- Packages allow you to hide implementation details from client programs.

  Hiding implementation details from client programs is a widely accepted best practice. Many Oracle customers follow this practice strictly, allowing client programs to access the database only by invoking PL/SQL subprograms. Some customers allow client programs to use SELECT statements to retrieve information from database tables, but require them to invoke PL/SQL subprograms for all business functions that change the database.

- Package subprograms must be qualified with package names when invoked from outside the package, which ensures that their names will always work when invoked from outside the package.

  For example, suppose that you developed a schema-level procedure named CONTINUE before Oracle Database 11*g* . Oracle Database 11*g* introduced the CONTINUE statement. Therefore, if you ported your code to Oracle Database 11*g* , it would no longer compile. However, if you had developed your procedure inside a package, your code would refer to the procedure as *package_name*.CONTINUE, so the code would still compile.

> **Note:**
>
> Oracle Database supplies many PL/SQL packages to extend database functionality and provide PL/SQL access to SQL features. You can use the supplied packages when creating your applications or for ideas in creating your own stored procedures. For information about these packages, see *Oracle Database PL/SQL Packages and Types Reference*.

> **See Also:**
>
> - *Oracle Database Concepts* for general information about packages
> - *Oracle Database PL/SQL Language Reference* for more reasons to use packages
> - *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL packages
> - *Oracle Database PL/SQL Packages and Types Reference* for complete information about the PL/SQL packages that Oracle provides

About PL/SQL Identifiers

Every PL/SQL subprogram, package, parameter, variable, constant, exception, and declared cursor has a name, which is a PL/SQL identifier.

The minimum length of an identifier is one character; the maximum length is 30 characters. The first character must be a letter, but each later character can be either a letter, numeral, dollar sign ($), underscore (_), or number sign (#). For example, these are acceptable identifiers:

```
X
t2
```

```
phone#
credit_limit
LastName
oracle$number
money$$$tree
SN##
try_again_
```

PL/SQL is not case-sensitive for identifiers. For example, PL/SQL considers these to be the same:

```
lastname
LastName
LASTNAME
```

You cannot use a PL/SQL reserved word as an identifier. You can use a PL/SQL keyword as an identifier, but it is not recommended. For lists of PL/SQL reserved words and keywords, see *Oracle Database PL/SQL Language Reference*.

See Also:

- *Oracle Database PL/SQL Language Reference* for additional general information about PL/SQL identifiers
- *Oracle Database PL/SQL Language Reference* for additional information about PL/SQL naming conventions
- *Oracle Database PL/SQL Language Reference* for information about the scope and visibility of PL/SQL identifiers
- *Oracle Database PL/SQL Language Reference* for information how to collect data on PL/SQL identifiers
- *Oracle Database PL/SQL Language Reference* for information about how PL/SQL resolves identifier names

About PL/SQL Data Types

Every PL/SQL constant, variable, subprogram parameter, and function return value has a data type that determines its storage format, constraints, valid range of values, and operations that can be performed on it.

A PL/SQL data type is either a SQL data type (such as VARCHAR2, NUMBER, or DATE) or a PL/SQL-only data type. The latter include BOOLEAN, RECORD, REF CURSOR, and many predefined subtypes. PL/SQL also lets you define your own subtypes.

A **subtype** is a subset of another data type, which is called its **base type**. A subtype has the same valid operations as its base type, but only a subset of its valid values. Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.

The predefined numeric subtype PLS_INTEGER is especially useful, because its operations use hardware arithmetic, rather than the library arithmetic that its base type uses.

You cannot use PL/SQL-only data types at schema level (that is, in tables or standalone subprograms). Therefore, to use these data types in a stored subprogram, you must put them in a package.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for general information about PL/SQL data types
> - *Oracle Database PL/SQL Language Reference* for information about the PLS_INTEGER data type
> - "About SQL Data Types"

Creating and Managing Standalone Subprograms

You can create and manage standalone PL/SQL subprograms.

> **Note:**
>
> To do the tutorials in this document, the `hr` sample schema must be installed and you must be connected to Oracle Database as the user HR from SQL Developer.

About Subprogram Structure

A subprogram follows PL/SQL block structure; that is, it has:

- **Declarative part** (optional)

  The declarative part contains declarations of types, constants, variables, exceptions, declared cursors, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

- **Executable part** (required)

  The executable part contains statements that assign values, control execution, and manipulate data.

- **Exception-handling part** (optional)

  The exception-handling part contains code that handles exceptions (runtime errors).

Comments can appear anywhere in PL/SQL code. The PL/SQL compiler ignores them. Adding comments to your program promotes readability and aids understanding. A **single-line comment** starts with a double hyphen (`--`) and extends to the end of the line. A **multiline comment** starts with a slash and asterisk (`/*`) and ends with an asterisk and a slash (`*/`).

The structure of a procedure is:

```
PROCEDURE name [ ( parameter_list ) ]
{ IS | AS }
  [ declarative_part ]
BEGIN  -- executable part begins
  statement; [ statement; ]...
[ EXCEPTION -- executable part ends, exception-handling part begins]
  exception_handler; [ exception_handler; ]... ]
END; /* exception-handling part ends if it exists;
        otherwise, executable part ends */
```

The structure of a function is like that of a procedure, except that it includes a `RETURN` clause and at least one `RETURN` statement (and some optional clauses that are beyond the scope of this document):

```
FUNCTION name [ ( parameter_list ) ] RETURN data_type [ clauses ]
{ IS | AS }
  [ declarative_part ]
BEGIN  -- executable part begins
  -- at least one statement must be a RETURN statement
  statement; [ statement; ]...
[ EXCEPTION -- executable part ends, exception-handling part begins]
  exception_handler; [ exception_handler; ]... ]
END; /* exception-handling part ends if it exists;
        otherwise, executable part ends */
```

The code that begins with PROCEDURE or FUNCTION and ends before IS or AS is the **subprogram signature**. The declarative, executable, and exception-handling parts comprise the **subprogram body**. The syntax of exception-handler is in "About Exceptions and Exception Handlers".

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about subprogram parts

Tutorial: Creating a Standalone Procedure

This tutorial shows how to use the Create Procedure tool to create a standalone procedure named ADD_EVALUATION that adds a row to the EVALUATIONS table.

The EVALUATIONS table was created in Example 4-1.

To create a standalone procedure, use either the SQL Developer tool Create Procedure or the DDL statement CREATE PROCEDURE.

To create a standalone procedure using Create Procedure tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, right-click **Procedures**.
3. In the list of choices, click **New Procedure**.

   The Create Procedure window opens.
4. For Schema, accept the default value, HR.
5. For Name, change PROCEDURE1 to `ADD_EVALUATION`.

6. Click the icon **Add Parameter**.

   A row appears under the column headings. Its fields have these default values: Name, PARAM1; Mode, IN; No Copy, deselected; Data Type, VARCHAR2; Default Value, empty.

7. For Name, change PARAM1 to `EVALUATION_ID`.

8. For Mode, accept the default value, IN.

9. For Data Type, select **NUMBER** from the menu.

10. Leave Default Value empty.

11. Add a second parameter by repeating steps 6 through 10 with the Name `EMPLOYEE_ID` and the Data Type NUMBER.

12. Add a third parameter by repeating steps 6 through 10 with the Name `EVALUATION_DATE` and the Data Type DATE.

13. Add a fourth parameter by repeating steps 6 through 10 with the Name `JOB_ID` and the Data Type VARCHAR2.

14. Add a fifth parameter by repeating steps 6 through 10 with the Name `MANAGER_ID` and the Data Type NUMBER.

15. Add a sixth parameter by repeating steps 6 through 10 with the Name `DEPARTMENT_ID` and the Data Type NUMBER.

16. Add a seventh parameter by repeating steps 6 through 10 with the Name `TOTAL_SCORE` and the Data Type NUMBER.

17. Click **OK**.

```
CREATE OR REPLACE PROCEDURE ADD_EVALUATION
(
  EVALUATION_ID IN NUMBER
, EMPLOYEE_ID IN NUMBER
, EVALUATION_DATE IN DATE
, JOB_ID IN VARCHAR2
, MANAGER_ID IN NUMBER
, DEPARTMENT_ID IN NUMBER
, TOTAL_SCORE IN NUMBER
) AS
BEGIN
  NULL;
END ADD_EVALUATION;
```

   The title of the ADD_EVALUATION pane is in italic font, indicating that the procedure is not yet saved in the database.

   Because the execution part of the procedure contains only the NULL statement, the procedure does nothing.

18. Replace the NULL statement with this statement:

```
INSERT INTO EVALUATIONS (
   evaluation_id,
   employee_id,
   evaluation_date,
   job_id,
   manager_id,
   department_id,
   total_score
)
```

```
VALUES (
  ADD_EVALUATION.evaluation_id,
  ADD_EVALUATION.employee_id,
  ADD_EVALUATION.evaluation_date,
  ADD_EVALUATION.job_id,
  ADD_EVALUATION.manager_id,
  ADD_EVALUATION.department_id,
  ADD_EVALUATION.total_score
);
```

(Qualifying the parameter names with the procedure name ensures that they are not confused with the columns that have the same names.)

19. From the File menu, select **Save**.

    Oracle Database compiles the procedure and saves it. The title of the ADD_EVALUATION pane is no longer in italic font. The Message - Log pane has the message `Compiled`.

> **See Also:**
>
> - *Oracle SQL Developer User's Guide* for another example of using SQL Developer to create a standalone procedure
> - "About Data Definition Language (DDL) Statements" for general information that applies to the CREATE PROCEDURE statement
> - *Oracle Database PL/SQL Language Reference* for information about the CREATE PROCEDURE statement

Tutorial: Creating a Standalone Function

This tutorial shows how to use the Create Function tool to create a standalone function named CALCULATE_SCORE that has three parameters and returns a value of type NUMBER.

To create a standalone function, use either the SQL Developer tool Create Function or the DDL statement CREATE FUNCTION.

To create a standalone function using Create Function tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, right-click **Functions**.
3. In the list of choices, click **New Function**.

   The Create Function window opens.
4. For Schema, accept the default value, HR.
5. For Name, change FUNCTION1 to `CALCULATE_SCORE`.
6. For Return Type, select **NUMBER** from the menu.
7. Click the icon **Add Parameter**.

   A row appears under the column headings. Its fields have these default values: Name, PARAM1; Mode, IN; No Copy, deselected; Data Type, VARCHAR2; Default Value, empty.

8. For Name, change PARAM1 to `cat`.
9. For Mode, accept the default value, IN.
10. For Data Type, accept the default, VARCHAR2.
11. Leave Default Value empty.
12. Add a second parameter by repeating steps 7 through 11 with the Name `score` and the Data Type NUMBER.
13. Add a third parameter by repeating steps 7 through 11 with the Name `weight` and the Data Type NUMBER.
14. Click **OK**.

    The CALCULATE_SCORE pane opens, showing the CREATE FUNCTION statement that created the function:

```
CREATE OR REPLACE FUNCTION CALCULATE_SCORE
(
  CAT IN VARCHAR2
, SCORE IN NUMBER
, WEIGHT IN NUMBER
) RETURN NUMBER AS
BEGIN
  RETURN NULL;
END CALCULATE_SCORE;
```

    The title of the CALCULATE_SCORE pane is in italic font, indicating that the function is not yet saved in the database.

    Because the only statement in the execution part of the function is the statement RETURN NULL, the function does nothing.

15. Replace NULL with `score * weight`.
16. From the File menu, select **Save**.

    Oracle Database compiles the function and saves it. The title of the CALCULATE_SCORE pane is no longer in italic font. The Message - Log pane has the message `Compiled`.

> **See Also:**
>
> - "About Data Definition Language (DDL) Statements" for general information that applies to the CREATE FUNCTION statement
> - *Oracle Database PL/SQL Language Reference* for information about the CREATE FUNCTION statement

Changing Standalone Subprograms

To change a standalone subprogram, use either the SQL Developer tool Edit or the DDL statement ALTER PROCEDURE or ALTER FUNCTION.

To change a standalone subprogram using the Edit tool:

1. In the Connections frame, expand **hr_conn**.

2. In the list of schema object types, expand either **Functions** or **Procedures**.

   A list of functions or procedures appears.

3. Click the function or procedure to change.

   To the right of the Connections frame, a frame appears. Its top tab has the name of the subprogram to change. The Code pane shows the code that created the subprogram.

   The Code pane is in write mode. (Clicking the pencil icon switches the mode from write mode to read only, or the reverse.)

4. In the Code pane, change the code.

   The title of the pane changes to italic font, indicating that the change is not yet saved in the database.

5. From the File menu, select **Save**.

   Oracle Database compiles the subprogram and saves it. The title of the pane is no longer in italic font. The Message - Log pane has the message `Compiled`.

> **See Also:**
>
> - "About Data Definition Language (DDL) Statements" for general information that applies to the ALTER PROCEDURE and ALTER FUNCTION statements
> - *Oracle Database PL/SQL Language Reference* for information about the ALTER PROCEDURE statement
> - *Oracle Database PL/SQL Language Reference* for information about the ALTER FUNCTION statement

Tutorial: Testing a Standalone Function

This tutorial shows how to use the SQL Developer tool Run to test the standalone function CALCULATE_SCORE.

To test the CALCULATE_SCORE function using the Run tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Functions**.
3. In the list of functions, right-click **CALCULATE_SCORE**.
4. In the list of choices, click **Run**.

   The Run PL/SQL window opens. Its PL/SQL Block frame includes this code:

```
v_Return := CALCULATE_SCORE (
    CAT => CAT,
    SCORE => SCORE,
    WEIGHT => WEIGHT
  );
```

5. Change the values of SCORE and WEIGHT to `8` and `0.2`, respectively:

```
v_Return := CALCULATE_SCORE (
    CAT => CAT,
    SCORE => 8,
```

```
    WEIGHT => 0.2
  );
```

6. Click **OK**.

   Under the Code pane, the Running window opens, showing this result:

```
Connecting to the database hr_conn.
Process exited.
Disconnecting from the database hr_conn.
```

   To the right of the tab Running is the tab Output Variables.

7. Click the tab **Output Variables**.

   Two frames appear, Variable and Value, which contain the values <Return Value> and 1.6, respectively.

> **See Also:**
>
> *Oracle SQL Developer User's Guide* for information about using SQL Developer to run and debug procedures and functions

Dropping Standalone Subprograms

To drop a standalone subprogram, use either the SQL Developer Connections frame and Drop tool, or the DDL statement DROP PROCEDURE or DROP FUNCTION.

> **Caution:**
>
> Do not drop the procedure `ADD_EVALUATION` or the function `CALCULATE_SCORE`—you need them for later tutorials. If you want to practice dropping subprograms, create simple ones and then drop them.

To drop a standalone subprogram using the Drop tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand either **Functions** or **Procedures**.
3. In the list of functions or procedures, right-click the name of the function or procedure to drop.
4. In the list of choices, click **Drop**.
5. In the Drop window, click **Apply**.
6. In the Confirmation window, click **OK**.

> **See Also:**
>
> - "About Data Definition Language (DDL) Statements" for general information that applies to the DROP PROCEDURE and DROP FUNCTION statements
> - *Oracle Database SQL Language Reference* for information about the DROP PROCEDURE statement
> - *Oracle Database SQL Language Reference* for information about the DROP FUNCTION statement

Creating and Managing Packages

You can create and manage PL/SQL packages.

> **See Also:**
>
> "Tutorial: Declaring Variables and Constants in a Subprogram", which shows how to change a package body

About Package Structure

A package always has a specification, and usually has a body. The specification defines the package itself, and is an application program interface (API). The body defines the queries for the declared cursors, and the code for the subprograms, that are declared in the package specification.

The **package specification** defines the package, declaring the types, variables, constants, exceptions, declared cursors, and subprograms that can be referenced from outside the package. A package specification is an **application program interface (API)**: It has all the information that client programs need to invoke its subprograms, but no information about their implementation.

The **package body** defines the queries for the declared cursors, and the code for the subprograms, that are declared in the package specification (therefore, a package with neither declared cursors nor subprograms does not need a body). The package body can also define **local subprograms**, which are not declared in the specification and can be invoked only by other subprograms in the package. Package body contents are hidden from client programs. You can change the package body without invalidating the applications that call the package.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about the package specification
> - *Oracle Database PL/SQL Language Reference* for more information about the package body

Tutorial: Creating a Package Specification

This tutorial shows how to use the Create Package tool to create a specification for a package named EMP_EVAL, which appears in many tutorials and examples in this document.

To create a package specification, use either the SQL Developer tool Create Package or the DDL statement CREATE PACKAGE.

To create a package specification using Create Package tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, right-click **Packages**.
3. In the list of choices, click **New Package**.

   The Create Package window opens. The field Schema has the value HR, the field Name has the default value PACKAGE1, and the check box Add New Source In Lowercase is deselected.
4. For Schema, accept the default value, HR.
5. For Name, change the value PACKAGE1 to `EMP_EVAL`.
6. Click OK.

   The EMP_EVAL pane opens, showing the CREATE PACKAGE statement that created the package:

   ```
   CREATE OR REPLACE PACKAGE emp_eval AS

     /* TODO enter package declarations (types, exceptions, methods etc) here */

   END emp_eval;
   ```

   The title of the pane is in italic font, indicating that the package is not saved to the database.
7. (Optional) In the CREATE PACKAGE statement, replace the comment with declarations.

   If you do not do this step now, you can do it later, as in "Tutorial: Changing a Package Specification".
8. From the File menu, select **Save**.

   Oracle Database compiles the package and saves it. The title of the EMP_EVAL pane is no longer in italic font.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE statement (for the package specification)

Tutorial: Changing a Package Specification

This tutorial shows how to use the Edit tool to change the specification for the EMP_EVAL package, which appears in many tutorials and examples in this document. Specifically, the tutorial shows how to add declarations for a procedure, EVAL_DEPARTMENT, and a function, CALCULATE_SCORE.

To change a package specification, use either the SQL Developer tool Edit or the DDL statement CREATE PACKAGE with the OR REPLACE clause.

To change EMP_EVAL package specification using the Edit tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Packages**.
3. In the list of packages, right-click **EMP_EVAL**.
4. In the list of choices, click **Edit**.

   The EMP_EVAL pane opens, showing the CREATE PACKAGE statement that created the package:

   ```
   CREATE OR REPLACE PACKAGE emp_eval AS

     /* TODO enter package declarations (types, exceptions, methods etc) here */

   END emp_eval;
   ```

   The title of the pane is not in italic font, indicating that the package is saved in the database.
5. In the EMP_EVAL pane, replace the comment with this code:

   ```
   PROCEDURE eval_department ( dept_id IN NUMBER );

   FUNCTION calculate_score ( evaluation_id IN NUMBER
                            , performance_id IN NUMBER)
                            RETURN NUMBER;
   ```

   The title of the EMP_EVAL pane changes to italic font, indicating that the changes have not been saved to the database.
6. Click the icon **Compile**.

   The changed package specification compiles and is saved to the database. The title of the EMP_EVAL pane is no longer in italic font.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE statement with the `OR REPLACE` clause

Tutorial: Creating a Package Body

This tutorial shows how to use the Create Body tool to create a body for the EMP_EVAL package, which appears in many examples and tutorials in this document.

To create a package body, use either the SQL Developer tool Create Body or the DDL statement CREATE PACKAGE BODY.

To create a body for the package EMP_EVAL using the Create Body tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Packages**.
3. In the list of packages, right-click **EMP_EVAL**.
4. In the list of choices, click **Create Body**.

   The EMP_EVAL Body pane appears, showing the automatically generated code for the package body:

```
CREATE OR REPLACE
PACKAGE BODY EMP_EVAL AS

  PROCEDURE eval_department(dept_id IN NUMBER) AS
  BEGIN
    -- TODO implementation required for PROCEDURE EMP_EVAL.eval_department
    NULL;
  END eval_department;

  FUNCTION calculate_score ( evaluation_id IN NUMBER
                           , performance_id IN NUMBER)
                           RETURN NUMBER AS
  BEGIN
    -- TODO implementation required for FUNCTION EMP_EVAL.calculate_score
    RETURN NULL;
  END calculate_score;

END EMP_EVAL;
```

   The title of the pane is in italic font, indicating that the code is not saved in the database.

5. (Optional) In the CREATE PACKAGE BODY statement:
   - Replace the comments with executable statements.
   - (Optional) In the executable part of the procedure, either delete NULL or replace it with an executable statement.
   - (Optional) In the executable part of the function, either replace NULL with another expression.

   If you do not do this step now, you can do it later, as in "Tutorial: Declaring Variables and Constants in a Subprogram".

6. Click the icon **Compile**.

   The changed package body compiles and is saved to the database. The title of the EMP_EVAL Body pane is no longer in italic font.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE BODY statement (for the package body)

Dropping a Package

To drop a package (both specification and body), use either the SQL Developer Connections frame and Drop tool, or the DDL statement DROP PACKAGE.

> **Caution:**
>
> Do not drop the package EMP_EVAL—you need it for later tutorials. If you want to practice dropping packages, create simple ones and then drop them.

To drop a package using the Drop tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Packages**.

   A list of packages appears.
3. In the list of packages, right-click the name of the package to drop.
4. In the list of choices, click **Drop Package**.
5. In the Drop window, click **Apply**.
6. In the Confirmation window, click **OK**.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about the DROP PACKAGE statement

Declaring and Assigning Values to Variables and Constants

A variable or constant declared in a package specification is available to any program that has access to the package. A variable or constant declared in a package body or subprogram is local to that package or subprogram. When declaring a constant, you must assign it an initial value.

One significant advantage that PL/SQL has over SQL is that PL/SQL lets you declare and use variables and constants.

A variable or constant declared in a package specification is available to any program that has access to the package. A variable or constant declared in a package body or subprogram is local to that package or subprogram.

A **variable** holds a value of a particular data type. Your program can change the value at runtime. A **constant** holds a value that cannot be changed.

A variable or constant can have any PL/SQL data type. When declaring a variable, you can assign it an initial value; if you do not, its initial value is NULL. When declaring a constant, you must assign it an initial value. To assign an initial value to a variable or constant, use the assignment operator (`:=`).

> **Tip:**
>
> Declare all values that do not change as constants. This practice optimizes your compiled code and makes your source code easier to maintain.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for general information about variables and constants

Tutorial: Declaring Variables and Constants in a Subprogram

This tutorial shows how to use the SQL Developer tool Edit to declare variables and constants in the EMP_EVAL.CALCULATE_SCORE function. (This tutorial is also an example of changing a package body.)

The EMP_EVAL.CALCULATE_SCORE function is specified in "Tutorial: Creating a Package Specification").

To declare variables and constants in CALCULATE_SCORE function:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Packages**.
3. In the list of packages, expand **EMP_EVAL**.
4. In the list of choices, right-click **EMP_EVAL Body**.

   A list of choices appears.
5. In the list of choices, click **Edit**.

   The EMP_EVAL Body pane appears, showing the code for the package body:

```
CREATE OR REPLACE
PACKAGE BODY EMP_EVAL AS

  PROCEDURE eval_department ( dept_id IN NUMBER ) AS

  BEGIN
    -- TODO implementation required for PROCEDURE EMP_EVAL.eval_department
    NULL;
  END eval_department;

  FUNCTION calculate_score ( evaluation_id IN NUMBER
                           , performance_id IN NUMBER)
```

```
                              RETURN NUMBER AS
  BEGIN
    -- TODO implementation required for FUNCTION EMP_EVAL.calculate_score
    RETURN NULL;
  END calculate_score;

END EMP_EVAL;
```

6. Between `RETURN NUMBER AS` and `BEGIN`, add these variable and constant declarations:

```
n_score       NUMBER(1,0);               -- variable
n_weight      NUMBER;                    -- variable
max_score     CONSTANT NUMBER(1,0) := 9;  -- constant, initial value 9
max_weight    CONSTANT NUMBER(8,8) := 1;  -- constant, initial value 1
```

   The title of the EMP_EVAL Bodypane changes to italic font, indicating that the code is not saved in the database.

7. From the File menu, select **Save**.

   Oracle Database compiles and saves the changed package body. The title of the EMP_EVAL Body pane is no longer in italic font.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for general information about declaring variables and constants
> - "Assigning Values to Variables with the Assignment Operator"

Ensuring that Variables, Constants, and Parameters Have Correct Data Types

Ensure that variables, constants, and parameters have the correct data types by declaring them with the %TYPE attribute.

After "Tutorial: Declaring Variables and Constants in a Subprogram", the code for the EMP_EVAL.CALCULATE_SCORE function is:

```
FUNCTION calculate_score ( evaluation_id IN NUMBER
                         , performance_id IN NUMBER )
                         RETURN NUMBER AS
  n_score       NUMBER(1,0);               -- variable
  n_weight      NUMBER;                    -- variable
  max_score     CONSTANT NUMBER(1,0) := 9;  -- constant, initial value 9
  max_weight    CONSTANT NUMBER(8,8) := 1;  -- constant, initial value 1
  BEGIN
    -- TODO implementation required for FUNCTION EMP_EVAL.calculate_score
    RETURN NULL;
  END calculate_score;
```

The variables, constants, and parameters of the function represent values from the tables SCORES and PERFORMANCE_PARTS (created in "Creating Tables"):

- Variable n_score will hold a value from the column SCORE.SCORES and constant max_score will be compared to such values.

- Variable n_weight will hold a value from the column PERFORMANCE_PARTS.WEIGHT and constant max_weight will be compared to such values.
- Parameter evaluation_id will hold a value from the column SCORE.EVALUATION_ID.
- Parameter performance_id will hold a value from the column SCORE.PERFORMANCE_ID.

Therefore, each variable, constant, and parameter has the same data type as its corresponding column.

If the data types of the columns change, you want the data types of the variables, constants, and parameters to change to the same data types; otherwise, the CALCULATE_SCORE function is invalidated.

To ensure that the data types of the variables, constants, and parameters always match those of the columns, declare them with the %TYPE attribute. The %TYPE attribute supplies the data type of a table column or another variable, ensuring the correct data type assignment.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about the `%TYPE` attribute
- *Oracle Database PL/SQL Language Reference* for the syntax of the %TYPE attribute

Tutorial: Changing Declarations to Use the %TYPE Attribute

This tutorial shows how to use the SQL Developer tool Edit to change the declarations of the variables, constants, and formal parameters of the EMP_EVAL.CALCULATE_SCORE function to use the %TYPE attribute.

The EMP_EVAL.CALCULATE_SCORE function is shown in "Tutorial: Declaring Variables and Constants in a Subprogram".

To change the declarations in CALCULATE_SCORE to use %TYPE:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Packages**.
3. In the list of packages, expand **EMP_EVAL**.
4. In the list of choices, right-click **EMP_EVAL Body**.
5. In the list of choices, click **Edit**.

   The EMP_EVAL Bodypane appears, showing the code for the package body:

```
CREATE OR REPLACE
PACKAGE BODY emp_eval AS

  PROCEDURE eval_department ( dept_id IN NUMBER ) AS
  BEGIN
```

```
    -- TODO implementation required for PROCEDURE EMP_EVAL.eval_department
    NULL;
  END eval_department;

  FUNCTION calculate_score ( evaluation_id IN NUMBER
                           , performance_id IN NUMBER )
                           RETURN NUMBER AS
  n_score       NUMBER(1,0);               -- variable
  n_weight      NUMBER;                    -- variable
  max_score     CONSTANT NUMBER(1,0) := 9;  -- constant, initial value 9
  max_weight    CONSTANT NUMBER(8,8) := 1;  -- constant, initial value 1
  BEGIN
    -- TODO implementation required for FUNCTION EMP_EVAL.calculate_score
    RETURN NULL;
  END calculate_score;

END emp_eval;
```

6. In the code for the function, make the changes shown in bold font:

```
  FUNCTION calculate_score ( evaluation_id IN SCORES.EVALUATION_ID%TYPE
                           , performance_id IN SCORES.PERFORMANCE_ID%TYPE)
                           RETURN NUMBER AS
  n_score       SCORES.SCORE%TYPE;
  n_weight      PERFORMANCE_PARTS.WEIGHT%TYPE;
  max_score     CONSTANT SCORES.SCORE%TYPE := 9;
  max_weight    CONSTANT PERFORMANCE_PARTS.WEIGHT%TYPE := 1;
```

7. Right-click **EMP_EVAL**.
8. In the list of choices, click **Edit**.

   The EMP_EVAL paneopens, showing the CREATE PACKAGE statement that created the package:

```
CREATE OR REPLACE PACKAGE EMP_EVAL AS

PROCEDURE eval_department(dept_id IN NUMBER);
FUNCTION calculate_score(evaluation_id IN NUMBER
                       , performance_id IN NUMBER)
                         RETURN NUMBER;

END EMP_EVAL;
```

9. In the code for the function, make the changes shown in bold font:

```
FUNCTION calculate_score(evaluation_id IN scores.evaluation_id%TYPE
                       , performance_id IN scores.performance_id%TYPE)
```

10. Right-click **EMP_EVAL**.
11. In the list of choices, click **Compile**.
12. Right-click **EMP_EVAL Body**.
13. In the list of choices, click **Compile**.

Assigning Values to Variables

You can assign a value to a variable in these ways:

- Use the assignment operator to assign it the value of an expression.
- Use the SELECT INTO or FETCH statement to assign it a value from a table.

- Pass it to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram.
- Bind the variable to a value.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about assigning values to variables
> - *Oracle Database Get Started with Java Development* for information about binding variables

Assigning Values to Variables with the Assignment Operator

With the assignment operator (:=), you can assign the value of an expression to a variable in either the declarative or executable part of a subprogram.

In the declarative part of a subprogram, you can assign an initial value to a variable when you declare it. The syntax is:

```
variable_name data_type := expression;
```

In the executable part of a subprogram, you can assign a value to a variable with an assignment statement. The syntax is:

```
variable_name := expression;
```

Example 5-1 shows, in bold font, the changes to make to the EMP_EVAL.CALCULATE_SCORE function to add a variable, running_total, and use it as the return value of the function. The assignment operator appears in both the declarative and executable parts of the function. (The data type of running_total must be NUMBER, rather than SCORES.SCORE%TYPE or PERFORMANCE_PARTS.WEIGHT%TYPE, because it holds the product of two NUMBER values with different precisions and scales.)

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for variable declaration syntax
> - *Oracle Database PL/SQL Language Reference* for assignment statement syntax

Example 5-1 Assigning Values to a Variable with Assignment Operator

```
FUNCTION calculate_score(evaluation_id IN SCORES.EVALUATION_ID%TYPE
                       , performance_id IN SCORES.PERFORMANCE_ID%TYPE)
                       RETURN NUMBER AS
  n_score       SCORES.SCORE%TYPE;
  n_weight      PERFORMANCE_PARTS.WEIGHT%TYPE;
  running_total NUMBER := 0;
  max_score     CONSTANT SCORES.SCORE%TYPE := 9;
```

```
  max_weight    CONSTANT PERFORMANCE_PARTS.WEIGHT%TYPE:= 1;
BEGIN
  running_total := max_score * max_weight;
  RETURN running_total;
END calculate_score;
```

Assigning Values to Variables with the SELECT INTO Statement

To use table values in subprograms or packages, you must assign them to variables with SELECT INTO statements.

Example 5-2 shows, in bold font, the changes to make to the EMP_EVAL.CALCULATE_SCORE function to have it calculate running_total from table values.

The ADD_EVAL procedure in Example 5-3 inserts a row into the EVALUATIONS table, using values from the corresponding row in the EMPLOYEES table. Add the ADD_EVAL procedure to the body of the EMP_EVAL package, but not to the specification. Because it is not in the specification, ADD_EVAL is local to the package—it can be invoked only by other subprograms in the package, not from outside the package.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the SELECT INTO statement

Example 5-2 Assigning Table Values to Variables with SELECT INTO

```
FUNCTION calculate_score ( evaluation_id IN scores.evaluation_id%TYPE
                         , performance_id IN scores.performance_id%TYPE )
                         RETURN NUMBER AS

  n_score       scores.score%TYPE;
  n_weight      performance_parts.weight%TYPE;
  running_total NUMBER := 0;
  max_score     CONSTANT scores.score%TYPE := 9;
  max_weight    CONSTANT performance_parts.weight%TYPE:= 1;
BEGIN
  SELECT s.score INTO n_score
  FROM SCORES s
  WHERE evaluation_id = s.evaluation_id
  AND performance_id = s.performance_id;

  SELECT p.weight INTO n_weight
  FROM PERFORMANCE_PARTS p
  WHERE performance_id = p.performance_id;

  running_total := n_score * n_weight;
  RETURN running_total;
END calculate_score;
```

Example 5-3 Inserting a Table Row with Values from Another Table

```
PROCEDURE add_eval ( employee_id IN EMPLOYEES.EMPLOYEE_ID%TYPE
                   , today IN DATE )
AS
  job_id         EMPLOYEES.JOB_ID%TYPE;
```

```
  manager_id     EMPLOYEES.MANAGER_ID%TYPE;
  department_id  EMPLOYEES.DEPARTMENT_ID%TYPE;
BEGIN
  INSERT INTO EVALUATIONS (
    evaluation_id,
    employee_id,
    evaluation_date,
    job_id,
    manager_id,
    department_id,
    total_score
  )
  SELECT
    evaluations_sequence.NEXTVAL,   -- evaluation_id
    add_eval.employee_id,      -- employee_id
    add_eval.today,            -- evaluation_date
    e.job_id,                  -- job_id
    e.manager_id,              -- manager_id
    e.department_id,           -- department_id
    0                          -- total_score
  FROM employees e;

  IF SQL%ROWCOUNT = 0 THEN
    RAISE NO_DATA_FOUND;
  END IF;
END add_eval;
```

Controlling Program Flow

Unlike SQL, which runs statements in the order in which you enter them, PL/SQL has control statements that let you control the flow of your program.

About Control Statements

PL/SQL has three categories of control statements: conditional selection statements, loop statements, and sequential control statements.

Conditional selection statements let you run different statements for different data values. The conditional selection statements are `IF` and `CASE`.

Loop statements let you repeat the same statements with a series of different data values. The loop statements are `FOR LOOP`, `WHILE LOOP`, and basic `LOOP`. The `EXIT` statement transfers control to the end of a loop. The `CONTINUE` statement exits the current iteration of a loop and transfers control to the next iteration. Both `EXIT` and `CONTINUE` have an optional `WHEN` clause, in which you can specify a condition.

Sequential control statements let you go to a specified labeled statement or to do nothing. The sequential control statements are `GOTO` and `NULL`.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for an overview of PL/SQL control statements

Using the IF Statement

The IF statement either runs or skips a sequence of statements, depending on the value of a Boolean expression.

The IF statement has this syntax:

```
IF boolean_expression THEN statement [, statement ]
[ ELSIF boolean_expression THEN statement [, statement ] ]...
[ ELSE  statement [, statement ] ]
END IF;
```

Suppose that your company evaluates employees twice a year in the first 10 years of employment, but only once a year afterward. You want a function that returns the evaluation frequency for an employee. You can use an IF statement to determine the return value of the function, as in Example 5-4.

Add the EVAL_FREQUENCY function to the body of the EMP_EVAL package, but not to the specification. Because it is not in the specification, EVAL_FREQUENCY is local to the package—it can be invoked only by other subprograms in the package, not from outside the package.

> **Tip:**
>
> When using a PL/SQL variable in a SQL statement, as in the second SELECT statement in Example 5-4, qualify the variable with the subprogram name to ensure that it is not mistaken for a table column.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for the syntax of the IF statement
> - *Oracle Database PL/SQL Language Reference* for more information about using the IF statement

Example 5-4 IF Statement that Determines Return Value of Function

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date     EMPLOYEES.HIRE_DATE%TYPE;
  today      EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq  PLS_INTEGER;
BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = eval_frequency.emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
```

```
    eval_freq := 1;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

Using the CASE Statement

The CASE statement chooses from a sequence of conditions, and runs the corresponding statement.

The simple CASE statement evaluates a single expression and compares it to several potential values. It has this syntax:

```
CASE expression
WHEN value THEN statement
[ WHEN value THEN statement ]...
[ ELSE statement [, statement ]... ]
END CASE;
```

The searched CASE statement evaluates multiple Boolean expressions and chooses the first one whose value is TRUE. For information about the searched CASE statement, see *Oracle Database PL/SQL Language Reference*.

> **Tip:**
>
> When you can use either a CASE statement or nested IF statements, use a CASE statement—it is both more readable and more efficient.

Suppose that, if an employee is evaluated only once a year, you want the EVAL_FREQUENCY function to suggest a salary increase, which depends on the JOB_ID.

Change the EVAL_FREQUENCY function as shown in bold font in Example 5-5. (For information about the procedures that prints the strings, DBMS_OUTPUT.PUT_LINE, see *Oracle Database PL/SQL Packages and Types Reference*.)

Example 5-5 CASE Statement that Determines Which String to Print

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date     EMPLOYEES.HIRE_DATE%TYPE;
  today      EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq  PLS_INTEGER;
  j_id       EMPLOYEES.JOB_ID%TYPE;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE, JOB_ID INTO h_date, j_id
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = eval_frequency.emp_id;

  IF ((h_date + (INTERVAL '12' MONTH)) < today) THEN
```

```
    eval_freq := 1;

    CASE j_id
       WHEN 'PU_CLERK' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 8% salary increase for employee # ' || emp_id);
       WHEN 'SH_CLERK' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 7% salary increase for employee # ' || emp_id);
       WHEN 'ST_CLERK' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 6% salary increase for employee # ' || emp_id);
       WHEN 'HR_REP' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 5% salary increase for employee # ' || emp_id);
       WHEN 'PR_REP' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 5% salary increase for employee # ' || emp_id);
       WHEN 'MK_REP' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 4% salary increase for employee # ' || emp_id);
       ELSE DBMS_OUTPUT.PUT_LINE(
         'Nothing to do for employee #' || emp_id);
    END CASE;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

See Also:

- "Using CASE Expressions in Queries"
- *Oracle Database PL/SQL Language Reference* for the syntax of the CASE statement
- *Oracle Database PL/SQL Language Reference* for more information about using the CASE statement

Using the FOR LOOP Statement

The FOR LOOP statement repeats a sequence of statements once for each integer in the range lower_bound through upper_bound.

The syntax of the FOR LOOP is:

```
FOR counter IN lower_bound..upper_bound LOOP
  statement [, statement ]...
END LOOP;
```

The statements between LOOP and END LOOP can use counter, but cannot change its value.

Suppose that, instead of only suggesting a salary increase, you want the EVAL_FREQUENCY function to report what the salary would be if it increased by the suggested amount every year for five years.

Change the EVAL_FREQUENCY function as shown in bold font in Example 5-6. (For information about the procedure that prints the strings, `DBMS_OUTPUT.PUT_LINE`, see *Oracle Database PL/SQL Packages and Types Reference*.)

Example 5-6 FOR LOOP Statement that Computes Salary After Five Years

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date      EMPLOYEES.HIRE_DATE%TYPE;
  today       EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq   PLS_INTEGER;
  j_id        EMPLOYEES.JOB_ID%TYPE;
  sal         EMPLOYEES.SALARY%TYPE;
  sal_raise   NUMBER(3,3) := 0;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE, JOB_ID, SALARY INTO h_date, j_id, sal
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = eval_frequency.emp_id;

  IF ((h_date + (INTERVAL '12' MONTH)) < today) THEN
    eval_freq := 1;

    CASE j_id
      WHEN 'PU_CLERK' THEN sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN sal_raise := 0.07;
      WHEN 'ST_CLERK' THEN sal_raise := 0.06;
      WHEN 'HR_REP'   THEN sal_raise := 0.05;
      WHEN 'PR_REP'   THEN sal_raise := 0.05;
      WHEN 'MK_REP'   THEN sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (sal_raise != 0) THEN
      BEGIN
        DBMS_OUTPUT.PUT_LINE('If salary ' || sal || ' increases by ' ||
          ROUND((sal_raise * 100),0) ||
          '% each year for 5 years, it will be:');

        FOR i IN 1..5 LOOP
          sal := sal * (1 + sal_raise);
          DBMS_OUTPUT.PUT_LINE(ROUND(sal, 2) || ' after ' || i || ' year(s)');
        END LOOP;
      END;
    END IF;

  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for the syntax of the FOR LOOP statement
> - *Oracle Database PL/SQL Language Reference* for more information about using the FOR LOOP statement

Using the WHILE LOOP Statement

The WHILE LOOP statement repeats a sequence of statements while a condition is TRUE.

The syntax of the WHILE LOOP statement is:

```
WHILE condition LOOP
  statement [, statement ]...
END LOOP;
```

> **Note:**
>
> If the statements between LOOP and END LOOP never cause condition to become FALSE, then the WHILE LOOP statement runs indefinitely.

Suppose that the EVAL_FREQUENCY function uses the WHILE LOOP statement instead of the FOR LOOP statement and ends after the proposed salary exceeds the maximum salary for the JOB_ID.

Change the EVAL_FREQUENCY function as shown in bold font in Example 5-7. (For information about the procedures that prints the strings, DBMS_OUTPUT.PUT_LINE, see *Oracle Database PL/SQL Packages and Types Reference*.)

Example 5-7 WHILE LOOP Statement that Computes Salary to Maximum

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date      EMPLOYEES.HIRE_DATE%TYPE;
  today       EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq   PLS_INTEGER;
  j_id        EMPLOYEES.JOB_ID%TYPE;
  sal         EMPLOYEES.SALARY%TYPE;
  sal_raise   NUMBER(3,3) := 0;
  sal_max     JOBS.MAX_SALARY%TYPE;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE, j.JOB_ID, SALARY, MAX_SALARY INTO h_date, j_id, sal, sal_max
  FROM EMPLOYEES e, JOBS j
  WHERE EMPLOYEE_ID = eval_frequency.emp_id AND JOB_ID = eval_frequency.j_id;

  IF ((h_date + (INTERVAL '12' MONTH)) < today) THEN
    eval_freq := 1;
```

```
    CASE j_id
      WHEN 'PU_CLERK' THEN sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN sal_raise := 0.07;
      WHEN 'ST_CLERK' THEN sal_raise := 0.06;
      WHEN 'HR_REP'   THEN sal_raise := 0.05;
      WHEN 'PR_REP'   THEN sal_raise := 0.05;
      WHEN 'MK_REP'   THEN sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (sal_raise != 0) THEN
      BEGIN
        DBMS_OUTPUT.PUT_LINE('If salary ' || sal || ' increases by ' ||
          ROUND((sal_raise * 100),0) ||
          '% each year, it will be:');

        WHILE sal <= sal_max LOOP
          sal := sal * (1 + sal_raise);
          DBMS_OUTPUT.PUT_LINE(ROUND(sal, 2));
        END LOOP;

        DBMS_OUTPUT.PUT_LINE('Maximum salary for this job is ' || sal_max);
      END;
    END IF;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

See Also:

- *Oracle Database PL/SQL Language Reference* for the syntax of the WHILE LOOP statement
- *Oracle Database PL/SQL Language Reference* for more information about using the WHILE LOOP statement

Using the Basic LOOP and EXIT WHEN Statements

The basic LOOP statement repeats a sequence of statements.

The syntax of the basic LOOP statement is:

```
LOOP
  statement [, statement ]...
END LOOP;
```

At least one statement must be an EXIT statement; otherwise, the LOOP statement runs indefinitely.

The EXIT WHEN statement (the EXIT statement with its optional WHEN clause) exits a loop when a condition is TRUE and transfers control to the end of the loop.

In the EVAL_FREQUENCY function, in the last iteration of the WHILE LOOP statement, the last computed value usually exceeds the maximum salary.

Change the WHILE LOOP statement to a basic LOOP statement that includes an EXIT WHEN statement, as in Example 5-8.

Example 5-8 Using the EXIT WHEN Statement

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date      EMPLOYEES.HIRE_DATE%TYPE;
  today       EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq   PLS_INTEGER;
  j_id        EMPLOYEES.JOB_ID%TYPE;
  sal         EMPLOYEES.SALARY%TYPE;
  sal_raise   NUMBER(3,3) := 0;
  sal_max     JOBS.MAX_SALARY%TYPE;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE, j.JOB_ID, SALARY, MAX_SALARY INTO h_date, j_id, sal, sal_max
  FROM EMPLOYEES e, JOBS j
  WHERE EMPLOYEE_ID = eval_frequency.emp_id AND JOB_ID = eval_frequency.j_id;

  IF ((h_date + (INTERVAL '12' MONTH)) < today) THEN
    eval_freq := 1;

    CASE j_id
      WHEN 'PU_CLERK' THEN sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN sal_raise := 0.07;
      WHEN 'ST_CLERK' THEN sal_raise := 0.06;
      WHEN 'HR_REP'   THEN sal_raise := 0.05;
      WHEN 'PR_REP'   THEN sal_raise := 0.05;
      WHEN 'MK_REP'   THEN sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (sal_raise != 0) THEN
      BEGIN
        DBMS_OUTPUT.PUT_LINE('If salary ' || sal || ' increases by ' ||
          ROUND((sal_raise * 100),0) ||
          '% each year, it will be:');

        LOOP
          sal := sal * (1 + sal_raise);
          EXIT WHEN sal > sal_max;
          DBMS_OUTPUT.PUT_LINE(ROUND(sal,2));
        END LOOP;

        DBMS_OUTPUT.PUT_LINE('Maximum salary for this job is ' || sal_max);
      END;
    END IF;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

See Also:

- *Oracle Database PL/SQL Language Reference* for the syntax of the LOOP statement
- *Oracle Database PL/SQL Language Reference* for the syntax of the EXIT statement
- *Oracle Database PL/SQL Language Reference* for more information about using the LOOP and EXIT statements

Using Records and Cursors

You can store data values in records, and use a cursor as a pointer to a result set and related processing information.

See Also:

Oracle Database PL/SQL Language Reference for more information about records

About Records

A **record** is a PL/SQL composite variable that can store data values of different types. You can treat Internal components (**fields**) like scalar variables. You can pass entire records as subprogram parameters. Records are useful for holding data from table rows, or from certain columns of table rows.

A **record** is a PL/SQL composite variable that can store data values of different types, similar to a struct type in C, C++, or Java. The internal components of a record are called **fields**. To access a record field, you use **dot notation**: record_name.field_name.

You can treat record fields like scalar variables. You can also pass entire records as subprogram parameters.

Records are useful for holding data from table rows, or from certain columns of table rows. Each record field corresponds to a table column.

There are three ways to create a record:

- Declare a **RECORD** type and then declare a variable of that type.

  The syntax is:

```
TYPE record_name IS RECORD
  ( field_name data_type [:= initial_value]
 [, field_name data_type [:= initial_value ] ]... );

variable_name record_name;
```

- Declare a variable of the type table_name%ROWTYPE.

The fields of the record have the same names and data types as the columns of the table.

- Declare a variable of the type cursor_name%ROWTYPE.

  The fields of the record have the same names and data types as the columns of the table in the FROM clause of the cursor SELECT statement.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about defining RECORD types and declaring records of that type
> - *Oracle Database PL/SQL Language Reference* for the syntax of a RECORD type definition
> - *Oracle Database PL/SQL Language Reference* for more information about the %ROWTYPE attribute
> - *Oracle Database PL/SQL Language Reference* for the syntax of the %ROWTYPE attribute

Tutorial: Declaring a RECORD Type

This tutorial shows how to use the SQL Developer tool Edit to declare a RECORD type, sal_info, whose fields can hold salary information for an employee—job ID, minimum and maximum salary for that job ID, current salary, and suggested raise.

To declare RECORD type sal_info:

1. In the Connections frame, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Right-click **EMP_EVAL**.

   A list of choices appears.

4. Click **Edit**.

   The EMP_EVAL pane opens, showing the `CREATE PACKAGE` statement that created the package:

   ```
   CREATE OR REPLACE PACKAGE EMP_EVAL AS

   PROCEDURE eval_department(dept_id IN NUMBER);
   FUNCTION calculate_score(evaluation_id IN NUMBER
                          , performance_id IN NUMBER)
                            RETURN NUMBER;

   END EMP_EVAL;
   ```

5. In the EMP_EVAL pane, immediately before `END EMP_EVAL`, add this code:

   ```
   TYPE sal_info IS RECORD
     ( j_id     jobs.job_id%type
   ```

```
, sal_min  jobs.min_salary%type
, sal_max  jobs.max_salary%type
, sal      employees.salary%type
, sal_raise NUMBER(3,3) );
```

The title of the pane is in italic font, indicating that the changes have not been saved to the database.

6. Click the icon **Compile**.

   The changed package specification compiles and is saved to the database. The title of the EMP_EVAL pane is no longer in italic font.

   Now you can declare records of the type sal_info, as in "Tutorial: Creating and Invoking a Subprogram with a Record Parameter".

Tutorial: Creating and Invoking a Subprogram with a Record Parameter

This tutorial shows how to use the SQL Developer tool Edit to create and invoke a subprogram with a parameter of the record type sal_info.

The record type sal_info was created in "Tutorial: Declaring a RECORD Type".

This tutorial shows how to use the SQL Developer tool Edit to do the following:

- Create a procedure, SALARY_SCHEDULE, which has a parameter of type sal_info.
- Change the EVAL_FREQUENCY function so that it declares a record, emp_sal, of the type sal_info, populates its fields, and passes it to the SALARY_SCHEDULE procedure.

Because EVAL_FREQUENCY will invoke SALARY_SCHEDULE, the declaration of SALARY_SCHEDULE must precede the declaration of EVAL_FREQUENCY (otherwise the package will not compile). However, the definition of SALARY_SCHEDULE can be anywhere in the package body.

To create SALARY_SCHEDULE and change EVAL_FREQUENCY:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Packages**.
3. In the list of packages, expand **EMP_EVAL**.
4. In the list of choices, right-click **EMP_EVAL Body**.
5. In the list of choices, click **Edit**.

   The EMP_EVAL Bodypane appears, showing the code for the package body.

6. In the EMP_EVAL Body pane, immediately before `END EMP_EVAL`, add this definition of the SALARY_SCHEDULE procedure:

```
PROCEDURE salary_schedule (emp IN sal_info) AS
  accumulating_sal  NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('If salary ' || emp.sal ||
    ' increases by ' || ROUND((emp.sal_raise * 100),0) ||
    '% each year, it will be:');
```

```
  accumulating_sal := emp.sal;

  WHILE accumulating_sal <= emp.sal_max LOOP
    accumulating_sal := accumulating_sal * (1 + emp.sal_raise);
    DBMS_OUTPUT.PUT_LINE(ROUND(accumulating_sal,2) ||', ');
  END LOOP;
END salary_schedule;
```

The title of the pane is in italic font, indicating that the changes have not been saved to the database.

7. In the EMP_EVAL Body pane, enter the code shown in bold font, in this position:

```
CREATE OR REPLACE
PACKAGE BODY EMP_EVAL AS

FUNCTION eval_frequency (emp_id EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER;
PROCEDURE salary_schedule(emp IN sal_info);
PROCEDURE add_eval(employee_id IN employees.employee_id%type, today IN DATE);

PROCEDURE eval_department (dept_id IN NUMBER) AS
```

8. Edit the `EVAL_FREQUENCY` function, making the changes shown in bold font:

```
FUNCTION eval_frequency (emp_id EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date     EMPLOYEES.HIRE_DATE%TYPE;
  today      EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq  PLS_INTEGER;
  emp_sal    SAL_INFO;  -- replaces sal, sal_raise, and sal_max

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = eval_frequency.emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
     eval_freq := 1;

     /* populate emp_sal */

     SELECT j.JOB_ID, j.MIN_SALARY, j.MAX_SALARY, e.SALARY
     INTO emp_sal.j_id, emp_sal.sal_min, emp_sal.sal_max, emp_sal.sal
     FROM EMPLOYEES e, JOBS j
     WHERE e.EMPLOYEE_ID = eval_frequency.emp_id
     AND j.JOB_ID = eval_frequency.emp_id;

     emp_sal.sal_raise := 0;  -- default

     CASE emp_sal.j_id
       WHEN 'PU_CLERK' THEN emp_sal.sal_raise := 0.08;
       WHEN 'SH_CLERK' THEN emp_sal.sal_raise := 0.07;
       WHEN 'ST_CLERK' THEN emp_sal.sal_raise := 0.06;
       WHEN 'HR_REP' THEN emp_sal.sal_raise := 0.05;
       WHEN 'PR_REP' THEN emp_sal.sal_raise := 0.05;
       WHEN 'MK_REP' THEN emp_sal.sal_raise := 0.04;
       ELSE NULL;
     END CASE;
```

```
      IF (emp_sal.sal_raise != 0) THEN
        salary_schedule(emp_sal);
      END IF;
    ELSE
      eval_freq := 2;
    END IF;

    RETURN eval_freq;
  END eval_frequency;
```

9. Click **Compile**.

About Cursors

When Oracle Database runs a SQL statement, it stores the result set and processing information in an unnamed private SQL area. A pointer to this unnamed area, called a **cursor**, lets you retrieve the result set one row at a time. **Cursor attributes** return information about the state of the cursor.

Every time you run either a SQL DML statement or a PL/SQL SELECT INTO statement, PL/SQL opens an **implicit cursor**. You can get information about this cursor from its attributes, but you cannot control it. After the statement runs, the database closes the cursor; however, its attribute values remain available until another DML or SELECT INTO statement runs.

PL/SQL also lets you declare cursors. A **declared cursor** has a name and is associated with a query (SQL SELECT statement)—usually one that returns multiple rows. After declaring a cursor, you must process it, either implicitly or explicitly. To process the cursor implicitly, use a cursor FOR LOOP. The syntax is:

```
FOR record_name IN cursor_name LOOP
  statement
  [ statement ]...
END LOOP;
```

To process the cursor explicitly, open it (with the OPEN statement), fetch rows from the result set either one at a time or in bulk (with the FETCH statement), and close the cursor (with the CLOSE statement). After closing the cursor, you can neither fetch records from the result set nor see the cursor attribute values.

The syntax for the value of an implicit cursor attribute is SQL%attribute (for example, SQL%FOUND). SQL%attribute always refers to the most recently run DML or SELECT INTO statement.

The syntax for the value of a declared cursor attribute is cursor_name%attribute (for example, c1%FOUND).

Table 5-1 lists the cursor attributes and the values that they can return. (Implicit cursors have additional attributes that are beyond the scope of this book.)

Table 5-1 Cursor Attribute Values

Attribute	Values for Declared Cursor	Values for Implicit Cursor
%FOUND	If cursor is open[1] but no fetch was attempted, NULL.	If no DML or SELECT INTO statement has run, NULL.
	If the most recent fetch returned a row, TRUE.	If the most recent DML or SELECT INTOstatement returned a row, TRUE.
	If the most recent fetch did not return a row, FALSE.	If the most recent DML or SELECT INTOstatement did not return a row, FALSE.
%NOTFOUND	If cursor is open[1] but no fetch was attempted, NULL.	If no DML or SELECT INTO statement has run, NULL.
	If the most recent fetch returned a row, FALSE.	If the most recent DML or SELECT INTOstatement returned a row, `FALSE`.
	If the most recent fetch did not return a row, TRUE.	If the most recent DML or SELECT INTO statement did not return a row, `TRUE`.
%ROWCOUNT	If cursor is open[1], a number greater than or equal to zero.	NULL if no DML or SELECT INTO statement has run; otherwise, a number greater than or equal to zero.
%ISOPEN	If cursor is open, TRUE; if not, FALSE.	Always FALSE.

1 If the cursor is not open, the attribute raises the predefined exception INVALID_CURSOR.

See Also:

- "About Queries"
- "About Data Manipulation Language (DML) Statements"
- *Oracle Database PL/SQL Language Reference* for more information about the SELECT INTO statement
- *Oracle Database PL/SQL Language Reference* for more information about managing cursors in PL/SQL

Using a Declared Cursor to Retrieve Result Set Rows One at a Time

You can use a declared cursor to retrieve result set rows one at a time.

The following procedure uses each necessary statement in its simplest form, but provides references to its complete syntax.

To use a declared cursor to retrieve result set rows one at a time:

1. In the declarative part:
   a. Declare the cursor:

      ```
      CURSOR cursor_name IS query;
      ```

      For complete declared cursor declaration syntax, see *Oracle Database PL/SQL Language Reference*.

b. Declare a record to hold the row returned by the cursor:

```
record_name cursor_name%ROWTYPE;
```

For complete %ROWTYPE syntax, see *Oracle Database PL/SQL Language Reference*.

2. In the executable part:

a. Open the cursor:

```
OPEN cursor_name;
```

For complete OPEN statement syntax, see *Oracle Database PL/SQL Language Reference*.

b. Fetch rows from the cursor (rows from the result set) one at a time, using a LOOP statement that has syntax similar to this:

```
LOOP
  FETCH cursor_name INTO record_name;
  EXIT WHEN cursor_name%NOTFOUND;
  -- Process row that is in record_name:
  statement;
  [ statement; ]...
END LOOP;
```

For complete FETCH statement syntax, see *Oracle Database PL/SQL Language Reference*.

c. Close the cursor:

```
CLOSE cursor_name;
```

For complete CLOSE statement syntax, see *Oracle Database PL/SQL Language Reference*.

Tutorial: Using a Declared Cursor to Retrieve Result Set Rows One at a Time

This tutorial shows how to implement the procedure EMP_EVAL.EVAL_DEPARTMENT, which uses a declared cursor, emp_cursor.

To implement the EMP_EVAL.EVAL_DEPARTMENT procedure:

1. In the EMP_EVAL package specification, change the declaration of the EVAL_DEPARTMENT procedure as shown in bold font:

```
PROCEDURE eval_department(dept_id IN employees.department_id%TYPE);
```

2. In the EMP_EVAL package body, change the definition of the EVAL_DEPARTMENT procedure as shown in bold font:

```
PROCEDURE eval_department (dept_id IN employees.department_id%TYPE)
AS
  CURSOR emp_cursor IS
    SELECT * FROM EMPLOYEES
    WHERE DEPARTMENT_ID = eval_department.dept_id;

  emp_record  EMPLOYEES%ROWTYPE;  -- for row returned by cursor
  all_evals   BOOLEAN;  -- true if all employees in dept need evaluations
```

```
  today       DATE;

BEGIN
  today := SYSDATE;

  IF (EXTRACT(MONTH FROM today) < 6) THEN
    all_evals := FALSE; -- only new employees need evaluations
  ELSE
    all_evals := TRUE;  -- all employees need evaluations
  END IF;

  OPEN emp_cursor;

  DBMS_OUTPUT.PUT_LINE (
    'Determining evaluations necessary in department # ' ||
    dept_id );

  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;

    IF all_evals THEN
      add_eval(emp_record.employee_id, today);
    ELSIF (eval_frequency(emp_record.employee_id) = 2) THEN
      add_eval(emp_record.employee_id, today);
    END IF;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Processed ' || emp_cursor%ROWCOUNT || ' records.');

  CLOSE emp_cursor;
END eval_department;
```

(For a step-by-step example of changing a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram".)

3. Compile the EMP_EVAL package specification.
4. Compile the EMP_EVAL package body.

About Cursor Variables

A **cursor variable** is like a cursor that it is not limited to one query. You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query. Cursor variables are useful for passing query results between subprograms.

For information about cursors, see "About Cursors".

To declare a cursor variable, you declare a REF CURSOR type, and then declare a variable of that type (therefore, a cursor variable is often called a **REF CURSOR**). A REF CURSOR type can be either strong or weak.

A **strong REF CURSOR type** specifies a **return type**, which is the RECORD type of its cursor variables. The PL/SQL compiler does not allow you to use these **strongly typed** cursor variables for queries that return rows that are not of the return type. Strong REF CURSOR types are less error-prone than weak ones, but weak ones are more flexible.

A **weak REF CURSOR type** does not specify a return type. The PL/SQL compiler accepts **weakly typed** cursor variables in any queries. Weak REF CURSOR types are

interchangeable; therefore, instead of creating weak REF CURSOR types, you can use the predefined type weak cursor type SYS_REFCURSOR.

After declaring a cursor variable, you must open it for a specific query (with the OPEN FOR statement), fetch rows one at a time from the result set (with the FETCH statement), and then either close the cursor (with the CLOSE statement) or open it for another specific query (with the OPEN FOR statement). Opening the cursor variable for another query closes it for the previous query. After closing a cursor variable for a specific query, you can neither fetch records from the result set of that query nor see the cursor attribute values for that query.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about using cursor variables
- *Oracle Database PL/SQL Language Reference* for the syntax of cursor variable declaration

Using a Cursor Variable to Retrieve Result Set Rows One at a Time

You can use a cursor variable to retrieve result set rows one at a time.

The following procedure uses each of the necessary statements in its simplest form, but provides references to their complete syntax.

To use a cursor variable to retrieve result set rows one at a time:

1. In the declarative part:
   a. Declare the REF CURSOR type:

      ```
      TYPE cursor_type IS REF CURSOR [ RETURN return_type ];
      ```

      For complete REF CURSOR type declaration syntax, see *Oracle Database PL/SQL Language Reference*.
   b. Declare a cursor variable of that type:

      ```
      cursor_variable cursor_type;
      ```

      For complete cursor variable declaration syntax, see *Oracle Database PL/SQL Language Reference*.
   c. Declare a record to hold the row returned by the cursor:

      ```
      record_name return_type;
      ```

      For complete information about record declaration syntax, see *Oracle Database PL/SQL Language Reference*.
2. In the executable part:
   a. Open the cursor variable for a specific query:

      ```
      OPEN cursor_variable FOR query;
      ```

For complete information about OPEN FOR statement syntax, see *Oracle Database PL/SQL Language Reference*.

b. Fetch rows from the cursor variable (rows from the result set) one at a time, using a LOOP statement that has syntax similar to this:

```
LOOP
  FETCH cursor_variable INTO record_name;
  EXIT WHEN cursor_variable%NOTFOUND;
  -- Process row that is in record_name:
  statement;
  [ statement; ]...
END LOOP;
```

For complete information about FETCH statement syntax, see *Oracle Database PL/SQL Language Reference*.

c. Close the cursor variable:

```
CLOSE cursor_variable;
```

Alternatively, you can open the cursor variable for another query, which closes it for the current query.

For complete information about CLOSE statement syntax, see *Oracle Database PL/SQL Language Reference*.

Tutorial: Using a Cursor Variable to Retrieve Result Set Rows One at a Time

This tutorial shows how to change the EMP_EVAL.EVAL_DEPARTMENT procedure so that it uses a cursor variable instead of a declared cursor (which lets it process multiple departments) and how to make EMP_EVAL.EVAL_DEPARTMENT and EMP_EVAL.ADD_EVAL more efficient.

How this tutorial makes EMP_EVAL.EVAL_DEPARTMENT and EMP_EVAL.ADD_EVAL more efficient: Instead of passing one field of a record to ADD_EVAL and having ADD_EVAL use three queries to extract three other fields of the same record, EVAL_DEPARTMENT passes the entire record to ADD_EVAL, and ADD_EVAL uses dot notation to access the values of the other three fields.

To change the EMP_EVAL.EVAL_DEPARTMENT procedure to use a cursor variable:

1. In the EMP_EVAL package specification, add the procedure declaration and the REF CURSOR type definition, as shown in bold font:

```
CREATE OR REPLACE
PACKAGE emp_eval AS

  PROCEDURE eval_department (dept_id IN employees.department_id%TYPE);

  PROCEDURE eval_everyone;

  FUNCTION calculate_score(eval_id IN scores.evaluation_id%TYPE
                         , perf_id IN scores.performance_id%TYPE)
                           RETURN NUMBER;
  TYPE SAL_INFO IS RECORD
      ( j_id jobs.job_id%type
      , sal_min jobs.min_salary%type
      , sal_max jobs.max_salary%type
```

```
      , salary employees.salary%type
      , sal_raise NUMBER(3,3));


  TYPE emp_refcursor_type IS REF CURSOR RETURN employees%ROWTYPE;
END emp_eval;
```

2. In the EMP_EVAL package body, add a forward declaration for the procedure EVAL_LOOP_CONTROL and change the declaration of the procedure ADD_EVAL, as shown in bold font:

```
CREATE OR REPLACE
PACKAGE BODY EMP_EVAL AS

  FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
    RETURN PLS_INTEGER;

  PROCEDURE salary_schedule(emp IN sal_info);

  PROCEDURE add_eval(emp_record IN EMPLOYEES%ROWTYPE, today IN DATE);

  PROCEDURE eval_loop_control(emp_cursor IN emp_refcursor_type);
...
```

(For a step-by-step example of changing a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram".)

3. Change the EVAL_DEPARTMENT procedure to retrieve three separate result sets based on the department, and to invoke the EVAL_LOOP_CONTROL procedure, as shown in bold font:

```
PROCEDURE eval_department(dept_id IN employees.department_id%TYPE) AS
  emp_cursor    emp_refcursor_type;
  current_dept  departments.department_id%TYPE;

BEGIN
  current_dept := dept_id;

  FOR loop_c IN 1..3 LOOP
    OPEN emp_cursor FOR
      SELECT *
      FROM employees
      WHERE current_dept = eval_department.dept_id;

    DBMS_OUTPUT.PUT_LINE
      ('Determining necessary evaluations in department #' ||
       current_dept);

    eval_loop_control(emp_cursor);

    DBMS_OUTPUT.PUT_LINE
      ('Processed ' || emp_cursor%ROWCOUNT || ' records.');

    CLOSE emp_cursor;
    current_dept := current_dept + 10;
  END LOOP;
END eval_department;
```

4. Change the ADD_EVAL procedure as shown in bold font:

```
PROCEDURE add_eval(emp_record IN employees%ROWTYPE, today IN DATE)
AS
-- (Delete local variables)
```

```
BEGIN
  INSERT INTO EVALUATIONS (
    evaluation_id,
    employee_id,
    evaluation_date,
    job_id,
    manager_id,
    department_id,
    total_score
  )
  VALUES (
    evaluations_sequence.NEXTVAL,   -- evaluation_id
    emp_record.employee_id,    -- employee_id
    today,                     -- evaluation_date
    emp_record.job_id,         -- job_id
    emp_record.manager_id,     -- manager_id
    emp_record.department_id,  -- department_id
    0                           -- total_score
);
END add_eval;
```

5. Before `END EMP_EVAL`, add the following procedure, which fetches the individual records from the result set and processes them:

```
PROCEDURE eval_loop_control (emp_cursor IN emp_refcursor_type) AS
   emp_record      EMPLOYEES%ROWTYPE;
   all_evals       BOOLEAN;
   today           DATE;
BEGIN
  today := SYSDATE;

  IF (EXTRACT(MONTH FROM today) < 6) THEN
    all_evals := FALSE;
  ELSE
    all_evals := TRUE;
  END IF;

  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;

    IF all_evals THEN
      add_eval(emp_record, today);
    ELSIF (eval_frequency(emp_record.employee_id) = 2) THEN
      add_eval(emp_record, today);
    END IF;
  END LOOP;
END eval_loop_control;
```

6. Before `END EMP_EVAL`, add the following procedure, which retrieves a result set that contains all employees in the company:

```
PROCEDURE eval_everyone AS
  emp_cursor emp_refcursor_type;
BEGIN
  OPEN emp_cursor FOR SELECT * FROM employees;
  DBMS_OUTPUT.PUT_LINE('Determining number of necessary evaluations.');
  eval_loop_control(emp_cursor);
  DBMS_OUTPUT.PUT_LINE('Processed ' || emp_cursor%ROWCOUNT || ' records.');
  CLOSE emp_cursor;
END eval_everyone;
```

7. Compile the `EMP_EVAL` package specification.
8. Compile the `EMP_EVAL` package body.

Using Associative Arrays

An associative array is a type of collection.

> **See Also:**
>
> For more information about collections:
>
> - *Oracle Database Concepts*
> - *Oracle Database PL/SQL Language Reference*

About Collections

A **collection** is a PL/SQL composite variable that stores elements of the same type in a specified order, similar to a one-dimensional array. The internal components of a collection are called **elements**. Each element has a unique subscript that identifies its position in the collection.

To access a collection element, you use **subscript notation**: collection_name(element_subscript).

You can treat collection elements like scalar variables. You can also pass entire collections as subprogram parameters (if neither the sending nor receiving subprogram is a standalone subprogram).

A **collection method** is a built-in PL/SQL subprogram that either returns information about a collection or operates on a collection. To invoke a collection method, you use **dot notation**: collection_name.method_name. For example, collection_name.COUNT returns the number of elements in the collection.

PL/SQL has three types of collections:

- Associative arrays (formerly called "PL/SQL tables" or "index-by tables")
- Nested tables
- Variable arrays (varrays)

This document explains only associative arrays.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about PL/SQL collection types
> - *Oracle Database PL/SQL Language Reference* for more information about collection methods

About Associative Arrays

An **associative array** is an unbounded set of key-value pairs. Each key is unique, and serves as the subscript of the element that holds the corresponding value. Therefore, you can access elements without knowing their positions in the array, and without traversing the array.

The data type of the key can be either PLS_INTEGER or VARCHAR2 (length).

If the data type of the key is PLS_INTEGER and the associative array is **indexed by integer** and is **dense** (that is, has no gaps between elements), then every element between the first and last element is defined and has a value (which can be NULL).

If the key type is VARCHAR2 (length), the associative array is **indexed by string** (of length characters) and is **sparse**; that is, it might have gaps between elements.

When traversing a dense associative array, you need not beware of gaps between elements; when traversing a sparse associative array, you do.

To assign a value to an associative array element, you can use an assignment operator:

```
array_name(key) := value
```

If key is not in the array, then the assignment statement adds the key-value pair to the array. Otherwise, the statement changes the value of array_name(key) to value.

Associative arrays are useful for storing data temporarily. They do not use the disk space or network operations that tables require. However, because associative arrays are intended for temporary storage, you cannot manipulate them with DML statements.

If you declare an associative array in a package and assign values to the variable in the package body, then the associative array exists for the life of the database session. Otherwise, it exists for the life of the subprogram in which you declare it.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about associative arrays

Declaring Associative Arrays

To declare an associative array, you declare an associative array type and then declare a variable of that type.

The simplest syntax is:

```
TYPE array_type IS TABLE OF element_type INDEX BY key_type;

array_name  array_type;
```

An efficient way to declare an associative array is with a cursor, using the following procedure. The procedure uses each necessary statement in its simplest form, but provides references to its complete syntax.

To use a cursor to declare an associative array:

1. In the declarative part:
   a. Declare the cursor:

      ```
      CURSOR cursor_name IS query;
      ```

      For complete declared cursor declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Declare the associative array type:

      ```
      TYPE array_type IS TABLE OF cursor_name%ROWTYPE
        INDEX BY { PLS_INTEGER | VARCHAR2 length }
      ```

      For complete associative array type declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   c. Declare an associative array variable of that type:

      ```
      array_name  array_type;
      ```

      For complete variable declaration syntax, see *Oracle Database PL/SQL Language Reference*.

Example 5-9 uses the preceding procedure to declare two associative arrays, employees_jobs and jobs_, and then declares a third associative array, job_titles, without using a cursor. The first two arrays are indexed by integer; the third is indexed by string.

> **Note:**
>
> The ORDER BY clause in the declaration of employees_jobs_cursor determines the storage order of the elements of the associative array employee_jobs.

Example 5-9 Declaring Associative Arrays

```
DECLARE
  -- Declare cursor:

  CURSOR employees_jobs_cursor IS
    SELECT FIRST_NAME, LAST_NAME, JOB_ID
    FROM EMPLOYEES
    ORDER BY JOB_ID, LAST_NAME, FIRST_NAME;

  -- Declare associative array type:

  TYPE employees_jobs_type IS TABLE OF employees_jobs_cursor%ROWTYPE
    INDEX BY PLS_INTEGER;

  -- Declare associative array:

  employees_jobs  employees_jobs_type;

  -- Use same procedure to declare another associative array:
```

```
  CURSOR jobs_cursor IS
    SELECT JOB_ID, JOB_TITLE
    FROM JOBS;

  TYPE jobs_type IS TABLE OF jobs_cursor%ROWTYPE
    INDEX BY PLS_INTEGER;

  jobs_  jobs_type;

-- Declare associative array without using cursor:

  TYPE job_titles_type IS TABLE OF JOBS.JOB_TITLE%TYPE
    INDEX BY JOBS.JOB_ID%TYPE;  -- jobs.job_id%type is varchar2(10)

  job_titles  job_titles_type;

BEGIN
  NULL;
END;
/
```

> **See Also:**
>
> - "About Cursors"
> - *Oracle Database PL/SQL Language Reference* for associative array declaration syntax

Populating Associative Arrays

The most efficient way to populate a dense associative array is usually with a SELECT statement with a BULK COLLECT INTO clause.

> **Note:**
>
> If a dense associative array is so large that a SELECT statement would a return a result set too large to fit in memory, then do not use a SELECT statement. Instead, populate the array with a cursor and the FETCH statement with the clauses BULK COLLECT INTO and LIMIT. For information about using the FETCH statement with BULK COLLECT INTO clause, see *Oracle Database PL/SQL Language Reference*.

You cannot use a SELECT statement to populate a sparse associative array (such as job_titles in "Declaring Associative Arrays"). Instead, you must use an assignment statement inside a loop statement. For information about loop statements, see "Controlling Program Flow".

Example 5-10 uses SELECT statements to populate the associative arrays employees_jobs and jobs_, which are indexed by integer. Then it uses an assignment statement inside a FOR LOOP statement to populate the associative array job_titles, which is indexed by string.

Example 5-10 Populating Associative Arrays

```
-- Declarative part from Example 5-9 goes here.

BEGIN
  -- Populate associative arrays indexed by integer:

SELECT FIRST_NAME, LAST_NAME, JOB_ID BULK COLLECT INTO employees_jobs
  FROM EMPLOYEES ORDER BY JOB_ID, LAST_NAME, FIRST_NAME;

SELECT JOB_ID, JOB_TITLE BULK COLLECT INTO jobs_ FROM JOBS;

  -- Populate associative array indexed by string:

  FOR i IN 1..jobs_.COUNT() LOOP
    job_titles(jobs_(i).job_id) := jobs_(i).job_title;
  END LOOP;
END;
/
```

> **See Also:**
>
> "About Cursors"

Traversing Dense Associative Arrays

A **dense associative array** (indexed by integer) has no gaps between elements—every element between the first and last element is defined and has a value (which can be NULL).

You can traverse a dense array with a FOR LOOP statement, as in Example 5-11.

When inserted in the executable part of Example 5-10, after the code that populates the employees_jobs array, the FOR LOOP statement in Example 5-11 prints the elements of the employees_jobs array in the order in which they were stored. Their storage order was determined by the ORDER BY clause in the declaration of employees_jobs_cursor, which was used to declare employees_jobs (see Example 5-9).

The upper bound of the FOR LOOP statement, employees_jobs.COUNT, invokes a collection method that returns the number of elements in the array. For more information about COUNT, see *Oracle Database PL/SQL Language Reference*.

Example 5-11 Traversing a Dense Associative Array

```
-- Code that populates employees_jobs must precede this code:

FOR i IN 1..employees_jobs.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(
    RPAD(employees_jobs(i).first_name, 23) ||
    RPAD(employees_jobs(i).last_name,  28) ||     employees_jobs(i).job_id);
  END LOOP;
```

Result:

```
William                 Gietz                       AC_ACCOUNT
Shelley                 Higgins                     AC_MGR
```

```
Jennifer                  Whalen                    AD_ASST
Steven                    King                      AD_PRES
Lex                       De Haan                   AD_VP
Neena                     Kochhar                   AD_VP
John                      Chen                      FI_ACCOUNT
...
Jose Manuel               Urman                     FI_ACCOUNT
Nancy                     Greenberg                 FI_MGR
Susan                     Mavris                    HR_REP
David                     Austin                    IT_PROG
...
Valli                     Pataballa                 IT_PROG
Michael                   Hartstein                 MK_MAN
Pat                       Fay                       MK_REP
Hermann                   Baer                      PR_REP
Shelli                    Baida                     PU_CLERK
...
Sigal                     Tobias                    PU_CLERK
Den                       Raphaely                  PU_MAN
Gerald                    Cambrault                 SA_MAN
...
Eleni                     Zlotkey                   SA_MAN
Ellen                     Abel                      SA_REP
...
Clara                     Vishney                   SA_REP
Sarah                     Bell                      SH_CLERK
...
Peter                     Vargas                    ST_CLERK
Adam                      Fripp                     ST_MAN
...
Matthew                   Weiss                     ST_MAN
```

Traversing Sparse Associative Arrays

A **sparse associative array** (indexed by string) might have gaps between elements.

You can traverse it with a WHILE LOOP statement, as in Example 5-12.

To run the code in Example 5-12, which prints the elements of the job_titles array:

1. At the end of the declarative part of Example 5-9, insert this variable declaration:

   ```
   i jobs.job_id%TYPE;
   ```

2. In the executable part of Example 5-10, after the code that populates the job_titles array, insert the code from Example 5-12.

Example 5-12 Traversing a Sparse Associative Array

```
/* Declare this variable in declarative part:

   i jobs.job_id%TYPE;

   Add this code to the executable part,
   after code that populates job_titles:
*/

i := job_titles.FIRST;

WHILE i IS NOT NULL LOOP
  DBMS_OUTPUT.PUT_LINE(RPAD(i, 12) || job_titles(i));
```

```
  i := job_titles.NEXT(i);
END LOOP;
```

Result:

```
AC_ACCOUNT  Public Accountant
AC_MGR      Accounting Manager
AD_ASST     Administration Assistant
AD_PRES     President
AD_VP       Administration Vice President
FI_ACCOUNT  Accountant
FI_MGR      Finance Manager
HR_REP      Human Resources Representative
IT_PROG     Programmer
MK_MAN      Marketing Manager
MK_REP      Marketing Representative
PR_REP      Public Relations Representative
PU_CLERK    Purchasing Clerk
PU_MAN      Purchasing Manager
SA_MAN      Sales Manager
SA_REP      Sales Representative
SH_CLERK    Shipping Clerk
ST_CLERK    Stock Clerk
ST_MAN      Stock Manager
```

Example 5-12 includes two collection method invocations, job_titles.FIRST and job_titles.NEXT(i). job_titles.FIRST returns the first element of job_titles, and job_titles.NEXT(i) returns the subscript that succeeds `i`. For more information about FIRST, see *Oracle Database PL/SQL Language Reference*. For more information about NEXT, see *Oracle Database PL/SQL Language Reference*.

Handling Exceptions (Runtime Errors)

You can handle exceptions that occur at run time with PL/SQL code.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about handling PL/SQL errors

About Exceptions and Exception Handlers

When a runtime error occurs in PL/SQL code, an **exception** is raised. If the subprogram (or block) in which the exception is raised has an exception-handling part, then control transfers to it; otherwise, execution stops.

Runtime errors can arise from design faults, coding mistakes, hardware failures, and many other sources.

Oracle Database has many **predefined exceptions**, which it raises automatically when a program violates database rules or exceeds system-dependent limits. For example, if a SELECT INTO statement returns no rows, then Oracle Database raises the predefined exception NO_DATA_FOUND. For a summary of predefined PL/SQL exceptions, see *Oracle Database PL/SQL Language Reference*.

PL/SQL lets you define (declare) your own exceptions. An exception declaration has this syntax:

```
exception_name EXCEPTION;
```

Unlike a predefined exception, a **user-defined exception** must be raised explicitly, using either the RAISE statement or the DBMS_STANDARD.RAISE_APPLICATION_ERROR. procedure. For example:

```
IF condition THEN RAISE exception_name;
```

For information about the DBMS_STANDARD.RAISE_APPLICATION_ERROR procedure, see *Oracle Database PL/SQL Language Reference*.

The exception-handling part of a subprogram contains one or more exception handlers. An **exception handler** has this syntax:

```
WHEN { exception_name [ OR exception_name ]... | OTHERS } THEN
  statement; [ statement; ]...
```

("About Subprogram Structure" shows where to put the exception-handling part of a subprogram.)

A WHEN OTHERS exception handler handles unexpected runtime errors. If used, it must be last. For example:

```
EXCEPTION
  WHEN exception_1 THEN
    statement; [ statement; ]...
  WHEN exception_2 OR exception_3 THEN
    statement; [ statement; ]...
  WHEN OTHERS THEN
    statement; [ statement; ]...
    RAISE;  -- Reraise the exception (very important).
END;
```

An alternative to the WHEN OTHERS exception handler is the EXCEPTION_INIT pragma, which associates a user-defined exception name with an Oracle Database error number.

See Also:

- *Oracle Database PL/SQL Language Reference* for more information about exception declaration syntax
- *Oracle Database PL/SQL Language Reference* for more information about exception handler syntax
- *Oracle Database PL/SQL Language Reference* for more information about the EXCEPTION_INIT pragma

When to Use Exception Handlers

Oracle recommends using exception handlers only in these situations.

- You expect an exception and want to handle it.

For example, you expect that eventually, a SELECT INTO statement will return no rows, causing Oracle Database to raise the predefined exception NO_DATA_FOUND. You want your subprogram or block to handle that exception (which is not an error) and then continue, as in Example 5-13.

- You must relinquish or close a resource.

  For example:

```
...
file := UTL_FILE.OPEN ...
BEGIN
  statement statement]...  -- If this code fails for any reason,
EXCEPTION
  WHEN OTHERS THEN
    UTL_FILE.FCLOSE(file);     -- then you want to close the file.
    RAISE;                     -- Reraise the exception (very important).
END;
UTL_FILE.FCLOSE(file);
...
```

- At the top level of the code, you want to log the error.

  For example, a client process might issue this block:

```
BEGIN
  proc(...);
EXCEPTION
  WHEN OTHERS THEN
    log_error_using_autonomous_transaction(...);
    RAISE;  -- Reraise the exception (very important).
END;
/
```

  Alternatively, the standalone subprogram that the client invokes can include the same exception-handling logic—but only at the top level.

Handling Predefined Exceptions

You can handle predefined exceptions.

Example 5-13 shows, in bold font, how to change the EMP_EVAL.EVAL_DEPARTMENT procedure to handle the predefined exception NO_DATA_FOUND. Make this change and compile the changed procedure. (For an example of how to change a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram".)

Example 5-13 Handling Predefined Exception NO_DATA_FOUND

```
PROCEDURE eval_department(dept_id IN employees.department_id%TYPE) AS
  emp_cursor    emp_refcursor_type;
  current_dept  departments.department_id%TYPE;

BEGIN
  current_dept := dept_id;

  FOR loop_c IN 1..3 LOOP
    OPEN emp_cursor FOR
      SELECT *
      FROM employees
      WHERE current_dept = eval_department.dept_id;
```

```
    DBMS_OUTPUT.PUT_LINE
      ('Determining necessary evaluations in department #' ||
       current_dept);

    eval_loop_control(emp_cursor);

    DBMS_OUTPUT.PUT_LINE
      ('Processed ' || emp_cursor%ROWCOUNT || ' records.');

    CLOSE emp_cursor;
    current_dept := current_dept + 10;
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('The query did not return a result set');
END eval_department;
```

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about predefined exceptions

Declaring and Handling User-Defined Exceptions

You can declare and handle user-defined exceptions.

Example 5-14 shows, in bold font, how to change the EMP_EVAL.CALCULATE_SCORE function to declare and handle two user-defined exceptions, wrong_weight and wrong_score. Make this change and compile the changed function. (For an example of how to change a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram".)

Example 5-14 Handling User-Defined Exceptions

```
FUNCTION calculate_score ( evaluation_id IN scores.evaluation_id%TYPE
                         , performance_id IN scores.performance_id%TYPE )
                         RETURN NUMBER AS

  weight_wrong  EXCEPTION;
  score_wrong   EXCEPTION;
  n_score       scores.score%TYPE;
  n_weight      performance_parts.weight%TYPE;
  running_total NUMBER := 0;
  max_score     CONSTANT scores.score%TYPE := 9;
  max_weight    CONSTANT performance_parts.weight%TYPE:= 1;
BEGIN
  SELECT s.score INTO n_score
  FROM SCORES s
  WHERE evaluation_id = s.evaluation_id
  AND performance_id = s.performance_id;

  SELECT p.weight INTO n_weight
  FROM PERFORMANCE_PARTS p
  WHERE performance_id = p.performance_id;

  BEGIN
    IF (n_weight > max_weight) OR (n_weight < 0) THEN
      RAISE weight_wrong;
```

```
    END IF;
  END;

  BEGIN
    IF (n_score > max_score) OR (n_score < 0) THEN
      RAISE score_wrong;
    END IF;
  END;

  running_total := n_score * n_weight;
  RETURN running_total;

EXCEPTION
  WHEN weight_wrong THEN
    DBMS_OUTPUT.PUT_LINE(
      'The weight of a score must be between 0 and ' || max_weight);
    RETURN -1;
  WHEN score_wrong THEN
    DBMS_OUTPUT.PUT_LINE(
      'The score must be between 0 and ' || max_score);
    RETURN -1;
END calculate_score;
```

See Also:

Oracle Database PL/SQL Language Reference for more information about user-defined exceptions