1

PL/SQL Programming Concepts: Review

ORACLE

Objectives

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

- **Identify PL/SQL block structure**
- **Describe PL/SQL basics**
- **List restrictions and guidelines on calling functions from SQL expressions**
- **Identify how explicit cursors are processed**
- **Handle exceptions**
- **Use the `raise_application_error` procedure**
- **Manage dependencies**

ORACLE

Objectives

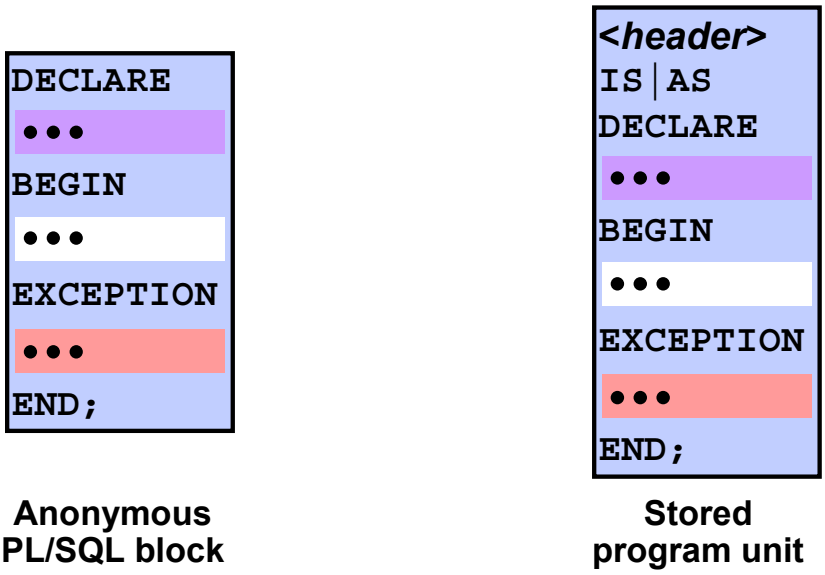
PL/SQL supports various programming constructs. This lesson reviews the basic concept of PL/SQL programming. This lesson also reviews how to:

- Create subprograms
- Use cursors
- Handle exceptions
- Identify predefined Oracle server errors
- Manage dependencies

A quiz at the end of the lesson will assess your knowledge of PL/SQL.

Note: The quiz is optional. Solutions to the quiz are provided in Appendix A.

PL/SQL Block Structure



Anonymous PL/SQL block

Stored program unit

ORACLE

PL/SQL Block Structure

An anonymous PL/SQL block structure consists of an optional `DECLARE` section, a mandatory `BEGIN-END` block, and an optional `EXCEPTION` section before the `END` statement of the main block.

A stored program unit has a mandatory header section. This section defines whether the program unit is a function, procedure, or a package. A stored program unit also has other sections mentioned for the anonymous PL/SQL block.

Every PL/SQL construct is made from one or more blocks. These blocks can be entirely separate, or nested within one another. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Naming Conventions

Advantages of proper naming conventions:

- **Easier to read**
- **Understandable**
- **Give information about the functionality**
- **Easier to debug**
- **Ensure consistency**

ORACLE

Naming Conventions

A proper naming convention makes the code easier to read and more understandable. It helps you understand the functionality of the identifier. If the code is written using proper naming conventions, you can easily find an error and rectify it. Most importantly, it ensures consistency among the code written by different developers.

The following table shows the naming conventions followed in this course:

Identifier	Convention	Example
Variable	`v_prefix`	`v_product_name`
Constant	`c_prefix`	`c_tax`
Parameter	`p_prefix`	`p_cust_id`
Exception	`e_prefix`	`e_check_credit_limit`
Cursor	`cur_prefix`	`cur_orders`
Type	`typ_prefix`	`typ_customer`

Procedures

A procedure is:

- **A named PL/SQL block that performs a sequence of actions**
- **Stored in the database as a schema object**
- **Used to promote reusability and maintainability**

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; …]
BEGIN
  -- actions;
END [procedure_name];
```

ORACLE

Procedures

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. A procedure is compiled and stored in the database as a schema object. Procedures promote reusability and maintainability.

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.

Functions

A function is:

- **A block that returns a value**
- **Stored in the database as a schema object**
- **Called as part of an expression or used to provide a parameter value**

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, ...)]
RETURN datatype IS|AS
 [local_variable_declarations; …]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

ORACLE

Functions

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.

Function: Example

- **Create the function:**

```
CREATE OR REPLACE FUNCTION get_credit
 (v_id customers.customer_id%TYPE) RETURN NUMBER IS
  v_credit customers.credit_limit%TYPE := 0;
BEGIN
  SELECT credit_limit
  INTO   v_credit
  FROM   customers
  WHERE  customer_id = v_id;
  RETURN v_credit;
END get_credit;
/
```

- **Invoke the function as an expression or as a parameter value:**

```
EXECUTE dbms_output.put_line(get_credit(101))
```

Function: Example

The `get_credit` function is created with a single input parameter and returns the credit limit as a number.

The `get_credit` function follows the common programming practice of assigning a returning value to a local variable and uses a single `RETURN` statement in the executable section of the code to return the value stored in the 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 `get_credit` function. In this case, `get_credit` is invoked first to calculate the credit limit of the customer with ID 101. The `credit_limit` 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.

Ways to Execute Functions

- **Invoke as part of a PL/SQL expression**
  - **Using a host variable to obtain the result:**

```
VARIABLE v_credit NUMBER
EXECUTE :v_credit := get_credit(101)
```

  - **Using a local variable to obtain the result:**

```
DECLARE v_credit customers.credit_limit%type;
BEGIN
  v_credit := get_credit(101); ...
END;
```

- **Use as a parameter to another subprogram**

```
EXECUTE dbms_output.put_line(get_credit(101))
```

- **Use in a SQL statement (subject to restrictions)**

```
SELECT get_credit(customer_id) FROM customers;
```

ORACLE

Ways to Execute Functions

If functions are designed thoughtfully, 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.
- **As a parameter to another subprogram:** The third example in the slide demonstrates this usage. The `get_credit` 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 *Oracle Database 10g: SQL Fundamentals I* course.
- **As an expression in a SQL statement:** The last example shows how a function can be used as a single-row function in a SQL statement.

Note: The restrictions and guidelines that apply to functions when used in a SQL statement are discussed in the next few pages.

Restrictions on Calling Functions from SQL Expressions

- **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:**
  - **Parameters must be specified with positional notation**
  - **You must own the function or have the `EXECUTE` privilege**

Restrictions on Calling Functions from SQL Expressions

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 input parameters and should be 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.

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.
- It cannot be used to specify a default value for a column.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called unless invoked from a function that meets the preceding requirements.

Guidelines for Calling Functions from SQL Expressions

Functions called from:

- **A `SELECT` statement cannot contain DML statements**
- **An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`**
- **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.

ORACLE

Guidelines for Calling Functions from SQL Expressions

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. In particular, when a function is called from:

- A `SELECT` statement or a parallel `UPDATE` or `DELETE` statement, the function cannot modify any database table
- An `UPDATE` or `DELETE` statement, the function cannot query or modify any database table modified by that statement
- A `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute directly or indirectly through another subprogram, a SQL transaction control statement 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

PL/SQL Packages: Review

PL/SQL packages:

- **Group logically related components:**
  - **PL/SQL types**
  - **Variables, data structures, and exceptions**
  - **Subprograms: procedures and functions**
- **Consist of two parts:**
  - **A specification**
  - **A body**

- **Enable the Oracle server to read multiple objects into memory simultaneously**

ORACLE

PL/SQL Packages: Review

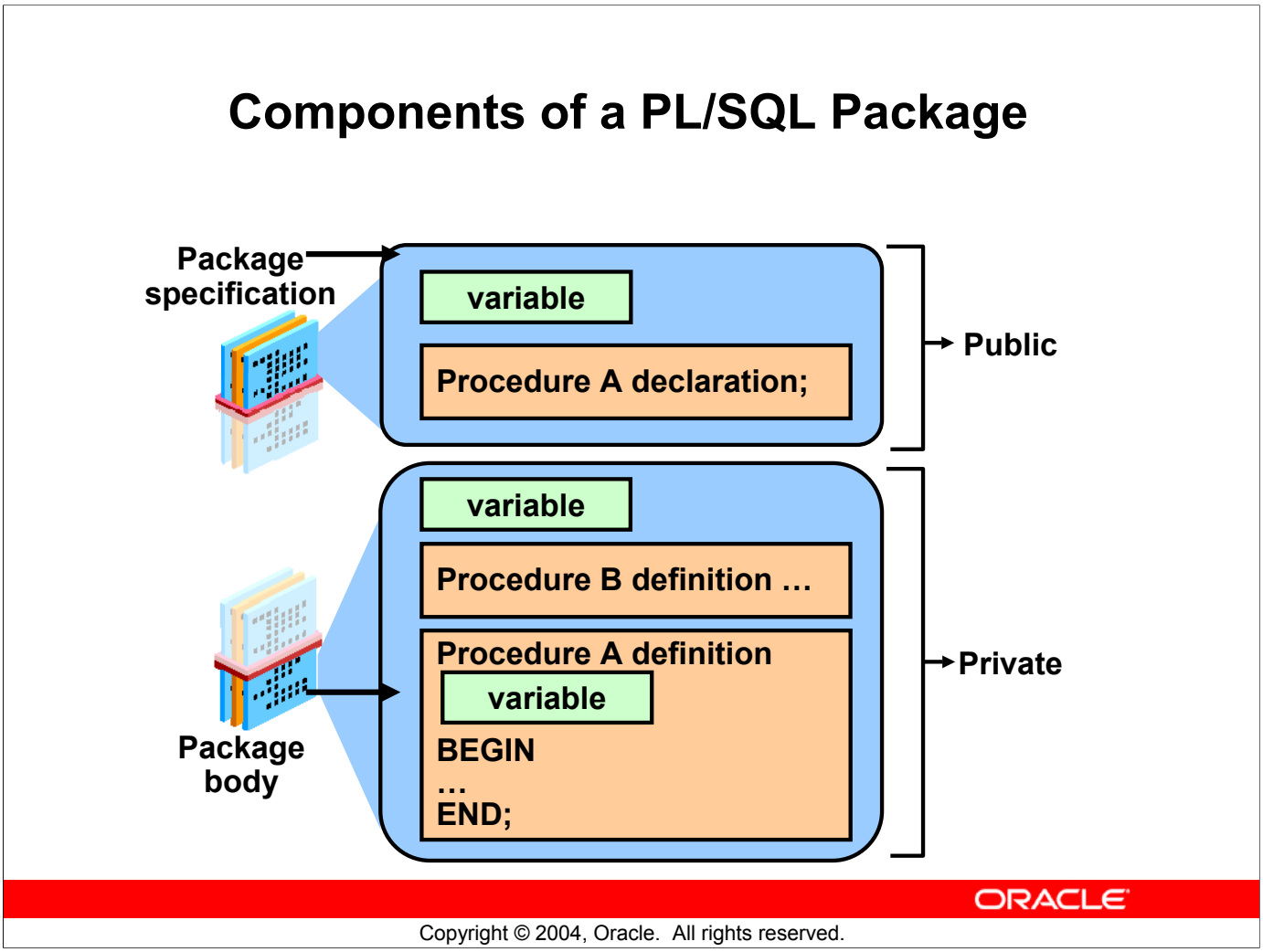
PL/SQL packages enable you to bundle related PL/SQL types, variables, data structures, exceptions, and subprograms into one container. For example, an Order Entry package can contain procedures for adding and deleting customers and orders, functions for calculating annual sales, and credit limit variables.

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).



Components of a PL/SQL Package

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 the package.

Note: If a package specification does not contain subprogram declarations, then there is no requirement for a package body.

Creating the Package Specification

Syntax:

```
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.**

ORACLE

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 specifications`*** specifies the public procedure or function declarations.

Note: 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.

Creating the Package Body

Syntax:

```
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.**

ORACLE

Creating the Package Body

Create a package body to define and implement all public subprograms and supporting private constructs. When creating a package body, do the following:

- 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.
- The package body must complete the implementation for all procedures or functions declared in the package specification.

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 bodies*** 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.

Cursor

- **A cursor is a pointer to the private memory area allocated by the Oracle server.**
- **There are two types of cursors:**
  - **Implicit cursors: Created and managed internally by the Oracle server to process SQL statements**
  - **Explicit cursors: Explicitly declared by the programmer**

ORACLE

Cursor

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 Oracle 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. Information required for processing and information retrieved after processing are all stored in this area. Because this area is internally managed by the Oracle server, you have no control over this area. 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 contains a SQL statement, an implicit cursor is created.

There are two types of cursors:

- **Implicit cursors:** Implicit cursors are created and managed by the Oracle server. You do not have access to them. The Oracle server creates such a cursor when it has to execute a SQL statement.

Cursor (continued)

- **Explicit cursors:** As a programmer, you may want to retrieve multiple rows from a database table, have a pointer to each row that is retrieved, and work on the rows one at a time. In such cases, you can declare cursors explicitly depending on your business requirements. Such cursors that are declared by programmers are called explicit cursors. You declare these cursors in the declarative section of a PL/SQL block. Remember that you can also declare variables and exceptions in the declarative section.

Processing Explicit Cursors

The following three commands are used to process an explicit cursor:

- **OPEN**
- **FETCH**
- **CLOSE**

Alternatively, you can also use a cursor FOR loops.

ORACLE

Processing Explicit Cursors

You declare an explicit cursor when you need exact control over query processing. You use three commands to control a cursor:

- OPEN
- FETCH
- CLOSE

You initialize the cursor with the OPEN command, which recognizes the result set. Then you execute the FETCH command repeatedly in a loop until all rows have been retrieved. Alternatively, you can use a BULK COLLECT clause to fetch all rows at once. After the last row has been processed, you release the cursor by using the CLOSE command.

Explicit Cursor Attributes

Every explicit cursor has the following four attributes:

- `cursor_name%FOUND`
- `cursor_name%ISOPEN`
- `cursor_name%NOTFOUND`
- `cursor_name%ROWCOUNT`

Cursor Attributes

When cursor attributes are appended to the cursors, they return useful information regarding the execution of the DML statement. The following are the four cursor attributes:

- **`cursor_name%FOUND`:** Returns `TRUE` if the last fetch returned a row. Returns `NULL` before the first fetch from an `OPEN` cursor. Returns `FALSE` if the last fetch failed to return a row.
- **`cursor_name%ISOPEN`:** Returns `TRUE` if the cursor is open, otherwise returns `FALSE`.
- **`cursor_name%NOTFOUND`:** Returns `FALSE` if the last fetch returned a row. Returns `NULL` before the first fetch from an `OPEN` cursor. Returns `TRUE` if the last fetch failed to return a row.
- **`cursor_name%ROWCOUNT`:** Returns zero before the first fetch. After every fetch returns the number of rows fetched so far.

Cursor FOR Loops

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.**

Cursor FOR Loops

A cursor FOR loop 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:

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

Guidelines

- Do not declare the record that controls the loop because 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: Example

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR cur_cust IS
   SELECT cust_first_name, credit_limit
   FROM customers
   WHERE credit_limit > 4000;
BEGIN
   FOR v_cust_record IN cur_cust
    LOOP
     DBMS_OUTPUT.PUT_LINE
     (v_cust_record.cust_first_name ||' '||
      v_cust_record.credit_limit);
    END LOOP;
END;
/
```

ORACLE

Cursor: Example

The example shows the use of a cursor `FOR` loop.

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

Handling Exceptions

- An exception is an error in PL/SQL 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

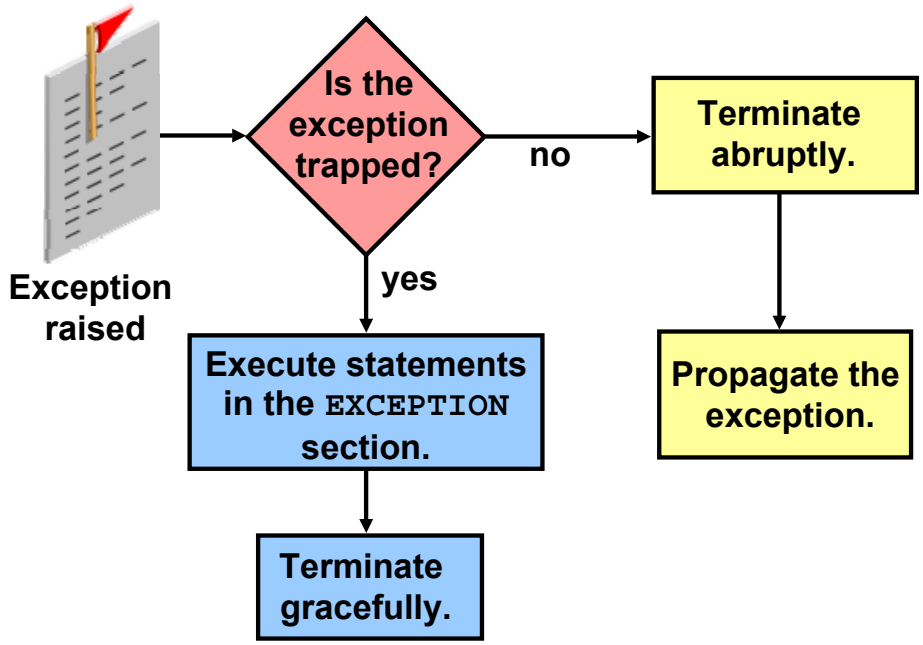
Handling Exceptions

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.

Methods for Raising an Exception

- An Oracle error occurs and the associated exception is raised automatically. For example, if the error `ORA-01403` occurs when no rows are retrieved from the database in a `SELECT` statement, then PL/SQL raises the `NO_DATA_FOUND` exception. These errors are converted into predefined exceptions.
- Depending on the business functionality your program is implementing, you may have to explicitly raise an exception. You raise an exception explicitly by issuing the `RAISE` statement within the block. The exception being raised may be either user-defined or predefined.
- There are 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



Handling Exceptions (continued)

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, then 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.

Exceptions: Example

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT cust_last_name INTO v_lname FROM customers
  WHERE cust_first_name='Ally';
  DBMS_OUTPUT.PUT_LINE ('Ally''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;
/
```

ORACLE

Exceptions: Example

You have written PL/SQL blocks with a declarative section (beginning with the keyword DECLARE) and an executable section (beginning and ending with the keywords BEGIN and END, respectively). For exception handling, include another optional section called the EXCEPTION section. This section begins with the keyword EXCEPTION. If present, this is the last section in a PL/SQL block. Examine the code in the slide to see the EXCEPTION section.

The output of this code is shown below:

```
Your select statement retrieved multiple rows. Consider using a
cursor.

PL/SQL procedure successfully completed.
```

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.

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`

ORACLE

Predefined Oracle Server Errors

You can reference predefined Oracle server errors by using its predefined name within the corresponding exception-handling routine.

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

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

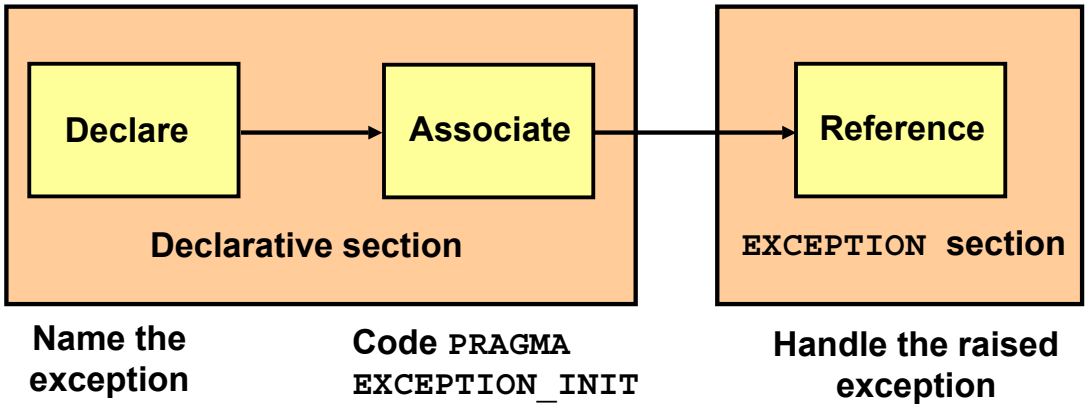
Predefined Oracle Server Errors (continued)

Exception Name	Oracle Server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or varray
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails
LOGIN_DENIED	ORA-01017	Logging on to the Oracle server with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single-row SELECT returned no data
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to the Oracle server
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types

Predefined Oracle Server Errors (continued)

Exception Name	Oracle Server Error Number	Description
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or varray element by using an index number larger than the number of elements in the collection
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or varray element by using an index number that is outside the legal range (for example –1)
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while the Oracle server was waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Trapping Non-Predefined Oracle Server Errors



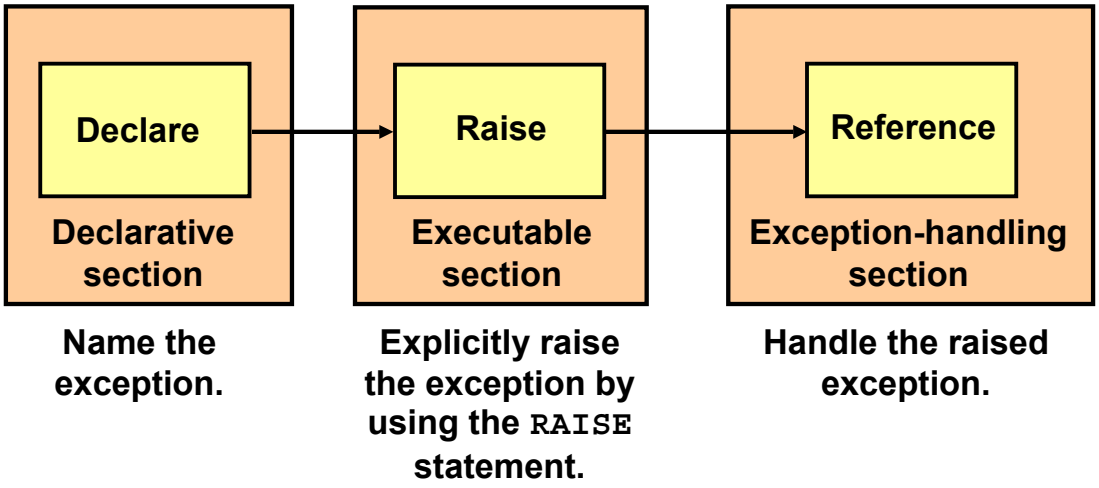
Trapping Non-Predefined Oracle Server Errors

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 can 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` instructs the compiler to associate an exception name with an Oracle error number. That allows 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.

Trapping User-Defined Exceptions



Trapping User-Defined Exceptions

With PL/SQL, you can define your own exceptions. You define 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, then 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

The 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.**

The RAISE_APPLICATION_ERROR Procedure

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:

error_number	Is a user-specified number for the exception between –20,000 and –20,999
message	Is the user-specified message for the exception. It 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, the default, the error replaces all previous errors.)

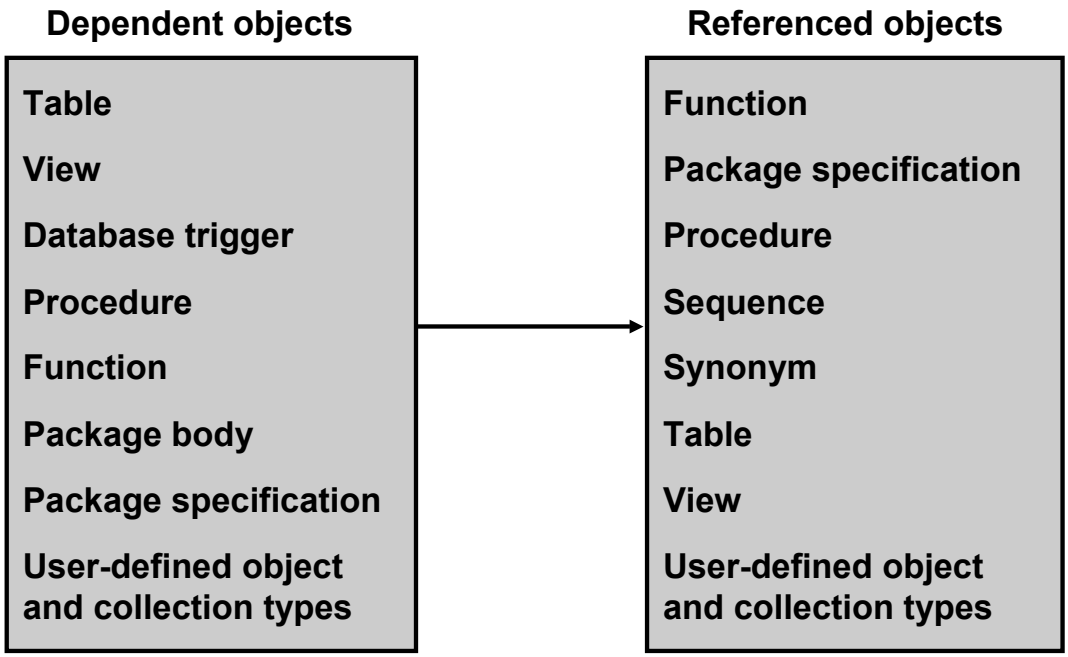
The RAISE_APPLICATION_ERROR Procedure

- 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.

The RAISE_APPLICATION_ERROR Procedure (continued)

The RAISE_APPLICATION_ERROR 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.

Dependencies



ORACLE

Dependent and Referenced Objects

Some objects reference other objects as part of their definitions. For example, a stored procedure could contain a `SELECT` statement that selects columns from a table. For this reason, the stored procedure is called a dependent object, whereas the table is called a referenced object.

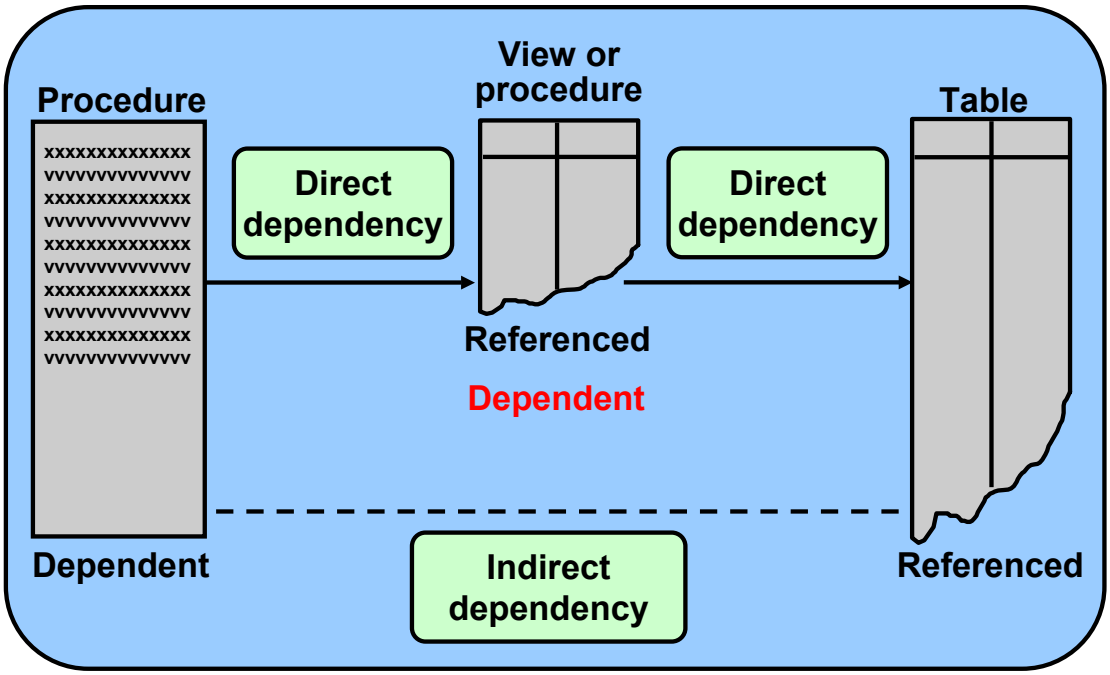
Dependency Issues

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, procedure may or may not continue to work without an 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.

Status	Significance
`VALID`	The schema object has been compiled and can be immediately used when referenced.
`INVALID`	The schema object must be compiled before it can be used.

Dependencies



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

Displaying Direct and Indirect Dependencies

1. **Run the `utldtree.sql` script to create the objects that enable you to display the direct and indirect dependencies.**
2. **Execute the `DEPTREE_FILL` procedure:**

```
EXECUTE deptree_fill('TABLE','OE','CUSTOMERS')
```


Displaying Direct and Indirect Dependencies by Using Views

Display direct and indirect dependencies from additional user views called `DEPTREE` and `IDEPTREE`; these views are provided by the Oracle database.

Example

1. Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/rdbms/admin` folder.
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_type`	Type of the referenced object
`object_owner`	Schema of the referenced object
`object_name`	Name of the referenced object

Using Oracle-Supplied Packages

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.

ORACLE

Using Oracle-Supplied Packages

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`.

Some of the Oracle Supplied Packages

Here is an abbreviated list of some Oracle-supplied packages:

- **DBMS_ALERT**
- **DBMS_LOCK**
- **DBMS_SESSION**
- **DBMS_OUTPUT**
- **HTP**
- **UTL_FILE**
- **UTL_MAIL**
- **DBMS_SCHEDULER**

ORACLE

List of Some of the Oracle Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. It would be impossible to cover the exhaustive set of packages and their functionality in this course. For more information, refer to *PL/SQL Packages and Types Reference 10g* manual (previously known as the *PL/SQL Supplied Packages Reference)*.

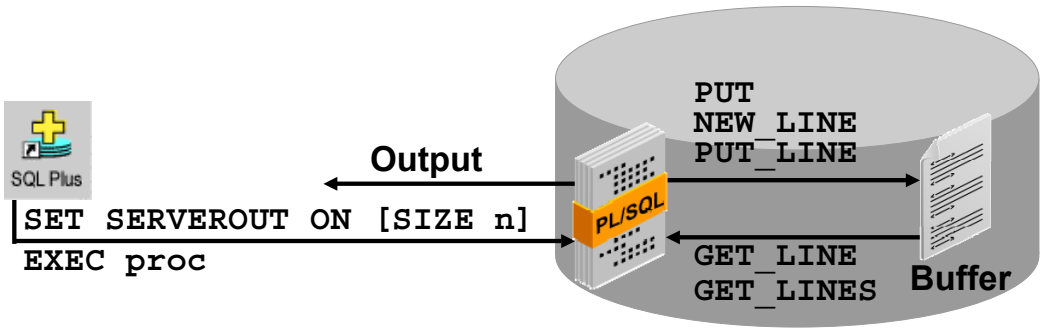
The following is a brief description of some listed packages:

- The `DBMS_ALERT` package supports asynchronous notification of database events. Messages or alerts are sent on a `COMMIT` command.
- The `DBMS_LOCK` package is used to request, convert, and release locks through Oracle Lock Management services.
- The `DBMS_SESSION` package enables programmatic use of the `ALTER SESSION` SQL statement and other session-level commands.
- The `DBMS_OUTPUT` package provides debugging and buffering of text data.
- The `HTP` package writes HTML-tagged data into database buffers.
- The `UTL_FILE` package enables reading and writing of operating system text files.
- The `UTL_MAIL` package enables composing and sending of e-mail messages.
- The `DBMS_SCHEDULER` package enables scheduling and automated execution of PL/SQL blocks, stored procedures, and external procedures or executables.

DBMS_OUTPUT Package

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.**
- **Use SET SERVEROUTPUT ON to display messages in SQL*Plus.**



ORACLE

DBMS_OUTPUT Package

The DBMS_OUTPUT package sends textual messages from any PL/SQL block into a buffer in the database. The procedures provided by the package include:

- PUT to append text from the procedure to the current line of the line output buffer
- NEW_LINE to place an end-of-line marker in the output buffer
- PUT_LINE to combine the action of PUT and NEW_LINE; to trim leading spaces
- GET_LINE to retrieve the current line from the buffer into a procedure variable
- GET_LINES to retrieve an array of lines into a procedure-array variable
- ENABLE/DISABLE to enable or disable calls to the 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 between 2,000 (the default) and 1,000,000 (1 million characters)
- An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

Practical Uses

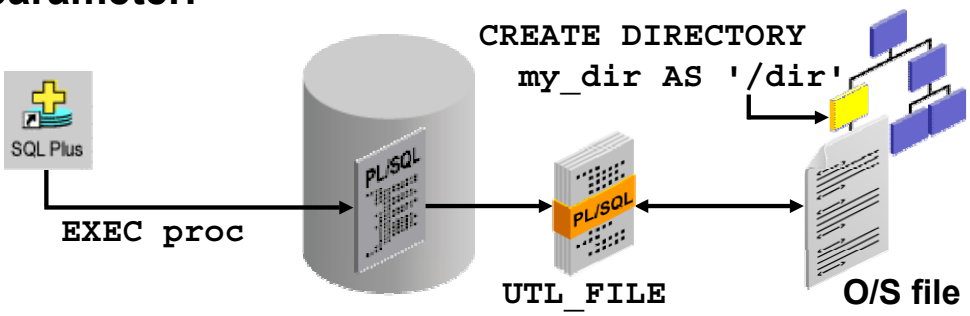
- You can output results to the window for debugging purposes.
- You can trace the code execution path for a function or procedure.
- You can send messages between subprograms and triggers.

Note: There is no mechanism to flush output during the execution of a procedure.

UTL_FILE Package

The UTL_FILE package extends PL/SQL programs to read and write operating system text files. UTL_FILE:

- **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. You can also use the utl_file_dir database parameter.**



UTL_FILE Package

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

The preferred approach is to use the directory alias created by the CREATE DIRECTORY statement, which does not require the database to be restarted. 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.

Note: The DBMS_LOB package can be used to read binary files on the operating system.

Summary

In this lesson, you should have learned how to:

- **Identify a PL/SQL block**
- **Create subprograms**
- **List restrictions and guidelines on calling functions from SQL expressions**
- **Use cursors**
- **Handle exceptions**
- **Use the `raise_application_error` procedure**
- **Identify Oracle-supplied packages**

ORACLE

Summary

This lesson reviewed some basic PL/SQL concepts such as:

- PL/SQL block structure
- Subprograms
- Cursors
- Exceptions
- Oracle-supplied packages

The quiz on the following pages is designed to test and review your PL/SQL knowledge. This knowledge is necessary as a base line for the subsequent chapters to build upon.

Practice 1: PL/SQL Knowledge Quiz

The questions are designed as a refresher. Use the space provided for your answers. If you do not know the answer, go on to the next question. For solutions to this quiz, see Appendix A.

PL/SQL Basics

1. What are the four key areas of the basic `PL/SQL` block? What happens in each area?

2. What is a variable and where is it declared?

3. What is a constant and where is it declared?

4. What are the different modes for parameters and what does each mode do?

5. How does a function differ from a procedure?

6. What are the two main components of a PL/SQL package?

   a. In what order are they defined?

   b. Are both required?

7. How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL*Plus?

8. What is a record?

9. What is an `index-by` table?

10. How are loops implemented in PL/SQL?

11. How is branching logic implemented in PL/SQL?

Practice 1: PL/SQL Knowledge Quiz (continued)

Cursor Basics

12. What is an explicit cursor?

13. Where do you define an explicit cursor?

14. Name the five steps for using an explicit cursor.

15. What is the syntax used to declare a cursor?

16. What does the `FOR UPDATE` clause do within a cursor definition?

17. What command opens an explicit cursor?

18. What command closes an explicit cursor?

19. Name five implicit actions that a cursor `FOR` loop provides.

20. Describe what the following cursor attributes do:
    - `cursor_name%ISOPEN`
    - `cursor_name%FOUND`
    - `cursor_name%NOTFOUND`
    - `cursor_name%ROWCOUNT`

Practice 1: PL/SQL Knowledge Quiz (continued)

Exceptions

21. An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?

22. An exception handler is mandatory within a PL/SQL subprogram. (True/False)

23. What syntax do you use in the exception handler area of a subprogram?

24. How do you code for a `NO_DATA_FOUND` error?

25. Name three types of exceptions.

26. To associate an exception identifier with an Oracle error code, what pragma do you use and where?

27. How do you explicitly raise an exception?

28. What types of exceptions are implicitly raised?

29. What does the `raise_application_error` procedure do?

Practice 1: PL/SQL Knowledge Quiz (continued)

Dependencies

30. Which objects can a procedure or function directly reference?

31. What are the two statuses that a schema object can have and where are they recorded?

32. The Oracle server automatically recompiles invalid procedures when they are called from the same _____. To avoid compile problems with remote database calls, you can use the _____ model instead of the timestamp model.

33. What data dictionary contains information on direct dependencies?

34. What script do you run to create the views `deptree` and `ideptree`?

35. What does the `deptree_fill` procedure do and what are the arguments that you need to provide?

Oracle-Supplied Packages

36. What does the `dbms_output` package do?

37. How do you write "This procedure works." from within a PL/SQL program by using the `dbms_output`?

38. What does `dbms_sql` do and how does this compare with Native Dynamic SQL?