17 Managing PL/SQL Code

Source text wrapping: A: READ ONLY

✅Describe and use conditional compilation

Conditional compilation: 2.9

- Conditional compilation lets you customize the functionality of a PL/SQL application without removing source text
- For example, you can:
  - Use new features with the latest database release and disable them when running the application in an older database release
  - Activate debugging or tracing statements in the development environment and hide them when running the application at a production site

How Conditional Compilation Works

- Conditional compilation uses selection directives, which are similar to `IF` statements, to select source text for compilation
- The condition in a selection directive usually includes an inquiry directive. Error directives raise user-defined errors. All conditional compilation directives are built from preprocessor control tokens and PL/SQL text

Preprocessor Control Tokens

- A preprocessor control token identifies code that is processed before the PL/SQL unit is compiled
- syntax: `$plsql_identifier`
- There cannot be space between `$` and `plsql_identifier`
- The character `$` can also appear inside `plsql_identifier`, but it has no special meaning there
- These preprocessor control tokens are reserved:
  - `$IF, $THEN, $ELSE, $ELSIF, $ERROR`

Selection Directives

- A selection directive selects source text to compile
- syntax:

```
$IF boolean_static_expression $THEN
   text
[ $ELSIF boolean_static_expression $THEN
   text
]...
[ $ELSE
   text
$END
]
```

- The text can be anything, but typically, it is either a statement or an error directive
- The selection directive evaluates the `BOOLEAN` static expressions in the order that they appear until either one expression has the value `TRUE` or the list of expressions is exhausted
- If one expression has the value `TRUE`, its text is compiled, the remaining expressions are not evaluated, and their text is not analyzed
- If no expression has the value `TRUE`, then if `$ELSE` is present, its text is compiled; otherwise, no text is compiled

Error Directives

- An error directive produces a user-defined error message during compilation
- syntax:

```
$ERROR varchar2_static_expression $END
```

- It produces this compile-time error message, where string is the value of varchar2_static_expression: `PLS-00179: $ERROR: string`

Inquiry Directives

- An inquiry directive provides information about the compilation environment
- syntax: `$$name`
- An inquiry directive typically appears in the boolean_static_expression of a selection directive, but it can appear anywhere that a variable or literal of its type can appear. Moreover, it can appear where regular PL/SQL allows only a literal (not a variable)—for example, to specify the size of a `VARCHAR2` variable
- Predefined Inquiry Directives
    - `$$PLSQL_LINE`
        - A `PLS_INTEGER` literal whose value is the number of the source line on which the directive appears in the current PL/SQL unit. E.g. `$IF $$PLSQL_LINE = 32 $THEN ...`
    - `$$PLSQL_UNIT`
        - A `VARCHAR2` literal that contains the name of the current PL/SQL unit
        - If the current PL/SQL unit is an anonymous block, then `$$PLSQL_UNIT` contains a `NULL` value
    - `$$PLSQL_UNIT_OWNER`
        - A `VARCHAR2` literal that contains the name of the owner of the current PL/SQL unit
        - If the current PL/SQL unit is an anonymous block, then `$$PLSQL_UNIT_OWNER` contains a `NULL` value

- $$PLSQL_UNIT_TYPE
  - A VARCHAR2 literal that contains the type of the current PL/SQL unit—ANONYMOUS BLOCK, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, TRIGGER, TYPE, or TYPE BODY
  - Inside an anonymous block or non-DML trigger, $$PLSQL_UNIT_TYPE has the value ANONYMOUS BLOCK
- $$plsql_compilation_parameter
  - The name plsql_compilation_parameter is a PL/SQL compilation parameter (for example, PLSCOPE_SETTINGS)
  - $$PLSCOPE_SETTINGS, $$PLSQL_CCFLAGS, $$PLSQL_CODE_TYPE, $$PLSQL_OPTIMIZE_LEVEL, $$PLSQL_WARNINGS, $$NLS_LENGTH_SEMANTICS
- Because a selection directive needs a BOOLEAN static expression, you cannot use $$PLSQL_UNIT, $$PLSQL_UNIT_OWNER, or $$PLSQL_UNIT_TYPE in a VARCHAR2 comparison such as:

```
$IF $$PLSQL_UNIT = 'AWARD_BONUS' $THEN ...
$IF $$PLSQL_UNIT_OWNER IS HR $THEN ...
$IF $$PLSQL_UNIT_TYPE IS FUNCTION $THEN ...
```

However, you can compare the preceding directives to NULL:

```
$IF $$PLSQL_UNIT IS NULL $THEN ...
$IF $$PLSQL_UNIT_OWNER IS NOT NULL $THEN ...
$IF $$PLSQL_UNIT_TYPE IS NULL $THEN ...
```

- Assigning Values to Inquiry Directives
  - You can assign values to inquiry directives with the PLSQL_CCFLAGS compilation parameter
  - For example:

```
ALTER SESSION SET PLSQL_CCFLAGS =
  'name1:value1, name2:value2, ... namen:valuen'
```

  - Each value must be either a BOOLEAN literal (TRUE, FALSE, or NULL) or PLS_INTEGER literal. The data type of value determines the data type of name
  - The same name can appear multiple times, with values of the same or different data types. Later assignments override earlier assignments
  - For example, this command sets the value of $$flag to 5 and its data type to PLS_INTEGER:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'flag:TRUE, flag:5'
```

```
```

  - Oracle recommends against using `PLSQL_CCFLAGS` to assign values to predefined inquiry directives, including compilation parameters
  - To assign values to compilation parameters, Oracle recommends using the `ALTER SESSION` statement
  - > Note: The compile-time value of `PLSQL_CCFLAGS` is stored with the metadata of stored PL/SQL units, which means that you can reuse the value when you explicitly recompile the units

- Unresolvable Inquiry Directives
  - If the source text is not wrapped, PL/SQL issues a warning if the value of an inquiry directive cannot be determined
  - If an inquiry directive (`$$name`) cannot be resolved, and the source text is not wrapped, then PL/SQL issues the warning `PLW-6003` and substitutes `NULL` for the value of the unresolved inquiry directive
  - If the source text is wrapped, the warning message is disabled, so that the unresolved inquiry directive is not revealed

DBMS_DB_VERSION Package

- The `DBMS_DB_VERSION` package specifies the Oracle version numbers and other information useful for simple conditional compilation selections based on Oracle versions
- The `DBMS_DB_VERSION` package provides these static constants:
  - The `PLS_INTEGER` constant `VERSION` identifies the current Oracle Database version
  - The `PLS_INTEGER` constant `RELEASE` identifies the current Oracle Database release number
  - Each `BOOLEAN` constant of the form `VER_LE_v` has the value `TRUE` if the database version is less than or equal to `v`; otherwise, it has the value `FALSE`
  - Each `BOOLEAN` constant of the form `VER_LE_v_r` has the value `TRUE` if the database version is less than or equal to `v` and release is less than or equal to `r`; otherwise, it has the value `FALSE`

Retrieving and Printing Post-Processed Source Text

- The `DBMS_PREPROCESSOR` package provides subprograms that retrieve and print the source text of a PL/SQL unit in its post-processed form

Conditional Compilation Directive Restrictions

- Conditional compilation directives are subject to these semantic restrictions
- A conditional compilation directive cannot appear in the specification of a schema-level user-defined type (created with the `CREATE TYPE` statement)
- This type specification specifies the attribute structure of the type, which determines the attribute structure of dependent types and the column structure of dependent tables
- > Caution: Using a conditional compilation directive to change the attribute structure of a type can cause dependent objects to "go out of sync" or dependent tables to become inaccessible. Oracle

> recommends that you change the attribute structure of a type only with the `ALTER TYPE` statement. The `ALTER TYPE` statement propagates changes to dependent objects

- If a conditional compilation directive is used in a schema-level type specification, the compiler raises the error `PLS-00180: preprocessor directives are not supported in this context`
- As all conditional compiler constructs are processed by the PL/SQL preprocessor, the SQL Parser imposes the following restrictions on the location of the first conditional compilation directive in a stored PL/SQL unit or anonymous block:
    - In a package specification, a package body, a type body, a schema-level function and in a schema-level procedure, at least one nonwhitespace PL/SQL token must appear after the identifier of the unit name before a conditional compilation directive is valid (ways to do this: left parenthesis of parameter list is a valid token, package or subprogram property (e.g. authid, accessible by) is also a valid PL/SQL token)
        - > Note: The PL/SQL comments, "--" or "/*", are counted as whitespace tokens. If the token is invalid in PL/SQL, then a `PLS-00103` error is issued. But if a conditional compilation directive is used in violation of this rule, then an `ORA` error is produced
    - In a trigger or an anonymous block, the first conditional compilation directive cannot appear before the keyword `DECLARE` or `BEGIN`, whichever comes first
- The SQL parser also imposes this restriction: If an anonymous block uses a placeholder, the placeholder cannot appear in a conditional compilation directive. For example:

```
BEGIN
  :n := 1; -- valid use of placeholder
  $IF ... $THEN
    :n := 1; -- invalid use of placeholder
$END
```

✅Code-based access control: granting roles to program units

Security for DR and IR: READ ONLY

Using code based access control

CBAC granting roles: oracle-base: READ ONLY

PL/SQL security blog: Feuerstein: READ ONLY

About Using Code Based Access Control for Applications

- You can use code based access control (CBAC) to better manage definer's rights program units
- Applications must often run program units in the caller's environment, while requiring elevated privileges. PL/SQL programs traditionally make use of definer's rights to temporarily elevate the privileges of the program

- However, definer's rights based program units run in the context of the definer or the owner of the program unit, as opposed to the invoker's context. Also, using definer's rights based programs often leads to the program unit getting more privileges than required
- Code based access control (CBAC) provides the solution by enabling you to attach database roles to a PL/SQL function, procedure, or package
- These database roles are enabled at run time, enabling the program unit to execute with the required privileges in the calling user's environment
- You can create privilege analysis policies that capture the use of CBAC roles

Who Can Grant Code Based Access Control Roles to a Program Unit?

- Code based access control roles can be granted to a program unit if a set of conditions are met
- These conditions are as follows:
    - The grantor is user `SYS` or owns the program unit
    - If the grantor owns the program unit, then the grantor must have the `GRANT ANY ROLE` system privilege, or have the `ADMIN` or `DELEGATE` option for the roles that they want to grant to program units
    - The roles to be granted are directly granted roles to the owner
    - The roles to be granted are standard database roles
- If these three conditions are not met, then error `ORA-28702: Program unit string is not owned by the grantor` is raised if the first condition is not met, and error `ORA-1924: role 'string' not granted or does not exist` is raised if the second and third conditions are not met

How Code Based Access Control Works with Invoker's Rights Program Units

READ

How Code Based Access Control Works with Definer's Rights Program Units

READ

Grants of Database Roles to Users for Their CBAC Grants

- The `DELEGATE` option in the `GRANT` statement can limit privilege grants to roles by users responsible for CBAC grants
- When you grant a database role to a user who is responsible for CBAC grants, you can include the `DELEGATE` option in the `GRANT` statement to prevent giving the grantee additional privileges on the roles
- The `DELEGATE` option enables the roles to be granted to program units, but it does not permit the granting of the role to other principals or the administration of the role itself
- You also can use the `ADMIN` option for the grants, which does permit the granting of the role to other principals
- Both the `ADMIN` and `DELEGATE` options are compatible; that is, you can grant both to a user, though you must do this in separate `GRANT` statements for each option

- To find if a user has been granted a role with these options, query the `DELEGATE_OPTION` column or the `ADMIN_OPTION` column of either the `USER_ROLE_PRIVS` or `DBA_ROLE_PRIVS` for the user
- The syntax for using the `DELEGATE` and `ADMIN` option is as follows:

```
GRANT role_list to user_list WITH DELEGATE OPTION;
GRANT role_list to user_list WITH ADMIN OPTION;
```

- Be aware that CBAC grants themselves can only take place locally in a PDB

Grants and Revokes of Database Roles to a Program Unit

- The `GRANT` and `REVOKE` statements can grant database roles to or revoke database roles from a program unit
- The following syntax to grants or revokes database roles for a PL/SQL function, procedure, or package:

```
GRANT role_list TO code_list
REVOKE {role_list | ALL} FROM code_list
```

  - role_list

```
code-based_role_name[, role_list]
```

  - code_list

```
{ {FUNCTION [schema.]function_name}
| {PROCEDURE [schema.]procedure_name}
| {PACKAGE [schema.]package_name}
} [, code_list]
```

✅Whitelist code access with the ACCESSIBLE BY clause

ACCESSIBLE BY: 13.1

- The `ACCESSIBLE BY` clause restricts access to a unit or subprogram by other units
- The accessor list explicitly lists those units which may have access
- The accessor list can be defined on individual subprograms in a package. This list is checked in addition to the accessor list defined on the package itself (if any)
- This list may only restrict access to the subprogram – it cannot expand access

- This code management feature is useful to prevent inadvertent use of internal subprograms. For example, it may not be convenient or feasible to reorganize a package into two packages: one for a small number of procedures requiring restricted access, and another one for the remaining units requiring public access
- The `ACCESSIBLE BY` clause may appear in the declarations of object types, object type bodies, packages, and subprograms
- The ACCESSIBLE BY clause can appear in the following SQL statements: ALTER TYPE Statement CREATE FUNCTION Statement CREATE PROCEDURE Statement CREATE PACKAGE Statement CREATE TYPE Statement CREATE TYPE BODY Statement
- syntax:

```
ACCESSIBLE BY ( accessor [,...] )
```

  - accessor

```
[FUNCTION | PROCEDURE | PACKAGE | TRIGGER | TYPE] [schema.] unit_name
```

    - Each accessor specifies another PL/SQL entity that may access the entity which includes the `ACCESSIBLE BY` clause
    - When an `ACCESSIBLE BY` clause appears, only entities named in the clause may access the entity in which the clause appears
    - An accessor may appear more than once in the `ACCESSIBLE BY` clause
    - The `ACCESSIBLE BY` clause can appear only once in the unit declaration
    - An entity named in an accessor is not required to exist
    - When an entity with an `ACCESSIBLE BY` clause is invoked, it imposes an additional access check after all other checks have been performed. These checks are:
      - The invoked unit must include an accessor with the same unit_name and unit_kind as the invoking unit
      - If the accessor includes a schema, the invoking unit must be in that schema
      - If the accessor does not include a schema, the invoker must be from the same schema as the invoked entity
    - unit_kind specifies if the unit is a `FUNCTION, PACKAGE, PROCEDURE, TRIGGER, or TYPE`
    - The unit_kind is optional, but it is recommended to specify it to avoid ambiguity when units have the same name. For example, it is possible to define a trigger with the same name as a function
    - The `ACCESSIBLE BY` clause allows access only when the call is direct. The check will fail if the access is through static SQL, `DBMS_SQL`, or dynamic SQL
    - Any call to the initialization procedure of a package specification or package body will be checked against the accessor list of the package specification
    - A unit can always access itself. An item in a unit can reference another item in the same unit

- RPC calls to a protected subprogram will always fail, since there is no context available to check the validity of the call, at either compile-time or run-time
- Calls to a protected subprogram from a conditional compilation directive will fail

☑ Mark code as deprecated

DEPRECATE pragma: 13.22

- The DEPRECATE pragma marks a PL/SQL element as deprecated. The compiler issues warnings for uses of pragma DEPRECATE or of deprecated elements
- The associated warnings tell users of a deprecated element that other code may need to be changed to account for the deprecation
- syntax:

```
PRAGMA DEPRECATE ( pls_identifier [, character_literal] ) ;
```

- The DEPRECATE pragma may only appear in the declaration sections of a package specification, an object specification, a top level procedure, or a top level function
- PL/SQL elements of these kinds may be deprecated: subprograms, packages, variables, constants, types, subtypes, exceptions, cursors
- The DEPRECATE pragma may only appear in the declaration section of a PL/SQL unit. It must appear immediately after the declaration of an item to be deprecated

```
PACKAGE trans_data AUTHID DEFINER AS
   min_balance constant real := 10.0;
   PRAGMA DEPRECATE(min_balance , 'Minimum balance requirement removed.');
   insufficient_funds EXCEPTION;
   PRAGMA DEPRECATE (insufficient_funds , 'Exception no longer raised.');
END trans_data;
```

- The DEPRECATE pragma applies to the PL/SQL element named in the declaration which precedes the pragma
- When the DEPRECATE pragma applies to a package specification, object specification, or subprogram, the pragma must appear immediately after the keyword IS or AS that terminates the declaration portion of the definition
- When the DEPRECATE pragma applies to a package or object specification, references to all the elements (of the kinds that can be deprecated) that are declared in the specification are also deprecated
- If the DEPRECATE pragma applies to a subprogram declaration, only that subprogram is affected; other overloads with the same name are not deprecated

- If the optional custom message appears in a use of the `DEPRECATE` pragma, the custom message will be added to the warning issued for any reference to the deprecated element
- The identifier in a `DEPRECATE` pragma must name the element in the declaration to which it applies
- Deprecation is inherited during type derivation. A child object type whose parent is deprecated is not deprecated. Only the attributes and methods that are inherited are deprecated
- When the base type is not deprecated but individual methods or attributes are deprecated, and when a type is derived from this type and the deprecated type or method is inherited, then references to these through the derived type will cause the compiler to issue a warning
- A reference to a deprecated element appearing anywhere except in the unit with the deprecation pragma or its body, will cause the PL/SQL compiler to issue a warning for the referenced elements. A reference to a deprecated element in an anonymous block will not cause the compiler to issue a warning; only references in named entities will draw a warning
- When a deprecated entity is referenced in the definition of another deprecated entity then no warning will be issued
- When an older client code refers to a deprecated entity, it is invalidated and recompiled. No warning is issued
- There is no effect when SQL code directly references a deprecated element
- A reference to a deprecated element in a PL/SQL static SQL statement may cause the PL/SQL compiler to issue a warning. However, such references may not be detectable
- pls_identifier: Identifier of the PL/SQL element being deprecated
- character_literal: An optional compile-time warning message

DEPRECATE Pragma Compilation Warnings

- The PL/SQL compiler issues warnings when the `DEPRECATE` pragma is used and when deprecated items are referenced
    - 6019 — The entity was deprecated and could be removed in a future release. Do not use the deprecated entity
    - 6020 — The referenced entity was deprecated and could be removed in a future release. Do not use the deprecated entity. Follow the specific instructions in the warning if any are given
    - 6021 — Misplaced pragma. The pragma `DEPRECATE` should follow immediately after the declaration of the entity that is being deprecated. Place the pragma immediately after the declaration of the entity that is being deprecated
    - 6022 — This entity cannot be deprecated. Deprecation only applies to entities that may be declared in a package or type specification as well as to top-level procedure and function definitions. Remove the pragma
- The `DEPRECATE` pragma warnings may be managed with the `PLSQL_WARNINGS` parameter or with the `DBMS_WARNING` package