a package in two parts:

The **package specification** is the interface to your applications. It declares the public types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGMAs, which are directives to the compiler.

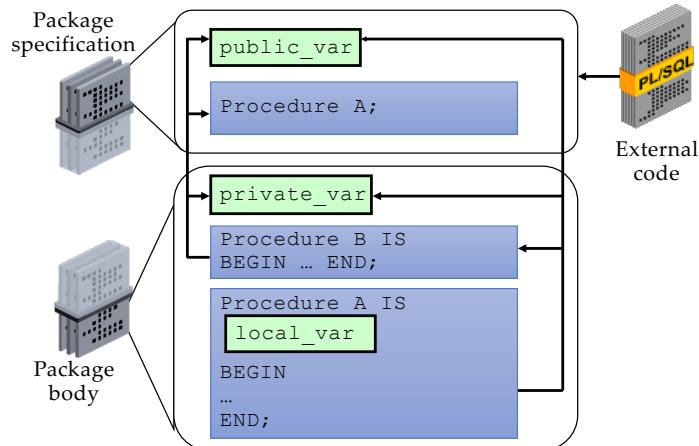
The **package body** defines its own subprograms and must fully implement subprograms declared in the specification part. The package body may also define PL/SQL constructs, such as types, variables, constants, exceptions, and cursors.

Public components are declared in the package specification. The specification defines a public application programming interface (API) for users of package features and functionality—that is, public components can be referenced from any Oracle server environment that is external to the package.

Private components are placed in the package body and can be referenced only by other constructs within the same package body. Private components can reference the public components of a package.

Note: If a package specification does not contain subprogram declarations, then there is no requirement for a package body.

Internal and External Visibility of a Package's Components



266

The visibility of a component describes whether that component can be seen, that is, referenced and used by other components or objects. The visibility of components depends on whether they are *locally* or *globally* declared.

Local components are visible within the structure in which they are declared, such as:

Variables defined in a subprogram can be referenced within that subprogram, and are not visible to external components—for example, `local_var` can be used in procedure A.

Private package variables, which are declared in a package body, can be referenced by other components in the same package body. They are not visible to any subprograms or objects that are outside the package. For example, `private_var` can be used by procedures A and B within the package body, but not outside the package.

Globally declared components are visible internally and externally to the package, such as:

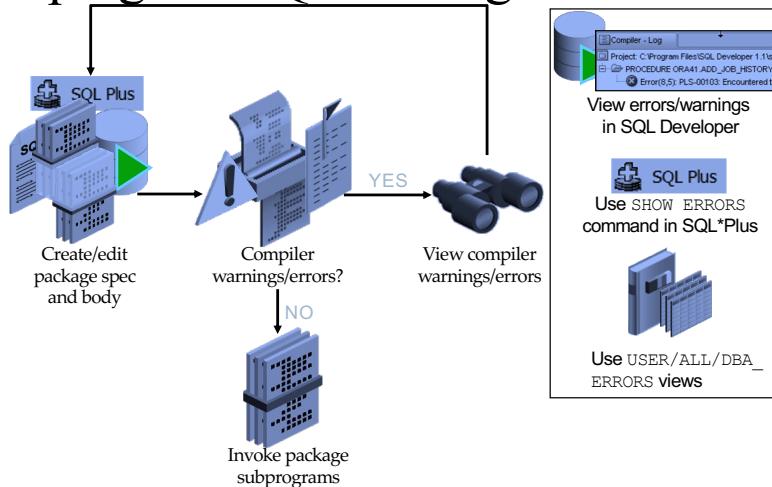
A public variable, which is declared in a package specification, can be referenced and changed outside the package (for example, `public_var` can be referenced externally).

A package subprogram in the specification can be called from external

code sources (for example, procedure A can be called from an environment external to the package).

Note: Private subprograms, such as procedure B, can be invoked only with public subprograms, such as procedure A, or other private package constructs. A public variable declared in the package specification is a global variable.

Developing PL/SQL Packages: Overview



267

Developing PL/SQL Packages

The graphic in the slide illustrates the basic steps involved in developing and using a package:

1. Create the procedure using SQL Developer's Object Navigator tree or the SQL Worksheet area.
2. Compile the package. The package is created in the database. The CREATE PACKAGE statement creates and stores source code and the compiled *m-code* in the database. To compile the package, right-click the package's name in the Object Navigator tree, and then click Compile.
3. If there are no compilation warnings or errors, you execute any public construct within the package specification from an Oracle Server environment.
4. If there are compilation warning or errors, you can view (and then correct) the warnings or errors using one of the following methods:
 - Using the SQL Developer interface (the Compiler – Log tab)
 - Using the SHOW ERRORS SQL*Plus command
 - Using the USER/ALL/DBA_ERRORS views

```
CREATE [OR REPLACE] PACKAGE package_name IS|AS  
    public type and variable declarations  
    subprogram specifications  
END [package_name];
```

- The OR REPLACE option drops and re-creates the package specification.
- Variables declared in the package specification are initialized to NULL by default.
- All the constructs declared in a package specification are visible to users who are granted privileges on the package.

268

Creating the Package Specification

To create packages, you declare all public constructs within the package specification.

Specify the OR REPLACE option if overwriting an existing package specification.

Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.

The following are definitions of items in the package syntax:

- **package_name** specifies a name for the package that must be unique among objects within the owning schema. Including the package name after the END keyword is optional.

- **public type and variable declarations** declares public variables, constants, cursors, exceptions, user-defined types, and subtypes.

- **subprogram specification** specifies the public procedure or function declarations.

The package specification should contain procedure and function headings terminated by a semicolon, without the IS (or AS) keyword and its PL/SQL

block. The implementation of a procedure or function that is declared in a package specification is done in the package body.

The Oracle database stores the specification and body of a package separately. This enables you to change the implementation of a program construct in the package body without invalidating other schema objects that call or reference the program construct.

Example of a Package Specification: *comm_pkg*

```
-- The package spec with a public variable and a
-- public procedure that are accessible from
-- outside the package.

CREATE OR REPLACE PACKAGE comm_pkg IS
    v_std_comm NUMBER := 0.10;  --initialized to 0.10
    PROCEDURE reset_comm(p_new_comm NUMBER);
END comm_pkg;
/
```

- *V_STD_COMM* is a *public* global variable initialized to 0.10.
- *RESET_COMM* is a *public* procedure used to reset the standard commission based on some business rules. It is implemented in the package body.

269

The example in the slide creates a package called *comm_pkg* used to manage business processing rules for commission calculations.

The *v_std_comm* public (global) variable is declared to hold a maximum allowable percentage commission for the user session, and it is initialized to 0.10 (that is, 10%).

The *reset_comm* public procedure is declared to accept a new commission percentage that updates the standard commission percentage if the commission validation rules are accepted. The validation rules for resetting the commission are not made public and do not appear in the package specification. The validation rules are managed by using a private function in the package body.

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS  
    private type and variable declarations  
    subprogram bodies  
    [BEGIN initialization statements]  
END [package_name];
```

- The OR REPLACE option drops and re-creates the package body.
- Identifiers defined in the package body are *private* and not visible outside the package body.
- All *private* constructs must be declared before they are referenced.
- Public constructs are visible to the package body.

270

Create a package body to define and implement all public subprograms and supporting private constructs. When creating a package body, perform the following steps:

Specify the OR REPLACE option to overwrite an existing package body.

Define the subprograms in an appropriate order. The basic principle is that you must declare a variable or subprogram before it can be referenced by other components in the same package body. It is common to see all private variables and subprograms defined first and the public subprograms defined last in the package body.

Complete the implementation for all procedures or functions declared in the package specification within the package body.

The following are definitions of items in the package body syntax:

•*package_name* specifies a name for the package that must be the same as its package specification. Using the package name after the END keyword is optional.

•*private type and variable declarations* declares private variables, constants, cursors, exceptions, user-defined types, and subtypes.

- **subprogram specification** specifies the full implementation of any private and/or public procedures or functions.
- **[BEGIN initialization statements]** is an optional block of initialization code that executes when the package is first referenced.

Example of a Package Body: comm_pkg

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS
  FUNCTION validate(p_comm NUMBER) RETURN BOOLEAN IS
    v_max_comm      employees.commission_pct%type;
  BEGIN
    SELECT MAX(commission_pct) INTO v_max_comm
    FROM   employees;
    RETURN (p_comm BETWEEN 0.0 AND v_max_comm);
  END validate;

  PROCEDURE reset_comm (p_new_comm NUMBER) IS
  BEGIN
    IF validate(p_new_comm) THEN
      v_std_comm := p_new_comm; -- reset public var
    ELSE
      RAISE_APPLICATION_ERROR(
        -20210, 'Bad Commission');
    END IF;
  END reset_comm;
END comm_pkg;
```

271

The slide shows the complete package body for `comm_pkg`, with a private function called `validate` to check for a valid commission. The validation requires that the commission be positive and less than the highest commission among existing employees. The `reset_comm` procedure invokes the private validation function before changing the standard commission in `v_std_comm`. In the example, note the following:

- The `v_std_comm` variable referenced in the `reset_comm` procedure is a public variable. Variables declared in the package specification, such as `v_std_comm`, can be directly referenced without qualification.
- The `reset_comm` procedure implements the public definition in the specification.
- In the `comm_pkg` body, the `validate` function is private and is directly referenced from the `reset_comm` procedure without qualification.

Note: The `validate` function appears before the `reset_comm` procedure because the `reset_comm` procedure references the `validate` function. It is possible to create forward declarations for subprograms in the package body if their order of appearance needs to be changed. If a package

specification declares only types, constants, variables, and exceptions without any subprogram specifications, then the package body is unnecessary. However, the body can be used to initialize items declared in the package specification.

Invoking the Package Subprograms: Examples

```
-- Invoke a function within the same package:  
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...  
    PROCEDURE reset_comm(p_new_comm NUMBER) IS  
        BEGIN  
            IF validate(p_new_comm) THEN  
                v_std_comm := p_new_comm;  
            ELSE ...  
            END IF;  
        END reset_comm;  
    END comm_pkg;
```

```
-- Invoke a package procedure from SQL*Plus:  
EXECUTE comm_pkg.reset_comm(0.15)
```

```
-- Invoke a package procedure in a different schema:  
EXECUTE scott.comm_pkg.reset_comm(0.15)
```

272

Invoking Package Subprograms

After the package is stored in the database, you can invoke public or private subprograms within the same package, or public subprograms if external to the package. Fully qualify the subprogram with its package name when invoked externally from the package. Use the `package_name.subprogram` syntax.

Fully qualifying a subprogram when invoked within the same package is optional.

Example 1: Invokes the `validate` function from the `reset_comm` procedure within the same package. The package name prefix is not required; it is optional.

Example 2: Calls the `reset_comm` procedure from SQL*Plus (an environment external to the package) to reset the prevailing commission to 0.15 for the user session.

Example 3: Calls the `reset_comm` procedure that is owned by a schema user called SCOTT. Using SQL*Plus, the qualified package procedure is prefixed with the schema name. This can be simplified by using a synonym that references the `schema.package_name`.

Assume that a database link named NY has been created for a remote

database in which the `reset_comm` package procedure is created. To invoke the remote procedure, use:

```
EXECUTE comm_pkg.reset_comm@NY(0.15)
```

Creating and Using Bodiless Packages

```
CREATE OR REPLACE PACKAGE global_consts IS
    c_mile_2_kilo CONSTANT NUMBER := 1.6093;
    c_kilo_2_mile CONSTANT NUMBER := 0.6214;
    c_yard_2_meter CONSTANT NUMBER := 0.9144;
    c_meter_2_yard CONSTANT NUMBER := 1.0936;
END global_consts;

SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
        20 * global_consts.c_mile_2_kilo || ' km');
END;

SET SERVEROUTPUT ON
CREATE FUNCTION mtr2yrd(p_m NUMBER) RETURN NUMBER IS
BEGIN
    RETURN (p_m * global_consts.c_meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

273

The variables and constants declared within stand-alone subprograms exist only for the duration that the subprogram executes. To provide data that exists for the duration of the user session, create a package specification containing public (global) variables and constant declarations. In this case, create a package specification without a package body, known as a *bodiless package*. As discussed earlier in this lesson, if a specification declares only types, constants, variables, and exceptions, then the package body is unnecessary.

Examples

The first code box in the slide creates a bodiless package specification with several constants to be used for conversion rates. A package body is not required to support this package specification. It is assumed that the `SET SERVEROUTPUT ON` statement was issued before executing the code examples in the slide.

The second code box references the `c_mile_2_kilo` constant in the `global_consts` package by prefixing the package name to the identifier of the constant.

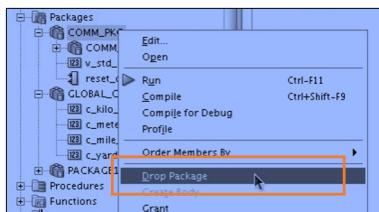
The third example creates a stand-alone function `c_mtr2yrd` to convert meters to yards, and uses the constant conversion rate `c_meter_2_yard`

declared in the global_consts package. The function is invoked in a DBMS_OUTPUT.PUT_LINE parameter.

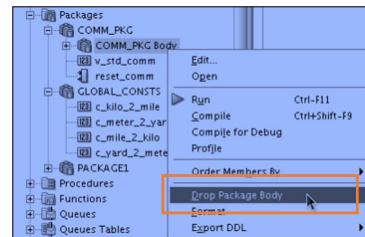
Rule to be followed: When referencing a variable, cursor, constant, or exception from outside the package, you must qualify it with the name of the package.

Removing Packages: Using SQL Developer or the SQL DROP Statement

Drop package specification and body



Drop package body only



```
-- Remove the package specification and body  
DROP PACKAGE package_name;
```

```
-- Remove the package body only  
DROP PACKAGE BODY package_name;
```

274

Removing Packages

When a package is no longer required, you can use a SQL statement in SQL Developer to remove it. A package has two parts; therefore, you can remove the whole package, or you can remove only the package body and retain the package specification.

Viewing Packages Using the Data Dictionary

```
-- View the package specification.  
SELECT text  
FROM user_source  
WHERE name = 'COMM_PKG' AND type = 'PACKAGE'  
ORDER BY LINE;
```

```
TEXT  
1 PACKAGE comm_pkg IS  
2 std_comm NUMBER := 0.10; --initialized to 0.10  
3 PROCEDURE reset_comm(new_comm NUMBER);  
4 END comm_pkg;
```

```
-- View the package body.  
SELECT text  
FROM user_source  
WHERE name = 'COMM_PKG' AND type = 'PACKAGE BODY'  
ORDER BY LINE;
```

```
TEXT  
1 PACKAGE BODY comm_pkg IS  
2 FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS  
3 max_comm employees.commission_pct%TYPE;  
4 BEGIN  
5 SELECT MAX(commission_pct) INTO max_comm  
6 FROM employees;  
7 RETURN (comm BETWEEN 0.0 AND max_comm);
```

275

Viewing Packages in the Data Dictionary

The source code for PL/SQL packages is also stored in the `USER_SOURCE` and `ALL_SOURCE` data dictionary views. The `USER_SOURCE` table is used to display PL/SQL code that you own. The `ALL_SOURCE` table is used to display PL/SQL code to which you have been granted the `EXECUTE` right by the owner of that subprogram code and provides an `OWNER` column in addition to the preceding columns.

When querying the package, use a condition in which the `TYPE` column is:

Equal to '`PACKAGE`' to display the source code for the package specification

Equal to '`PACKAGE BODY`' to display the source code for the package body

You can also view the package specification and body in SQL Developer using the package name in the Packages node.

Note: You cannot display the source code for Oracle PL/SQL built-in packages, or PL/SQL whose source code has been wrapped by using a `WRAP` utility or obfuscation. Obfuscating and wrapping PL/SQL source code is covered in a later lesson. Clicking the Execute Statement (F9) icon (instead of the Run Script icon) in the SQL Worksheet toolbar, sometimes displays a better

formatted output in the Results tab as shown in the slide examples.

- Develop packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.
- The fine-grain dependency management reduces the need to recompile referencing subprograms when a package specification changes.
- The package specification should contain as few constructs as possible.

276

Keep your packages as general as possible, so that they can be reused in future applications. Also, avoid writing packages that duplicate features provided by the Oracle server.

Package specifications reflect the design of your application, so define them before defining the package bodies. The package specification should contain only those constructs that must be visible to the users of the package. Thus, other developers cannot misuse the package by basing code on irrelevant details.

Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions. For example, declare a variable called `NUMBER_EMPLOYED` as a private variable if each call to a procedure that uses the variable needs to be maintained. When declared as a global variable in the package specification, the value of that global variable is initialized in a session the first time a construct from the package is invoked.

Before Oracle Database 19c, changes to the package body did not require recompilation of dependent constructs, whereas changes to the package specification required the recompilation of every stored subprogram that references the package. Oracle Database 19c reduces this dependency.

Dependencies are now tracked at the level of element within unit. Fine-Grain Dependency Management is covered in a later lesson.

Topics covered in this module:

- Describe packages and list their components
- Create a package to group related variables, cursors, constants, exceptions, procedures, and functions
- Designate a package construct as either public or private
- Invoke a package construct
- Describe the use of a bodiless package

277

You group related procedures and functions in a package. Packages improve organization, management, security, and performance.

A package consists of a package specification and a package body. You can change a package body without affecting its package specification.

Packages enable you to hide source code from users. When you invoke a package for the first time, the entire package is loaded into memory. This reduces the disk access for subsequent calls.



Oracle 19c: PL/SQL



Module 12: Working with Packages

Topics covered in this module:

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Use associative arrays (index-by tables) and records in packages

279

This lesson introduces the more advanced features of PL/SQL, including overloading, forward referencing, one-time-only procedures, and the persistency of variables, constants, exceptions, and cursors. It also explains the effect of packaging functions that are used in SQL statements.

- Enables you to create two or more subprograms with the same name
- Requires that the subprogram's formal parameters differ in number, order, or data type family
- Enables you to build flexible ways for invoking subprograms with different data
- Provides a way to extend functionality without loss of existing code; that is, adding new parameters to existing subprograms
- Provides a way to overload local subprograms, package subprograms, and type methods, but not stand-alone subprograms

280

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the `TO_CHAR` function has more than one way to be called, enabling you to convert a number or a date to a character string.

PL/SQL allows overloading of package subprogram names and object type methods.

The key rule is that you can use the same name for different subprograms as long as their formal parameters differ in *number, order, or data type family*.

Consider using overloading when:

Processing rules for two or more subprograms are similar, but the type or number of parameters used varies

Providing alternative ways for finding different data with varying search criteria. For example, you may want to find employees by their employee ID and also provide a way to find employees by their last name. The logic is intrinsically the same, but the parameters or search criteria differ.

Extending functionality when you do not want to replace existing code

Note: Stand-alone subprograms cannot be overloaded. Writing local subprograms in object type methods is not discussed in this course.

Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department
    (p_deptno departments.department_id%TYPE,
     p_name departments.department_name%TYPE := 'unknown',
     p_loc   departments.location_id%TYPE := 1700);

  PROCEDURE add_department
    (p_name departments.department_name%TYPE := 'unknown',
     p_loc   departments.location_id%TYPE := 1700);
END dept_pkg;
/
```

281

Overloading: Example

The slide shows the `dept_pkg` package specification with an overloaded procedure called `add_department`. The first declaration takes three parameters that are used to provide data for a new department record inserted into the `department` table. The second declaration takes only two parameters because this version internally generates the department ID through an Oracle sequence.

It is better to specify data types using the `%TYPE` attribute for variables that are used to populate columns in database tables, as shown in the example in the slide; however, you can also specify the data types as follows:

```
CREATE OR REPLACE PACKAGE dept_pkg_method2 IS
  PROCEDURE add_department(p_deptno NUMBER,
                           p_name VARCHAR2 := 'unknown', p_loc NUMBER := 1700);
  ...
  ...
```

Overloading Procedures Example: Creating the Package Body

```
-- Package body of package defined on previous slide.
CREATE OR REPLACE PACKAGE BODY dept_pkg IS
PROCEDURE add_department -- First procedure's declaration
  (p_deptno departments.department_id%TYPE,
   p_name    departments.department_name%TYPE := 'unknown',
   p_loc     departments.location_id%TYPE := 1700) IS
BEGIN
  INSERT INTO departments(department_id,
                         department_name, location_id)
  VALUES (p_deptno, p_name, p_loc);
END add_department;
PROCEDURE add_department -- Second procedure's declaration
  (p_name    departments.department_name%TYPE := 'unknown',
   p_loc     departments.location_id%TYPE := 1700) IS
BEGIN
  INSERT INTO departments (department_id,
                          department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_department;
END dept_pkg; /
```

282

Overloading: Example (continued)

If you call `add_department` with an explicitly provided department ID, then PL/SQL uses the first version of the procedure. Consider the following example:

```
EXECUTE  
dept_pkg.add_department(980,'Education',2500)  
SELECT * FROM departments  
WHERE department_id = 980;
```

```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME           MANAGER_ID LOCATION_ID
-----  -----
      980 Education                      2500

uses the
ping'. 2400)
```

```
SELECT * FROM departments  
WHERE department_name = 'Training';
```

```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME          MANAGER_ID LOCATION_ID
----- -----
      280 Training                         2400
```

Overloading and the **STANDARD** Package

- A package named STANDARD defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. An example is the TO_CHAR function:

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 DATE, p2 VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, p2 VARCHAR2) RETURN
    VARCHAR2;
. . .
```

- A PL/SQL subprogram with the same name as a built-in subprogram overrides the standard declaration in the local context, unless qualified by the package name.

283

A package named STANDARD defines the PL/SQL environment and globally declares types, exceptions, and subprograms that are available automatically to PL/SQL programs. Most of the built-in functions that are found in the STANDARD package are overloaded. For example, the TO_CHAR function has four different declarations, as shown in the slide. The TO_CHAR function can take either the DATE or the NUMBER data type and convert it to the character data type. The format to which the date or number has to be converted can also be specified in the function call.

If you re-declare a built-in subprogram in another PL/SQL program, then your local declaration overrides the standard or built-in subprogram. To be able to access the built-in subprogram, you must qualify it with its package name. For example, if you re-declare the TO_CHAR function to access the built-in function, you refer to it as STANDARD.TO_CHAR.

If you re-declare a built-in subprogram as a stand-alone subprogram, then, to access your subprogram, you must qualify it with your schema name: for example, SCOTT.TO_CHAR.

In the example in the slide, PL/SQL resolves a call to TO_CHAR by matching the number and data types of the formal and actual parameters.

- Block-structured languages such as PL/SQL must declare identifiers before referencing them.
- Example of a referencing problem:

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE award_bonus(. . .) IS
    BEGIN
      calc_rating(. . .);          --illegal reference
    END;

  PROCEDURE calc_rating(. . .) IS
    BEGIN
      ...
    END;
END forward_pkg;
/
```

284

Using Forward Declarations

In general, PL/SQL is like other block-structured languages and does not allow forward references. You must declare an identifier before using it. For example, a subprogram must be declared before you can call it.

Coding standards often require that subprograms be kept in alphabetical sequence to make them easy to find. In this case, you may encounter problems, as shown in the slide, where the `calc_rating` procedure cannot be referenced because it has not yet been declared.

You can solve the illegal reference problem by reversing the order of the two procedures. However, this easy solution does not work if the coding rules require subprograms to be declared in alphabetical order.

The solution in this case is to use forward declarations provided in PL/SQL. A forward declaration enables you to declare the heading of a subprogram, that is, the subprogram specification terminated by a semicolon.

Note: The compilation error for `calc_rating` occurs only if `calc_rating` is a private packaged procedure. If `calc_rating` is declared in the package specification, it is already declared as if it is a forward declaration, and its reference can be resolved by the compiler.



Using Forward Declarations

In the package body, a forward declaration is a private subprogram specification terminated by a semicolon.

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE calc_rating (...) ; -- forward declaration
  -- Subprograms defined in alphabetical order
  PROCEDURE award_bonus (...) IS
  BEGIN
    calc_rating (...);           -- reference resolved!
    .
    .
    END;
  PROCEDURE calc_rating (...) IS -- implementation
  BEGIN
    .
    .
    END;
END forward_pkg;
```

285

Using Forward Declarations (continued)

As previously mentioned, PL/SQL enables you to create a special subprogram declaration called a forward declaration. A forward declaration may be required for private subprograms in the package body, and consists of the subprogram specification terminated by a semicolon. Forward declarations help to:

Define subprograms in logical or alphabetical order

Define mutually recursive subprograms. Mutually recursive programs are programs that call each other directly or indirectly.

Group and logically organize subprograms in a package body

When creating a forward declaration:

The formal parameters must appear in both the forward declaration and the subprogram body

The subprogram body can appear anywhere after the forward declaration, but both must appear in the same program unit

Forward Declarations and Packages

Typically, the subprogram specifications go in the package specification, and the subprogram bodies go in the package body. The public subprogram declarations in the package specification do not require forward declarations.

The block at the end of the package body executes once and is used to initialize public and private package variables.

```
CREATE OR REPLACE PACKAGE taxes IS
    v_tax    NUMBER;
    ... -- declare all public procedures/functions
END taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes IS
    ... -- declare all private variables
    ... -- define public/private procedures/functions
BEGIN
    SELECT    rate_value INTO v_tax
    FROM      tax_rates
    WHERE     rate name = 'TAX';
END taxes;
```

286

Package Initialization Block

The first time a component in a package is referenced, the entire package is loaded into memory for the user session. By default, the initial value of variables is NULL (if not explicitly initialized). To initialize package variables, you can:

Use assignment operations in their declarations for simple initialization tasks

Add code block to the end of a package body for more complex initialization tasks

Consider the block of code at the end of a package body as a package initialization block that executes once, when the package is first invoked within the user session.

The example in the slide shows the `v_tax` public variable being initialized to the value in the `tax_rates` table the first time the `taxes` package is referenced.

Note: If you initialize the variable in the declaration by using an assignment operation, it is overwritten by the code in the initialization block at the end of the package body. The initialization block is terminated by the `END` keyword

for the package body.

- You use package functions in SQL statements.
- To execute a SQL statement that calls a member function, the Oracle database must know the function's purity level.
- Purity level is the extent to which the function is free of side effects, which refers to accessing database tables, package variables, and so on, for reading or writing.
- It is important to control side effects because they can:
 - Prevent the proper parallelization of a query
 - Produce order-dependent and, therefore, indeterminate results
 - Require impermissible actions such as the maintenance of package state across user sessions

287

Using Package Functions in SQL and Restrictions

To execute a SQL statement that calls a stored function, the Oracle Server must know the purity level of the function, or the extent to which the function is free of side effects. The term *side effect* refers to accessing database tables, package variables, and so forth for reading or writing. It is important to control side effects because they can prevent the proper parallelization of a query, produce order-dependent and therefore indeterminate results, or require impermissible actions such as the maintenance of package state across user sessions.

In general, restrictions are changes to database tables or public package variables (those declared in a package specification). Restrictions can delay the execution of a query, yield order-dependent (therefore indeterminate) results, or require that the package state variables be maintained across user sessions. Various restrictions are not allowed when a function is called from a SQL query or a DML statement.

To be callable from SQL statements, a stored function must obey the following purity rules to control side effects:

- When called from a SELECT or a parallelized DML statement, the function cannot modify any database tables.
- When called from a DML statement, the function cannot query or modify any database tables modified by that statement.
- When called from a SELECT or DML statement, the function cannot execute SQL transaction control, session control, or system control statements.

288

The fewer side effects a function has, the better it can be optimized within a query, particularly when the PARALLEL_ENABLE or DETERMINISTIC hints are used.

To be callable from SQL statements, a stored function (and any subprograms that it calls) must obey the purity rules listed in the slide. The purpose of those rules is to control side effects.

If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed).

To check for purity rule violations at compile time, use the RESTRICT_REFERENCES pragma to assert that a function does not read or write database tables or package variables.

Note

In the slide, a DML statement refers to an INSERT, UPDATE, or DELETE statement.

For information about using the RESTRICT_REFERENCES pragma, refer to the *Oracle Database PL/SQL Language Reference*.

Package Function in SQL: Example

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER IS
        v_rate NUMBER := 0.08;
    BEGIN
        RETURN (p_value * v_rate);
    END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM employees;
```

289

The first code example in the slide shows how to create the package specification and the body encapsulating the tax function in the `taxes_pkg` package. The second code example shows how to call the packaged tax function in the `SELECT` statement. The results are as follows:

TAXES_PKG.TAX(SALARY)	SALARY	LAST_NAME
208	2600	OConnell
208	2600	Grant
352	4400	Whalen
1040	13000	Hartstein
480	6000	Fay
520	6500	Mavris
800	10000	Baer
960	12000	Higgins
664	8300	Gietz
80	1000	Harris
80	1000	Joplin
1920	24000	King

111 rows selected

The collection of package variables and the values define the package state. The package state is:

- Initialized when the package is first loaded
- Persistent (by default) for the life of the session:
 - Stored in the User Global Area (UGA)
 - Unique to each session
 - Subject to change when package subprograms are called or public variables are modified
- Not persistent for the session but persistent for the life of a subprogram call when using PRAGMA SERIALLY_RESUSABLE in the package specification

The collection of public and private package variables represents the package state for the user session. That is, the package state is the set of values stored in all the package variables at a given point in time. In general, the package state exists for the life of the user session.

Package variables are initialized the first time a package is loaded into memory for a user session. The package variables are, by default, unique to each session and hold their values until the user session is terminated. In other words, the variables are stored in the User Global Area (UGA) memory allocated by the database for each user session. The package state changes when a package subprogram is invoked and its logic modifies the variable state. Public package state can be directly modified by operations appropriate to its type.

PRAGMA signifies that the statement is a compiler directive. PRAGMAS are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler. If you add PRAGMA SERIALLY_RESUSABLE to the package specification, then the database stores package variables in the System Global Area (SGA) shared across user sessions. In this case, the package state is maintained for the life of a subprogram call or a single reference to a package construct. The

SERIALLY_REUSEABLE directive is useful if you want to conserve memory and if the package state does not need to persist for each user session.

Persistent State of Package Variables: Example

Time	Events	v_std_comm [variable]	MAX (commission_pct) [column]	v_std_comm [variable]	MAX (commission_pct) [Column]
9:00	Scott> EXECUTE comm_pkg.reset_comm(0.25)	0.10 0.25	0.4	-	0.4
9:30	Jones> INSERT INTO employees(last_name, commission_pct) VALUES('Madonna', 0.8);	0.25	0.4		0.8
9:35	Jones> EXECUTE comm_pkg.reset_comm (0.5)	0.25	0.4	0.10 0.5	0.8
10:00	Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'	0.25	0.4	0.5	0.8
11:00	Jones> ROLLBACK;	0.25	0.4	0.5	0.4
11:01	EXIT ...	0.25	0.4	-	0.4
12:00	EXEC comm_pkg.reset_comm(0.2)	0.25	0.4	0.2	0.4

291

The slide sequence is based on two different users, Scott and Jones, executing `comm_pkg` (covered in the lesson titled “Creating Packages”), in which the `reset_comm` procedure invokes the `validate` function to check the new commission. The example shows how the persistent state of the `v_std_comm` package variable is maintained in each user session.

At 9:00: Scott calls `reset_comm` with a new commission value of 0.25, the package state for `v_std_comm` is initialized to 0.10 and then set to 0.25, which is validated because it is less than the database maximum value of 0.4.

At 9:30: Jones inserts a new row into the `EMPLOYEES` table with a new maximum `v_commission_pct` value of 0.8. This is not committed, so it is visible to Jones only. Scott’s state is unaffected.

At 9:35: Jones calls `reset_comm` with a new commission value of 0.5. The state for Jones’s `v_std_comm` is first initialized to 0.10 and then set to the new value 0.5 that is valid for his session with the database maximum value of 0.8.

At 10:00: Scott calls `reset_comm` with a new commission value of 0.6, which is greater than the maximum database commission visible to his session, that is, 0.4. (Jones did not commit the 0.8 value.)

Between 11:00 and 12:00: Jones rolls back the transaction (`INSERT` statement) and exits the session. Jones logs in at 11:45 and successfully

executes the procedure, setting his state to 0.2.

Persistent State of a Package Cursor: Example

```
CREATE OR REPLACE PACKAGE curs_pkg IS -- Package spec
  PROCEDURE open;
  FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN;
  PROCEDURE close;
END curs_pkg;

CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  -- Package body
  CURSOR cur_c IS
    SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT cur_c%ISOPEN THEN
      OPEN cur_c;
    END IF;
  END open;
  . . . -- code continued on next slide
```

292

The example in the slide shows the CURS_PKG package specification and body. The body declaration is continued in the next slide.

To use this package, perform the following steps to process the rows:

1. Call the open procedure to open the cursor.

Persistent State of a Package Cursor: Example

```
 . .
FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN IS
    v_emp_id employees.employee_id%TYPE;
BEGIN
    FOR count IN 1 .. p_n LOOP
        FETCH cur_c INTO v_emp_id;
        EXIT WHEN cur_c%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Id: ' || (v_emp_id));
    END LOOP;
    RETURN cur_c%FOUND;
END next;
PROCEDURE close IS
BEGIN
    IF cur_c%ISOPEN THEN
        CLOSE cur_c;
    END IF;
    END close;
END curs_pkg;
```

293

2. Call the `next` procedure to fetch one or a specified number of rows. If you request more rows than actually exist, the procedure successfully handles termination.
It returns TRUE if more rows need to be processed; otherwise it returns FALSE.
3. Call the `close` procedure to close the cursor, before or at the end of processing the rows.

Note: The cursor declaration is private to the package. Therefore, the cursor state can be influenced by invoking the package procedure and functions listed in the slide.

Executing the CURS_PKG Package

The screenshot shows the Oracle SQL Developer interface. In the top window (Worksheet), the following PL/SQL code is displayed:

```
1 SET SERVEROUTPUT ON
2
3 EXECUTE curs_pkg.open
4 DECLARE
5   v_more BOOLEAN := curs_pkg.next(3);
6 BEGIN
7   IF NOT v_more THEN
8     curs_pkg.close;
9   END IF;
10 END;
11 /
```

In the bottom window (Script Output), the results of the execution are shown:

```
anonymous block completed
anonymous block completed
Id: 100
Id: 101
Id: 102

anonymous block completed
anonymous block completed
Id: 103
Id: 104
Id: 105
```

294

Recall that the state of a package variable or cursor persists across transactions within a session. However, the state does not persist across different sessions for the same user. The database tables hold data that persists across sessions and users. The call to `curs_pkg.open` opens the cursor, which remains open until the session is terminated, or the cursor is explicitly closed. The anonymous block executes the `next` function in the Declaration section, initializing the BOOLEAN variable `b_more` to TRUE, as there are more than three rows in the `EMPLOYEES` table. The block checks for the end of the result set and closes the cursor, if appropriate. When the block executes, it displays the first three rows:

```
Id :100
Id :101
Id :102
```

If you click the Run Script (F5) icon again, the next three rows are displayed:

```
Id :103
Id :104
Id :105
```

To close the cursor, you can issue the following command to close the cursor in the package, or exit the session:

```
EXECUTE curs_pkg.close
```

Using Associative Arrays in Packages

```
CREATE OR REPLACE PACKAGE emp_pkg IS
    TYPE emp_table_type IS TABLE OF employees%ROWTYPE
        INDEX BY BINARY_INTEGER;
    PROCEDURE get_employees(p_emps OUT emp_table_type);
END emp_pkg;
```

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    PROCEDURE get_employees(p_emps OUT emp_table_type) IS
        v_i BINARY_INTEGER := 0;
    BEGIN
        FOR emp_record IN (SELECT * FROM employees)
        LOOP
            emps(v_i) := emp_record;
            v_i:= v_i + 1;
        END LOOP;
    END get_employees;
END emp_pkg;
```

295

Associative arrays used to be known as index by tables.

The `emp_pkg` package contains a `get_employees` procedure that reads rows from the `EMPLOYEES` table and returns the rows using the `OUT` parameter, which is an associative array (PL/SQL table of records). The key points include the following:

- `employee_table_type` is declared as a public type.
- `employee_table_type` is used for a formal output parameter in the procedure, and the `employees` variable in the calling block (shown below).

In SQL Developer, you can invoke the `get_employees` procedure in an anonymous PL/SQL block by using the `v_employees` variable, as shown in the following example and output:

```
SET SERVEROUTPUT ON
DECLARE
    v_employees emp_pkg.emp_table_type;
BEGIN
    emp_pkg.get_employees(v_employees);
    DBMS_OUTPUT.PUT_LINE('Emp 5:'
```

```
anonymous block completed
Emp 5: Fay
```

```
'||v_employees(4).last_name);  
END;
```

Topics covered in this module:

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Use associative arrays (index-by tables) and records in packages

296

Overloading is a feature that enables you to define different subprograms with the same name. It is logical to give two subprograms the same name when the processing in both the subprograms is the same but the parameters passed to them vary.

PL/SQL permits a special subprogram declaration called a forward declaration. A forward declaration enables you to define subprograms in logical or alphabetical order, define mutually recursive subprograms, and group subprograms in a package.

A package initialization block is executed only when the package is first invoked within the other user session. You can use this feature to initialize variables only once per session.

You can keep track of the state of a package variable or cursor, which persists throughout the user session, from the time the user first references the variable or cursor to the time the user disconnects.

Using the PL/SQL wrapper, you can obscure the source code stored in the database to protect your intellectual property.

Oracle PL/SQL 19c

Module 13: Oracle-Supplied Packages



Objectives

Topics covered in this module:

- Describe how the DBMS_OUTPUT package works
- Use UTL_FILE to direct output to operating system files
- Describe the main features of UTL_MAIL

298

In this lesson, you learn how to use some of the Oracle-supplied packages and their capabilities.

- The Oracle-supplied packages:
 - Are provided with the Oracle server
 - Extend the functionality of the database
 - Enable access to certain SQL features that are normally restricted for PL/SQL
- For example, the DBMS_OUTPUT package was originally designed to debug PL/SQL programs.

299

Packages are provided with the Oracle server to allow either of the following:

PL/SQL access to certain SQL features

The extension of the functionality of the database

You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

Most of the standard packages are created by running `catproc.sql`. The DBMS_OUTPUT package is the one that you will be most familiar with during this course. You should already be familiar with this package if you attended the *Oracle Database: PL/SQL Fundamentals* course.

Some Oracle-supplied packages:

- o DBMS_OUTPUT
- o UTL_FILE
- o UTL_MAIL
- o DBMS_ALERT
- o DBMS_LOCK
- o DBMS_SESSION
- o DBMS_APPLICATION_INFO
- o HTP
- o DBMS_SCHEDULER

300

List of Some Oracle-Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. This lesson covers the first three packages in the slide. For more information, refer to the *Oracle Database PL/SQL Packages and Types Reference*. The following is a brief description about the remaining listed packages in the slide:

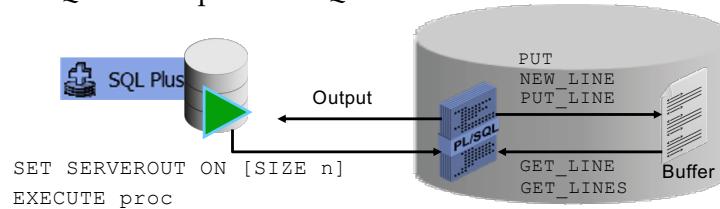
- DBMS_OUTPUT provides debugging and buffering of text data.
- UTL_FILE enables reading and writing of operating system text files.
- UTL_MAIL enables composing and sending of email messages.
- DBMS_ALERT supports asynchronous notification of database events. Messages or alerts are sent on a COMMIT command.
- DBMS_LOCK is used to request, convert, and release locks through Oracle Lock Management services.
- DBMS_SESSION enables programmatic use of the ALTER SESSION SQL statement and other session-level commands.
- DBMS_APPLICATION_INFO can be used with Oracle Trace and the SQL trace facility to record names of executing modules or transactions in the database for later use when tracking the

performance of various modules and debugging.

How the DBMS_OUTPUT Package Works

The DBMS_OUTPUT package enables you to send messages from stored subprograms and triggers.

- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until the sending subprogram or trigger completes.
- Use SET SERVEROUTPUT ON to display messages in SQL Developer and SQL*Plus.



301

The DBMS_OUTPUT package sends textual messages from any PL/SQL block into a buffer in the database. Procedures provided by the package include the following:

- PUT appends text from the procedure to the current line of the line output buffer.
- NEW_LINE places an end-of-line marker in the output buffer.
- PUT_LINE combines the action of PUT and NEW_LINE (to trim leading spaces).
- GET_LINE retrieves the current line from the buffer into a procedure variable.
- GET_LINES retrieves an array of lines into a procedure-array variable.
- ENABLE/DISABLE enables and disables calls to DBMS_OUTPUT procedures.

The buffer size can be set by using:

The SIZE *n* option appended to the SET SERVEROUTPUT ON command where *n* is the number of characters. The minimum is 2,000 and the maximum is unlimited. The default is 20,000. An integer parameter between 2,000 and 1,000,000 in the ENABLE

procedure

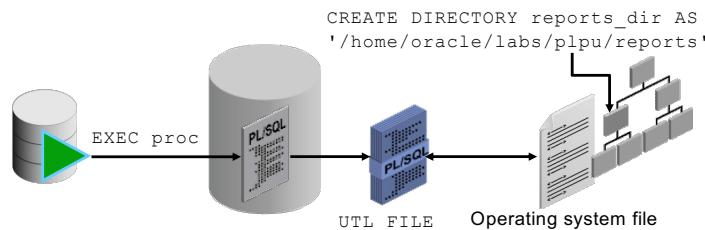
You can output results to the window for debugging purposes. You can trace a code execution path for a function or procedure. You can send messages between subprograms and triggers.

Note: There is no mechanism to flush output during the execution of a procedure.

Using the UTL_FILE Package

The UTL_FILE package extends PL/SQL programs to read and write operating system text files:

- Provides a restricted version of operating system stream file I/O for text files
- Can access files in operating system directories defined by a CREATE DIRECTORY statement



302

Interacting with Operating System Files

The Oracle-supplied UTL_FILE package is used to access text files in the operating system of the database server. The database provides read and write access to specific operating system directories by using:

A CREATE DIRECTORY statement that associates an alias with an operating system directory. The database directory alias can be granted the READ and WRITE privileges to control the type of access to files in the operating system. For example:

```
CREATE DIRECTORY my_dir AS '/temp/my_files';
GRANT READ, WRITE ON DIRECTORY my_dir TO public.
```

The paths specified in the utl_file_dir database initialization parameter

It is recommended that you use the CREATE DIRECTORY feature instead of UTL_FILE_DIR for directory access verification. Directory objects offer more flexibility and granular control to the UTL_FILE application administrator, can be maintained dynamically (that is, without shutting down the database), and are consistent with other Oracle tools. The CREATE DIRECTORY privilege is granted only to SYS and SYSTEM by default.

The operating system directories specified by using either of these techniques

should be accessible to and on the same machine as the database server processes. The path (directory) names may be case-sensitive for some operating systems.

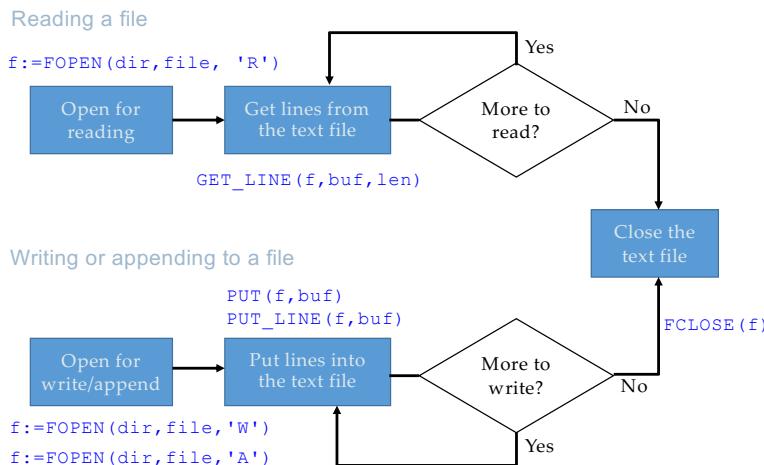
Some of the UTL_FILE Procedures and Functions

Subprogram	Description
ISOPEN function	Determines if a file handle refers to an open file
FOPEN function	Opens a file for input or output
FCLOSE function	Closes all open file handles
FCOPY procedure	Copies a contiguous portion of a file to a newly created file
FGETATTR procedure	Reads and returns the attributes of a disk file
GET_LINE procedure	Reads text from an open file
FREMOVE procedure	Deletes a disk file, if you have sufficient privileges
FRENAME procedure	Renames an existing file to a new name
PUT procedure	Writes a string to a file
PUT_LINE procedure	Writes a line to a file, and so appends an operating system-specific line terminator

303

The table in the slide lists some of the UTL_FILE Package subprograms. For a complete list of the package's subprograms, see *Oracle Database PL/SQL Packages and Types Reference*.

File Processing Using the UTL_FILE Package: Overview



304

You can use the procedures and functions in the `UTL_FILE` package to open files with the `FOPEN` function. You can then either read from or write or append to the file until processing is done. After you finish processing the file, close the file by using the `FCLOSE` procedure. The following are the subprograms:

The `FOPEN` function opens a file in a specified directory for input/output (I/O) and returns a file handle used in subsequent I/O operations.

The `IS_OPEN` function returns a Boolean value whenever a file handle refers to an open file. Use `IS_OPEN` to check whether the file is already open before opening the file.

The `GET_LINE` procedure reads a line of text from the file into an output buffer parameter. (The maximum input record size is 1,023 bytes unless you specify a larger size in the overloaded version of `FOPEN`.)

The `PUT` and `PUT_LINE` procedures write text to the opened file.

The `PUTF` procedure provides formatted output with two format specifiers: `%s` to substitute a value into the output string and `\n` for a new line character.

The NEW_LINE procedure terminates a line in an output file.
The FFLUSH procedure writes all data buffered in memory to a file.
The FCLOSE procedure closes an opened file.
The FCLOSE_ALL procedure closes all opened file handles for the session.

Using the Available Declared Exceptions in the UTL_FILE Package

Exception Name	Description
INVALID_PATH	File location invalid
INVALID_MODE	The open_mode parameter in FOPEN is invalid
INVALID_FILEHANDLE	File handle invalid
INVALID_OPERATION	File could not be opened or operated on as requested
READ_ERROR	Operating system error occurred during the read operation
WRITE_ERROR	Operating system error occurred during the write operation
INTERNAL_ERROR	Unspecified PL/SQL error

305

The UTL_FILE package declares fifteen exceptions that indicate an error condition in the operating system file processing. You may have to handle one of these exceptions when using UTL_FILE subprograms.

A subset of the exceptions are displayed in the slide. For additional information about the remaining exceptions, refer to *Oracle Database PL/SQL Packages and Types Reference*.

Note: These exceptions must always be prefixed with the package name.

UTL_FILE procedures can also raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR.

The NO_DATA_FOUND exception is raised when reading past the end of a file by using UTL_FILE.GET_LINE or UTL_FILE.GET_LINES.

FOPEN and IS_OPEN Functions: Example

- This FOPEN function opens a file for input or output.

```
FUNCTION FOPEN (p_location  IN VARCHAR2,
               p_filename   IN VARCHAR2,
               p_open_mode  IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

- The IS_OPEN function determines whether a file handle refers to an open file.

```
FUNCTION IS_OPEN (p_file IN FILE_TYPE)
RETURN BOOLEAN;
```

306

The parameters include the following:

- p_location parameter:** Specifies the name of a directory alias defined by a CREATE DIRECTORY statement, or an operating system-specific path specified by using the utl_file_dir database parameter
- p_filename parameter:** Specifies the name of the file, including the extension, without any path information
- open_mode string:** Specifies how the file is to be opened. Values are:

```
'R' for reading text (use GET_LINE)  
'W' for writing text (PUT, PUT_LINE, NEW_LINE,  
PUTF, FFLUSH)  
'A' for appending text (PUT, PUT_LINE, NEW_LINE,  
PUTF, FFLUSH)
```

The return value from FOPEN is a file handle whose type is UTL_FILE.FILE_TYPE. The handle must be used on subsequent calls to routines that operate on the opened file.

The IS_OPEN function parameter is the file handle. The IS_OPEN function tests a file handle to see whether it identifies an opened file. It returns a

Boolean value of TRUE if the file has been opened; otherwise it returns a value of FALSE indicating that the file has not been opened. The example in the slide shows how to combine the use of the two subprograms. For the full syntax, refer to *Oracle Database PL/SQL Packages and Types Reference*.

Using UTL_FILE: Example

```
CREATE OR REPLACE PROCEDURE sal_status(
    p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
    f_file UTL_FILE.FILE_TYPE;
    CURSOR cur_emp IS
        SELECT last_name, salary, department_id
        FROM employees ORDER BY department_id;
    v_newdeptno employees.department_id%TYPE;
    v_olddeptno employees.department_id%TYPE := 0;
BEGIN
    f_file:=UTL_FILE.FOPEN(p_dir, p_filename, 'W');
    UTL_FILE.PUT_LINE(f_file,
        'REPORT: GENERATED ON ' || SYSDATE);
    UTL_FILE.NEW_LINE (f_file);
    .
.
```

307

In the slide example, the `sal_status` procedure creates a report of employees for each department, along with their salaries. The data is written to a text file by using the `UTL_FILE` package. In the code example, the `file` variable is declared as `UTL_FILE.FILE_TYPE`, a package type that is a record with a field called `ID` of the `BINARY_INTEGER` data type. For example:

```
TYPE file_type IS RECORD (id BINARY_INTEGER);
```

The field of `FILE_TYPE` record is private to the `UTL_FILE` package and should never be referenced or changed. The `sal_status` procedure accepts two parameters:

- The `p_dir` parameter for the name of the directory in which to write the text file

- The `p_filename` parameter to specify the name of the file

For example, to call the procedure, use the following after ensuring that the external file and the database are on the same PC:

```
EXECUTE sal_status('REPORTS_DIR', 'salreport2.txt')
```

Note: The directory location used (`REPORTS_DIR`) must be in uppercase characters if it is a directory alias created by a `CREATE DIRECTORY` statement. When reading a file in a loop, the loop should exit when it detects

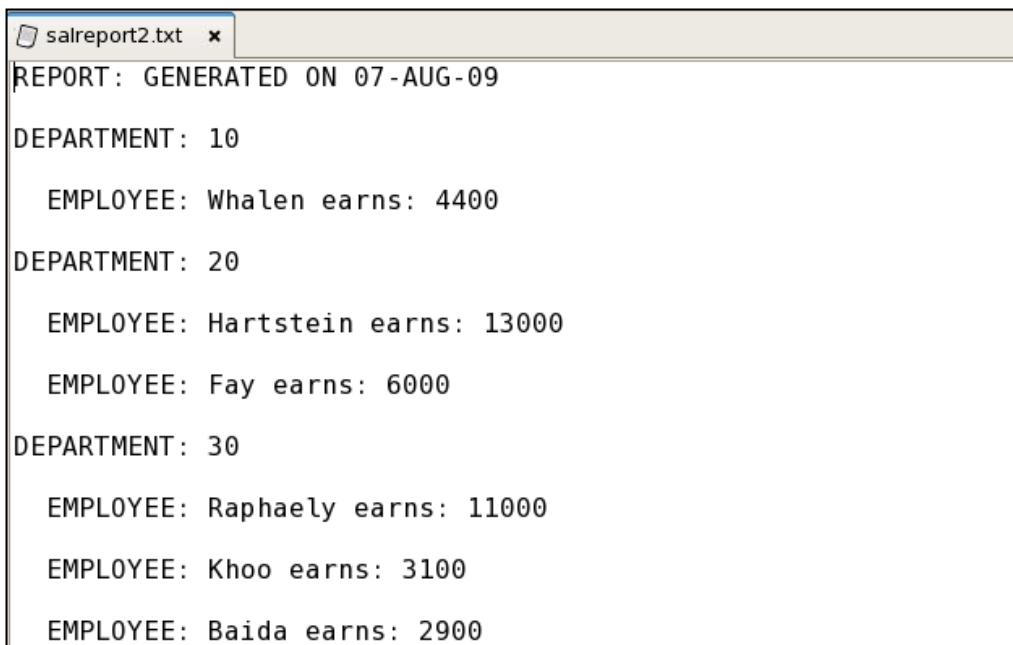
the NO_DATA_FOUND exception. The UTL_FILE output is sent synchronously. A DBMS_OUTPUT procedure does not produce an output until the procedure is completed.

Using UTL_FILE: Example

```
.
.
FOR emp_rec IN cur_emp LOOP
    IF emp_rec.department_id <> v_olddeptno THEN
        UTL_FILE.PUT_LINE (f_file,
                           'DEPARTMENT: ' || emp_rec.department_id);
        UTL_FILE.NEW_LINE (f_file);
    END IF;
    UTL_FILE.PUT_LINE (f_file,
                       'EMPLOYEE: ' || emp_rec.last_name ||
                           ' earns: ' || emp_rec.salary);
    v_olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (f_file);
END LOOP;
UTL_FILE.PUT_LINE(f_file,'*** END OF REPORT ***');
UTL_FILE.FCLOSE (f_file);
EXCEPTION
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
    WHEN UTL_FILE.WRITE_ERROR THEN
        RAISE_APPLICATION_ERROR (-20002, 'Unable to write to file');
END sal_status;/
```

308

The following is a sample of the `salreport2.txt` output file:



```
salreport2.txt x
REPORT: GENERATED ON 07-AUG-09

DEPARTMENT: 10

EMPLOYEE: Whalen earns: 4400

DEPARTMENT: 20

EMPLOYEE: Hartstein earns: 13000

EMPLOYEE: Fay earns: 6000

DEPARTMENT: 30

EMPLOYEE: Raphaely earns: 11000

EMPLOYEE: Khoo earns: 3100

EMPLOYEE: Baida earns: 2900
```

What Is the UTL_MAIL Package?

- A utility for managing email
- Requires the setting of the SMTP_OUT_SERVER database initialization parameter
- Provides the following procedures:
 - SEND for messages without attachments
 - SEND_ATTACH_RAW for messages with binary attachments
 - SEND_ATTACH_VARCHAR2 for messages with text attachments

309

Using UTL_MAIL

The UTL_MAIL package is a utility for managing email that includes commonly used email features such as attachments, CC, BCC, and return receipt.

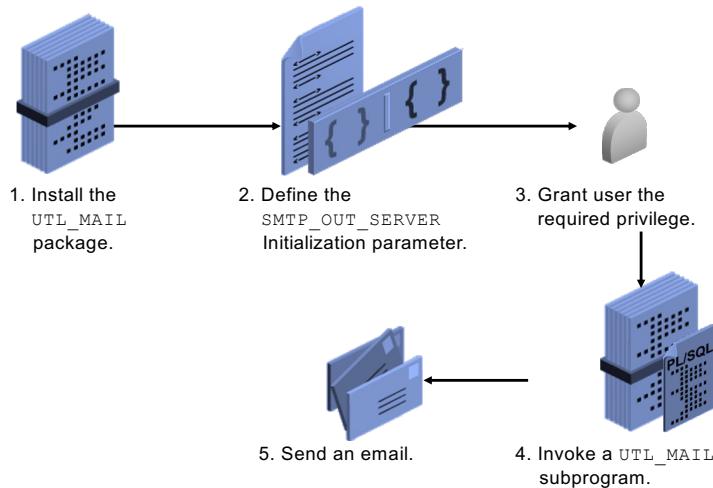
The UTL_MAIL package is not installed by default because of the SMTP_OUT_SERVER configuration requirement and the security exposure this involves. When installing UTL_MAIL, you should take steps to prevent the port defined by SMTP_OUT_SERVER being swamped by data transmissions. To install UTL_MAIL, log in as a DBA user in SQL*Plus and execute the following scripts:

```
@$ORACLE_HOME/rdbms/admin/utlmail.sql  
@$ORACLE_HOME/rdbms/admin/prvtmail.plb
```

You should define the SMTP_OUT_SERVER parameter in the init.ora file database initialization file:

```
SMTP_OUT_SERVER=mystmpserver.mydomain.com
```

Setting Up and Using the UTL_MAIL: Overview



310

In Oracle Database 19c, the `UTL_MAIL` package is now an invoker's rights package and the invoking user will need the connect privilege granted in the access control list assigned to the remote network host to which he wants to connect. The Security Administrator performs this task.

Note

For information about how a user with `SYSDBA` capabilities grants a user the required fine-grained privileges required for using this package, refer to the “Managing Fine-Grained Access to External Network Services” topic in *Oracle Database Security Guide 19c Release 2 (11.2)* and the *Oracle Database 19c Advanced PL/SQL* instructor-led training course.

Due to firewall restrictions, the `UTL_MAIL` examples in this lesson cannot be demonstrated; therefore, no labs were designed to use `UTL_MAIL`.

Summary of UTL_MAIL Subprograms

Subprogram	Description
SEND procedure	Packages an email message, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients
SEND_ATTACH_RAW Procedure	Represents the SEND procedure overloaded for RAW attachments
SEND_ATTACH_VARCHAR2 Procedure	Represents the SEND procedure overloaded for VARCHAR2 attachments

- o As SYSDBA, using SQL Developer or SQL*Plus:
 - Install the UTL_MAIL package

```
@?/rdbms/admin/utlmail.sql
@?/rdbms/admin/prvtmail.plb
```

- Set the SMTP_OUT_SERVER

```
ALTER SYSTEM SET SMTP_OUT_SERVER='smtp.server.com'
SCOPE=SPFILE
```

- o As a developer, invoke a UTL_MAIL procedure:

```
BEGIN
  UTL_MAIL.SEND('otn@oracle.com','user@oracle.com',
    message => 'For latest downloads visit OTN',
    subject => 'OTN Newsletter');
END;
```

312

The slide shows how to configure the SMTP_OUT_SERVER parameter to the name of the SMTP host in your network, and how to install the UTL_MAIL package that is not installed by default. Changing the SMTP_OUT_SERVER parameter requires restarting the database instance. These tasks are performed by a user with SYSDBA capabilities.

The last example in the slide shows the simplest way to send a text message by using the UTL_MAIL.SEND procedure with at least a subject and a message. The first two required parameters are the following :

The sender email address (in this case, otn@oracle.com)

The recipients email address (for example, user@oracle.com). The value can be a comma-separated list of addresses.

The UTL_MAIL.SEND procedure provides several other parameters, such as cc, bcc, and priority with default values, if not specified. In the example, the message parameter specifies the text for the email, and the subject parameter contains the text for the subject line. To send an HTML message with HTML tags, add the mime_type parameter (for example, mime_type=>'text/html').

Note: For details about all the UTL_MAIL procedure parameters, refer to

Oracle Database PL/SQL Packages and Types Reference.

Packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients.

```
UTL_MAIL.SEND (
    sender      IN      VARCHAR2 CHARACTER SET ANY_CS,
    recipients   IN      VARCHAR2 CHARACTER SET ANY_CS,
    cc          IN      VARCHAR2 CHARACTER SET ANY_CS
                      DEFAULT NULL,
    bcc         IN      VARCHAR2 CHARACTER SET ANY_CS
                      DEFAULT NULL,
    subject     IN      VARCHAR2 CHARACTER SET ANY_CS
                      DEFAULT NULL,
    message     IN      VARCHAR2 CHARACTER SET ANY_CS,
    mime_type   IN      VARCHAR2
                      DEFAULT 'text/plain; charset=us-ascii',
    priority    IN      PLS_INTEGER DEFAULT NULL);
```

313

The *SEND* Procedure

This procedure packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients. It hides the SMTP API and exposes a one-line email facility for ease of use.

The *SEND* Procedure Parameters

- **sender:** The email address of the sender.
- **recipients:** The email addresses of the recipient(s), separated by commas.
- **cc:** The email addresses of the CC recipient(s), separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipient(s), separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **message:** A text message body.
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'.
- **priority:** The message priority. The default is NULL.

The *SEND_ATTACH_RAW* Procedure

This procedure is the *SEND* procedure overloaded for *RAW* attachments.

```
UTL_MAIL.SEND_ATTACH_RAW (
    sender      IN  VARCHAR2 CHARACTER SET ANY_CS,
    recipients   IN  VARCHAR2 CHARACTER SET ANY_CS,
    cc          IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    bcc         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    subject     IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    message     IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    mime_type   IN  VARCHAR2 DEFAULT CHARACTER SET ANY_CS
                      DEFAULT 'text/plain; charset=us-ascii',
    priority    IN  PLS_INTEGER DEFAULT 3,
    attachment  IN  RAW,
    att_inline   IN  BOOLEAN DEFAULT TRUE,
    att_mime_type IN  VARCHAR2 CHARACTER SET ANY_CS
                      DEFAULT 'text/plain; charset=us-ascii',
    att_filename IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```

314

The *SEND_ATTACH_RAW* Procedure Parameters

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipient(s), separated by commas
- **cc:** The email addresses of the CC recipient(s), separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipient(s), separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **message:** A text message body
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'
- **priority:** The message priority. The default is NULL.
- **attachment:** A RAW attachment
- **att_inline:** Specifies whether the attachment is viewable inline with the message body. The default is TRUE.

Sending Email with a Binary Attachment: Example

```
CREATE OR REPLACE PROCEDURE send_mail_logo IS
BEGIN
    UTL_MAIL.SEND_ATTACH_RAW(
        sender => 'me@oracle.com',
        recipients => 'you@somewhere.net',
        message =>
            '<HTML><BODY>See attachment</BODY></HTML>',
        subject => 'Oracle Logo',
        mime_type => 'text/html',
        attachment => get_image('oracle.gif'),
        att_inline => true,
        att_mime_type => 'image/gif',
        att_filename => 'oralogo.gif');
END;
/
```

315

The slide shows a procedure calling the `UTL_MAIL.SEND_ATTACH_RAW` procedure to send a textual or an HTML message with a binary attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the email message, the `SEND_ATTACH_RAW` procedure has the following highlighted parameters:

- The `attachment` parameter (required) accepts a `RAW` data type, with a maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is Boolean (default `TRUE`) to indicate that the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_image` function in the example uses a `BFILE` to read the image data. Using a `BFILE` requires creating a logical directory name in the database by using the `CREATE DIRECTORY` statement. The code for `get_image` is shown on the following page.

The *SEND_ATTACH_VARCHAR2* Procedure

This procedure is the *SEND* procedure overloaded for *VARCHAR2* attachments.

```
UTL_MAIL.SEND_ATTACH_VARCHAR2 (
    sender          IN  VARCHAR2 CHARACTER SET ANY_CS,
    recipients      IN  VARCHAR2 CHARACTER SET ANY_CS,
    cc              IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    bcc             IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    subject         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    message          IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    mime_type        IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    priority         IN  PLS_INTEGER DEFAULT 3,
    attachment       IN  VARCHAR2 CHARACTER SET ANY_CS,
    att_inline       IN  BOOLEAN DEFAULT TRUE,
    att_mime_type    IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    att_filename     IN  VARCHAR2CHARACTER SET ANY_CS DEFAULT NULL);
```

316

The *SEND_ATTACH_VARCHAR2* Procedure Parameters

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipient(s), separated by commas
- **cc:** The email addresses of the CC recipient(s), separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipient(s), separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **Message:** A text message body
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'
- **priority:** The message priority. The default is NULL.
- **attachment:** A text attachment
- **att_inline:** Specifies whether the attachment is inline. The default is TRUE.
- **att_mime_type:** The mime type of the attachment, default is

'text/plain; charset=us-ascii'
`.att_filename`: The string specifying a file name containing the attachment. The default is NULL.

Sending Email with a Text Attachment: Example

```
CREATE OR REPLACE PROCEDURE send_mail_file IS
BEGIN
    UTL_MAIL.SEND_ATTACH_VARCHAR2(
        sender => 'me@oracle.com',
        recipients => 'you@somewhere.net',
        message =>
            '<HTML><BODY>See attachment</BODY></HTML>',
        subject => 'Oracle Notes',
        mime_type => 'text/html'
        attachment => get_file('notes.txt'),
        att_inline => false,
        att_mime_type => 'text/plain',
        att_filename => 'notes.txt');
END;
/
```

317

Sending Email with a Text Attachment

The slide shows a procedure that calls the `UTL_MAIL.SEND_ATTACH_VARCHAR2` procedure to send a textual or an HTML message with a text attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the e-mail message, the `SEND_ATTACH_VARCHAR2` procedure has the following parameters highlighted:

The `attachment` parameter (required) accepts a `VARCHAR2` data type with a maximum size of 32,767 binary characters.

The `att_inline` parameter (optional) is a Boolean (default `TRUE`) to indicate that the attachment is viewable with the message body.

The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.

The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_file` function in the example uses a `BFILE` to read a text file from the operating system directories for the value of the attachment

parameter, which could simply be populated from a VARCHAR2 variable. The code for `get_file` is shown on the following page.

Topics covered in this module:

- How the DBMS_OUTPUT package works
- How to use UTL_FILE to direct output to operating system files
- About the main features of UTL_MAIL

318

This lesson covers a small subset of packages provided with the Oracle database. You have extensively used DBMS_OUTPUT for debugging purposes and displaying procedurally generated information on the screen in SQL*Plus. In this lesson, you should have learned how to use the power features provided by the database to create text files in the operating system by using UTL_FILE. You also learned how to send email with or without binary or text attachments by using the UTL_MAIL package.

Note: For more information about all PL/SQL packages and types, refer to

PL/SQL

Packages and Types Reference.



Oracle PL/SQL 19c



Module 14: Dynamic SQL

Topics covered in this module:

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

320

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.

- All SQL statements go through some or all of the following stages:
 - Parse
 - Bind
 - Execute
 - Fetch
- Some stages may not be relevant for all statements:
 - The fetch phase is applicable to queries.
 - For embedded SQL statements such as SELECT, DML, MERGE, COMMIT, SAVEPOINT, and ROLLBACK, the parse and bind phases are done at compile time.
 - For dynamic SQL statements, all phases are performed at run time.

321

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 non-query statements, this is the last phase.

Fetch: In the fetch stage, which is applicable to queries, the rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched.

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, string variable, or string expression within the application
- Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)
- Enables DDL, DCL, 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, MERGE, COMMIT, and ROLLBACK, all of which are parsed at compile time—that is, they have a fixed structure. You need to use dynamic SQL functionality if you require:

The structure of a SQL statement to be altered at run time

Access to data definition language (DDL) statements and other SQL functionality in PL/SQL

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

Native Dynamic SQL statements with EXECUTE IMMEDIATE

The DBMS_SQL package

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

session control statement (such as ALTER SESSION).

- Use dynamic SQL when the full text of the dynamic SQL statement is unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.
- Use dynamic SQL when one of the following items is unknown at precompile time:
 - Text of the SQL statement such as commands, clauses, and so on
 - The number and data types of host variables
 - References to database objects such as tables, columns, indexes, sequences, usernames, and views
- Use dynamic SQL to make your PL/SQL programs more general and flexible.

323

In PL/SQL, you need dynamic SQL to execute the following SQL statements

where the full text is unknown at compile time such as:

- A SELECT statement that includes an identifier that is unknown at compile time (such as a table name)
- A WHERE clause in which the column name is unknown at compile time

Note

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

*Pro*C/C++ Programmer's Guide*

Lesson 13, Oracle Dynamic SQL, covers the four available methods that you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, and then offers guidelines for choosing the right method. Later sections in the same guide show you how to use the methods, and include example programs that you can study.

Lesson 15, Oracle Dynamic SQL: Method 4, contains very detailed information about Method 4 when defining dynamic SQL statements.

Oracle PL/SQL Programming book by Steven Feuerstein and Bill Pribyl.
Lesson 16, Dynamic SQL and Dynamic PL/SQL, contains additional
information about dynamic SQL.

- 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.
- If the dynamic SQL statement is a SELECT statement that returns multiple rows, NDS gives you the following choices:
 - Use the EXECUTE IMMEDIATE statement with the BULK COLLECT INTO clause
 - Use the OPEN-FOR, FETCH, and CLOSE statements
- In Oracle Database 19c, NDS supports statements larger than 32 KB by accepting a CLOB argument.

324

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

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

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

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

Using the **EXECUTE IMMEDIATE** Statement

Use the **EXECUTE IMMEDIATE** statement for NDS 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**.

325

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).

Available Methods for Using NDS

Method #	SQL Statement Type	NDS SQL Statements Used
Method 1	Non-query without host variables	EXECUTE IMMEDIATE without the USING and INTO clauses
Method 2	Non-query with known number of input host variables	EXECUTE IMMEDIATE with a USING clause
Method 3	Query with known number of select-list items and input host variables	EXECUTE IMMEDIATE with the USING and INTO clauses
Method 4	Query with unknown number of select-list items or input host variables	Use the DBMS_SQL package

326

The four available methods for NDS that are listed in the slide are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and Method 4 encompasses Methods 1, 2, and 3. However, each method is most useful for handling a certain kind of SQL statement, as follows:

Method 1

This method lets your program accept or build a dynamic SQL statement, and then immediately execute it using the EXECUTE IMMEDIATE command.

The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

- DELETE FROM EMPLOYEES WHERE DEPTNO = 20
- GRANT SELECT ON EMPLOYEES TO scott

With Method 1, the SQL statement is parsed every time it is executed.

Note

Examples of non-queries include data definition language (DDLS) statements, UPDATES, INSERTS, or DELETES.

The term *select-list item* includes column names and expressions such as SAL * 1.10 and MAX(SAL).

Dynamic SQL with a DDL Statement: Examples

```
-- Create a table using dynamic SQL

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

```
-- Call the procedure

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

327

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

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

An ID column with a `NUMBER` data type used as a primary key

A name column of up to 40 characters for the employee name

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

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

/

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

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

Dynamic SQL with DML Statements

```
-- Delete rows from any table:  
CREATE FUNCTION del_rows(p_table_name VARCHAR2)  
RETURN NUMBER IS  
BEGIN  
    EXECUTE IMMEDIATE 'DELETE FROM '|| p_table_name;  
    RETURN SQL%ROWCOUNT;  
END;  
/  
BEGIN DBMS_OUTPUT.PUT_LINE(  
    del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');//  
END;  
/  
  
-- Insert a row into a table with two columns:  
CREATE PROCEDURE add_row(p_table_name VARCHAR2,  
    p_id NUMBER, p_name VARCHAR2) IS  
BEGIN  
    EXECUTE IMMEDIATE 'INSERT INTO '|| p_table_name ||  
        ' VALUES (:1, :2)' USING p_id, p_name;  
END;
```

328

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

The `del_rows` function deletes rows from a specified table and returns the number of rows deleted by using the implicit SQL cursor `%ROWCOUNT` attribute. Executing the function is shown below the example for creating a function.

The `add_row` procedure shows how to provide input values to a dynamic SQL statement with the `USING` clause. The bind variable names `:1` and `:2` are not important; however, the order of the parameter names (`p_id` and `p_name`) in the `USING` clause is associated with the bind variables by position, in the order of their respective appearance. Therefore, the PL/SQL parameter `p_id` is assigned to the `:1` placeholder, and the `p_name` parameter is assigned to the `:2` placeholder. Placeholder or bind variable names can be alphanumeric but must be preceded with a colon.

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

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

Dynamic SQL with a Single-Row Query: Example

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

FUNCTION get_emp(p_emp_id Compiled.
anonymous block completed
Emp: King

329

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

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

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

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

Note

For an example of “Dynamic SQL with a Multirow Query: Example” using `REF CURSORS`, see the `demo_06_13_a` in the `/home/oracle/labs/plpu/demo` folder.

For an example on using `REF CURSORS`, see the `demo_06_13_b` in the `/home/oracle/labs/plpu/demo` folder.

`REF CURSORS` are covered in the *Oracle Database 19c: Advanced PL/SQL* or *Oracle Database 10g: Advanced PL/SQL*.

Executing a PL/SQL Anonymous Block Dynamically

```
CREATE FUNCTION annual_sal( p_emp_id NUMBER)
RETURN NUMBER IS
v_plsql varchar2(200) := 
'DECLARE |||
  rec_emp employees%ROWTYPE; ||
BEGIN ||
  rec_emp := get_emp(:empid); ||
  :res:= rec_emp.salary * 12; ||
END;';
v_result NUMBER;
BEGIN
  EXECUTE IMMEDIATE v_plsql
    USING IN p_emp_id, OUT v_result; (2)
  RETURN v_result;
END;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

330

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.

The output of the slide example is as follows:

```
FUNCTION annual_sal compiled
anonymous block completed
288000
```

Compile PL/SQL code with the ALTER statement:

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

```
CREATE PROCEDURE compile_plsql(p_name VARCHAR2,
    p_plsql_type VARCHAR2, p_options VARCHAR2 := NULL) IS
    v_stmt varchar2(200) := 'ALTER '|| p_plsql_type || 
                           ' '|| p_name || ' COMPILE';
BEGIN
    IF p_options IS NOT NULL THEN
        v_stmt := v_stmt || ' '|| p_options;
    END IF;
    EXECUTE IMMEDIATE v_stmt;
END;
/
```

331

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')
```

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

332

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

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

When generating dynamic SQL, you can either use the DBMS_SQL supplied package when dealing with Method 4 situations, or you can use native dynamic SQL. Before Oracle Database 19c, each of these methods had functional limitations. In Oracle Database 19c, functionality is added to both methods to make them more complete.

The features for executing dynamic SQL from PL/SQL had some restrictions in Oracle Database 10g. DBMS_SQL was needed for Method 4 scenarios but it could not handle the full range of data types and its cursor representation was not usable by a client to the database. Native dynamic SQL was more convenient for non-Method 4 scenarios, but it did not support statements

bigger than 32 KB. Oracle Database 19c removes these and other restrictions to make the support of dynamic SQL from PL/SQL functionally complete.

Examples of the package procedures and functions:

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

333

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

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

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

Using DBMS_SQL with a DML Statement: Deleting Rows

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

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

334

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.

Using DBMS_SQL with a Parameterized DML Statement

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

335

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

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

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

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

each selected column value into each PL/SQL variable for processing.

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

Topics covered in this module:

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

336

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.



Oracle PL/SQL 19c



Module 15: PL/SQL Design Considerations

Topics covered in this module:

- Create standard constants and exceptions
- Write and call local subprograms
- Control the run-time privileges of a subprogram
- Perform autonomous transactions
- Pass parameters by reference using a NOCOPY hint
- Use the PARALLEL ENABLE hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the DETERMINISTIC clause with functions
- Use the RETURNING clause and bulk binding with DML

338

In this lesson, you learn to use package specifications to standardize names for constant values and exceptions. You learn how to create subprograms in the Declaration section of any PL/SQL block for using locally in the block. The AUTHID compiler directive is discussed to show how you can manage run-time privileges of the PL/SQL code, and create independent transactions by using the AUTONOMOUS TRANSACTION directive for subprograms. This lesson also covers some performance considerations that can be applied to PL/SQL applications, such as bulk binding operations with a single SQL statement, the RETURNING clause, and the NOCOPY and PARALLEL ENABLE hints.

Constants and exceptions are typically implemented using a bodiless package (that is, a package specification).

- Standardizing helps to:
 - Develop programs that are consistent
 - Promote a higher degree of code reuse
 - Ease code maintenance
 - Implement company standards across entire applications
- Start with standardization of:
 - Exception names
 - Constant definitions

339

When several developers are writing their own exception handlers in an application, there could be inconsistencies in the handling of error situations. Unless certain standards are adhered to, the situation can become confusing because of the different approaches followed in handling the same error or because of the display of conflicting error messages that confuse users. To overcome these, you can:

Implement company standards that use a consistent approach to error handling across the entire application

Create predefined, generic exception handlers that produce consistency in the application

Write and call programs that produce consistent error messages

All good programming environments promote naming and coding standards. In PL/SQL, a good place to start implementing naming and coding standards is with commonly used constants and exceptions that occur in the application domain.

The PL/SQL package specification construct is an excellent component to support standardization because all identifiers declared in the package specification are public. They are visible to the subprograms that are developed by the owner of the package and all code with EXECUTE rights to

the package specification.

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
    e_fk_err      EXCEPTION;
    e_seq_nbr_err EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
    PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
    ...
END error_pkg;
/
```

340

In the example in the slide, the `error_pkg` package is a standardized exception package. It declares a set of programmer-defined exception identifiers. Because many of the Oracle database predefined exceptions do not have identifying names, the example package shown in the slide uses the `PRAGMA EXCEPTION_INIT` directive to associate selected exception names with an Oracle database error number. This enables you to refer to any of the exceptions in a standard way in your applications, as in the following example:

```
BEGIN
    DELETE FROM departments
    WHERE department_id = deptno;
    ...
EXCEPTION
    WHEN error_pkg.e_fk_err THEN
        ...
    WHEN OTHERS THEN
        ...
END;
```

/

► Consider writing a subprogram for common exception handling to:

- Display errors based on SQLCODE and SQLERRM values for exceptions
- Track run-time errors easily by using parameters in your code to identify:
 - The procedure in which the error occurred
 - The location (line number) of the error
 - RAISE_APPLICATION_ERROR using stack trace capabilities, with the third argument set to TRUE

341

Standardized exception handling can be implemented either as a stand-alone subprogram or a subprogram added to the package that defines the standard exceptions. Consider creating a package with:

- Every named exception that is to be used in the application
- All unnamed, programmer-defined exceptions that are used in the application. These are error numbers -20000 through -20999.
- A program to call RAISE_APPLICATION_ERROR based on package exceptions
- A program to display an error based on the values of SQLCODE and SQLERRM
- Additional objects, such as error log tables, and programs to access the tables

A common practice is to use parameters that identify the name of the procedure and the location in which the error has occurred. This enables you to keep track of run-time errors more easily. An alternative is to use the RAISE_APPLICATION_ERROR built-in procedure to keep a stack trace of exceptions that can be used to track the call sequence leading to the error. To do this, set the third optional argument to TRUE. For example:

```
RAISE_APPLICATION_ERROR(-20001, 'My first error',
```

TRUE);

This is meaningful when more than one exception is raised in this manner.

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS  
    c_order_received CONSTANT VARCHAR(2) := 'OR';  
    c_order_shipped   CONSTANT VARCHAR(2) := 'OS';  
    c_min_sal         CONSTANT NUMBER(3) := 900;  
END constant_pkg;
```

342

By definition, a variable's value changes, whereas a constant's value cannot be changed. If you have programs that use local variables whose values should not and do not change, then convert the variables to constants. This can help with the maintenance and debugging of your code.

Consider creating a single shared package with all your constants in it. This makes maintenance and change of the constants much easier. This procedure or package can be loaded on system startup for better performance.

The example in the slide shows the `constant_pkg` package containing a few constants. Refer to any of the package constants in your application as required. Here is an example:

```
BEGIN  
    UPDATE employees  
        SET salary = salary + 200  
    WHERE salary <= constant_pkg.c_min_sal;  
END;  
/
```

Local Subprograms

A local subprogram is a PROCEDURE or FUNCTION defined at the end of the declarative section in a subprogram.

```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
  v_emp employees%ROWTYPE;
  FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN p_salary * 0.825;
  END tax;
BEGIN
  SELECT * INTO v_emp
  FROM EMPLOYEES WHERE employee_id = p_id;
  DBMS_OUTPUT.PUT_LINE('Tax: '|| tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```

```
PROCEDURE employee_sal compiled
anonymous block completed
Tax: 19800
```

343

Local subprograms can drive top-down design. They reduce the size of a module by removing redundant code. This is one of the main reasons for creating a local subprogram. If a module needs the same routine several times, but only this module needs the routine, then define it as a local subprogram.

You can define a named PL/SQL block in the declarative section of any PL/SQL program, procedure, function, or anonymous block *if it is declared at the end of the Declaration section*. Local subprograms have the following characteristics:

They are accessible to only the block in which they are defined.

They are compiled as part of their enclosing blocks.

The benefits of local subprograms are:

Reduction of repetitive code

Improved code readability and ease of maintenance

Less administration because there is one program to maintain instead of two

The concept is simple. The example shown in the slide illustrates this with a basic example of an income tax calculation of an employee's salary.

Definer's Rights Versus Invoker's Rights

Definer's rights:

- Programs execute with the privileges of the creating user.
- User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.

Invoker's rights:

- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the procedure accesses.

344

Definer's Rights Model

By default, all programs executed with the privileges of the user who created the subprogram. This is known as the definer's rights model, which:

Allows a caller of the program the privilege to execute the procedure, but no privileges on the underlying objects that the procedure accesses

Requires the owner to have all the necessary object privileges for the objects that the procedure references

For example, if user Scott creates a PL/SQL subprogram `get_employees` that is subsequently invoked by Sarah, then the `get_employees` procedure runs with the privileges of the definer Scott.

Invoker's Rights Model

In the invoker's rights model, which was introduced in Oracle8i, programs are executed with the privileges of the calling user. A user of a procedure running with invoker's rights requires privileges on the underlying objects that the procedure references.

For example, if Scott's PL/SQL subprogram `get_employees` is invoked by Sarah, then the `get_employees` procedure runs with the privileges of the invoker Sarah.

```
CREATE OR REPLACE PROCEDURE add_dept(
    p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
    INSERT INTO departments
    VALUES (p_id, p_name, NULL, NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema
- Calls to other packages, functions, and procedures are resolved in the definer's schema

345

Specifying Invoker's Rights

You can set the invoker's rights for different PL/SQL subprogram constructs as follows:

```
CREATE FUNCTION name RETURN type AUTHID
CURRENT_USER IS...
CREATE PROCEDURE name AUTHID CURRENT_USER IS...
CREATE PACKAGE name AUTHID CURRENT_USER IS...
CREATE TYPE name AUTHID CURRENT_USER IS
OBJECT...
```

The default is AUTHID DEFINER, which specifies that the subprogram executes with the privileges of its owner. Most supplied PL/SQL packages such as DBMS_LOB, DBMS_ROWID, and so on, are invoker-rights packages.

Name Resolution

For a definer's rights procedure, all external references are resolved in the definer's schema. For an invoker's rights procedure, the resolution of external references depends on the kind of statement in which they appear:

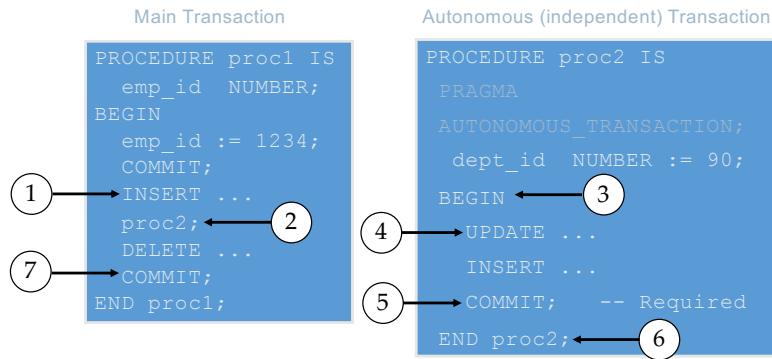
Names used in queries, data manipulation language (DML) statements, dynamic SQL, and DBMS_SQL are resolved in the invoker's schema.

All other statements, such as calls to packages, functions, and

procedures, are resolved in the definer's schema.

Autonomous Transactions

- Are independent transactions started by another main transaction
- Are specified with PRAGMA AUTONOMOUS_TRANSACTION



346

A transaction is a series of statements doing a logical unit of work that completes or fails as an integrated unit. Often, one transaction starts another that may need to operate outside the scope of the transaction that started it. That is, in an existing transaction, a required independent transaction may need to commit or roll back changes without affecting the outcome of the starting transaction. For example, in a stock purchase transaction, the customer's information must be committed regardless of whether the overall stock purchase completes. Or, while running that same transaction, you want to log messages to a table even if the overall transaction rolls back.

Since Oracle8i, autonomous transactions were added to make it possible to create an independent transaction. An autonomous transaction (AT) is an independent transaction started by another main transaction (MT). The slide depicts the behavior of an AT:

1. The main transaction begins.
2. A proc2 procedure is called to start the autonomous transaction.
3. The main transaction is suspended.
4. The autonomous transactional operation begins.

5. The autonomous transaction ends with a commit or roll back operation.
6. The main transaction is resumed.
7. The main transaction ends.

Features of Autonomous Transactions

- Are independent of the main transaction
- Suspend the calling transaction until the autonomous transactions are completed
- Are not nested transactions
- Do not roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are started and ended by individual subprograms and not by nested or anonymous PL/SQL blocks

347

Autonomous transactions exhibit the following features:

- Although called within a transaction, autonomous transactions are independent of that transaction. That is, they are not nested transactions.
- If the main transaction rolls back, autonomous transactions do not. Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits.
- With their stack-like functionality, only the “top” transaction is accessible at any given time. After completion, the autonomous transaction is popped, and the calling transaction is resumed.
- There are no limits other than resource limits on how many autonomous transactions can be recursively called.
- Autonomous transactions must be explicitly committed or rolled back; otherwise, an error is returned when attempting to return from the autonomous block.
- You cannot use PRAGMA to mark all subprograms in a package as autonomous. Only individual routines can be marked autonomous.
- You cannot mark a nested or anonymous PL/SQL block as autonomous.

Using Autonomous Transactions: Example

```
CREATE TABLE usage (card_id NUMBER, loc NUMBER)
/
CREATE TABLE txn (acc_id NUMBER, amount NUMBER)
/
CREATE OR REPLACE PROCEDURE log_usage (p_card_id NUMBER, p_loc NUMBER)
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO usage
        VALUES (p_card_id, p_loc);
    COMMIT;
END log_usage;
/
CREATE OR REPLACE PROCEDURE bank_trans(p_cardnbr NUMBER,p_loc NUMBER) IS
BEGIN
    INSERT INTO txn VALUES (9001, 1000);
    log_usage (p_cardnbr, p_loc);
END bank_trans;
/
EXECUTE bank_trans(50, 2000)
```

348

Using Autonomous Transactions

To define autonomous transactions, you use PRAGMA AUTONOMOUS_TRANSACTION. PRAGMA instructs the PL/SQL compiler to mark a routine as autonomous (independent). In this context, the term “routine” includes top-level (not nested) anonymous PL/SQL blocks; local, stand-alone, and packaged functions and procedures; methods of a SQL object type; and database triggers. You can code PRAGMA anywhere in the declarative section of a routine. However, for readability, it is best placed at the top of the Declaration section.

In the example in the slide, you track where the bankcard is used, regardless of whether the transaction is successful. The following are the benefits of autonomous transactions:

After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit dependencies with the main transaction, so you can log events, increment retry counters, and so on even if the main transaction rolls back.

More importantly, autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling

application need not know about a procedure's autonomous operations, and the procedure need not know about the application's transaction context. That makes autonomous transactions less error-prone than regular transactions and easier to use.

- Allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE      rec_emp_type IS TABLE OF employees%ROWTYPE;
  rec_emp  rec_emp_type;
  PROCEDURE populate(p_tab IN OUT NOCOPY empstabtype) IS
    BEGIN
      . . .
      END;
  BEGIN
    populate(rec_emp);
  END;
/
```

349

Note that PL/SQL subprograms support three parameter-passing modes: IN, OUT, and IN OUT. By default.

The IN parameter is passed by reference. A pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both the parameters reference the same memory location, which holds the value of the actual parameter.

The OUT and IN OUT parameters are passed by value. The value of the OUT or IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

Copying parameters that represent large data structures (such as collections, records, and instances of object types) with OUT and IN OUT parameters slows down execution and uses up memory. To prevent this overhead, you can specify the NOCOPY hint, which enables the PL/SQL compiler to pass the OUT and IN OUT parameters by reference.

The slide shows an example of declaring an IN OUT parameter with the NOCOPY hint.

- If the subprogram exits with an exception that is not handled:
 - You cannot rely on the values of the actual parameters passed to a NOCOPY parameter
 - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.

As a trade-off for better performance, the NOCOPY hint enables you to trade well-defined exception semantics for better performance. Its use affects exception handling in the following ways:

Because NOCOPY is a hint and not a directive, the compiler can pass NOCOPY parameters to a subprogram by value or by reference. So, if the subprogram exits with an unhandled exception, you cannot rely on the values of the NOCOPY actual parameters.

By default, if a subprogram exits with an unhandled exception, the values assigned to its OUT and IN OUT formal parameters are not copied to the corresponding actual parameters, and the changes appear to roll back. However, when you specify NOCOPY, assignments to the formal parameters immediately affect the actual parameters as well. So, if the subprogram exits with an unhandled exception, the (possibly unfinished) changes are not “rolled back.”

Currently, the RPC protocol enables you to pass parameters only by value. So, exception semantics can change without notification when you partition applications. For example, if you move a local procedure with NOCOPY parameters to a remote site, those parameters are no longer passed by reference.

The NOCOPY hint has no effect if:

- The actual parameter:
 - Is an element of associative arrays (index-by tables)
 - Is constrained (for example, by scale or NOT NULL)
 - And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ
 - Requires an implicit data type conversion
- The subprogram is involved in an external or remote procedure call

351

In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method (with no error generated).

The actual parameter is an element of associative arrays (index-by tables). This restriction does not apply to entire associative arrays.

The actual parameter is constrained (by scale or NOT NULL). This restriction does not extend to constrained elements or attributes. Also, it does not apply to size-constrained character strings.

The actual and formal parameters are records; one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.

The actual and formal parameters are records; the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.

Passing the actual parameter requires an implicit data type conversion.
The subprogram is involved in an external or remote procedure call.

Using the **PARALLEL_ENABLE** Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```

```
FUNCTION f2 Compiled.
```

352

The **PARALLEL_ENABLE** keyword can be used in the syntax for declaring a function. It is an optimization hint that indicates that the function can be used in a parallelized query or parallelized DML statement. Oracle's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement that is run in parallel can have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process. For DML statements, before Oracle8i, the parallelization optimization looked to see whether a function was noted as having all four of RNDS, WNDS, RNPS, and WNPS specified in a **PRAGMA RESTRICT_REFERENCES** declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a **CREATE FUNCTION** statement had their code implicitly examined to determine whether they were actually pure enough; parallelized execution might occur even though a **PRAGMA** cannot be specified on these functions.

The **PARALLEL_ENABLE** keyword is placed after the return value type in the declaration of the function, as shown in the example in the slide.

Note: The function should not use session state, such as package variables,

because those variables may not be shared among the parallel execution servers.

Using the Cross-Session PL/SQL Function Result Cache

- Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in cache.
- The function result cache is stored in a shared global area (SGA), making it available to any session that runs your application.
- Subsequent calls to the same function with the same parameters uses the result from cache.
- Performance and scalability are improved.
- This feature is used with functions that are called frequently and dependent on information that changes infrequently.

353

Starting in Oracle Database 19c, you can use the PL/SQL cross-session

function result caching mechanism. This caching mechanism provides you with a language-supported and system-managed means for storing the results of PL/SQL functions in a shared global area (SGA), which is available to every session that runs your application. The caching mechanism is both efficient and easy to use, and it relieves you of the burden of designing and developing your own caches and cache-management policies.

Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in the cache.

Subsequently, when the same function is called with the same parameter values, the result is retrieved from the cache, instead of being recomputed. If a database object that was used to compute a cached result is updated, the cached result becomes invalid and must be recomputed.

Use the result-caching feature with functions that are called frequently and are dependent on information that never changes or changes infrequently.

Note: For additional information about *Cross-Session PL/SQL Function Result Cache*, refer to the *Oracle Database 19c Advanced PL/SQL* course, the *Oracle Database 19c SQL and PL/SQL New Features* course, or *Oracle Database PL/SQL Language Reference 19c Release 2 (11.2)*.

Declaring and Defining a Result-Cached Function: Example

```
CREATE OR REPLACE FUNCTION emp_hire_date (p_emp_id  
NUMBER) RETURN VARCHAR  
RESULT CACHE RELIES ON (employees) IS  
    v_date_hired DATE;  
BEGIN  
    SELECT hire_date INTO v_date_hired  
    FROM HR.Employees  
    WHERE Employee_ID = p_emp_ID;  
    RETURN to_char(v_date_hired);  
END;
```

```
FUNCTION emp_hire_date Compiled.
```

354

If a function depends on settings that might vary from session to session such as `NLS_DATE_FORMAT` and `TIME_ZONE`, make the function result-cached only if you can modify it to handle the various settings.

In the example in the slide, the `emp_hire_date` function uses the `to_char` function to convert a `DATE` item to a `VARCHAR` item.

`emp_hire_date` does not specify a format mask, so the format mask defaults to the one that `NLS_DATE_FORMAT` specifies. If sessions that call `emp_hire_date` have different `NLS_DATE_FORMAT` settings, cached results can have different formats. If a cached result computed by one session ages out, and another session recomputes it, the format might vary even for the same parameter value. If a session gets a cached result whose format differs from its own format, that result will probably be incorrect.

Some possible solutions to this problem are:

Change the return type of `emp_hire_date` to `DATE` and have each session call the `to_char` function.

If a common format is acceptable to all sessions, specify a format mask, removing the dependency on `NLS_DATE_FORMAT`—for example, `to_char(date_hired, 'mm/dd/yy');`.



Using the DETERMINISTIC Clause with Functions

- Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its arguments.
- This helps the optimizer avoid redundant function calls.
- If a function was called previously with the same arguments, the optimizer can elect to use the previous result.
- Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects.

355

You can use the DETERMINISTIC function clause to indicate that the function returns the same result value whenever it is called with the same values for its arguments.

You must specify this keyword if you intend to call the function in the expression of a function-based index or from the query of a materialized view that is marked REFRESH FAST or ENABLE QUERY REWRITE. When Oracle Database encounters a deterministic function in one of these contexts, it attempts to use previously calculated results when possible rather than execute the function again. If you subsequently change the semantics of the function, you must manually rebuild all dependent function-based indexes and materialized views.

Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the return result of the function. The results of doing so will not be captured if Oracle Database chooses not to re-execute the function.

Note

Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects because results might vary across calls. Instead, consider making the function result-

cached.

For more information about the DETERMINISTIC clause, refer to the *Oracle Database SQL Language Reference*.

Using the **RETURNING** Clause

- Improves performance by returning column values with INSERT, UPDATE, and DELETE statements
- Eliminates the need for a SELECT statement

```

CREATE OR REPLACE PROCEDURE update_salary(p_emp_id
    NUMBER) IS
    v_name      employees.last_name%TYPE;
    v_new_sal   employees.salary%TYPE;
BEGIN
    UPDATE employees
        SET salary = salary * 1.1
    WHERE employee_id = p_emp_id
    RETURNING last_name, salary INTO v_name, v_new_sal;
    DBMS_OUTPUT.PUT_LINE(v_name || ' new salary is ' ||
        v_new_sal);
END update_salary;
/

```

356

Often, applications need information about the row affected by a SQL operation—for example, to generate a report or to take a subsequent action.

The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to SELECT the row after an INSERT or UPDATE, or before a DELETE. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

The example in the slide shows how to update the salary of an employee and, at the same time, retrieve the employee's last name and new salary into a local PL/SQL variable. To test the code example, issue the following commands that check the salary of employee_id 108 before updating it. Next, the procedure is invoked passing the employee_id 108 as the parameter.

```

1 SET SERVEROUTPUT ON
2 /
3 SELECT last_name, salary
4 FROM employees
5 WHERE employee_id = 108;
6 /
7 EXECUTE update_salary(108)

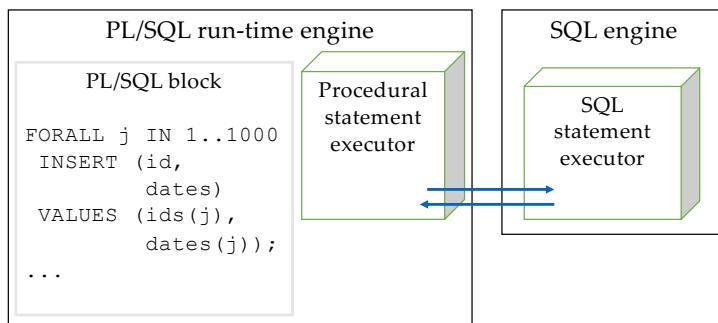
```

LAST_NAME	SALARY
Greenberg	12000

1 rows selected
anonymous block completed
Greenberg new salary is 13200

Using Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



357

The Oracle server uses two engines to run PL/SQL blocks and subprograms:

The PL/SQL run-time engine, which runs procedural statements but passes the SQL statements to the SQL engine

The SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine

During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty for excessive amounts of SQL processing. This is typical of applications that have a SQL statement in a loop that uses values from indexed collections. Collections include nested tables, varying arrays, index-by tables, and host arrays.

Performance can be substantially improved by minimizing the number of context switches through the use of bulk binding. Bulk binding causes an entire collection to be bound in one call, a context switch, to the SQL engine.

That is, a bulk bind process passes the entire collection of values back and forth between the two engines in a single context switch, compared with incurring a context switch for each collection element in an iteration of a loop. The more rows affected by a SQL statement, the greater the performance gain with bulk binding.

Bulk Binding: Syntax and Keywords

- The FORALL keyword instructs the *PL/SQL engine* to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound  
[SAVE EXCEPTIONS]  
sql_statement;
```

- The BULK COLLECT keyword instructs the *SQL engine* to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO  
collection_name[,collection_name] ...
```

358

Use bulk binds to improve the performance of:

DML statements that reference collection elements

•SELECT statements that reference collection elements

Cursor FOR loops that reference collections and the RETURNING INTO clause

The FORALL keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is not a FOR loop.

The BULK COLLECT keyword instructs the SQL engine to bulk bind output collections, before returning them to the PL/SQL engine. This enables you to bind locations into which SQL can return the retrieved values in bulk. Thus, you can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses.

The SAVE EXCEPTIONS keyword is optional. However, if some of the DML operations succeed and some fail, you would want to track or report on those that fail. Using the SAVE EXCEPTIONS keyword causes failed operations to be stored in a cursor attribute called %BULK_EXCEPTIONS, which is a collection of records indicating the bulk DML iteration number and corresponding error code.

Bulk Binding FORALL: Example

```
CREATE PROCEDURE raise_salary(p_percent NUMBER) IS
    TYPE numlist_type IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;
    v_id  numlist_type; -- collection
BEGIN
    v_id(1):= 100; v_id(2):= 102; v_id(3):= 104; v_id(4) := 110;
    -- bulk-bind the PL/SQL table
    FORALL i IN v_id.FIRST .. v_id.LAST
        UPDATE employees
            SET salary = (1 + p_percent/100) * salary
            WHERE employee_id = v_id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

```
anonymous block completed
```

359

Note: Before you can run the example in the slide, you must disable the update_job_history trigger as follows:

```
ALTER TRIGGER update_job_history DISABLE;
```

In the example in the slide, the PL/SQL block increases the salary for employees with IDs 100, 102, 104, or 110. It uses the `FORALL` keyword to bulk bind the collection. Without bulk binding, the PL/SQL block would have sent a SQL statement to the SQL engine for each employee record that is updated. If there are many employee records to update, the large number of context switches between the PL/SQL engine and the SQL engine can affect performance. However, the `FORALL` keyword bulk binds the collection to improve performance.

Note: A looping construct is no longer required when using this feature.

The SELECT statement supports the BULK COLLECT INTO syntax.

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
  TYPE dept_tab_type IS
    TABLE OF departments%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  SELECT * BULK COLLECT INTO v_depts
  FROM departments
  WHERE location_id = p_loc;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
      ||' '|| v_depts(i).department_name);
  END LOOP;
END;
```

360

Starting with Oracle Database 10g, when using a SELECT statement in PL/SQL, you can use the bulk collection syntax shown in the example in the slide. Thus, you can quickly obtain a set of rows without using a cursor mechanism.

The example reads all the department rows for a specified region into a PL/SQL table, whose contents are displayed with the FOR loop that follows the SELECT statement.

The `FETCH` statement has been enhanced to support the `BULK COLLECT INTO` syntax.

```
CREATE OR REPLACE PROCEDURE get_departments(p_loc NUMBER) IS
  CURSOR cur_dept IS
    SELECT * FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tab_type IS TABLE OF cur_dept%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  OPEN cur_dept;
  FETCH cur_dept BULK COLLECT INTO v_depts;
  CLOSE cur_dept;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
      ||'-'|| v_depts(i).department_name);
  END LOOP;
END;
```

361

In Oracle Database 10g, when using cursors in PL/SQL, you can use a form of the `FETCH` statement that supports the bulk collection syntax shown in the example in the slide.

This example shows how `BULK COLLECT INTO` can be used with cursors. You can also add a `LIMIT` clause to control the number of rows fetched in each operation. The code example in the slide could be modified as follows:

```
CREATE OR REPLACE PROCEDURE
  get_departments(p_loc NUMBER,
  p_nrows NUMBER) IS
  CURSOR dept_csr IS SELECT *
    FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts LIMIT
  nrows;
  CLOSE dept_csr;
```

```
DBMS_OUTPUT.PUT_LINE(depts.COUNT||' rows  
read');  
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

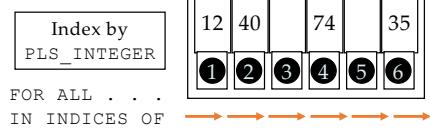
```
CREATE OR REPLACE PROCEDURE raise_salary(p_rate NUMBER)
IS
TYPE emplist_type IS TABLE OF NUMBER;
TYPE numlist_type IS TABLE OF employees.salary%TYPE
INDEX BY BINARY_INTEGER;
v_emp_ids emplist_type :=
emplist_type(100,101,102,104);
v_new_sals numlist_type;
BEGIN
FORALL i IN v_emp_ids.FIRST .. v_emp_ids.LAST
UPDATE employees
SET commission_pct = p_rate * salary
WHERE employee_id = v_emp_ids(i)
RETURNING salary BULK COLLECT INTO v_new_sals;
FOR i IN 1 .. v_new_sals.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(v_new_sals(i));
END LOOP;
END;
```

362

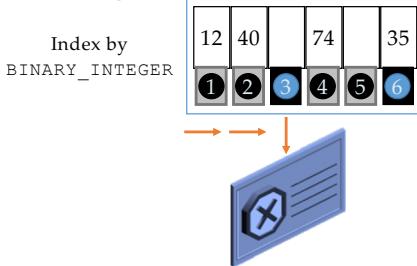
Bulk binds can be used to improve the performance of FOR loops that reference collections and return DML. If you have, or plan to have, PL/SQL code that does this, then you can use the FORALL keyword along with the RETURNING and BULK COLLECT INTO keywords to improve performance. In the example shown in the slide, the salary information is retrieved from the EMPLOYEES table and collected into the new_sals array. The new_sals collection is returned in bulk to the PL/SQL engine. The example in the slide shows an incomplete FOR loop that is used to iterate through the new salary data received from the UPDATE operation and then process the results.

Using Bulk Binds in Sparse Collections

- Current releases



- Prior to 10g:



363

FORALL Support for Sparse Collections

In earlier releases, PL/SQL did not allow sparse collections to be used with the `FORALL` statement.

If the `SAVE EXCEPTIONS` keyword was not specified, the statement was terminated when the first deleted element was encountered. Even when `SAVE EXCEPTION` was used, the PL/SQL engine tried to iterate over all elements (existing and non-existing). This substantially degraded the performance of the DML operation if the relative percentage of the deleted elements was high.

When you use the keyword `INDICES` (available with Oracle Database 10g and later), you can successfully loop over a sparse collection with the `FORALL` statement. This syntax binds sparse collections more efficiently and it also supports a more general approach where an index array can be specified to iterate over the collections.

Using sparse collection and index arrays in bulk operations improves performance.

Using Bulk Binds in Sparse Collections

```
-- The INDICES OF syntax allows the bound arrays
-- themselves to be sparse.

FORALL index_name IN INDICES OF sparse_array_name
    BETWEEN LOWER_BOUND AND UPPER_BOUND -- optional
    SAVE EXCEPTIONS -- optional, but recommended
        INSERT INTO table_name VALUES
            sparse_array(index_name);
. . .
```

```
-- The VALUES OF syntax lets you indicate a subset
-- of the binding arrays.

FORALL index_name IN VALUES OF index_array_name
    SAVE EXCEPTIONS -- optional, but recommended
        INSERT INTO table_name VALUES
            binding_array_name(index_name);
. . .
```

364

FORALL Support for Sparse Collections

You can use the INDICES OF and VALUES OF syntax with the FORALL statement.

The bulk bind for sparse array syntax can be used in all DML syntaxes.

In the syntax, the index array must be dense, and the binding arrays may be dense or sparse and the indicated elements must exist.

The typical application for this feature is an order entry and order processing system where:

- Users enter orders through the Web
- Orders are placed in a staging table before validation
- Data is later validated based on complex business rules (usually implemented programmatically using PL/SQL)
- Invalid orders are separated from valid ones
- Valid orders are inserted into a permanent table for processing

This feature can be used in any application that starts with a dense PL/SQL table or records or table of scalar that are populated using a bulk collect. This is used as the binding array. A dense array (pointer), whose elements denote the indices of the binding array, is made sparse based on the application logic. This pointer array is then used in the FORALL statement to perform bulk DML with the binding arrays. Any exceptions encountered can be saved and further processed in the exception-handling section, perhaps by using another FORALL statement.

Using Bulk Bind with Index Array

```
CREATE OR REPLACE PROCEDURE ins_emp2 AS
  TYPE emptab_type IS TABLE OF employees%ROWTYPE;
  v_emp emptab_type;
  TYPE values_of_tab_type IS TABLE OF PLS_INTEGER
    INDEX BY PLS_INTEGER;
  v_num values_of_tab_type;
  .
  .
  .
BEGIN
  .
  .
  .
  FORALL k IN VALUES OF v_num
    INSERT INTO new_employees VALUES v_emp(k);
END;
```

366

You can use an index collection of PLS_INTEGER or BINARY_INTEGER (or one of its subtypes) whose values are the indexes of the collections involved in the bulk-bind DML operation using FORALL. These index collections can then be used in a FORALL statement to process bulk DML using the VALUES OF clause.

In the example shown above, V_NUM is a collection whose type is PLS_INTEGER. In the example, you are creating a procedure INS_EMP2, which identifies only one employee for each occurrence of the first letter of the last name. This procedure then inserts into the NEW_EMPLOYEES table created earlier using the FORALL..IN VALUES OF syntax.

Topics covered in this module:

- Create standard constants and exceptions
- Write and call local subprograms
- Control the run-time privileges of a subprogram
- Perform autonomous transactions
- Pass parameters by reference using a NOCOPY hint
- Use the PARALLEL ENABLE hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the DETERMINISTIC clause with functions
- Use the RETURNING clause and bulk binding with DML

367

The lesson provides insights into managing your PL/SQL code by defining constants and exceptions in a package specification. This enables a high degree of reuse and standardization of code.

Local subprograms can be used to simplify and modularize a block of code where the subprogram functionality is repeatedly used in the local block.

The run-time security privileges of a subprogram can be altered by using definer's or invoker's rights.

Autonomous transactions can be executed without affecting an existing main transaction.

You should understand how to obtain performance gains by using the NOCOPY hint, bulk binding and the RETURNING clauses in SQL statements, and the PARALLEL_ENABLE hint for optimization of functions.



Oracle PL/SQL 19c



Module 09: Creating Triggers

Topics covered in this module:

- Describe database triggers and their uses
- Describe the different types of triggers
- Create database triggers
- Describe database trigger-firing rules
- Remove database triggers
- Display trigger information

- A trigger is a PL/SQL block that is stored in the database and fired (executed) in response to a specified event.
- The Oracle database automatically executes a trigger when specified conditions occur.



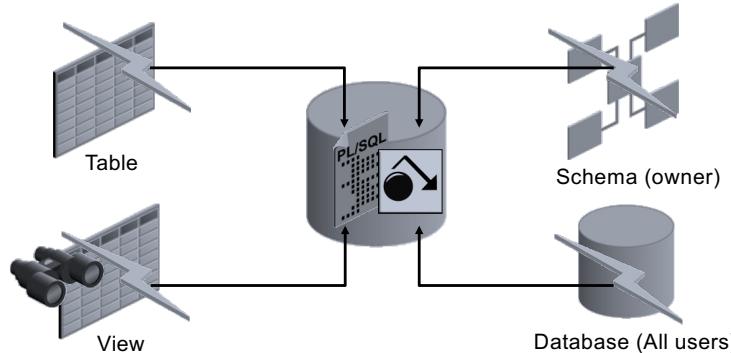
370

Working with Triggers: Overview

Triggers are similar to stored procedures. A trigger stored in the database contains PL/SQL in the form of an anonymous block, a call statement, or a compound trigger block. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger. Triggers are implicitly fired by the Oracle database when a triggering event occurs, no matter which user is connected or which application is being used.

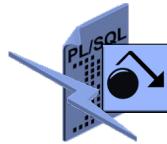
Defining Triggers

A trigger can be defined on the table, view, schema (schema owner), or database (all users).



You can write triggers that fire whenever one of the following operations occurs in the database:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation such as SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.



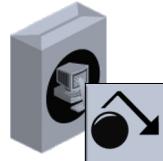
372

Triggering Event or Statement

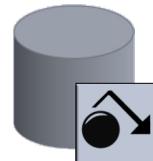
A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:

- An INSERT, UPDATE, or DELETE statement on a specific table (or view, in some cases)
- A CREATE, ALTER, or DROP statement on any schema object
- A database startup or instance shutdown
- A specific error message or any error message
- A user logon or logoff

- Database trigger (covered in this course):
 - Fires whenever a DML, a DLL, or system event occurs on a schema or database
- Application trigger:
 - Fires whenever an event occurs within a particular application



Application Trigger



Database Trigger

373

Types of Triggers

Application triggers execute implicitly whenever a particular data manipulation language (DML) event occurs within an application. An example of an application that uses triggers extensively is an application developed with Oracle Forms Developer.

Database triggers execute implicitly when any of the following events occur:

- DML operations on a table
- DML operations on a view, with an INSTEAD OF trigger
- DDL statements, such as CREATE and ALTER

This is the case no matter which user is connected or which application is used. Database triggers also execute implicitly when some user actions or database system actions occur (for example, when a user logs on or the DBA shuts down the database).

Database triggers can be system triggers on a database or a schema (covered in the next lesson). For databases, triggers fire for each event for all users; for a schema, they fire for each event for that specific user. Oracle Forms can define, store, and run triggers of a different sort. However, do not confuse Oracle Forms triggers with the triggers discussed in this lesson.

You can use triggers for:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

374

Develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server or as alternatives to those provided by the Oracle server.

Security: The Oracle server allows table access to users or roles.

Triggers allow table access according to data values.

Auditing: The Oracle server tracks data operations on tables. Triggers track values for data operations on tables.

Data integrity: The Oracle server enforces integrity constraints. Triggers implement complex integrity rules.

Referential integrity: The Oracle server enforces standard referential integrity rules. Triggers implement nonstandard functionality.

Table replication: The Oracle server copies tables asynchronously into snapshots. Triggers copy tables synchronously into replicas.

Derived data: The Oracle server computes derived data values manually. Triggers compute derived data values automatically.

Event logging: The Oracle server logs events explicitly. Triggers log events transparently.



Available Trigger Types

- Simple DML triggers
 - BEFORE
 - AFTER
 - INSTEAD OF
- Compound triggers
- Non-DML triggers
 - DDL event triggers
 - Database event triggers

375

Note

In this lesson, we will discuss the BEFORE, AFTER, and INSTEAD OF triggers.

The other trigger types are discussed in the lesson titled “Creating Compound, DDL, and Event Database Triggers.”

- A trigger event type determines which DML statement causes the trigger to execute. The possible events are:
 - INSERT
 - UPDATE [OF column]
 - DELETE
- A trigger body determines what action is performed and is a PL/SQL block or a CALL to a procedure.

Triggering Event Types

The triggering event or statement can be an INSERT, UPDATE, or DELETE statement on a table.

When the triggering event is an UPDATE statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement because it always affects entire rows.

... UPDATE OF salary ...

The triggering event can contain one, two, or all three of these DML operations.

... INSERT or UPDATE or DELETE

... INSERT or UPDATE OF job_id ...

The trigger body defines the action—that is, what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Note: The size of a trigger cannot be greater than 32 KB.

Creating DML Triggers Using the CREATE TRIGGER Statement

```
CREATE [OR REPLACE] TRIGGER trigger_name  
timing -- when to fire the trigger  
event1 [OR event2 OR event3]  
ON object_name  
[REFERENCING OLD AS old | NEW AS new]  
FOR EACH ROW -- default is statement level trigger  
WHEN (condition)]  
DECLARE]  
BEGIN  
... trigger_body -- executable statements  
[EXCEPTION . . .]  
END [trigger_name];
```

```
timing = BEFORE | AFTER | INSTEAD OF
```

```
event = INSERT | DELETE | UPDATE | UPDATE OF column_list
```

377

Creating DML Triggers

The components of the trigger syntax are:

- *trigger_name* uniquely identifies the trigger.
- *timing* indicates when the trigger fires in relation to the triggering event. Values are BEFORE, AFTER, and INSTEAD OF.
- *event* identifies the DML operation causing the trigger to fire. Values are INSERT, UPDATE [OF column], and DELETE.
- *object_name* indicates the table or view associated with the trigger.

For row triggers, you can specify:

A REFERENCING clause to choose correlation names for referencing the old and new values of the current row (default values are OLD and NEW)

FOR EACH ROW to designate that the trigger is a row trigger
A WHEN clause to apply a conditional predicate, in parentheses, which is evaluated for each row to determine whether or not to execute the trigger body

You can specify the trigger timing as to whether to run the trigger's action before or after the triggering statement:

- BEFORE: Execute the trigger body before the triggering DML event on a table.
- AFTER: Execute the trigger body after the triggering DML event on a table.
- INSTEAD OF: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

Trigger Timing

The BEFORE trigger timing is frequently used in the following situations.

- To determine whether the triggering statement should be allowed to complete (This eliminates unnecessary processing and enables a rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing an INSERT or UPDATE statement
- To initialize global variables or flags, and to validate complex business rules

The AFTER triggers are frequently used in the following situations:

- To complete the triggering statement before executing the triggering action
- To perform different actions on the same triggering statement if a BEFORE trigger is already present

The INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because a view is not always modifiable. You can write appropriate DML statements inside the body of an INSTEAD OF trigger to perform actions directly on the underlying tables of views.

If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order you intend. If two or more triggers are defined with the same timing point, and the order in which they fire is important, then you can control the firing order using the `FOLLOWS` and `PRECEDES` clauses.

Statement-Level Triggers Versus Row-Level Triggers

Statement-Level Triggers	Row-Level Triggers
Is the default when creating a trigger	Use the FOR EACH ROW clause when creating a trigger.
Fires once for the triggering event	Fires once for each row affected by the triggering event
Fires once even if no rows are affected	Does not fire if the triggering event does not affect any rows

379

Types of DML Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple row UPDATE) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself (for example, a trigger that performs a complex security check on the current user).

Row Trigger

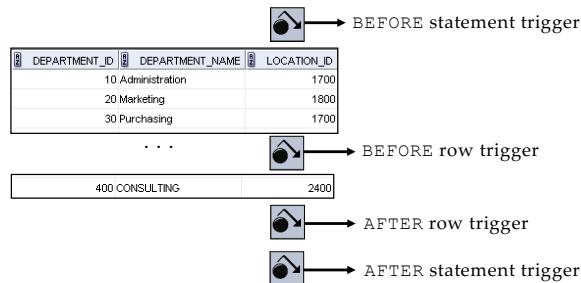
A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed. Row triggers are useful if the trigger action depends on data of the rows that are affected or on data provided by the triggering event itself.

Note: Row triggers use correlation names to access the old and new column values of the row being processed by the trigger.

Trigger-Firing Sequence:Single-Row Manipulation

Use the following firing sequence for a trigger on a table when a single row is manipulated:

```
INSERT INTO departments
(department_id, department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```



380

Create a statement trigger or a row trigger based on the requirement that the trigger must fire once for each row affected by the triggering statement, or just once for the triggering statement, regardless of the number of rows affected.

When the triggering DML statement affects a single row, both the statement trigger and the row trigger fire exactly once.

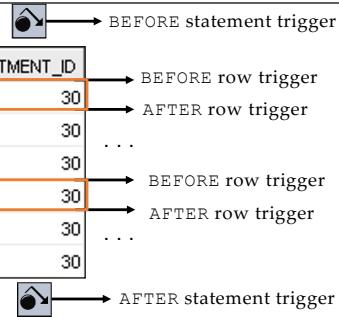
Example

The SQL statement in the slide does not differentiate statement triggers from row triggers because exactly one row is inserted into the table using the syntax for the `INSERT` statement shown in the slide.

Trigger-Firing Sequence: Multirow Manipulation

Use the following firing sequence for a trigger on a table when many rows are manipulated.

```
UPDATE employees  
SET salary = salary * 1.1  
WHERE department_id = 30;
```



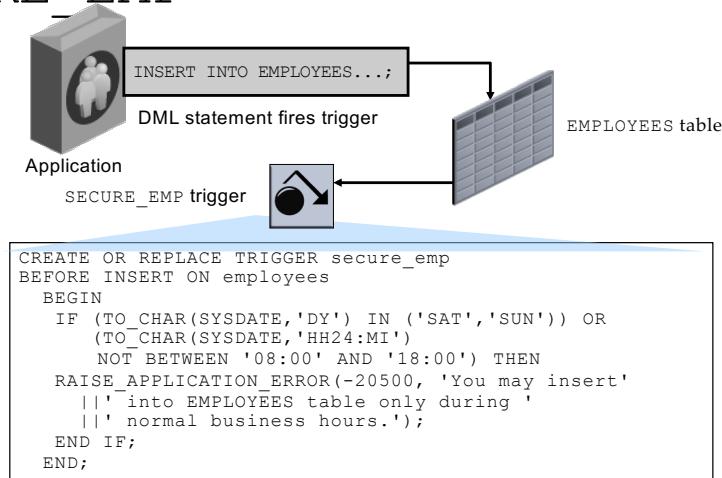
381

When the triggering DML statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Example

The SQL statement in the slide causes a row-level trigger to fire a number of times equal to the number of rows that satisfy the WHERE clause (that is, the number of employees reporting to department 30).

Creating a DML Statement Trigger Example: SECURE_EMP



382

Creating a DML Statement Trigger

In the example in the slide, the `SECURE_EMP` database trigger is a BEFORE statement trigger that prevents the `INSERT` operation from succeeding if the business condition is violated. In this case, the trigger restricts inserts into the `EMPLOYEES` table during certain business hours, Monday through Friday.

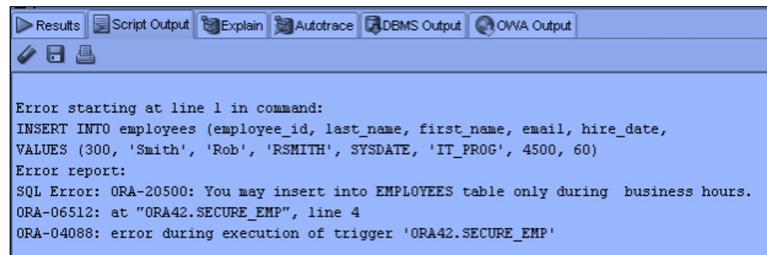
If a user attempts to insert a row into the `EMPLOYEES` table on Saturday, then the user sees an error message, the trigger fails, and the triggering statement is rolled back. Remember that the

`RAISE_APPLICATION_ERROR` is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server.

Testing Trigger SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
    first_name, email, hire_date, job_id, salary,
    department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
    'IT_PROG', 4500, 60);
```



The screenshot shows the Oracle SQL Developer interface. The top navigation bar includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the navigation bar is a toolbar with icons for New Connection, Run, Stop, and Refresh. The main area displays an error message:

```
Error starting at line 1 in command:
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date,
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60)
Error report:
SQL Error: ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "ORA42.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'ORA42.SECURE_EMP'
```

383

Testing SECURE_EMP

To test the trigger, insert a row into the EMPLOYEES table during nonbusiness hours. When the date and time are out of the business hours specified in the trigger, you receive the error message shown in the slide.

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees
BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24') NOT BETWEEN '08' AND '18') THEN
    IF DELETING THEN RAISE_APPLICATION_ERROR(
      -20502,'You may delete from EMPLOYEES table'|||
      'only during normal business hours.');
    ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
      -20500,'You may insert into EMPLOYEES table'|||
      'only during normal business hours.');
    ELSIF UPDATING ('SALARY') THEN
      RAISE_APPLICATION_ERROR(-20503, 'You may '|||
      'update SALARY only normal during business hours.');
    ELSE RAISE_APPLICATION_ERROR(-20504,'You may'|||
      ' update EMPLOYEES table only during'|||
      ' normal business hours.');
    END IF;
  END IF;
END;
```

384

Detecting the DML Operation That Fired a Trigger

If more than one type of DML operation can fire a trigger (for example, ON INSERT OR DELETE OR UPDATE OF Emp_tab), the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to check which type of statement fired the trigger.

You can combine several triggering events into one by taking advantage of the special conditional predicates INSERTING, UPDATING, and DELETING within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the EMPLOYEES table to certain business hours, 8 AM to 6 PM, Monday through Friday.

Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
        AND :NEW.salary > 15000 THEN
        RAISE_APPLICATION_ERROR (-20202,
            'Employee cannot earn more than $15,000.');
    END IF;
END;

UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';

Error starting at line 1 in command:
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell'
Error report:
SQL Error: ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "ORA62.RESTRICT_SALARY", line 4
ORA-04086: error during execution of trigger 'ORA62.RESTRICT_SALARY'
```

385

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

In the first example in the slide, a trigger is created to allow only employees whose job IDs are either AD_PRES or AD_VP to earn a salary of more than 15,000. If you try to update the salary of employee Russell whose employee ID is SA_MAN, the trigger raises the exception displayed in the slide.

Note: Before executing the first code example in the slide, make sure you disable the secure_emp and secure_employees triggers.

- When a row-level trigger fires, the PL/SQL run-time engine creates and populates two data structures:
 - *OLD*: Stores the original values of the record processed by the trigger
 - *NEW*: Contains the new values
- *NEW* and *OLD* have the same structure as a record declared using the `%ROWTYPE` on the table to which the trigger is attached.

Data Operations	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

386

Within a `ROW` trigger, you can reference the value of a column before and after the data change by prefixing it with the `OLD` and `NEW` qualifiers.

Note

The `OLD` and `NEW` qualifiers are available only in `ROW` triggers. Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement. There is no colon (:) prefix if the qualifiers are referenced in the `WHEN` restricting condition. Row triggers can decrease the performance if you perform many updates on larger tables.

Using OLD and NEW Qualifiers: Example

```
CREATE TABLE audit_emp (
    user_name      VARCHAR2(30),
    time_stamp     date,
    id             NUMBER(6),
    old_last_name VARCHAR2(25),
    new_last_name VARCHAR2(25),
    old_title      VARCHAR2(10),
    new_title      VARCHAR2(10),
    old_salary     NUMBER(8,2),
    new_salary     NUMBER(8,2) )
/
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp(user_name, time_stamp, id,
        old_last_name, new_last_name, old_title,
        new_title, old_salary, new_salary)
    VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary);
END;
```

387

In the example in the slide, the AUDIT_EMP_VALUES trigger is created on the EMPLOYEES table. The trigger adds rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

Using OLD and NEW Qualifiers: Example

```

INSERT INTO employees (employee_id, last_name, job_id,
salary, email, hire_date)
VALUES (999, 'Temp emp', 'SA_REP', 6000, 'TEMPEMP',
TRUNC(SYSDATE))
/
UPDATE employees
SET salary = 7000, last_name = 'Smith'
WHERE employee_id = 999
/
SELECT *
FROM audit_emp;

```

Results									
	USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
1	ORA61	04-JUN-09	(null)	Temp emp	Smith	SA_REP	SA_REP	6000	6000
2	ORA61	04-JUN-09	999 Temp emp	Smith	Smith	SA_REP	SA_REP	6000	7000

388

Using OLD and NEW Qualifiers: Example the Using AUDIT_EMP Table

Create a trigger on the EMPLOYEES table to add rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

The following is the result of inserting the employee record into the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
999 (null)	Smith	TEMPEMP	(null)	04-JUN-09	SA_REP	7000	(null)	(null)	(null)	(null)
300 Rob	Smith	RSMITH	(null)	04-JUN-09	IT_PROG	4500	(null)	(null)	(null)	60
206 William	Gietz	WGETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300	(null)	(null)	205	110

...

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
999 (null)	Smith	TEMPEMP	(null)	04-JUN-09	SA_REP	7000	(null)	(null)	(null)	(null)

Using the WHEN Clause to Fire a Row Trigger Based on a Condition

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
    IF INSERTING THEN
        :NEW.commission_pct := 0;
    ELSIF :OLD.commission_pct IS NULL THEN
        :NEW.commission_pct := 0;
    ELSE
        :NEW.commission_pct := :OLD.commission_pct+0.05;
    END IF;
END;
/
```

389

Restricting a Row Trigger: Example

Optionally, you can include a trigger restriction in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause. If you include a WHEN clause in the trigger, then the expression in the WHEN clause is evaluated for each row that the trigger affects.

If the expression evaluates to TRUE for a row, then the trigger body executes on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE for a row (unknown, as with nulls), then the trigger body does not execute for that row. The evaluation of the WHEN clause does not have an effect on the execution of the triggering SQL statement (in other words, the triggering statement is not rolled back if the expression in a WHEN clause evaluates to FALSE).

Note: A WHEN clause cannot be included in the definition of a statement trigger.

In the example in the slide, a trigger is created on the EMPLOYEES table to calculate an employee's commission when a row is added to the EMPLOYEES table, or when an employee's salary is modified.

The NEW qualifier cannot be prefixed with a colon in the WHEN clause because the WHEN clause is outside the PL/SQL blocks.

1. Execute all BEFORE STATEMENT triggers.
2. Loop for each row affected by the SQL statement:
 - a. Execute all BEFORE ROW triggers for that row.
 - b. Execute the DML statement and perform integrity constraint checking for that row.
 - c. Execute all AFTER ROW triggers for that row.
3. Execute all AFTER STATEMENT triggers.

390

Trigger Execution Model

A single DML statement can potentially fire up to four types of triggers.

- BEFORE and AFTER statement triggers
- BEFORE and AFTER row triggers

A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. However, you can defer constraint checking until a COMMIT operation is performed.

Triggers can also cause other triggers—known as cascading triggers—to fire. All actions and checks performed as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, then all actions performed because of the original SQL statement are rolled back (including actions performed by firing triggers). This guarantees that integrity constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may undergo changes by other users' transactions. In all cases, a read-consistent image is guaranteed for the modified values that the trigger needs to read (query) or write (update).

Note: Integrity checking can be deferred until the COMMIT operation is performed.

Implementing an Integrity Constraint with an After Trigger

```
-- Integrity constraint violation error -2291 raised.  
UPDATE employees SET department_id = 999  
WHERE employee_id = 170;
```

```
CREATE OR REPLACE TRIGGER employee_dept_fk_trg  
AFTER UPDATE OF department_id ON employees  
FOR EACH ROW  
BEGIN  
    INSERT INTO departments VALUES (:new.department_id,  
                                    'Dept '||:new.department_id, NULL, NULL);  
EXCEPTION  
    WHEN DUP_VAL_ON_INDEX THEN  
        NULL; -- mask exception if department exists  
END;  
/
```

```
-- Successful after trigger is fired  
UPDATE employees SET department_id = 999  
WHERE employee_id = 170;
```

```
1 rows updated
```

391

The example in the slide explains a situation in which the integrity constraint can be taken care of by using an AFTER trigger. The EMPLOYEES table has a foreign key constraint on the DEPARTMENT_ID column of the DEPARTMENTS table.

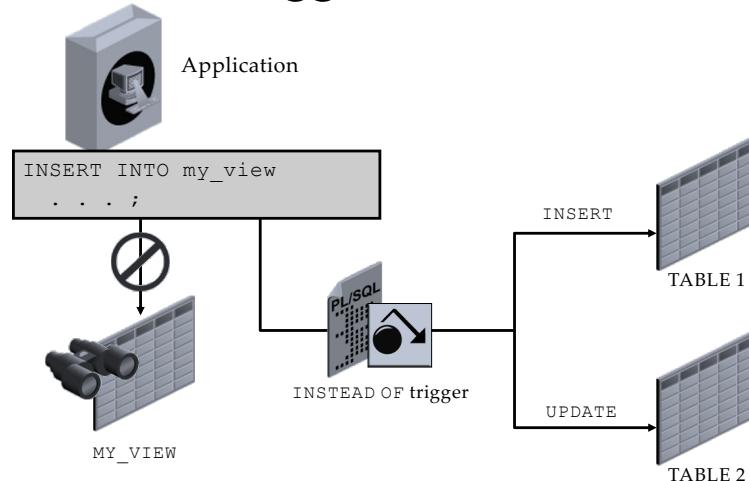
In the first SQL statement, the DEPARTMENT_ID of the employee 170 is modified to 999. Because department 999 does not exist in the DEPARTMENTS table, the statement raises exception –2291 for the integrity constraint violation.

The EMPLOYEE_DEPT_FK_TRG trigger is created and it inserts a new row into the DEPARTMENTS table by using : NEW.DEPARTMENT_ID for the value of the new department's DEPARTMENT_ID. The trigger fires when the UPDATE statement modifies the DEPARTMENT_ID of employee 170 to 999. When the foreign key constraint is checked, it is successful because the trigger inserted the department 999 into the DEPARTMENTS table. Therefore, no exception occurs unless the department already exists when the trigger attempts to insert the new row. However, the EXCEPTION handler traps and masks the exception allowing the operation to succeed.

Note: Although the example shown in the slide is somewhat contrived due to the limited data in the HR schema, the point is that if you defer the constraint

check until the commit, you then have the ability to engineer a trigger to detect that constraint failure and repair it prior to the commit action.

INSTEAD OF Triggers



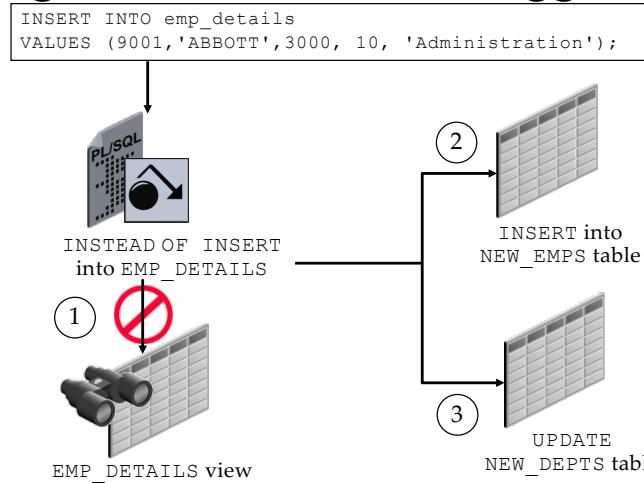
392

Use `INSTEAD OF` triggers to modify data in which the DML statement has been issued against an inherently un-updatable view. These triggers are called `INSTEAD OF` triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. These triggers are used to perform `INSERT`, `UPDATE`, and `DELETE` operations directly on the underlying tables. You can write `INSERT`, `UPDATE`, and `DELETE` statements against a view, and the `INSTEAD OF` trigger works invisibly in the background to make the right actions take place. A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as `GROUP BY`, `CONNECT BY`, `START`, the `DISTINCT` operator, or joins. For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So you write an `INSTEAD OF` trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatable and has `INSTEAD OF` triggers, then the triggers take precedence. `INSTEAD OF` triggers are row triggers. The `CHECK` option for views is not enforced when insertions or updates to the view are performed by using `INSTEAD OF` triggers. The `INSTEAD OF` trigger

body must enforce the check.

Creating an INSTEAD OF Trigger: Example



393

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based.

The example in the slide illustrates an employee being inserted into the view `EMP_DETAILS`, whose query is based on the `EMPLOYEES` and `DEPARTMENTS` tables. The `NEW_EMP_DEPT` (INSTEAD OF) trigger executes in place of the `INSERT` operation that causes the trigger to fire. The INSTEAD OF trigger then issues the appropriate `INSERT` and `UPDATE` to the base tables used by the `EMP_DETAILS` view. Therefore, instead of inserting the new employee record into the `EMPLOYEES` table, the following actions take place:

1. The `NEW_EMP_DEPT` INSTEAD OF trigger fires.
2. A row is inserted into the `NEW_EMPS` table.
3. The `DEPT_SAL` column of the `NEW_DEPTS` table is updated. The salary value supplied for the new employee is added to the existing total salary of the department to which the new employee has been assigned.

Note: Before you run the example in the slide, you must create the required structures shown on the next two pages.

Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```
CREATE TABLE new_emps AS
  SELECT employee_id, last_name, salary, department_id
    FROM employees;

CREATE TABLE new_depts AS
  SELECT d.department_id, d.department_name,
         sum(e.salary) dept_sal
    FROM employees e, departments d
   WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
  SELECT e.employee_id, e.last_name, e.salary,
         e.department_id, d.department_name
    FROM employees e, departments d
   WHERE e.department_id = d.department_id
  GROUP BY d.department_id, d.department_name;
```

394

The example in the slide creates two new tables, NEW_EMPS and NEW_DEPTS, that are based on the EMPLOYEES and DEPARTMENTS tables, respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables.

If a view has a complex query structure, then it is not always possible to perform DML directly on the view to affect the underlying tables. The example requires creation of an INSTEAD OF trigger, called NEW_EMP_DEPT, shown on the next page. The NEW_DEPT_EMP trigger handles DML in the following way:

When a row is inserted into the EMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, using the data values supplied with the INSERT statement.

When a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

Note: INSTEAD OF triggers can be written only for views, and the BEFORE and AFTER timing options are not valid.

A trigger is in either of two distinct modes:

- Enabled: The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).
- Disabled: The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.



- Before Oracle Database 19c, if you created a trigger whose body had a PL/SQL compilation error, then DML to the table failed.
- In Oracle Database 19c, you can create a disabled trigger and then enable it only when you know it will be compiled successfully.

```
CREATE OR REPLACE TRIGGER mytrg
  BEFORE INSERT ON mytable FOR EACH ROW
  DISABLE
BEGIN
  :New.ID := my_seq.Nextval;
  . . .
END;
/
```

396

Before Oracle Database 19c, if you created a trigger whose body had a PL/SQL compilation error, then DML to the table failed. The following error message was displayed:

ORA-04098: trigger 'TRG' is invalid and failed re-validation

In Oracle Database 19c, you can create a disabled trigger, and then enable it only when you know it will be compiled successfully.

You can also temporarily disable a trigger in the following situations:

An object it references is not available.

You need to perform a large data load, and you want it to proceed quickly without firing triggers.

You are reloading data.

Note: The code example in the slide assumes that you have an existing sequence named `my_seq`.

Managing Triggers Using the ALTER and DROP SQL Statements

```
-- Disable or reenable a database trigger:
```

```
ALTER TRIGGER trigger_name DISABLE | ENABLE;
```

```
-- Disable or reenable all triggers for a table:
```

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS;
```

```
-- Recompile a trigger for a table:
```

```
ALTER TRIGGER trigger_name COMPILE;
```

```
-- Remove a trigger from the database:
```

```
DROP TRIGGER trigger_name;
```

397

Managing Triggers

A trigger has two modes or states: ENABLED and DISABLED. When a trigger is first created, it is enabled by default. The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, manages the dependencies, and provides a two-phase commit process if a trigger updates remote tables in a distributed database.

Disabling a Trigger

Use the ALTER TRIGGER command to disable a trigger. You can also disable all triggers on a table by using the ALTER TABLE command. You can disable triggers to improve performance or to avoid data integrity checks when loading massive amounts of data with utilities such as SQL*Loader. You might also disable a trigger when it references a database object that is currently unavailable, due to a failed network connection, disk crash, offline data file, or offline tablespace.

Recompiling a Trigger

Use the ALTER TRIGGER command to explicitly recompile a trigger that is invalid.

Removing Triggers

When a trigger is no longer required, use a SQL statement in SQL Developer or SQL*Plus to remove it. When you remove a table, all triggers on that table are also removed.

- Test each triggering data operation, as well as non-triggering data operations.
- Test each case of the WHEN clause.
- Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.
- Test the effect of the trigger on other triggers.
- Test the effect of other triggers on the trigger.

398

Testing code can be a time-consuming process. Do the following when testing triggers.

Ensure that the trigger works properly by testing a number of cases separately:

Test the most common success scenarios first.

Test the most common failure conditions to see that they are properly managed.

The more complex the trigger, the more detailed your testing is likely to be. For example, if you have a row trigger with a WHEN clause specified, then you should ensure that the trigger fires when the conditions are satisfied. Or, if you have cascading triggers, you need to test the effect of one trigger on the other and ensure that you end up with the desired results.

Use the DBMS_OUTPUT package to debug triggers.

You can view the following trigger information:

Data Dictionary View	Description
USER_OBJECTS	Displays object information
USER/ALL/DBA_TRIGGERS	Displays trigger information
USER_ERRORS	Displays PL/SQL syntax errors for a trigger

399

The slide shows the data dictionary views that you can access to get information regarding the triggers.

The USER_OBJECTS view contains the name and status of the trigger and the date and time when the trigger was created.

The USER_ERRORS view contains the details about the compilation errors that occurred while a trigger was compiling. The contents of these views are similar to those for subprograms.

The USER_TRIGGERS view contains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

The SELECT Username FROM USER_USERS; statement gives the name of the owner of the trigger, not the name of the user who is updating the table.

Using USER TRIGGERS

```
DESCRIBE user_triggers
```

Name	Null	Type
TRIGGER_NAME		VARCHAR2(30)
TRIGGER_TYPE		VARCHAR2(10)
TRIGGERING_EVENT		VARCHAR2(227)
TABLE_OWNER		VARCHAR2(30)
BASE_OBJECT_TYPE		VARCHAR2(16)
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
REFERENCING_NAMES		VARCHAR2(128)
WHEN_CLAUSE		VARCHAR2(4000)
STATEMENTS		VARCHAR2(8)
DESCRIPTION		VARCHAR2(4000)
ACTION_TYPE		VARCHAR2(11)
TRIGGER_BODY		LONG
CROSSEDITION		VARCHAR2(7)
BEFORE_STATEMENT		VARCHAR2(3)
BEFORE_ROW		VARCHAR2(3)
AFTER_ROW		VARCHAR2(3)
AFTER_STATEMENT		VARCHAR2(3)
INSTEAD_OF_ROW		VARCHAR2(3)
FIRE_ONCE		VARCHAR2(3)
APPLY_SERVER_ONLY		VARCHAR2(3)

21 rows selected

```
SELECT trigger_type, trigger_body
FROM user_triggers
WHERE trigger_name = 'SECURE_EMP';
```

400

If the source file is unavailable, then you can use the SQL Worksheet in SQL Developer or SQL*Plus to regenerate it from USER_TRIGGERS. You can also examine the ALL_TRIGGERS and DBA_TRIGGERS views, each of which contains the additional column OWNER, for the owner of the object. The result for the second example in the slide is as follows:

TRIGGER_TYPE	TRIGGER_BODY
BEFORE STATEMENT BEGIN	IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR (TO_CHAR(SYSDATE, 'HH24

Topics covered in this module:

- Create database triggers that are invoked by DML operations
- Create statement and row trigger types
- Use database trigger-firing rules
- Enable, disable, and manage database triggers
- Develop a strategy for testing triggers
- Remove database triggers

401

This lesson covered creating database triggers that execute before, after, or instead of a specified DML operation. Triggers are associated with database tables or views. The BEFORE and AFTER timings apply to DML operations on tables. The INSTEAD OF trigger is used as a way to replace DML operations on a view with appropriate DML statements against other tables in the database.

Triggers are enabled by default but can be disabled to suppress their operation until enabled again. If business rules change, triggers can be removed or altered as required.

Oracle PL/SQL 19c

Module 17: Compound, DDL & Event Triggers

Topics covered in this module:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers

403

In this lesson, you learn how to create and use database triggers.



What Is a Compound Trigger?

A single trigger on a table that allows you to specify actions for each of the following four timing points:

- Before the firing statement
- Before each row that the firing statement affects
- After each row that the firing statement affects
- After the firing statement

404

~~Starting with Oracle Database 19c, you can use a compound trigger. A~~

~~compound trigger is a single trigger on a table that allows you to specify actions for each of the four triggering timing points:~~

- ~~Before the firing statement~~
- ~~Before each row that the firing statement affects~~
- ~~After each row that the firing statement affects~~
- ~~After the firing statement~~

Note: For additional information about triggers, refer to the *Oracle Database PL/SQL Language Reference 19c Release 2 (11.2)*.

- The compound trigger body supports a common PL/SQL state that the code for each timing point can access.
- The compound trigger common state is:
 - Established when the triggering statement starts
 - Destroyed when the triggering statement completes
- A compound trigger has a declaration section and a section for each of its timing points.

405

The compound trigger body supports a common PL/SQL state that the code for each timing point can access. The common state is automatically destroyed when the firing statement completes, even when the firing statement causes an error. Your applications can avoid the mutating table error by allowing rows destined for a second table (such as a history table or an audit table) to accumulate and then bulk-inserting them.

Before Oracle Database 19c Release 1 (11.1), you needed to model the common state with an ancillary package. This approach was both cumbersome to program and subject to memory leak when the firing statement caused an error and the after-statement trigger did not fire. Compound triggers make PL/SQL easier for you to use and improve run-time performance and scalability.

You can use compound triggers to:

- Program an approach where you want the actions you implement for the various timing points to share common data.
- Accumulate rows destined for a second table so that you can periodically bulk-insert them
- Avoid the mutating-table error (ORA-04091) by allowing rows destined for a second table to accumulate and then bulk-inserting them



Timing-Point Sections of a Table Compound Trigger

A compound trigger defined on a table has one or more of the following timing-point sections. Timing-point sections must appear in the order shown in the table.

Timing Point	Compound Trigger Section
Before the triggering statement executes	BEFORE statement
After the triggering statement executes	AFTER statement
Before each row that the triggering statement affects	BEFORE EACH ROW
After each row that the triggering statement affects	AFTER EACH ROW

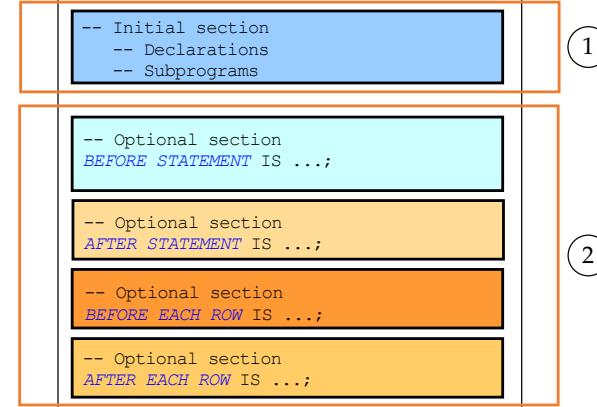
407

Note

Timing-point sections must appear in the order shown in the slide. If a timing-point section is absent, nothing happens at its timing point.

Compound Trigger Structure for Tables

```
CREATE OR REPLACE TRIGGER schema.trigger  
FOR dml_event_clause ON schema.table  
COMPOUND TRIGGER
```



408

A compound trigger has two main sections:

An initial section where variables and subprograms are declared. The code in this section executes before any of the code in the optional section.

An optional section that defines the code for each possible trigger point. Depending on whether you are defining a compound trigger for a table or for a view, these triggering points are different and are listed in the image shown above and on the following page. The code for the triggering points must follow the order shown above.

Note: For additional information about Compound Triggers, refer to *Oracle Database PL/SQL Language Reference 19c Release 2 (11.2)*.

Compound Trigger Structure for Views

```
CREATE OR REPLACE TRIGGER
schema.trigger
FOR dml_event_clause ON schema.view
COMPOUND TRIGGER
    -- Initial section
    -- Declarations
    -- Subprograms
    --
    -- Optional section (exclusive)
    INSTEAD OF EACH ROW IS
    ...;
```

409

With views, the only allowed section is an INSTEAD OF EACH ROW clause.

- A compound trigger must be a DML trigger and defined on either a table or a view.
- The body of a compound trigger must be compound trigger block, written in PL/SQL.
- A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- :OLD and :NEW cannot appear in the declaration, BEFORE STATEMENT, or the AFTER STATEMENT sections.
- Only the BEFORE EACH ROW section can change the value of :NEW.
- The firing order of compound triggers is not guaranteed unless you use the `FOLLOWS` clause.

410

The following are some of the restrictions when working with compound triggers.

The body of a compound trigger must compound trigger block, written in PL/SQL.

A compound trigger must be a DML trigger.

A compound trigger must be defined on either a table or a view.

A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section. This is not a problem, because the BEFORE STATEMENT section always executes exactly once before any other timing-point section executes.

An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.

`::OLD`, `:NEW`, and `:PARENT` cannot appear in the declaration section, the BEFORE STATEMENT section, or the AFTER STATEMENT section.

The firing order of compound triggers is not guaranteed unless you use the `FOLLOWS` clause.

- A mutating table is:
 - A table that is being modified by an UPDATE, DELETE, or INSERT statement, or
 - A table that might be updated by the effects of a DELETE CASCADE constraint
- The session that issued the triggering statement cannot query or modify a mutating table.
- This restriction prevents a trigger from seeing an inconsistent set of data.
- This restriction applies to all triggers that use the FOR EACH ROW clause.
- Views being modified in the INSTEAD OF triggers are not considered mutating.

411

Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of ON DELETE CASCADE.

Mutating Table

A mutating table is a table that is currently being modified by an UPDATE, DELETE, or INSERT statement, or a table that might need to be updated by the effects of a declarative DELETE CASCADE referential integrity action. For STATEMENT triggers, a table is not considered a mutating table.

A mutating table error (ORA-4091) occurs when a row-level trigger attempts to change or examine a table that is already undergoing change via a DML statement.

The triggered table itself is a mutating table, as well as any table referencing it with the FOREIGN KEY constraint. This restriction prevents a row trigger from seeing an inconsistent set of data.

Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id
  ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
    INTO v_minsalary, v_maxsalary
   FROM employees
  WHERE job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR :NEW.salary > v_maxsalary THEN
    RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
/
```

412

The CHECK_SALARY trigger in the slide example attempts to guarantee that whenever a new employee is added to the EMPLOYEES table or whenever an existing employee's salary or job ID is changed, the employee's salary falls within the established salary range for the employee's job.

When an employee record is updated, the CHECK_SALARY trigger is fired for each row that is updated. The trigger code queries the same table that is being updated. Therefore, it is said that the EMPLOYEES table is a mutating table.

Mutating Table: Example

```
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles';
```

```
TRIGGER check_salary Compiled.  
  
Error starting at line 1 in command:  
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles'  
  
Error report:  
SQL Error: ORA-04091: table ORA42.EMPLOYEES is mutating, trigger/function may not see it  
ORA-06512: at "ORA42.CHECK_SALARY", line 5  
ORA-04088: error during execution of trigger 'ORA42.CHECK_SALARY'  
04091. 00000 -  "table %s.%s is mutating, trigger/function may not see it"  
*Cause: A trigger (or a user defined plsql function that is referenced in  
this statement) attempted to look at (or modify) a table that was  
in the middle of being modified by the statement which fired it.  
*Action: Rewrite the trigger (or function) so it does not read that table.
```

413

In the example in the slide, the trigger code tries to read or select data from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value, then you get a run-time error. The EMPLOYEES table is mutating, or in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Possible Solutions

Possible solutions to this mutating table problem include the following:

- Use a compound trigger as described earlier in this lesson.

- Store the summary data (the minimum salaries and the maximum salaries) in another summary table, which is kept up-to-date with other DML triggers.

- Store the summary data in a PL/SQL package, and access the data from the package. This can be done in a BEFORE statement trigger.

Depending on the nature of the problem, a solution can become more convoluted and difficult to solve. In this case, consider implementing the rules in the application or middle tier and avoid using database triggers to perform

overly complex business rules. An insert statement in the code example in the slide will not generate a mutating table example.

Using a Compound Trigger to Resolve the Mutating Table Error

```
CREATE OR REPLACE TRIGGER check_salary
FOR INSERT OR UPDATE OF salary, job_id
ON employees
WHEN (NEW.job_id <> 'AD_PRES')
COMPOUND TRIGGER

TYPE salaries_t          IS TABLE OF employees.salary%TYPE;
min_salaries            salaries_t;
max_salaries            salaries_t;

TYPE department_ids_t   IS TABLE OF employees.department_id%TYPE;
department_ids          department_ids_t;

TYPE department_salaries_t IS TABLE OF employees.salary%TYPE
                           INDEX BY VARCHAR2(80);
department_min_salaries department_salaries_t;
department_max_salaries department_salaries_t;

-- example continues on next slide
```

414

The CHECK_SALARY compound trigger resolves the mutating table error in the earlier example. This is achieved by storing the values in PL/SQL collections, and then performing a bulk insert/update in the “before statement” section of the compound trigger. *This is a 19c only code.* In the example in the slide, PL/SQL collections are used. The element types used are based on the SALARY and DEPARTMENT_ID columns from the EMPLOYEES table. To create collections, you define a collection type, and then declare variables of that type. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. min_salaries is used to hold the minimum salary for each department and max_salaries is used to hold the maximum salary for each department. department_ids is used to hold the department IDs. If the employee who earns the minimum or maximum salary does not have an assigned department, you use the NVL function to store -1 for the department id instead of NULL. Next, you collect the minimum salary, maximum salary, and the department ID using a bulk insert into the min_salaries, max_salaries, and department_ids respectively grouped by department ID. The select statement returns 13 rows. The values of the

`department_ids` are used as an index for the
`department_min_salaries` and `department_max_salaries`
tables. Therefore, the index for those two tables (`VARCHAR2`) represents the
actual
`department_ids`.

Using a Compound Trigger to Resolve the Mutating Table Error

```
    .
    .
    .
BEFORE STATEMENT IS
BEGIN
    SELECT MIN(salary), MAX(salary), NVL(department_id, -1)
        BULK COLLECT INTO min_Salaries, max_salaries, department_ids
    FROM   employees
    GROUP BY department_id;
    FOR j IN 1..department_ids.COUNT() LOOP
        department_min_salaries(department_ids(j)) := min_salaries(j);
        department_max_salaries(department_ids(j)) := max_salaries(j);
    END LOOP;
END BEFORE STATEMENT;

AFTER EACH ROW IS
BEGIN
    IF :NEW.salary < department_min_salaries(:NEW.department_id)
        OR :NEW.salary > department_max_salaries(:NEW.department_id) THEN
        RAISE_APPLICATION_ERROR(-20505,'New Salary is out of acceptable
                                         range');
    END IF;
END AFTER EACH ROW;
END check_salary;
```

415

After each row is added, if the new salary is less than the minimum salary for that department or greater than the department's maximum salary, then an error message is displayed.

To test the newly created compound trigger, issue the following statement:

```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```

To ensure that the salary for employee Stiles was updated, issue the following query using the F9 key in SQL Developer:

```
1 rows updated
SELECT employee_id, first_name, last_name, job_id,
       department_id, salary
FROM employees
WHERE last_name = 'Stiles';
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	DEPARTMENT_ID	SALARY
1	138	Stephen	Stiles	ST_CLERK	50	3400

Creating Triggers on DDL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- Timing
[ddl_event1 [OR ddl_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Sample DDL Events	Fires When
CREATE	Any database object is created using the CREATE command.
ALTER	Any database object is altered using the ALTER command.
DROP	Any database object is dropped using the DROP command.

416

You can specify one or more types of DDL statements that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can also specify BEFORE and AFTER for the timing of the trigger. The Oracle database fires the trigger in the existing user transaction.

You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

The trigger body in the syntax in the slide represents a complete PL/SQL block.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, or user.

- Triggering user event:
 - CREATE, ALTER, or DROP
 - Logging on or off
- Triggering database or system event:
 - Shutting down or starting up the database
 - A specific error (or any error) being raised

Creating Database Triggers

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level.

Triggers on data definition language (DDL) statements, or a user logging on or off, can also be defined at either the database level or schema level. Triggers on data manipulation language (DML) statements are defined on a specific table or a view.

A trigger defined at the database level fires for all users whereas a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shutdown or startup
- Any error that occurs

Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- timing
[database_event1 [OR database_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Database Event	Triggers Fires When
AFTER SERVERERROR	An Oracle error is raised
AFTER LOGON	A user logs on to the database
BEFORE LOGOFF	A user logs off the database
AFTER STARTUP	The database is opened
BEFORE SHUTDOWN	The database is shut down normally

418

You can create triggers for the events listed in the table in the slide on DATABASE or SCHEMA, except SHUTDOWN and STARTUP, which apply only to DATABASE.

LOGON and LOGOFF Triggers: Example

```
-- Create the log_trig_table shown in the notes page
-- first

CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
    VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
    VALUES (USER, SYSDATE, 'Logging off');
END;
/
```

419

You can create these triggers to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify ON SCHEMA, the trigger fires for the specific user. If you specify ON DATABASE, the trigger fires for all users.

The definition of the log_trig_table used in the slide examples is as follows:

```
CREATE TABLE log_trig_table(
    user_id VARCHAR2(30),
    log_date DATE,
    action VARCHAR2(40))
/
```

CALL Statements in Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name  
timing  
event1 [OR event2 OR event3]  
ON table_name  
[REFERENCING OLD AS old | NEW AS new]  
[FOR EACH ROW]  
[WHEN condition]  
CALL procedure_name  
/
```

```
CREATE OR REPLACE PROCEDURE log_execution IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('log_execution: Employee Inserted');  
END;  
/  
CREATE OR REPLACE TRIGGER log_employee  
BEFORE INSERT ON EMPLOYEES  
CALL log_execution -- no semicolon needed  
/
```

420

A CALL statement enables you to call a stored procedure, rather than code the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.

The call can reference the trigger attributes :NEW and :OLD as parameters, as in the following example:

```
CREATE OR REPLACE TRIGGER salary_check  
BEFORE UPDATE OF salary, job_id ON  
employees  
FOR EACH ROW  
WHEN (NEW.job_id <> 'AD_PRES')  
CALL check_salary(:NEW.job_id,  
:NEW.salary)
```

Note: There is no semicolon at the end of the CALL statement.

In the preceding example, the trigger calls a `check_salary` procedure. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

- Improved data security:
 - Provides enhanced and complex security checks
 - Provides enhanced and complex auditing
- Improved data integrity:
 - Enforces dynamic data integrity constraints
 - Enforces complex referential integrity constraints
 - Ensures that related operations are performed together implicitly

421

You can use database triggers:

- As alternatives to features provided by the Oracle server
- If your requirements are more complex or more simple than those provided by the Oracle server
- If your requirements are not provided by the Oracle server at all

System Privileges Required to Manage Triggers

The following system privileges are required to manage triggers:

- The privileges that enable you to create, alter, and drop triggers in any schema:
 - GRANT CREATE TRIGGER TO ora61
 - GRANT ALTER ANY TRIGGER TO ora61
 - GRANT DROP ANY TRIGGER TO ora61
- The privilege that enables you to create a trigger on the database:
 - GRANT ADMINISTER DATABASE TRIGGER TO ora61
- The EXECUTE privilege (if your trigger refers to any objects that are not in your schema)

422

To create a trigger in your schema, you need the CREATE TRIGGER system privilege, and you must own the table specified in the triggering statement, have the ALTER privilege for the table in the triggering statement, or have the ALTER ANY TABLE system privilege. You can alter or drop your triggers without any further privileges being required.

If the ANY keyword is used, you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table. You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the EXECUTE privilege on the referenced procedures, functions, or packages, and not through roles.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, you can drop the trigger but you cannot alter it.

Note: Similar to stored procedures, statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

Guidelines for Designing Triggers

- You can design triggers to:
 - Perform related actions
 - Centralize global operations
- You must not design triggers:
 - Where functionality is already built into the Oracle server
 - That duplicate other triggers
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.
- Excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

423

Use triggers to guarantee that related actions are performed for a specific operation and for centralized, global operations that should be fired for the triggering statement, independent of the user or application issuing the statement.

Do not define triggers to duplicate or replace the functionality already built into the Oracle database. For example, implement integrity rules using declarative constraints instead of triggers. To remember the design order for a business rule:

Use built-in constraints in the Oracle server, such as primary key, and so on.

Develop a database trigger or an application, such as a servlet or Enterprise JavaBeans (EJB) on your middle tier.

Use a presentation interface, such as Oracle Forms, HTML, JavaServer Pages (JSP) and so on, for data presentation rules.

Excessive use of triggers can result in complex interdependencies, which may be difficult to maintain. Use triggers when necessary, and be aware of recursive and cascading effects.

Avoid lengthy trigger logic by creating stored procedures or packaged procedures that are invoked in the trigger body.

Database triggers fire for every user each time the event occurs on the trigger that is created.

Topics covered in this module:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers

Oracle PL/SQL 19c

Module 18: PL/SQL Compiler

Topics covered in this module:

- Use the PL/SQL compiler initialization parameters
- Use the PL/SQL compile-time warnings

Note that the content in this chapter is 19c specific.

Some of the examples listed may or may not work in a 10g environment.

- o PLSQL_CODE_TYPE
- o PLSQL_OPTIMIZE_LEVEL
- o PLSQL_CCFLAGS
- o PLSQL_WARNINGS



427

In releases before Oracle Database 10g, the PL/SQL compiler translated your code to machine code without applying many changes for performance. Oracle Database 19c onwards, PL/SQL uses an optimizing compiler that can rearrange code for better performance. You do not need to do anything to get the benefits of this new optimizer; it is enabled by default.

Note

The PLSQL_CCFLAGS initialization parameter is covered in the lesson titled “Managing PL/SQL Code.”

The PLSQL_WARNINGS initialization parameter is covered later in this lesson.



Using the Initialization Parameters for PL/SQL Compilation

- `PLSQL_CODE_TYPE`: Specifies the compilation mode for PL/SQL library units

```
PLSQL_CODE_TYPE = { INTERPRETED | NATIVE }
```

- `PLSQL_OPTIMIZE_LEVEL`: Specifies the optimization level to be used to compile PL/SQL library units

```
PLSQL_OPTIMIZE_LEVEL = { 0 | 1 | 2 | 3 }
```

428

The `PLSQL_CODE_TYPE` Parameter

This parameter specifies the compilation mode for PL/SQL library units. If you choose `INTERPRETED`, PL/SQL library units will be compiled to PL/SQL bytecode format. Such modules are executed by the PL/SQL interpreter engine. If you choose `NATIVE`, PL/SQL library units (with the possible exception of top-level anonymous PL/SQL blocks) will be compiled to native (machine) code. Such modules will be executed natively without incurring any interpreter overhead. When the value of this parameter is changed, it has no effect on PL/SQL library units that have already been compiled. The value of this parameter is stored persistently with each library unit. If a PL/SQL library unit is compiled natively, all subsequent automatic recompilations of that library unit will use native compilation. In Oracle Database 19c, native compilation is easier and more integrated, with fewer initialization parameters to set.

In rare cases, if the overhead of the optimizer makes compilation of very large applications take too long, you might lower the optimization level by setting the initialization parameter `PLSQL_OPTIMIZE_LEVEL` to 1 instead of its default value 2. In even rarer cases, you might see a change in exception behavior—either an exception that is not raised at all or one that is raised

earlier than expected. Setting `PLSQL_OPTIMIZE_LEVEL` to 0 prevents the code from being rearranged at all.

The Compiler Settings

Compiler Option	Description
PLSQL_CODE_TYPE	Specifies the compilation mode for PL/SQL library units.
PLSQL_OPTIMIZE_LEVEL	Specifies the optimization level to be used to compile PL/SQL library units.
PLSQL_WARNINGS	Enables or disables the reporting of warning messages by the PL/SQL compiler.
PLSQL_CCFLAGS	Controls conditional compilation of each PL/SQL library unit independently.

In general, for the fastest performance, use the following setting:

```
PLSQL_CODE_TYPE = NATIVE  
PLSQL_OPTIMIZE_LEVEL = 2
```

429

The new compiler increases the performance of PL/SQL code and allows it to execute approximately two times faster than an Oracle8*i* database and 1.5 times to 1.75 times as fast as Oracle9*i* Database Release 2.

To get the fastest performance, the compiler setting must be:

```
PLSQL_CODE_TYPE = NATIVE  
PLSQL_OPTIMIZE_LEVEL = 2
```



Displaying the PL/SQL Initialization Parameters

Use the `USER|ALL|DBA PLSQL_OBJECT_SETTINGS` data dictionary views to display the settings for a PL/SQL object:

```
DESCRIBE USER_PLSQL_OBJECT_SETTINGS
```

```
DESCRIBE USER_PLSQL_OBJECT_SETTINGS
Name          Null    Type
-----        -----
NAME          NOT NULL VARCHAR2(30)
TYPE          VARCHAR2(12)
PLSQL_OPTIMIZE_LEVEL NUMBER
PLSQL_CODE_TYPE      VARCHAR2(4000)
PLSQL_DEBUG      VARCHAR2(4000)
PLSQL_WARNINGS   VARCHAR2(4000)
NLS_LENGTH_SEMANTICS VARCHAR2(4000)
PLSQL_CFLAGS     VARCHAR2(4000)
PLSCOPE_SETTINGS VARCHAR2(4000)
```

430

The columns of the `USER_PLSQL_OBJECTS_SETTINGS` data dictionary view are:

Owner: The owner of the object. This column is not displayed in the `USER_PLSQL_OBJECTS_SETTINGS` view.

Name: The name of the object

Type: The available choices are: PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, TYPE, or TYPE BODY.

PLSQL_OPTIMIZE_LEVEL: The optimization level that was used to compile the object

PLSQL_CODE_TYPE: The compilation mode for the object

PLSQL_DEBUG: Specifies whether or not the object was compiled for debugging

PLSQL_WARNINGS: The compiler warning settings used to compile the object

NLS_LENGTH_SEMANTICS: The NLS length semantics used to compile the object

PLSQL_CFLAGS: The conditional compilation flag used to compile the object

PLSCOPE_SETTINGS: Controls the compile-time collection, cross

reference, and storage of PL/SQL source code identifier data (new in Oracle Database 19c)



Displaying and Setting the PL/SQL Initialization Parameters

```
SELECT name, type, plsql_code_type AS CODE_TYPE,
       plsql_optimize_level AS OPT_LVL
  FROM user_plsql_object_settings;
```

NAME	TYPE	CODE_TYPE	OPT_LVL
1 ADD_EMPLOYEE	PROCEDURE	INTERPRETED	2
2 ADD_JOB_HIST	PROCEDURE	INTERPRETED	2
3 ADD_JOB_HISTORY	PROCEDURE	INTERPRETED	2
4 BANK_TRANS	PROCEDURE	INTERPRETED	2
5 CHECK_SALARY	PROCEDURE	INTERPRETED	2

- o Set the ~~DBA_STORED_SETTINGS~~ parameter's value using the ALTER SYSTEM or ALTER SESSION statements.
- o The parameters' values are accessed when the CREATE OR REPLACE statement is executed.

431

Note

For additional information about the ALTER SYSTEM or ALTER SESSION statements, refer to *Oracle Database SQL Reference*.

The DBA_STORED_SETTINGS data dictionary view family is deprecated in Oracle Database 10g and is replaced with the DBA_PLSQL_OBJECT_SETTINGS data dictionary view family.

Changing PL/SQL Initialization Parameters: Example

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;
ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE';
```

session SET altered.
session SET altered.

```
-- code displayed in the notes page
CREATE OR REPLACE PROCEDURE add_job_history
. . .
```

```
@code_11_10_bn.sql
```

NAME	TYPE	CODE_TYPE	OPT_LVL
1 ADD_COL	PROCEDURE	INTERPRETED	2
2 ADD_EMPLOYEE	PROCEDURE	INTERPRETED	2
3 ADD_JOB_HIST	PROCEDURE	INTERPRETED	2
4 ADD_JOB_HISTORY	PROCEDURE	NATIVE	1
...			

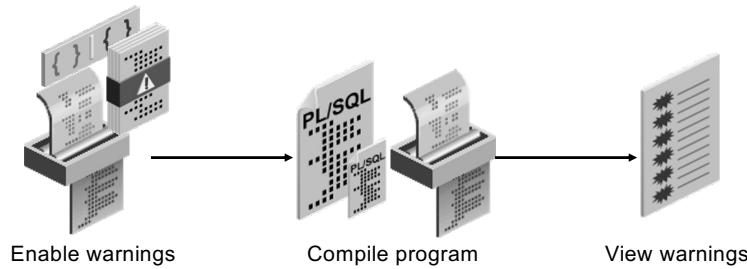
432

To change a compiled PL/SQL object from interpreted code type to native code type, you must first set the `PLSQL_CODE_TYPE` parameter to `NATIVE` (optionally set the other parameters) and then, recompile the program. To enforce native compilation to all PL/SQL code, you must recompile each one. Scripts (in the `rdmbs/admin` directory) are provided for you to achieve conversion to full native compilation (`dbmsupgnv.sql`) or full interpreted compilation (`dbmsupgln.sql`). The `add_job_history` procedure is created as follows:

```
CREATE OR REPLACE PROCEDURE add_job_history
( p_emp_id      job_history.employee_id%type
, p_start_date   job_history.start_date%type
, p_end_date     job_history.end_date%type
, p_job_id       job_history.job_id%type
, p_department_id
                job_history.department_id%type )
IS
BEGIN
  INSERT INTO job_history (employee_id, start_date,
                           end_date, job_id, department_id)
```

```
VALUES(p_emp_id, p_start_date, p_end_date,  
       p_job_id, p_department_id);  
END add_job_history;  
/
```

Starting with Oracle 10g, the PL/SQL compiler has been enhanced to produce warnings for subprograms.



433

To make your programs more robust and avoid problems at run time, you can turn on checking for certain warning conditions. These conditions are not serious enough to produce an error and keep you from compiling a subprogram. They may point out something in the subprogram that produces an undefined result or might create a performance problem.

In releases prior to Oracle Database 10g, compiling a PL/SQL program had two possible outcomes:

Success, producing a valid compiled unit

Failure, with compilation errors indicating that the program had either syntax or semantic errors

However, even when compilation of a program was successful, the program may have violated recommended best practices or could have been coded to be more efficient. Oracle Database 10g introduced a new ease-of-use feature that allows the PL/SQL compiler to communicate warning messages in these situations. Compiler warnings allow developers to avoid common coding pitfalls, thus improving productivity.

Benefits of Compiler Warnings

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



434

Using compiler warnings can help you to:

- Make your programs more robust and avoid problems at run time
- Identify potential performance problems
- Identify 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.

Warning messages can be issued during compilation of PL/SQL subprograms; anonymous blocks do not produce any warnings.

All PL/SQL warning messages use the prefix PLW.

- Severe
- Informational
- Performance
- All



435

You can set the compiler warning messages levels using one of the following methods:

Using the PLSQL_WARNINGS Initialization Parameter

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 settings for the PLSQL_WARNINGS parameter are stored along with each compiled subprogram. You can use the PLSQL_WARNINGS initialization parameter to do the following:

Enable or disable the reporting of all warnings, warnings of a selected category, or specific warning messages.

Treat all warnings, a selected category of warning, or specific warning messages as errors.

Any valid combination of the preceding

The keyword All is a shorthand way to refer to all warning messages: SEVERE, PERFORMANCE, and INFORMATIONAL.

Using the 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. This package is covered later in this lesson.

You can set warning levels using one of the following methods:

- Declaratively:
 - Using the `PLSQL_WARNINGS` initialization parameter
- Programmatically:
 - Using the `DBMS_WARNING` package

You can set the compiler warning messages levels using one of the following methods:

Using the `PLSQL_WARNINGS` Initialization Parameter

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 settings for the `PLSQL_WARNINGS` parameter are stored along with each compiled subprogram. You can use the `PLSQL_WARNINGS` initialization parameter to do the following:

Enable or disable the reporting of all warnings, warnings of a selected category, or specific warning messages.

Treat all warnings, a selected category of warning, or specific warning messages as errors.

Any valid combination of the preceding

The keyword `All` is a shorthand way to refer to all warning messages: `SEVERE`, `PERFORMANCE`, and `INFORMATIONAL`.

Using the `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. This package is covered later in this lesson.

Setting Compiler Warning Levels: Using PLSQL_WARNINGS

```
ALTER [SESSION|SYSTEM]
PLSQL_WARNINGS = 'value_clause1'[, 'value_clause2']...
```

```
value_clause = Qualifier Value : Modifier Value
```

```
Qualifier Value = { ENABLE | DISABLE | ERROR }

Modifier Value =
{ ALL | SEVERE | INFORMATIONAL | PERFORMANCE |
{ integer [, integer ] ... } }
```

437

Modifying Compiler Warning Settings

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**, **DISABLE**, and **ERROR**. 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.

Possible values for **ENABLE**, **DISABLE**, and **ERROR**:

- ALL**
- SEVERE**
- INFORMATIONAL**
- PERFORMANCE**
- numeric_value**

Values for **numeric_value** are in:

Range 5000–5999 for severe

Range 6000–6249 for informational

Range 7000–7249 for performance

Setting Compiler Warning Levels: Using PLSQL_WARNINGS, Examples

```
ALTER SESSION  
SET plsql_warnings = 'enable:severe',  
      'enable:performance',  
      'disable:informational';
```

```
ALTER SESSION succeeded.
```

```
ALTER SESSION  
SET plsql_warnings = 'enable:severe';
```

```
ALTER SESSION succeeded.
```

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE',  
      'DISABLE:PERFORMANCE' , 'ERROR:05003';
```

```
ALTER SESSION SET succeeded.
```

438

You can use the ALTER SESSION or ALTER SYSTEM command to change the PLSQL_WARNINGS initialization parameter. The graphic in the slide shows the various examples of enabling and disabling compiler warnings.

Example 1

In this example, you are enabling SEVERE and PERFORMANCE warnings and disabling INFORMATIONAL warnings.

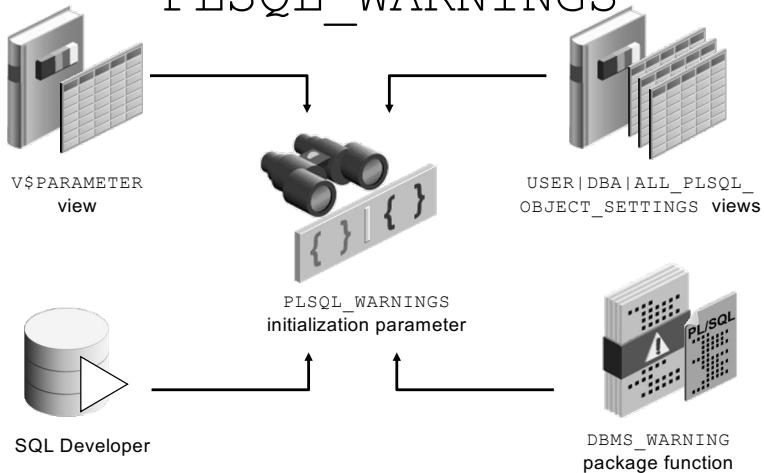
Example 2

In the second example, you are enabling only SEVERE warnings.

Example 3

You can also treat particular messages as errors instead of warnings. In this example, if you know that the warning message PLW-05003 represents a serious problem in your code, including 'ERROR:05003' in the PLSQL_WARNINGS setting makes that condition trigger an error message (PLS_05003) instead of a warning message. An error message causes the compilation to fail. In this example, you are also disabling PERFORMANCE warnings.

Viewing the Current Setting of PLSQL_WARNINGS



439

Viewing the Current Value of the PLSQL_WARNINGS Parameter

You can examine the current setting for the PLSQL_WARNINGS parameter by issuing a SELECT statement on the V\$PARAMETER view. For example:

```
ALTER SESSION SET plsql_warnings = 'enable:severe',
  'enable:performance','enable:informational';
```

Session altered.

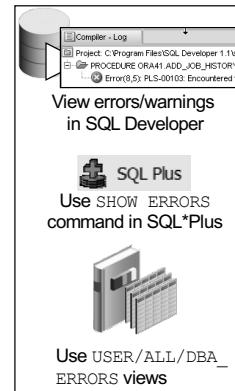
```
SELECT value FROM v$parameter WHERE
name='plsql_warnings';
VALUE
-----
ENABLE:ALL
```

Viewing the Compiler Warnings: Using SQL Developer, SQL*Plus, or Data Dictionary Views

Enable warnings
and compile program

Compiler
warnings/errors?

View compiler
warnings/errors



440

Viewing the Compiler Warnings

You can use SQL*Plus to see any warnings raised as a result of the compilation of a PL/SQL block. SQL*Plus indicates that a compilation warning has occurred. The "***SP2-08xx: <object> created with compilation warnings.***" message is displayed for objects compiled with the PERFORMANCE, INFORMATIONAL, or SEVERE modifiers. There is no differentiation between the three. You must enable the compiler warnings before compiling the program. You can display the compiler warning messages using one of the following methods:

Using the SQL*Plus SHOW ERRORS Command

This command displays any compiler errors including the new compiler warnings and informational messages. This command is invoked immediately after a CREATE [PROCEDURE | FUNCTION | PACKAGE] command is used. The SHOW ERRORS command displays warnings and compiler errors. New compiler warnings and informational messages are "interleaved" with compiler errors when SHOW ERRORS is invoked.

Using the Data Dictionary Views

You can select from the USER_ | ALL_ | DBA_ERRORS data dictionary views to display PL/SQL compiler warnings. The ATTRIBUTES column of these

views has a new attribute called `WARNING` and the warning message displays in the `TEXT` column.

SQL*Plus Warning Messages: Example

```
CREATE OR REPLACE PROCEDURE bad_proc(p_out ...) IS
BEGIN
. . .;
END;
/
SP2-0804: Procedure created with compilation warnings.
```

```
SHOW ERRORS;
Errors for PROCEDURE BAD_PROC:

LINE/COL      ERROR
----- -----
6/24          PLW-07203: parameter 'p_out' may benefit
                  from use of the NOCOPY compiler hint
```

441

Use the `SHOW ERRORS` command in SQL*Plus to display the compilation errors of a stored procedure. When you specify this option with no arguments, SQL*Plus displays the compilation errors for the most recently created or altered stored procedure. If SQL*Plus displays a compilation warnings message after you create or alter a stored procedure, you can use `SHOW ERRORS` commands to obtain more information.

With the introduction of the support for PL/SQL warnings, the range of feedback messages is expanded to include a third message as follows:

SP2-08xx: <object> created with compilation
warnings.

This enables you to differentiate between the occurrence of a compilation warning and a compilation error. You must correct an error if you want to use the stored procedure, whereas a warning is for informational purposes only. The `SP2` prefix is included with the warning message, because this provides you with the ability to look up the corresponding message number in the *SQL*Plus User's Guide and Reference* to determine the cause and action for the particular message.

Note: The `SHOW SQL*Plus` command is not supported in the SQL Developer 1.5.4 version that is used in this class. You can view the compiler errors and

warnings using the USER_ | ALL_ | DBA_ERRORS data dictionary views.

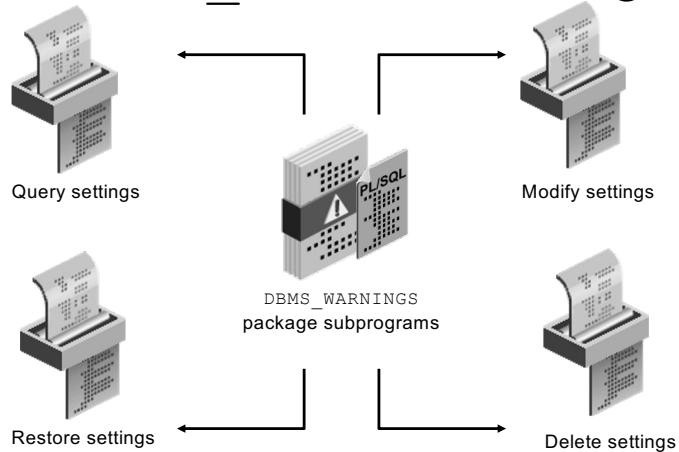
- The settings for the PLSQL_WARNINGS parameter are stored along with each compiled subprogram.
- If you recompile the subprogram using one of the following statements, the current settings for that session are used:
 - CREATE OR REPLACE
 - ALTER ... COMPILE
- If you recompile the subprogram using the ALTER ... COMPILE statement with the REUSE SETTINGS clause, the original setting stored with the program is used.

442

As already stated, the PLSQL_WARNINGS parameter can be set at the session level or the system level.

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.

Setting Compiler Warning Levels: Using the DBMS_WARNING Package



443

Use the DBMS_WARNING package to programmatically manipulate the behavior of current system or session PL/SQL warning settings. 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_WARNING 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.

Using the DBMS_WARNING Package Subprograms

Scenario	Subprograms to Use
Set warnings	ADD_WARNING_SETTING_CAT (procedure) ADD_WARNING_SETTING_NUM (procedure)
Query warnings	GET_WARNING_SETTING_CAT (function) GET_WARNING_SETTING_NUM (function) GET_WARNING_SETTING_STRING (function)
Replace warnings	SET_WARNING_SETTING_STRING (procedure)
Get the warnings' categories names	GET_CATEGORY (function)

444

Using the DBMS_WARNING Subprograms

The following is a list of the DBMS_WARNING subprograms:

- 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
- GET_CATEGORY: Returns the category name, given the message number
- GET_WARNING_SETTING_CAT: Returns the specific warning category in the session
- GET_WARNING_SETTING_NUM: Returns the specific warning number in the session
- GET_WARNING_SETTING_STRING: Returns the entire warning string for the current session
- SET_WARNING_SETTING_STRING: Replaces previous settings with the new value

Note: For additional information about the preceding subprograms, refer to

*Oracle Database PL/SQL Packages and Types Reference 19c Release 2 (11.2) or
Oracle Database PL/SQL Packages and Types Reference 10g (as applicable to
the version you are using).*

The DBMS_WARNING Procedures: Syntax, Parameters, and Allowed Values

```
EXECUTE DBMS_WARNING.ADD_WARNING_SETTING_CAT (-  
    warning_category      IN      VARCHAR2,  
    warning_value         IN      VARCHAR2,  
    scope                 IN      VARCAHR2);
```

```
EXECUTE DBMS_WARNING.ADD_WARNING_SETTING_NUM (-  
    warning_number        IN      NUMBER,  
    warning_value         IN      VARCHAR2,  
    scope                 IN      VARCAHR2);
```

```
EXECUTE DBMS_WARNING.SET_WARNING_SETTING_STRING (-  
    warning_value        IN      VARCHAR2,  
    scope                 IN      VARCHAR2);
```

445

The `warning_category` is the name of the category. The allowed values are:

- ALL
- INFORMATIONAL
- SEVERE
- PERFORMANCE.

The `warning_value` is the value for the category. The allowed values are:

- ENABLE
- DISABLE
- ERROR

The `warning_number` is the warning message number. The allowed values are all valid warning numbers.

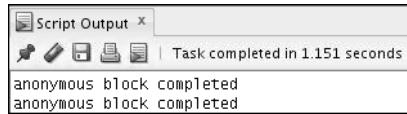
The `scope` specifies whether the changes are being performed in the session context or the system context. The allowed values are SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.

The DBMS_WARNING Procedures: Example

```
-- Establish the following warning setting string in the
-- current session:
-- ENABLE:INFORMATIONAL,
-- DISABLE:PERFORMANCE,
-- ENABLE:SEVERE

EXECUTE DBMS_WARNING.SET_WARNING_SETTING_STRING(
    'ENABLE:ALL', 'SESSION');

EXECUTE DBMS_WARNING.ADD_WARNING_SETTING_CAT(
    'PERFORMANCE', 'DISABLE', 'SESSION');
```



446

Using DBMS_WARNING Procedures: Example

Using the SET_WARNING_SETTING_STRING procedure, 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 more settings to the initial string.

The example in the slide establishes the following warning setting string in the current session:

ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE

The DBMS_WARNING Functions: Syntax, Parameters, and Allowed Values

```
DBMS_WARNING.GET_WARNING_SETTING_CAT (-  
warning_category IN VARCHAR2) RETURN warning_value;
```

```
DBMS_WARNING.GET_WARNING_SETTING_NUM (-  
warning_number IN NUMBER) RETURN warning_value;
```

```
DBMS_WARNING.GET_WARNING_SETTING_STRING  
RETURN pls_integer;
```

```
DBMS_WARNING.GET_CATEGORY (-  
warning_number IN pls_integer) RETURN VARCHAR2;
```

447

The `warning_category` is the name of the category. The allowed values are:

- ALL
- INFORMATIONAL
- SEVERE
- PERFORMANCE

The `warning_number` is the warning message number. The allowed values are all valid warning numbers.

The `scope` specifies whether the changes are being performed in the session context or the system context. The allowed values are SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.

Note: Use the `GET_WARNING_SETTING_STRING` function when you do not have the `SELECT` privilege on the `v$parameter` or `v$parameter2` fixed tables, or if you want to parse the warning string yourself and then modify and set the new value using `SET_WARNING_SETTING_STRING`.

The DBMS_WARNING Functions: Example

```
-- Determine the current session warning settings  
SET SERVEROUTPUT ON  
EXECUTE DBMS_OUTPUT.PUT_LINE( -  
DBMS_WARNING.GET_WARNING_SETTING_STRING);
```

```
anonymous block completed  
ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE
```

```
-- Determine the category for warning message number  
-- PLW-07203  
SET SERVEROUTPUT ON  
EXECUTE DBMS_OUTPUT.PUT_LINE( -  
DBMS_WARNING.GET_CATEGORY(7203));
```

```
anonymous block completed  
PERFORMANCE
```

448

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

```
CREATE OR REPLACE PROCEDURE compile_code(p_pkg_name VARCHAR2) IS
  v_warn_value  VARCHAR2(200);
  v_compile_stmt VARCHAR2(200) := 
    'ALTER PACKAGE'|| p_pkg_name ||' COMPILE';

BEGIN
  v_warn_value := DBMS_WARNING.GET_WARNING_SETTING_STRING;
  DBMS_OUTPUT.PUT_LINE('Current warning settings: '|| 
    v_warn_value);
  DBMS_WARNING.ADD_WARNING_SETTING_CAT(
    'PERFORMANCE', 'DISABLE', 'SESSION');
  DBMS_OUTPUT.PUT_LINE('Modified warning settings: '|| 
    DBMS_WARNING.GET_WARNING_SETTING_STRING);
  EXECUTE IMMEDIATE v_compile_stmt;
  DBMS_WARNING.SET_WARNING_SETTING_STRING(v_warn_value,
    'SESSION');
  DBMS_OUTPUT.PUT_LINE('Restored warning settings: '|| 
    DBMS_WARNING.GET_WARNING_SETTING_STRING);
END;
/
```

449

Note: Before you run the code provided in the example in the slide, you must create the MY_PKG script found in demo_11_33.sql. This demo script creates the MY_PKG package.

In the example in the slide, the compile_code procedure is designed to compile a named PL/SQL package. The code suppresses the PERFORMANCE category warnings. The calling environment's warning settings must be restored after the compilation is performed. The code does not know what the calling environment warning settings are; it uses the 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_code procedure alters the current session-warning level by disabling warnings for the PERFORMANCE category. The code also prints the original, modified, and the restored warning settings.

Using DBMS_WARNING: Example

```
EXECUTE DBMS_WARNING.SET_WARNING_SETTING_STRING(--  
'ENABLE:ALL', 'SESSION');
```

```
anonymous block completed
```

```
@/home/oracle/labs/plpu/code_ex/code_11_33_s.sql
```

```
PROCEDURE compile_code compiled
```

```
@code_11_34_cs.sql -- compiles the DEPT_PKG package
```

```
anonymous block completed  
Current warning settings: ENABLE:ALL  
Modified warning settings: ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE  
Restored warning settings: ENABLE:ALL
```

450

In the example in the slide, the example provided in the previous slide is tested. First, enable all compiler warnings. Next, run the script on the previous page. Finally, call the `compile_code` procedure and pass it an existing package name, `DEPT_PKG`, as a parameter.

Using the PLW 06009 Warning Message

- The PLW warning indicates that the OTHERS handler of your PL/SQL subroutine can exit without executing:
 - Some form of RAISE, or
 - A call to the standard procedure RAISE_APPLICATION_ERROR
- A good programming practice suggests that OTHERS handlers must always pass an exception upward.

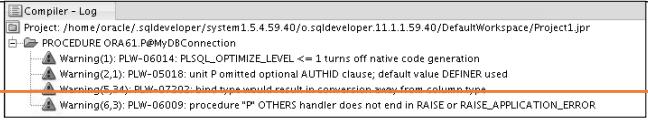
451

As a good programming practice, you should have your OTHERS exception handler pass the exception upward to the calling subroutine. If you fail to add this functionality, you run the risk of having exceptions go unnoticed. To avoid this flaw in your code, you can turn on warnings for your session and recompile the code that you want to verify. If the OTHERS handler does not handle the exception, the PLW 06009 warning will inform you. Please note that this warning is specific to Oracle Database 19c.

Note: PLW 06009 is not the only new warning message in Oracle Database 19c. For a complete list of all PLW warnings, see *Oracle Database Error Messages19c Release 2 (11.2)*.

The PLW 06009 Warning: Example

```
CREATE OR REPLACE PROCEDURE p(i IN VARCHAR2)
IS
BEGIN
    INSERT INTO t(col_a) VALUES (i);
EXCEPTION
    WHEN OTHERS THEN null;
END p;
/
ALTER PROCEDURE P COMPILE
PLSQL_warnings = 'enable:all' REUSE SETTINGS;
```



```
SELECT *
FROM user_errors
WHERE name = 'P'
```

NAME	TYPE	SEQUEN.	LINE	POSITION	TEXT	ATTR.	MESSAGE NUMBER
1 P	PROCEDURE	1	4	34	PLW-07202: bind type would result in conversion away from column type	WARNING	7202
2 P	PROCEDURE	2	1	1	PLW-05018: unit P omitted optional AUTHID clause; default value DEFINER used	WARNING	5018
3 P	PROCEDURE	3	0	0	PLW-06014: PLSQL_OPTIMIZE_LEVEL <= 1 turns off native code generation	WARNING	6014
4 P	PROCEDURE	4	5	3	PLW-06009: procedure "P" OTHERS handler does not end in RAISE or RAISE_APPLICATION... WARNING	WARNING	6009

452

After running the first code example in the slide and after compiling the procedure using the Object Navigation tree, the **Compiler – Log** tab displays the PLW-06009 warning.

You can also use the `user_error` data dictionary view to display the error. The definition of table `t` that is used in the slide example is as follows:

```
CREATE TABLE t (col_a NUMBER);
```

Topics covered in this module:

- Use the PL/SQL compiler initialization parameters
- Use the PL/SQL compile-time warnings



Oracle PL/SQL 19c



Module 19: Managing PL/SQL Code

Topics covered in this module:

- Describe and use conditional compilation
- Hide PL/SQL source code using dynamic obfuscation and the Wrap utility

455

This lesson introduces the conditional compilation and obfuscating or wrapping PL/SQL code.

What Is Conditional Compilation?

Enables you to customize the functionality in a PL/SQL application without removing any source code:

- Utilize the latest functionality with the latest database release or disable the new features to run the application against an older release of the database.
- Activate debugging or tracing functionality in the development environment and hide that functionality in the application while it runs at a production site.



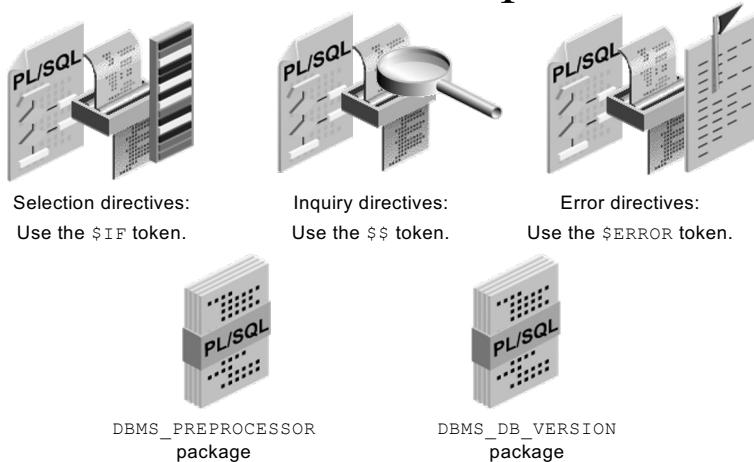
456

Conditional compilation enables you to selectively include code, depending on the values of the conditions evaluated during compilation. For example, conditional compilation enables you to determine which PL/SQL features in a PL/SQL application are used for specific database releases. The latest PL/SQL features in an application can be run on a new database release and at the same time those features can be conditional so that the same application is compatible with a previous database release. Conditional compilation is also useful when you want to execute debugging procedures in a development environment, but want to turn off the debugging routines in a production environment.

Benefits of Conditional Compilation

- Support for multiple versions of the same program in one source code
- Easy maintenance and debugging of code
- Easy migration of code to a different release of the database

How Does Conditional Compilation Work?



457

You can use conditional compilation by embedding directives in your PL/SQL source programs. When the PL/SQL program is submitted for compilation, a preprocessor evaluates these directives and selects parts of the program to be compiled. The selected program source is then handed off to the compiler for compilation.

Inquiry directives use the \$\$ token to make inquiries about the compilation environment such as the value of a PL/SQL compiler initialization parameters PLSQL_CCFLAGS or PLSQL_OPTIMIZE_LEVEL for the unit being compiled. This directive can be used in conjunction with the conditional selection directive to select the parts of the program to compile.

Selection directives can test inquiry directives or static package constants by using the \$IF construct to branch sections of code for possible compilation if a condition is satisfied.

Error directives issue a compilation error if an unexpected condition is encountered during conditional compilation using the \$ERROR token.

The DBMS_DB_VERSION package provides database version and release constants that can be used for conditional compilation.

The DBMS_PREPROCESSOR package provides subprograms for accessing the post-processed source text that is selected by conditional compilation

directives in a PL/SQL unit.

Using Selection Directives

```
$IF <Boolean-expression> $THEN Text  
$ELSEIF <Boolean-expression> $THEN Text  
.  
.  
$ELSE Text  
$END
```

```
DECLARE  
CURSOR cur IS SELECT employee_id FROM  
employees WHERE  
$IF myapp_tax_package.new_tax_code $THEN  
    salary > 20000;  
$ELSE  
    salary > 50000;  
$END  
BEGIN  
    OPEN cur;  
.  
END;
```

458

The conditional selection directive looks like and operates like the IF-THEN-ELSE mechanism in PL/SQL proper. When the preprocessor encounters \$THEN, it verifies that the text between \$IF and \$THEN is a static expression. If the check succeeds and the result of the evaluation is TRUE, then the PL/SQL program text between \$THEN and \$ELSE (or \$ELSIF) is selected for compilation.

The selection condition (the expression between \$IF and \$THEN) can be constructed by referring to constants defined in another package or an inquiry directive or some combination of the two.

In the example in the slide, conditional selection directive chooses between two versions of the cursor, `cur`, on the basis of the value of `MYAPP_TAX_PACKAGE.NEW_TAX_CODE`. If the value is TRUE, then employees with `salary > 20000` are selected, else employees with `salary > 50000` are selected.

Using Predefined and User-Defined Inquiry Directives

```
PLSQL_CCFLAGS  
PLSQL_CODE_TYPE  
PLSQL_OPTIMIZE_LEVEL  
PLSQL_WARNINGS  
NLS_LENGTH_SEMANTICS  
PLSQL_LINE  
PLSQL_UNIT
```

Predefined inquiry directives

```
PLSQL_CCFLAGS = 'plsql_ccflags:true,debug:true,debug:0';
```

User-defined inquiry directives

459

An inquiry directive can be predefined or user-defined. The following describes the order of the processing flow when conditional compilation attempts to resolve an inquiry directive:

1. The ID is used as an inquiry directive in the form `$$id` for the search key.
2. The two-pass algorithm proceeds as follows:
 - a. The string in the `PLSQL_CCFLAGS` initialization parameter is scanned from right to left, searching with ID for a matching name (not case sensitive); done if found.
 - b. The predefined inquiry directives are searched; done if found.
3. If the `$$ID` cannot be resolved to a value, then the `PLW-6003` warning message is reported if the source text is not wrapped. The literal `NULL` is substituted as the value for undefined inquiry directives. Note that if the PL/SQL code is wrapped, then the warning message is disabled so that the undefined inquiry directive is not revealed.

In the example in the slide, the value of `$$debug` is 0 and the value of `$$plsql_ccflags` is TRUE. Note that the value of `$$plsql_ccflags`

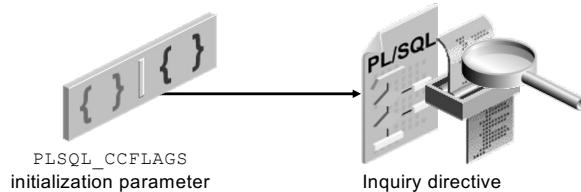
resolves to the user-defined `plsql_ccflags` inside the value of the `PLSQL_CCFLAGS` compiler parameter. This occurs because a user-defined directive overrides the predefined one.

The `PLSQL_CCFLAGS` Parameter and the Inquiry Directive

Use the `PLSQL_CCFLAGS` parameter to control conditional compilation of each PL/SQL library unit independently.

```
PLSQL_CCFLAGS = '<v1>:<c1>,<v2>:<c2>,...,<vn>:<cn>'
```

```
ALTER SESSION SET
PLSQL_CCFLAGS = 'plsql_ccflags:true, debug:true, debug:0';
```



460

Oracle Database 10g Release 2 introduced a new Oracle initialization parameter `PLSQL_CCFLAGS` for use with conditional compilation. This dynamic parameter enables you to set up name-value pairs. The names (called flag names) can then be referenced in inquiry directives. `PLSQL_CCFLAGS` provides a mechanism that allows PL/SQL programmers to control conditional compilation of each PL/SQL library unit independently.

Values

- `vi`: Has the form of an unquoted PL/SQL identifier. It is unrestricted and can be a reserved word or a keyword. The text is not case sensitive. Each one is known as a flag or flag name. Each `vi` can occur more than once in the string, each occurrence can have a different flag value, and the flag values can be of different kinds.
- `ci`: Can be any of the following:
 - A PL/SQL Boolean literal
 - A `PLS_INTEGER` literal
 - The literal `NULL` (default). The text is not case sensitive. Each one is known as a flag value and corresponds to a flag name.

Displaying the PLSQL_CCFLAGS Initialization Parameter Setting

```
SELECT name, type, plsql_ccflags  
FROM user_plsql_object_settings
```

NAME	TYPE	PLSQL_CCFLAGS
1 DEPT_PKG	PACKAGE	(null)
2 DEPT_PKG	PACKAGE BODY	(null)
3 TAXES_PKG	PACKAGE	(null)
4 TAXES_PKG	PACKAGE BODY	(null)
5 EMP_PKG	PACKAGE	(null)
6 EMP_PKG	PACKAGE BODY	(null)
7 SECURE_DML	PROCEDURE	(null)
8 SECURE_EMPLOYEES	TRIGGER	(null)
9 ADD_JOB_HISTORY	PROCEDURE	plsql_ccflags:true,debug:true,debug:0
10 UPDATE_JOB_HISTORY	TRIGGER	(null)

...

461

Use the USER | ALL | DBA_PLSQL_OBJECT_SETTINGS data dictionary views to display the settings of a PL/SQL object.

You can define any allowable value for PLSQL_CCFLAGS. However, Oracle recommends that this parameter be used for controlling the conditional compilation of debugging or tracing code.

The flag names can be set to any identifier, including reserved words and keywords. The values must be the literals TRUE, FALSE, or NULL, or a PLS_INTEGER literal. The flag names and values are not case sensitive. The PLSQL_CCFLAGS parameter is a PL/SQL compiler parameter (like other compiler parameters) and is stored with the PL/SQL program unit. Consequently, if the PL/SQL program gets recompiled later with the REUSE SETTINGS clause (example, ALTER PACKAGE ...REUSE SETTINGS), then the same value of PLSQL_CCFLAGS is used for the recompilation. Because the PLSQL_CCFLAGS parameter can be set to a different value for each PL/SQL unit, it provides a convenient method for controlling conditional compilation on a per unit basis.

The PLSQL_CCFLAGS Parameter and the Inquiry Directive: Example

```
ALTER SESSION SET PLSQL_CCFLAGS = 'Tracing:true';
CREATE OR REPLACE PROCEDURE P IS
BEGIN
    $IF $$tracing $THEN
        DBMS_OUTPUT.PUT_LINE ('TRACING');
    $END
END P;
```

```
ALTER SESSION SET succeeded.
PROCEDURE P Compiled.
```

```
SELECT name, plsql_ccflags
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE name = 'P';
```

```
Results:
```

NAME	PLSQL_CCFLAGS
P	Tracingtrue

462

In the example in the slide, the parameter is set and then the procedure is created. The setting is stored with each PL/SQL unit.

Using Conditional Compilation Error Directives to Raise User-Defined Errors

```
$ERROR varchar2_static_expression $END
```

```
ALTER SESSION SET Plsql_CCFlags = ' Trace_Level:3 '
/ CREATE PROCEDURE P IS
BEGIN
  $IF    $$Trace_Level = 0 $THEN ...
  $ELSIF $$Trace_Level = 1 $THEN ...
  $ELSIF $$Trace_Level = 2 $THEN ...
  $Else $error 'Bad: '||$$Trace_Level $END -- error
                                -- directive
$END -- selection directive ends
END P;
```

```
SHOW ERRORS
Errors for PROCEDURE P:
LINE/COL ERROR
-----
6/9      PLS-00179: $ERROR: Bad: 3
```

463

The \$ERROR error directive raises a user-defined error and is of the form:

```
$ERROR varchar2_static_expression $END
```

Note: varchar2_static_expression must be a VARCHAR2 static expression.``

- o Boolean static expressions:
 - TRUE, FALSE, NULL, IS NULL, IS NOT NULL
 - >, <, >=, <=, =, <>, NOT, AND, OR
- o PLS_INTEGER static expressions:
 - -2147483648 to 2147483647, NULL
- o VARCHAR2 static expressions include:
 - ||, NULL, TO_CHAR
- o Static constants:

```
static_constant CONSTANT datatype := static_expression;
```

464

As described earlier, a preprocessor processes conditional directives before proper compilation begins. Consequently, only expressions that can be fully evaluated at compile time are permitted in conditional compilation directives. Any expression that contains references to variables or functions that require the execution of the PL/SQL are not available during compilation and cannot be evaluated.

This subset of PL/SQL expressions allowed in conditional compilation directives is referred to as static expressions. Static expressions are carefully defined to guarantee that if a unit is automatically recompiled without any changes to the values it depends on, the expressions evaluate in the same way and the same source is compiled.

Generally, static expressions are composed of three sources:

Inquiry directives marked with \$\$

Constants defined in PL/SQL packages such as DBMS_DB_VERSION.

These values can be combined and compared using the ordinary operations of PL/SQL.

Literals such as TRUE, FALSE, 'CA', 123, NULL

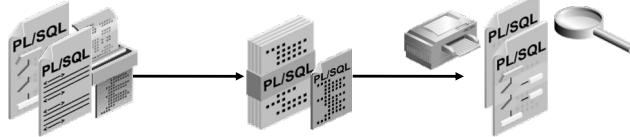
Static expressions can also contain operations that include comparisons, logical Boolean operations (such as OR and AND), or concatenations of static

character expression.

Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text

```
CALL  
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE('PACKAGE'  
, 'ORA61', 'MY_PKG');
```

```
CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE('PACKAGE', succeeded.  
PACKAGE my_pkg AS  
SUBTYPE my_real IS  
-- Check the database version, if >= 10g, use BINARY_DOUBLE data type,  
-- else use NUMBER data type  
--  
BINARY_DOUBLE;  
my_pi my_real; my_e my_real;  
END my_pkg;
```



465

DBMS_PREPROCESSOR subprograms print or retrieve the postprocessed source text of a PL/SQL unit after processing the conditional compilation directives. This postprocessed text is the actual source used to compile a valid PL/SQL unit. The example in the slide shows how to print the postprocessed form of my_pkg using the PRINT_POST_PROCESSED_SOURCE procedure.

When my_pkg is compiled on an Oracle Database 10g release or later database using the HR account, the resulting output is shown in the slide above.

The PRINT_POST_PROCESSED_SOURCE removes unselected text. The lines of code that are not included in the postprocessed text are removed. The arguments for the PRINT_POST_PROCESSED_SOURCE procedure are: object type, schema name (using student account ORA61), and object name.

Note: For additional information about the DBMS_PREPROCESSOR package, refer to *Oracle Database PL/SQL Packages and Types Reference*.

- Obfuscation (or wrapping) of a PL/SQL unit is the process of hiding the PL/SQL source code.
- Wrapping can be done with the wrap utility and DBMS_DDL subprograms.
- The Wrap utility is run from the command line and it processes an input SQL file, such as a SQL*Plus installation script.
- The DBMS_DDL subprograms wrap a single PL/SQL unit, such as a single CREATE PROCEDURE command, that has been generated dynamically.

Note: For additional information about obfuscation, refer to *Oracle Database PL/SQL Language Reference*.



Benefits of Obfuscating

- It prevents others from seeing your source code.
- Your source code is not visible through the `USER_SOURCE`, `ALL_SOURCE`, or `DBA_SOURCE` data dictionary views.
- SQL*Plus can process the obfuscated source files.
- The Import and Export utilities accept wrapped files.

Obfuscated PL/SQL Code: Example

```
BEGIN -- ALL_SOURCE view family obfuscates source code
DBMS_DDL.CREATE_WRAPPED (
    CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('I am wrapped now');
END P1;
);
END;
/
CALL p1();
```

```
anonymous block completed
p1 ) succeeded.
```

```
SELECT text FROM user_source
WHERE name = 'P1' ORDER BY line;
```

```
TEXT
-----
PROCEDURE P1 wrapped
a000000
b2
abcd
...
```

468

In the example in the slide, the DBMS_DDL.CREATE_WRAPPED package procedure is used to create the procedure P1.

The code is obfuscated when you use any of the views from the ALL_SOURCE view family to display the procedure's code as shown on the next page. When you check the *_SOURCE views, the source is wrapped, or hidden, so that others cannot view the code details as shown in the output of the command in the slide.

Dynamic Obfuscation: Example

```
SET SERVEROUTPUT ON

DECLARE
c_code CONSTANT VARCHAR2(32767) :=
' CREATE OR REPLACE PROCEDURE new_proc AS
  v_VDATE DATE;
BEGIN
  v_VDATE := SYSDATE;
  DBMS_OUTPUT.PUT_LINE(v_VDATE) ;
END; '
BEGIN
  DBMS_DDL.CREATE_WRAPPED (c_CODE);
END;
/
```

469

The example in the slide displays the creation of a dynamically obfuscated procedure called NEW_PROC. To verify that the code for NEW_PROC is obfuscated, you can query from the

DBA | ALL | USER_SOURCE dictionary views as shown below:

```
SELECT text FROM user_source
WHERE name = 'NEW_PROC';
```

TEXT

```
-----  
PROCEDURE new_proc wrapped  
a000000
```

```
?  
71 9e
```

```
hBWMPGeSsd58b4jCP3/0d04rof0wg5nnm7+fMr2ywFyFDGLQlhaXriu4dCuPCWnnx1J0Ulxp  
pvc8nsr7Seq/riQvHRsXAQovdohOK6ZvM1Kbskr+KLK957KzHQYwLK4k6rJLC55EyJ7qJB/2  
RDmm3j79Uw==
```

- The PL/SQL wrapper is a stand-alone utility that hides application internals by converting PL/SQL source code into portable object code.
- Wrapping has the following features:
 - Platform independence
 - Dynamic loading
 - Dynamic binding
 - Dependency checking
 - Normal importing and exporting when invoked

470

PL/SQL Wrapper

The PL/SQL wrapper is a stand-alone utility that converts PL/SQL source code into portable object code. Using it, you can deliver PL/SQL applications without exposing your source code, which may contain proprietary algorithms and data structures. The wrapper converts the readable source code into unreadable code. By hiding application internals, it prevents misuse of your application.

Wrapped code, such as PL/SQL stored programs, has several features:

- It is platform independent, so you do not need to deliver multiple versions of the same compilation unit.
- It permits dynamic loading, so users need not shut down and restart to add a new feature.
- It permits dynamic binding, so external references are resolved at load time.
- It offers strict dependency checking, so that invalidated program units are recompiled automatically when they are invoked.
- It supports normal importing and exporting, so the import/export utility can process wrapped files.

Running the Wrapper Utility

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

- Do not use spaces around the equal signs.
- The INAME argument is required.
- The default extension for the input file is .sql, unless it is specified with the name.
- The ONAME argument is optional.
- The default extension for output file is .plb, unless specified with the ONAME argument.

Examples

```
WRAP INAME=demo_04_hello.sql  
WRAP INAME=demo_04_hello  
WRAP INAME=demo_04_hello.sql ONAME=demo_04_hello.plb
```

471

Running the Wrapper

The wrapper is an operating system executable called WRAP. This is a 10c only code.

To run the wrapper, enter the following command at your operating system prompt:

```
WRAP INAME=input_file_name  
[ONAME=output_file_name]
```

Each of the examples shown in the slide takes a file called demo_04_hello.sql as input and creates an output file called demo_04_hello.plb.

After the wrapped file is created, execute the .plb file from SQL*Plus to compile and store the wrapped version of the source code, as you would execute SQL script files.

Note

Only the INAME argument is required. If the ONAME argument is not specified, then the output file acquires the same name as the input file with an extension of .plb.

The input file can have any extension, but the default is .sql.

Case sensitivity of the INAME and ONAME values depends on the

operating system.

Generally, the output file is much larger than the input file.

Do not put spaces around the equal signs in the INAME and ONAME arguments and values.

Results of Wrapping

```
-- Original PL/SQL source code in input file:  
  
CREATE PACKAGE banking IS  
    min_bal := 100;  
    no_funds EXCEPTION;  
    ...  
END banking;  
/
```

```
-- Wrapped code in output file:  
  
CREATE PACKAGE banking  
    wrapped  
012abc463e ...  
/
```

472

When it is wrapped, an object type, package, or subprogram has the following form: header, followed by the word `wrapped`, followed by the encrypted body.

The input file can contain any combination of SQL statements. However, the PL/SQL wrapper wraps only the following CREATE statements:

- `CREATE [OR REPLACE] TYPE`
- `CREATE [OR REPLACE] TYPE BODY`
- `CREATE [OR REPLACE] PACKAGE`
- `CREATE [OR REPLACE] PACKAGE BODY`
- `CREATE [OR REPLACE] FUNCTION`
- `CREATE [OR REPLACE] PROCEDURE`

All other SQL CREATE statements are passed intact to the output file.

- You must wrap only the package body, not the package specification.
- The wrapper can detect syntactic errors but cannot detect semantic errors.
- The output file should not be edited. You maintain the original source code and wrap again as required.
- To ensure that all the important parts of your source code are obfuscated, view the wrapped file in a text editor before distributing it.

473

Guidelines include the following:

When wrapping a package or object type, wrap only the body, not the specification. Thus, you give other developers the information that they need to use the package without exposing its implementation. If your input file contains syntactic errors, the PL/SQL wrapper detects and reports them. However, the wrapper cannot detect semantic errors because it does not resolve external references. For example, the wrapper does not report an error if the table or view `amp` does not exist:

```
CREATE PROCEDURE raise_salary (emp_id INTEGER,  
amount NUMBER) AS  
BEGIN  
    UPDATE amp -- should be emp  
        SET sal = sal + amount WHERE empno = emp_id;  
END;
```

However, the PL/SQL compiler resolves external references. Therefore, semantic errors are reported when the wrapper output file (.plb file) is compiled.

Because its contents are not readable, the output file should not be

edited. To change a wrapped object, you need to modify the original source code and wrap the code again.

DBMS_DDL Package Versus the Wrap Utility

Functionality	DBMS_DDL	Wrap Utility
Code obfuscation	Yes	Yes
Dynamic Obfuscation	Yes	No
Obfuscate multiple programs at a time	No	Yes

474

DBMS_DDL Versus the Wrap Utility

Both the Wrap utility and the DBMS_DDL package have distinct uses:

The Wrap utility is useful for obfuscating multiple programs with one execution of the utility. In essence, a complete application may be wrapped. However, the Wrap utility cannot be used to obfuscate dynamically generated code at run time. The Wrap utility processes an input SQL file and obfuscates only the PL/SQL units in the file, such as:

Package specification and body

Function and procedure

Type specification and body

The Wrap utility does not obfuscate PL/SQL content in:

Anonymous blocks

Triggers

Non-PL/SQL code

The DBMS_DDL package is intended to obfuscate a dynamically generated program unit from within another program unit. The DBMS_DDL package methods cannot obfuscate multiple program units at one execution. Each execution of these methods accepts only one CREATE OR REPLACE statement at a time as argument.

Topics covered in this module:

- Describe and use conditional compilation
- Hide PL/SQL source code using dynamic obfuscation and the Wrap utility

475

This lesson introduced the conditional compilation and obfuscating (or wrapping) of PL/SQL code.



Oracle PL/SQL 19c



Module 20: Managing Dependencies

Topics covered in this module:

- Track procedural dependencies
- Predict the effect of changing a database object on procedures and functions
- Manage procedural dependencies

477

This lesson introduces you to object dependencies and implicit and explicit recompilation of invalid objects.

Overview of Schema Object Dependencies

Object Type	Can Be Dependent or Referenced
Package body	Dependent only
Package specification	Both
Sequence	Referenced only
Subprogram	Both
Synonym	Both
Table	Both
Trigger	Both
User-defined object	Both
User-defined collection	Both
View	Both

478

Dependent and Referenced Objects

Some types of schema objects can reference other objects in their definitions. For example, a view is defined by a query that references tables or other views, and the body of a subprogram can include SQL statements that reference other objects. If the definition of object A references object B, then A is a dependent object (with respect to B) and B is a referenced object (with respect to A).

Dependency Issues

If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, the procedure may or may not continue to work without error.

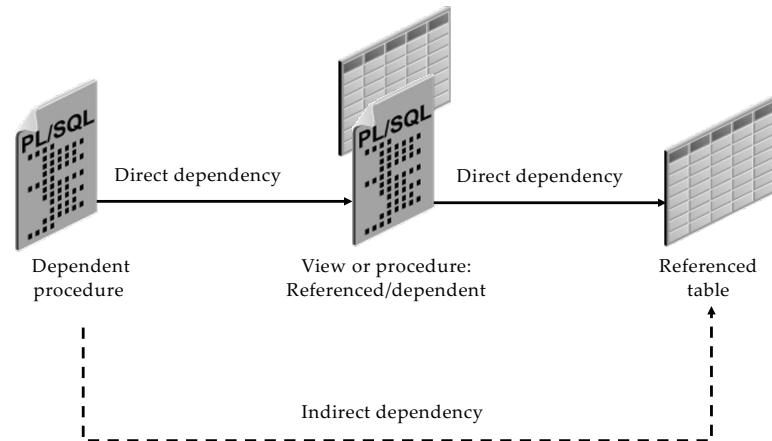
The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary, and you can view the status in the `USER_OBJECTS` data dictionary view.

If the status of a schema object is `VALID`, then the object has been compiled and can be immediately used when referenced.

If the status of a schema object is `INVALID`, then the schema object

must be compiled before it can be used.

Dependencies



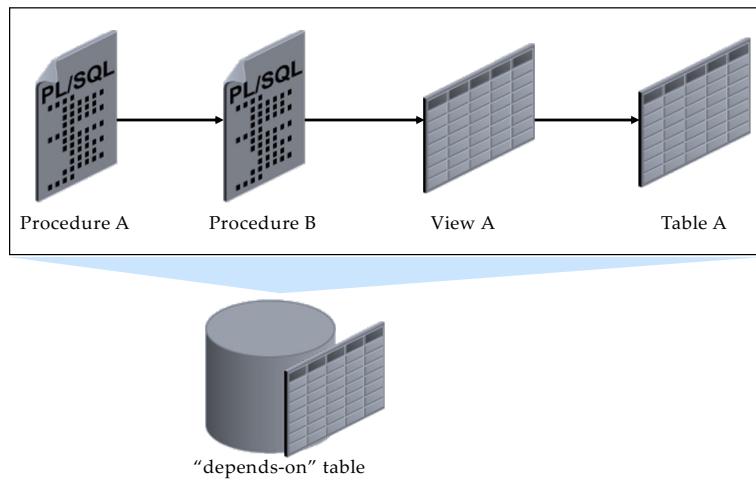
479

Dependent and Referenced Objects (continued)

A procedure or function can directly or indirectly (through an intermediate view, procedure, function, or packaged procedure or function) reference the following objects:

- Tables
- Views
- Sequences
- Procedures
- Functions
- Packaged procedures or functions

Direct Local Dependencies



480

Managing Local Dependencies

In the case of local dependencies, the objects are on the same node in the same database. The Oracle server automatically manages all local dependencies, using the database's internal “depends-on” table. When a referenced object is modified, the dependent objects are sometimes invalidated. The next time an invalidated object is called, the Oracle server automatically recompiles it.

If you alter the definition of a referenced object, dependent objects might or might not continue to function without error, depending on the type of alteration. For example, if you drop a table, no view based on the dropped table is usable.

Starting with Oracle Database 10g, the CREATE OR REPLACE SYNONYM command has been enhanced to minimize the invalidations to dependent PL/SQL program units and views that reference it. This is covered later in this lesson.

Starting with Oracle Database 19c, dependencies are tracked at the level of element within unit. This is referred to as fine-grained dependency. Fine-grained dependencies are covered later in this lesson.

Querying Direct Object Dependencies: Using the USER_DEPENDENCIES View

```
DESC User_dependencies
Name          Null    Type
-----
NAME          NOT NULL VARCHAR2(30)
TYPE          VARCHAR2(17)
REFERENCED_OWNER  VARCHAR2(30)
REFERENCED_NAME  VARCHAR2(64)
REFERENCED_TYPE  VARCHAR2(17)
REFERENCED_LINK_NAME  VARCHAR2(128)
SCHEMADID      NUMBER
DEPENDENCY_TYPE  VARCHAR2(4)

8 rows selected
```

```
SELECT name, type, referenced_name, referenced_type
FROM user_dependencies
WHERE referenced_name IN ('EMPLOYEES', 'EMP_VW');
```

Results:				
#	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
1	QUERY_EMP	PROCEDURE	EMPLOYEES	TABLE
2	DML_CALL_SQL	FUNCTION	EMPLOYEES	TABLE
3	QUERY_CALL_SQL	FUNCTION	EMPLOYEES	TABLE
4	COMM_PKG	PACKAGE BODY	EMPLOYEES	TABLE
5	CURS_PKG	PACKAGE BODY	EMPLOYEES	TABLE
6	EMP_PKG	PACKAGE	EMPLOYEES	TABLE
• 7	EMP_PKG	PACKAGE BODY	EMPLOYEES	TABLE

481

You can determine which database objects to recompile manually by displaying direct dependencies from the `USER_DEPENDENCIES` data dictionary view.

The `ALL_DEPENDENCIES` and `DBA_DEPENDENCIES` views contain the additional `OWNER` column, which references the owner of the object.

The `USER_DEPENDENCIES` Data Dictionary View Columns

The columns of the `USER_DEPENDENCIES` data dictionary view are as follows:

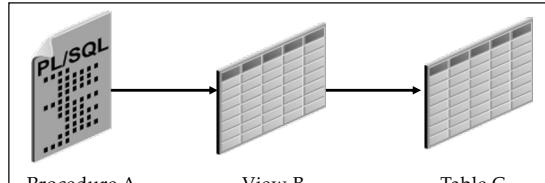
- **NAME:** The name of the dependent object
- **TYPE:** The type of the dependent object (PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, or VIEW)
- **REFERENCED_OWNER:** The schema of the referenced object
- **REFERENCED_NAME:** The name of the referenced object
- **REFERENCED_TYPE:** The type of the referenced object
- **REFERENCED_LINK_NAME:** The database link used to access the referenced object

Every database object has one of the following status values:

Status	Description
VALID	The object was successfully compiled, using the current definition in the data dictionary.
COMPILED WITH ERRORS	The most recent attempt to compile the object produced errors.
INVALID	The object is marked invalid because an object that it references has changed. (Only a dependent object can be invalid.)
UNAUTHORIZED	An access privilege on a referenced object was revoked. (Only a dependent object can be unauthorized.)

Every database object has one of the status values shown in the table in the slide.

Note: The USER_OBJECTS, ALL_OBJECTS, and DBA_OBJECTS static data dictionary views do not distinguish between Compiled with errors, Invalid, and Unauthorized; instead, they describe all these as INVALID.



- o Procedure A depends on Table C.
View B is a direct dependent of Table C.
Procedure A is an indirect dependent of Table C.
- o Direct dependents are invalidated only by changes to the referenced object that affect them.
- o Indirect dependents can be invalidated by changes to the reference object that do not affect them.

483

If object A depends on object B, which depends on object C, then A is a direct dependent of B, B is a direct dependent of C, and A is an indirect dependent of C.

In Oracle Database 19c, direct dependents are invalidated only by changes to the referenced object that affect them (changes to the signature of the referenced object).

Indirect dependents can be invalidated by changes to the reference object that do not affect them: If a change to Table C invalidates View B, it invalidates Procedure A (and all other direct and indirect dependents of View B). This is called cascading invalidation.

Assume that the structure of the table on which a view is based is modified. When you describe the view by using the SQL*Plus DESCRIBE command, you get an error message that states that the object is invalid to describe. This is because the command is not a SQL command; at this stage, the view is invalid because the structure of its base table is changed. If you query the view now, then the view is recompiled automatically and you can see the result if it is successfully recompiled.

Schema Object Change That Invalidates Some Dependents: Example

```
CREATE VIEW commissioned AS  
SELECT first_name, last_name, commission_pct FROM employees  
WHERE commission_pct > 0.00;
```

```
CREATE VIEW six_figure_salary AS  
SELECT * FROM employees  
WHERE salary >= 100000;
```

```
SELECT object_name, status  
FROM user_objects  
WHERE object_type = 'VIEW';
```

OBJECT_NAME	STATUS
1 SIX_FIGURE_SALARY	VALID
2 DEPTREE	VALID
3 EMP_DETAILS_VIEW	VALID
4 EMP_DETAILS	VALID
5 DEPTREE	VALID
6 COMMISSIONED	VALID

484

The example in the slide demonstrates an example of a schema object change that invalidates some dependents but not others. The two newly created views are based on the EMPLOYEES table in the HR schema. The status of the newly created views is VALID.

Schema Object Change That Invalidates Some Dependents: Example

```
ALTER TABLE employees MODIFY email VARCHAR2(50);

SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW';
```

Results:	
OBJECT_NAME	STATUS
1 EMP_DETAILS	VALID
2 COMMISSIONED	VALID
3 SIX FIGURE SALARY	INVALID
4 EMP_DETAILS_VIEW	VALID

485

Suppose you determine that the EMAIL column in the EMPLOYEES table needs to be lengthened from 25 to 50, you alter the table as shown in the slide above.

Because the COMMISSIONED view does not include the EMAIL column in its select list, it is not invalidated. However, the SIXFIGURES view is invalidated because all columns in the table are selected.

Displaying Direct and Indirect Dependencies

1. Run the `utldtree.sql` script that creates the objects that enable you to display the direct and indirect dependencies.

```
@/home/oracle/labs/plpu/labs/utldtree.sql
```

2. Execute the `DEPTREE_FILL` procedure.

```
EXECUTE deptree_fill('TABLE', 'ORA61', 'EMPLOYEES')
```

486

Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle

Display direct and indirect dependencies from additional user views called `DEPTREE` and `I_DEPTREE`; these views are provided by Oracle.

Example

1. Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/labs/plpu/labs` folder. You can run the script as follows:

```
@?/labs/plpu/labs/utldtree.sql
```

Note: In this class, this script is supplied in the `labs` folder of your class files. The code example above uses the student account `ORA61`. (This applies to a Linux environment. If the file is not found, locate the file in your `labs` subdirectory.)

2. Populate the `DEPTREE_TEMPTAB` table with information for a particular referenced object by invoking the `DEPTREE_FILL` procedure. There are three parameters for this procedure:

object	<i>object_type</i>	Type of the referenced
object	<i>object_owner</i>	Schema of the referenced
object	<i>object_name</i>	Name of the referenced

Displaying Dependencies Using the DEPTREE View

```
SELECT nested_level, type, name
FROM deptree
ORDER BY seq#;
```

NESTED_LEVEL	TYPE	NAME
0	TABLE	EMPLOYEES
1	VIEW	EMP_DETAILS_VIEW
1	TRIGGER	SECURE_EMPLOYEES
1	TRIGGER	UPDATE_JOB_HISTORY
1	PROCEDURE	RAISE_SALARY
2	PROCEDURE	PROCESS_EMPLOYEES
1	PROCEDURE	QUERY_EMP
1	PROCEDURE	PROCESS_EMPLOYEES
1	FUNCTION	DML_CALL_SQL
1	FUNCTION	QUERY_CALL_SQL
1	PACKAGE BODY	COMM_PKG
1	PACKAGE BODY	CURS_PKG
1	PACKAGE	EMP_PKG
2	PACKAGE BODY	EMP_PKG
1	PACKAGE BODY	EMP_PKG
1	PROCEDURE	SAL_STATUS

...

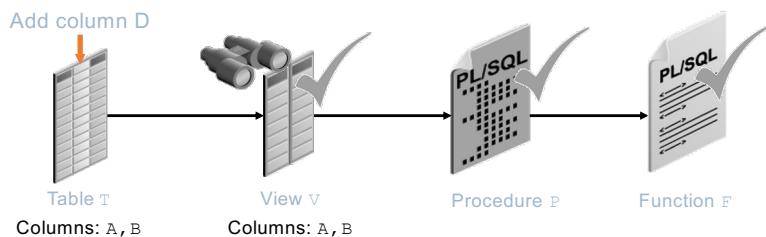
487

You can display a tabular representation of all dependent objects by querying the DEPTREE view. You can display an indented representation of the same information by querying the IDEPTREE view, which consists of a single column named DEPENDENCIES as follows:

```
SELECT *
FROM ideptree;
```

DEPENDENCIES
TRIGGER ORA61.SECURE_EMPLOYEES PROCEDURE ORA61.UPDATE_SALARY TRIGGER ORA61.AUDIT_EMP_VALUES PROCEDURE ORA61.EMP_LIST PROCEDURE ORA61.PROCESS_EMPLOYEES PACKAGE BODY ORA61.CURS_PKG PACKAGE BODY ORA61.EMP_PKG VIEW ORA61.EMP_DETAILS TRIGGER ORA61.NEW_EMP_DEPT TRIGGER ORA61.UPDATE_JOB_HISTORY PACKAGE BODY ORA61.EMP_PKG FUNCTION ORA61.EMP_HIRE_DATE PROCEDURE ORA61.SAL_STATUS TRIGGER ORA61.DERIVE_COMMISSION_PCT TRIGGER ORA61.CHECK_SALARY PROCEDURE ORA61.PROCESS_EMPLOYEES PROCEDURE ORA61.QUERY_EMP TRIGGER ORA61.RESTRICT_SALARY VIEW ORA61.COMMISSIONED

- Before 19c, adding column D to table T invalidated the dependent objects.
- Oracle Database 19c records additional, finer-grained dependency management:
 - Adding column D to table T does not impact view V and does not invalidate the dependent objects



488

Fine-Grained Dependencies

Starting with Oracle Database 19c, you have access to records that describe more precise dependency metadata. This is called fine-grained dependency and it enables you to see when the dependent objects are not invalidated without logical requirement.

Earlier Oracle Database releases record dependency metadata—for example, PL/SQL unit P depends on PL/SQL unit F, or that view V depends on table T—with the precision of the whole object. This means that dependent objects are sometimes invalidated without logical requirement. For example, if view V depends only on columns A and B in table T, and column D is added to table T, the validity of view V is not logically affected. Nevertheless, before Oracle Database Release 11.1, view V is invalidated by the addition of column D to table T. With Oracle Database Release 11.1, adding column D to table T does not invalidate view V. Similarly, if procedure P depends only on elements E1 and E2 within a package, adding element E99 to the package does not invalidate procedure P.

Reducing the invalidation of dependent objects in response to changes to the objects on which they depend increases application availability, both in the development environment and during online application upgrade.

- In Oracle Database 19c, dependencies are now tracked at the level of *element within unit*.
- Element-based dependency tracking covers the following:
 - Dependency of a single-table view on its base table
 - Dependency of a PL/SQL program unit (package specification, package body, or subprogram) on the following:
 - Other PL/SQL program units
 - Tables
 - Views

Fine-Grained Dependency Management: Example 1

```
CREATE TABLE t2 (col_a NUMBER, col_b NUMBER, col_c NUMBER);
CREATE VIEW v AS SELECT col_a, col_b FROM t2;
```

```
SELECT ud.name, ud.type, ud.referenced_name,
       ud.referenced_type, uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:				
	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
1	V	VIEW	T2	TABLE

```
ALTER TABLE t2 ADD (col_d VARCHAR2(20));
```

```
SELECT ud.name, ud.type, ud.referenced_name,
       ud.referenced_type, uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:				
	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
1	V	VIEW	T2	TABLE

490

Example of Dependency of a Single-Table View on Its Base Table

In the first example in the slide, table T2 is created with three columns: COL_A, COL_B, and COL_C. A view named V is created based on columns COL_A and COL_B of table T2. The dictionary views are queried and the view V is dependent on table T and its status is valid.

In the third example, table T2 is altered. A new column named COL_D is added. The dictionary views still report that the view V is dependent because element-based dependency tracking realizes that the columns COL_A and COL_B are not modified and, therefore, the view does not need to be invalidated.

Fine-Grained Dependency Management: Example 1

```
ALTER TABLE t2 MODIFY (col_a VARCHAR2(20));
SELECT ud.name, ud.referenced_name, ud.referenced_type,
       uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:				
	NAME	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1	V	T2	TABLE	INVALID

491

In the example in the slide, the view is invalidated because its element (COL_A) is modified in the table on which the view is dependent.

Fine-Grained Dependency Management: Example 2

```
CREATE PACKAGE pkg IS
    PROCEDURE proc_1;
END pkg;
/
CREATE OR REPLACE PROCEDURE p IS
BEGIN
    pkg.proc_1();
END p;
/
CREATE OR REPLACE PACKAGE pkg
IS
    PROCEDURE proc_1;
    PROCEDURE unheard_of;
END pkg;
/
```

```
PACKAGE pkg Compiled.
PROCEDURE p Compiled.
PACKAGE pkg Compiled.
```

492

In the example in the slide, you create a package named PKG that has procedure PROC_1 declared.

A procedure named P invokes PKG.PROC_1.

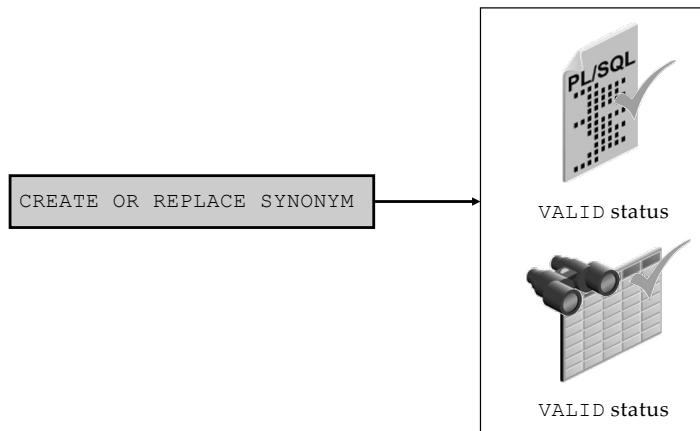
The definition of the PKG package is modified and another subroutine is added to the package declaration.

When you query the USER_OBJECTS dictionary view for the status of the P procedure, it is still valid as shown because the element you added to the definition of PKG is not referenced through procedure P.

```
SELECT status FROM user_objects
WHERE object_name = 'P';
```

Results:	
	STATUS
1	VALID

Changes to Synonym Dependencies



493

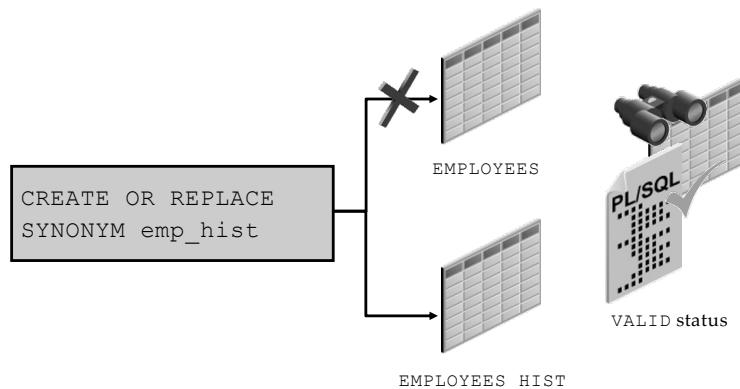
The Oracle Database minimizes down time during code upgrades or schema merges.

When certain conditions on columns, privileges, partitions, and so on are met, a table or object type is considered equivalent and dependent objects are no longer invalidated.

In Oracle Database 10g, the `CREATE OR REPLACE SYNONYM` command has been enhanced to minimize the invalidations to dependent PL/SQL program units and views that reference it. This eliminates the need for time-consuming recompilation of the program units after redefinition of the synonyms or during execution. You do not have to set any parameters or issue any special commands to enable this functionality; invalidations are minimized automatically.

Note: This enhancement applies only to synonyms pointing to tables.

Maintaining Valid PL/SQL Program Units and Views



494

Maintaining Valid PL/SQL Program Units

Starting with Oracle Database 10g Release 2, you can change the definition of a synonym, and the dependent PL/SQL program units are not invalidated under the following conditions:

The column order, column names, and column data types of the tables are identical.

The privileges on the newly referenced table and its columns are a superset of the set of privileges on the original table. These privileges must not be derived through roles alone.

The names and types of partitions and subpartitions are identical.

The tables are of the same organization type.

Object type columns are of the same type.

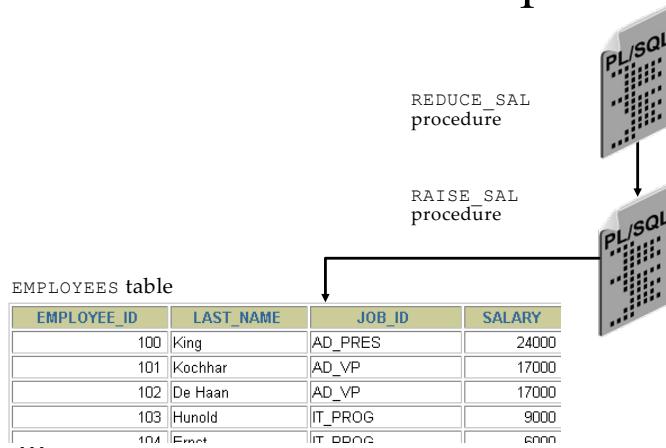
Maintaining Valid Views: As with dependent PL/SQL program units, you can change the definition of a synonym, and the dependent views are not invalidated under the conditions listed in the preceding paragraph. In addition, the following must be true to preserve the VALID status of dependent views, but not of dependent PL/SQL program units, when you redefine a synonym:

Columns and order of columns defined for primary key and unique indexes, NOT NULL constraints, and primary key and unique

constraints must be identical.

The dependent view cannot have any referential constraints.

Another Scenario of Local Dependencies



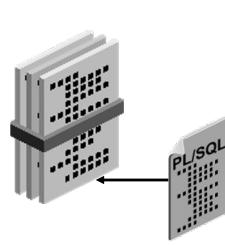
495

The example in the slide illustrates the effect that a change in the definition of a procedure has on the recompilation of a dependent procedure. Assume that the `RAISE_SAL` procedure updates the `EMPLOYEES` table directly, and that the `REDUCE_SAL` procedure updates the `EMPLOYEES` table indirectly by way of `RAISE_SAL`.

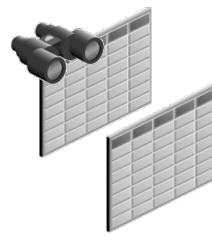
If the internal logic of the `RAISE_SAL` procedure is modified, `REDUCE_SAL` will successfully recompile if `RAISE_SAL` has successfully compiled.

If the formal parameters for the `RAISE_SAL` procedure are eliminated, `REDUCE_SAL` will not successfully recompile.

To reduce invalidation of dependent objects:



Add new items to the end of the package



Reference each table through a view

496

Add New Items to End of Package

When adding new items to a package, add them to the end of the package. This preserves the slot numbers and entry-point numbers of existing top-level package items, preventing their invalidation. For example, consider the following package:

```
CREATE OR REPLACE PACKAGE pkg1 IS
  FUNCTION get_var RETURN VARCHAR2;
  PROCEDURE set_var (v VARCHAR2);
END;
```

Adding an item to the end of `pkg1` does not invalidate dependents that reference `get_var`. Inserting an item between the `get_var` function and the `set_var` procedure invalidates dependents that reference the `set_var` function.

Reference Each Table Through a View

Reference tables indirectly, using views. This allows you to do the following:

- Add columns to the table without invalidating dependent views or dependent PL/SQL objects
- Modify or delete columns not referenced by the view without invalidating dependent objects

- An object that is not valid when it is referenced must be validated before it can be used.
- Validation occurs automatically when an object is referenced; it does not require explicit user action.
- If an object is not valid, its status is either COMPILED WITH ERRORS, UNAUTHORIZED, or INVALID.

497

The compiler cannot automatically revalidate an object that compiled with errors.

The compiler recompiles the object, and if it recompiles without errors, it is revalidated; otherwise, it remains invalid.

The compiler checks whether the unauthorized object has access privileges to all of its referenced objects. If so, the compiler revalidates the unauthorized object without recompiling it. If not, the compiler issues appropriate error messages.

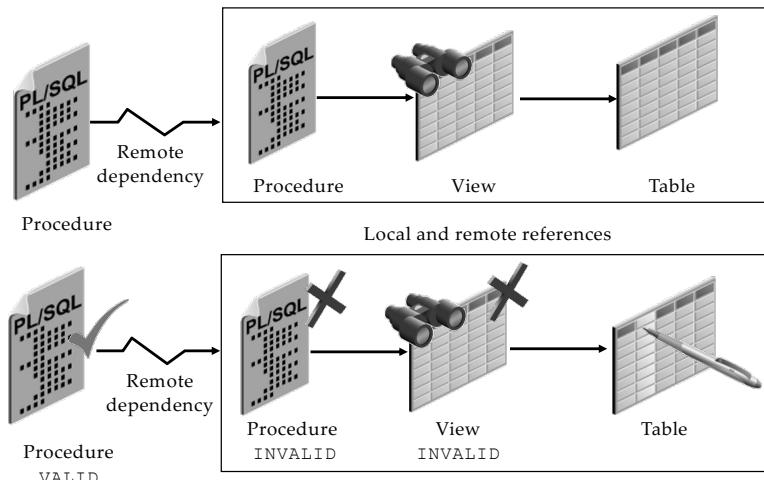
The SQL compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid.

For an invalid PL/SQL program unit (procedure, function, or package), the PL/SQL compiler checks whether any referenced object changed in a way that affects the invalid object.

If so, the compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid. If not, the compiler revalidates the invalid object without recompiling it.

If not, the compiler revalidates the invalid object without recompiling it. Fast revalidation is usually performed on objects that were invalidated due to cascading invalidation.

Remote Dependencies



498

In the case of remote dependencies, the objects are on separate nodes. The Oracle server does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies (including functions, packages, and triggers). The local stored procedure and all its dependent objects are invalidated but do not automatically recompile when called for the first time.

Recompilation of Dependent Objects: Local and Remote

Verify successful explicit recompilation of the dependent remote procedures and implicit recompilation of the dependent local procedures by checking the status of these procedures within the `USER_OBJECTS` view.

If an automatic implicit recompilation of the dependent local procedures fails, the status remains invalid and the Oracle server issues a run-time error. Therefore, to avoid disrupting production, it is strongly recommended that you recompile local dependent objects manually, rather than relying on an automatic mechanism.

Remote dependencies are governed by the mode that is chosen by the user:



TIMESTAMP checking



SIGNATURE checking

499

TIMESTAMP Checking

Each PL/SQL program unit carries a time stamp that is set when it is created or recompiled. Whenever you alter a PL/SQL program unit or a relevant schema object, all its dependent program units are marked as invalid and must be recompiled before they can execute. The actual time stamp comparison occurs when a statement in the body of a local procedure calls a remote procedure.

SIGNATURE Checking

For each PL/SQL program unit, both the time stamp and the signature are recorded. The signature of a PL/SQL construct contains information about the following:

- The name of the construct (procedure, function, or package)
- The base types of the parameters of the construct
- The modes of the parameters (IN, OUT, or IN OUT)
- The number of the parameters

The recorded time stamp in the calling program unit is compared with the current time stamp in the called remote program unit. If the time stamps match, the call proceeds. If they do not match, the remote procedure call (RPC) layer performs a simple comparison of the signature to determine

whether the call is safe or not. If the signature has not been changed in an incompatible manner, execution continues; otherwise, an error is returned.

Setting the *REMOTE_DEPENDENCIES_MODE* Parameter

- As an *init.ora* parameter:
`REMOTE_DEPENDENCIES_MODE = TIMESTAMP | SIGNATURE`
- At the system level:
`ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE = TIMESTAMP | SIGNATURE`
- At the session level:
`ALTER SESSION SET
REMOTE_DEPENDENCIES_MODE = TIMESTAMP |
SIGNATURE`

500

REMOTE_DEPENDENCIES_MODE Parameter

Setting the *REMOTE_DEPENDENCIES_MODE*

value TIMESTAMP
 SIGNATURE

Specify the value of the *REMOTE_DEPENDENCIES_MODE* parameter using one of the three methods described in the slide.

Note: The calling site determines the dependency model.

Recompilation:

- Is handled automatically through implicit run-time recompilation
- Is handled through explicit recompilation with the ALTER statement

```
ALTER PROCEDURE [SCHEMA.]procedure_name COMPILE;
```

```
ALTER FUNCTION [SCHEMA.]function_name COMPILE;
```

```
ALTER PACKAGE [SCHEMA.]package_name  
COMPILE [PACKAGE | SPECIFICATION | BODY];
```

```
ALTER TRIGGER trigger_name [COMPILE[DEBUG]] ;
```

501

Recompiling PL/SQL Objects

If the recompilation is successful, the object becomes valid. If not, the Oracle server returns an error and the object remains invalid. When you recompile a PL/SQL object, the Oracle server first recompiles any invalid object on which it depends.

Procedure: Any local objects that depend on a procedure (such as procedures that call the recompiled procedure or package bodies that define the procedures that call the recompiled procedure) are also invalidated.

Packages: The COMPILE PACKAGE option recompiles both the package specification and the body, regardless of whether it is invalid. The COMPILE SPECIFICATION option recompiles the package specification. Recompiling a package specification invalidates any local objects that depend on the specification, such as subprograms that use the package. Note that the body of a package also depends on its specification. The COMPILE BODY option recompiles only the package body.

Triggers: Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The DEBUG option instructs the PL/SQL compiler to generate and store the

code for use by the PL/SQL debugger.

Recompiling dependent procedures and functions is unsuccessful when:

- The referenced object is dropped or renamed
- The data type of the referenced column is changed
- The referenced column is dropped
- A referenced view is replaced by a view with different columns
- The parameter list of a referenced procedure is modified

Sometimes a recompilation of dependent procedures is unsuccessful (for example, when a referenced table is dropped or renamed).

The success of any recompilation is based on the exact dependency. If a referenced view is re-created, any object that is dependent on the view needs to be recompiled. The success of the recompilation depends on the columns that the view now contains, as well as the columns that the dependent objects require for their execution. If the required columns are not part of the new view, then the object remains invalid.

Recompiling dependent procedures and functions is successful if:

- The referenced table has new columns
- The data type of referenced columns has not changed
- A private table is dropped, but a public table that has the same name and structure exists
- The PL/SQL body of a referenced procedure has been modified and recompiled successfully

503

The recompilation of dependent objects is successful if:

New columns are added to a referenced table

All INSERT statements include a column list

No new column is defined as NOT NULL

When a private table is referenced by a dependent procedure and the private table is dropped, the status of the dependent procedure becomes invalid.

When the procedure is recompiled (either explicitly or implicitly) and a public table exists, the procedure can recompile successfully but is now dependent on the public table. The recompilation is successful only if the public table contains the columns that the procedure requires; otherwise, the status of the procedure remains invalid.

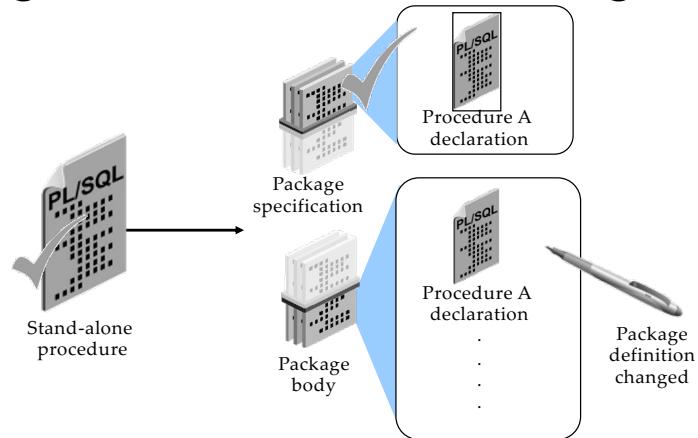
Minimize dependency failures by:

- Declaring records with the %ROWTYPE attribute
- Declaring variables with the %TYPE attribute
- Querying with the SELECT * notation
- Including a column list with INSERT statements

504

You can minimize recompilation failure by following the guidelines that are shown in the slide.

Packages and Dependencies: Subprogram References the Package



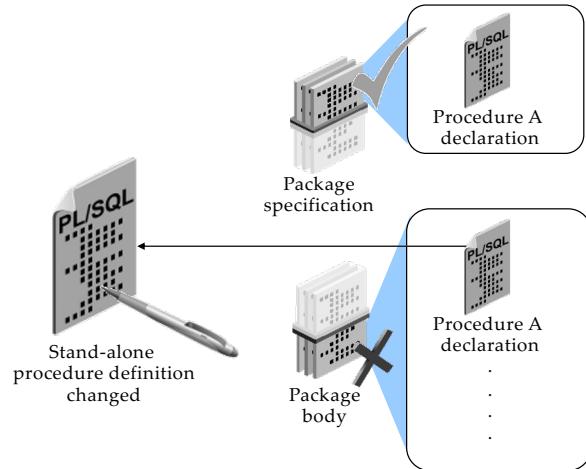
505

You can simplify dependency management with packages when referencing a package procedure or function from a stand-alone procedure or function.

If the package body changes and the package specification does not change, then the stand-alone procedure that references a package construct remains valid.

If the package specification changes, then the outside procedure referencing a package construct is invalidated, as is the package body.

Packages and Dependencies: Package Subprogram References Procedure



506

If a stand-alone procedure that is referenced within the package changes, then the entire package body is invalidated, but the package specification remains valid. Therefore, it is recommended that you bring the procedure into the package.

Topics covered in this module:

- Track procedural dependencies
- Predict the effect of changing a database object on procedures and functions
- Manage procedural dependencies

507

Avoid disrupting production by keeping track of dependent procedures and recompiling them manually as soon as possible after the definition of a database object changes.

