**4****Developing and Using Stored****Procedures**

**Creating and Using Standalone****Procedures and****Functions**

With Oracle Database, you can store programs in the database, so commonly used code can be written and tested once and then accessed by any application that requires it. Program units that reside in the database also ensure that when the code is invoked the data is processed consistently, which leads to ease and consistency of the application development process.

Schema-level, or standalone subprograms such as functions (which return a value) and procedures (which do not return a value) are compiled and stored in an Oracle Database. Once compiled, they become **stored procedure** or **stored function** schema objects, and can be referenced or called by any applications connected to Oracle Database. At invocation, both stored procedures and functions can accept parameters.

Procedures and functions follow the basic PL/SQL block structure, which consists of the following elements:

* A declarative part, sometimes starting with the keyword DECLARE, identifies variables and constants used in the application logic. This part is optional.
* An executable part, starting with BEGIN and ending with END, contains the application logic. This part is mandatory.
* An exception-handling part, starting with EXCEPTION, handles error conditions that may be raised in the executable part of the block. This part is optional.

The general form of a PL/SQL block follows. Note also that each stored program unit has a header that names the unit and identifies it as either a function, procedure, or a package.

*Header* AS

[declaration statements

...]

BEGIN

...

[EXCEPTION

...]

END;

**Creating****Procedures and****Functions**

The SQL statements for creating procedures and functions are CREATE PROCEDURE and CREATE FUNCTION, respectively. In practice, it is best to use a CREATE OR REPLACE statement. The general form of these statements follows.

CREATE OR REPLACE *procedure\_name*(*arg1* *data\_type,* ...) AS

BEGIN

....

END *procedure\_name*;

CREATE OR REPLACE *procedure\_name*(*arg1* *data\_type,* ...) AS

BEGIN

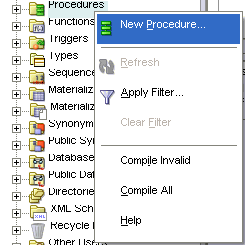
....

END *procedure\_name*;

**To create a procedure:**

You will create a procedure add\_evaluation that creates a new row in the evaluations table.

1. In the Connections navigation hierarchy, right-click **Procedures**.
2. Select **New Procedure**.



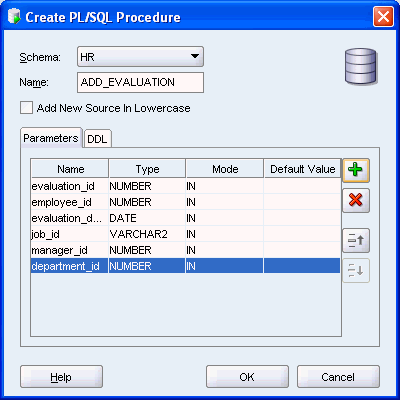
1. In the New Procedure window, set the following parameters:
   * Ensure that **Schema** is set to HR.
   * Set **Name** to ADD\_EVALUATION.

In the Parameters tab, click the **Add Column** icon ('plus' sign) and specify the first parameter of the procedure. Set **Name** toeval\_id, set **Type** to NUMBER, set **Mode** to IN, and leave **Default Value** empty.

Similarly, add the following parameters, in this order:

* + employee\_id: set **Type** to NUMBER, set **Mode** to IN, and leave **Default Value** empty.
  + evaluation\_date: set **Type** to DATE, set **Mode** to IN, and leave **Default Value** empty.
  + job\_id: set **Type** to VARCHAR2, set **Mode** to IN, and leave **Default Value** empty.
  + manager\_id: set **Type** to NUMBER, set **Mode** to IN, and leave **Default Value** empty
  + department\_id: set **Type** to NUMBER, set **Mode** to IN, and leave **Default Value** empty

Click **OK**.



1. The ADD\_EVALUATION pane opens with the following code.

Note that the tile of the pane is in italic font, which indicates that the procedure is not saved in the database.

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

) AS

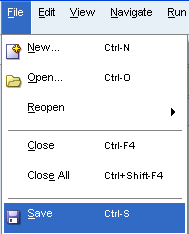
BEGIN

NULL;

END ADD\_EVALUATION;

1. From the **File** menu, select **Save** to save the new procedures. Alternatively, use the **CTRL + S** key combination.

Note that Oracle Database automatically compiles procedures prior to saving them.

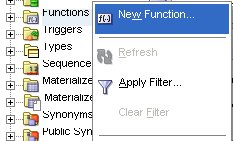


Note that the tile of the add\_evaluation pane is in regular font, not italic; this indicates that the procedure is saved to the database

**To create a****function:**

You will create a new function calculate\_score, which calculates the weighted score based on the performance in a particular category.

1. In the Connections navigation hierarchy, right-click **Functions**.
2. Select **New Function**.

  
In the New Function window, set the following parameters:

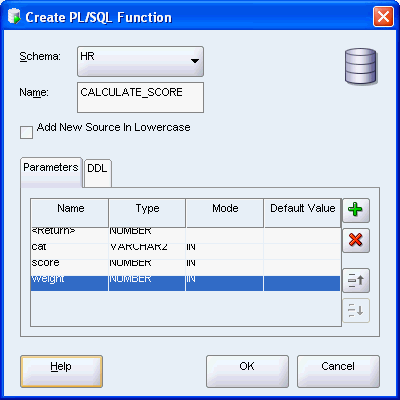
* + Ensure that **Schema** is set to HR.
  + Set **Name** to CALCULATE\_SCORE.

In the Parameters pane, set the **<return>** **Type** to NUMBER.

Similarly, add the following parameters, in this order:

* + cat: set **Type** to VARCHAR2, set **Mode** to IN, and leave **Default Value** empty.
  + score: set **Type** to NUMBER, set **Mode** to IN, and leave **Default Value** empty
  + weight: set **Type** to NUMBER, set **Mode** to IN, and leave **Default Value** empty

Click **OK**.



1. The calculate\_score pane opens with the following code.

Note that the tile of the pane is in italic font, which indicates that the procedure is not saved in the database.

CREATE OR REPLACE

FUNCTION calculate\_score

( cat IN VARCHAR2

, score IN NUMBER

, weight IN NUMBER

) RETURN NUMBER AS

BEGIN

RETURN NULL;

END calculate\_score;

1. From the **File** menu, select **Save** to save the new function. Alternatively, use the **CTRL + S** key combination.

Note that Oracle Database automatically compiles functions prior to saving them.

Note that the tile of the calculate\_score pane is in regular font, not italic; this indicates that the procedure is saved to the database

**Modifying****Procedures and Functions**

You already created a new procedure and a new function. However, they both consist of only the subprogram signature. In this section, you will edit a subprogram body.

**To modify a****function:**

You will edit the function calculate\_score to determine the weighted value of an evaluation for a particular category.

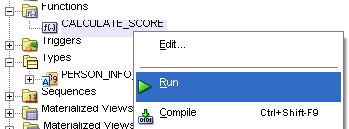
1. In the calculate\_score pane, replace the body of the function with the following code. The new code is in bold font.
2. BEGIN
3. RETURN **score \* weight**;
4. END calculate\_score;
5. Compile and save the function; you may use the **CTRL + S** key combination.

**Testing****Procedures and Functions**

Next, you will test the function that you just modified.

**To test a****function:** You will test the function calculate\_score.

1. In the Connections navigator hierarchy, right-click the calculate\_score function. Select **Run**.



1. In the Run PL/SQL window, click inside the PL/SQL Block pane, and edit the assignments for the score and weight variables. The new code is in bold font.
2. v\_Return := CALCULATE\_SCORE(
3. CAT => CAT,
4. SCORE => **8**,
5. WEIGHT => **0.2**
6. );

Click **OK**.

1. In the Running - Log pane, note the following results:
2. Connecting to the database hr\_conn.
3. v\_Return = 1.6
4. Process exited.
5. Disconnecting from the database hr\_conn.

**Dropping****Procedures and****Functions**

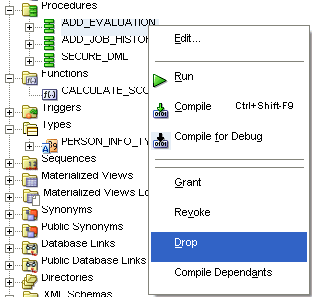
You can delete a procedure or function from the database using either the Connection Navigator, or the SQL DROP statement.

**To drop a procedure:**

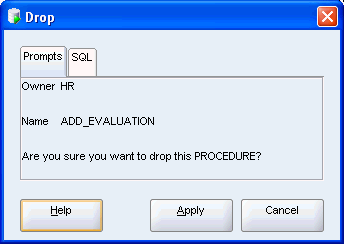
You will drop the procedure ADD\_EVALUATION.

1. In the Connections navigator hierarchy, right-click the ADD\_EVALUATION function.

Select **Drop**.



1. In the Drop window, click **Apply**.



1. In the Confirmation dialog box, click **OK**.

You dropped the ADD\_EVALUATION procedure from the database.

**Creating and Using Packages**

In the preceding section, you created and tested procedures and functions that were schema objects. This approach can be useful for testing subsets or small features of your application.

Enterprise level applications have much greater complexity: some of the interfaces and types are directly available to user, while others are used only by ohter functions and procedures and are never called by the user. PL/SQL enables you to formally state the relationship between these subprograms by placing them in the same **package**, which is a schema object that groups and name-qualifies logically related elements such as PL/SQL types, variables, functions and procedures. Encapsulating these elements inside a package also prevents, over the life time of the applications, unintended consequences such as name capture that is discussed in ["Overview of Stored Procedures"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHGDECD).

Procedures and functions that are defined within a package are known as **packaged subprograms**. Procedures and functions that are nested within other subprograms or within a PL/SQL block are called **local subprograms**; they exist only inside the enclosing block and cannot be referenced externally.

Another reason that standalone procedures and functions, like the ones in ["Creating and Using Standalone Procedures and Functions"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHCHIDB), are limited to large-scale development is that they can only send and receive scalar parameters (NUMBER, VARCHAR2, and DATE), but cannot use a composite structure, RECORD, unless it is defined in a package specification.

Packages usually have two parts: a specification and a body.

The package is defined by the **package specification**, which declares the types, variables, constants, exceptions, cursors, functions and procedures that can be referenced from outside of the package. The specification is the interface to the package. Applications that call the subprograms in a package only need to know the names and parameters from the package specification.

The standard package specification has this form:

CREATE OR REPLACE PACKAGE *package\_name* AS

*type definitions for records,* *index-by* *tables*

*constants*

*exceptions*

*global variable declarations*

procedure *procedure\_1*(*arg1*, ...);

...

function *function\_1*(*arg1*,...) return *datat\_ype*;

...

END *package\_name*;

The **package body** contains the code that implements these subprograms, the code for all private subprograms that can only be invoked from within the package, and the queries for the cursors. You can change the implementation details inside the package body without invalidating the calling applications.

The package body has this form:

CREATE OR REPLACE PACKAGE BODY *package\_name* AS

PROCEDURE *procedure\_1*(*arg1*,...) IS

BEGIN

...

EXCEPTION

...

END *procedure\_1*;

...

FUNCTION *function\_1*(*arg1*,...) RETURN *data\_type* IS *result\_variable* *data\_type*

BEGIN

...

RETURN *result\_variable*;

EXCEPTION

...

END *function\_1*;

...

END *package\_name*;

**Guidelines for Packages**

You should become familiar with the packages supplied with Oracle Database and avoid writing code that duplicates existing features.

You should design and define the package specification before writing the implementation in the package body. In the specification, include only those parts that must be publicly visible to calling programs, and hide private declarations within the package body. This prevents unsafe dependencies of other programs on your implementation details.

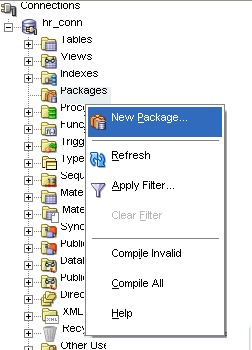
Because PL/SQL has a single-pass compiler, you may find that the dependencies between correct and valid subprograms within the package body prevent you from successfully compiling your package. You then need to declare these unknown subprograms near the top of the package body, and specify them later. For this reason, Oracle recommends that you add new elements at the end of the package specification or body to minimize possible invalidation of dependents.

**Creating a Package**

You will create a package that encapsulates all the functionality necessary to perform employee evaluations. After you create the package,["Modifying a Package"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHIEAGC) explains how you modify the package and to create the package body.

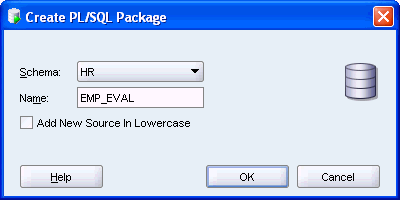
**To create a package in SQL Developer navigation hierarchy:**

1. In the Connections navigation hierarchy, right-click **Packages**.
2. Select **New Package**.



1. In the Create PL/SQL Package dialog, set the following parameters:
   * Ensure that **Schema** is set to HR.
   * Set **Name** to EMP\_EVAL.

Click **OK**.

1. 
2. The emp\_eval pane opens with the following code:
3. CREATE OR REPLACE PACKAGE emp\_eval AS
5. /\* TODO enter package declarations (types, exceptions, methods etc) here \*/
7. END emp\_eval;

Note that the title of the pane is in italic font, which indicates that the package is not saved to the database.

1. From the **File** menu, select **Save** to compile and save the new package. Alternatively, use the **CTRL + S** key combination.

In the Messages - Log pane, the system confirms that the package was created:

EMP\_EVAL Compiled.

Note that the title of the emp\_eval pane is in regular font, not italic; this indicates that the procedure is saved to the database.

[Example 4-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHFHIBJ) shows how to create a package directly in the SQL Worksheet.

***Example 4-1 Creating a******PL/SQL Package***

CREATE OR REPLACE PACKAGE eval AS

/\* package \*/

END eval;

The results of the script follow.

PACKAGE eval Compiled.

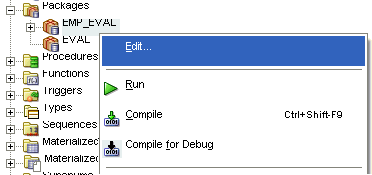
**Modifying a Package**

In this section, you will modify package emp\_eval.

**To change the package specification:**

You will change the package specification of emp\_eval by specifying some functions and procedures.

1. In the Connections navigation hierarchy, select Packages, and then right-click emp\_eval.
2. Select **Edit**.



1. In the EMP\_EVAL pane, edit the package. The new code is in bold font.
2. create or replace
3. PACKAGE emp\_eval AS
4. **PROCEDURE eval\_department(department\_id IN NUMBER);**
5. **FUNCTION calculate\_score(evaluation\_id IN NUMBER**
6. **, performance\_id IN NUMBER)**
7. **RETURN NUMBER;**
8. END emp\_eval;
9. Compile the package specification.

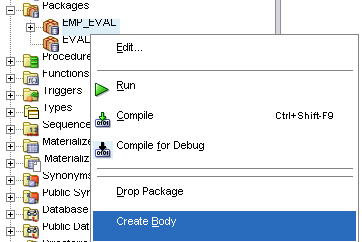
You will see the following message that confirms that the package compiled correctly.

EMP\_EVAL Compiled.

**To create a package body:**

You will create a package body for emp\_eval by specifying some functions and procedures.

1. In the Connections navigation hierarchy, right-click emp\_eval.
2. Select **Create Body**.



1. In the emp\_eval Body pane, you can see the automatically generated code for the package body.
2. CREATE OR REPLACE
3. PACKAGE BODY emp\_eval AS
5. PROCEDURE eval\_department(department\_id IN NUMBER) AS
6. BEGIN
7. /\* TODO implementation required \*/
8. NULL;
9. END eval\_department;
11. FUNCTION calculate\_score(evaluation\_id IN NUMBER
12. , performance\_id IN NUMBER)
13. RETURN NUMBER AS
14. BEGIN
15. /\* TODO implementation required \*/
16. RETURN NULL;
17. END calculate\_score;
18. END emp\_eval;
19. Compile and save the package body.

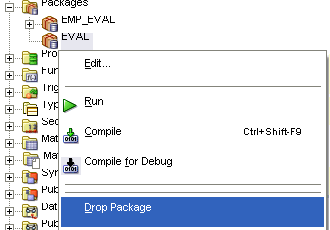
You will see the following message that confirms that the package body compiled correctly.

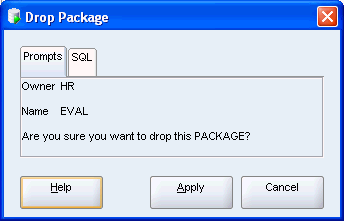
EMP\_EVAL Body Compiled.

**Dropping a Package**

You can delete a package from the database either by using the Connections navigator hierarchy or the SQL DROP statement. When you drop a package, you remove from the database both the package specification and its package body.

**To drop a package:**

1. In the Connections navigator hierarchy, select Packages, and then right-click the EVAL package.
2. Select **Drop Package**. 
3. In the Drop Package dialog, click **Apply**.



1. In the Confirmation dialog, click **OK**.

**Using Variables and Constants**

One of the significant advantages that PL/SQL offers over SQL is its ability to use variables and constants in programming constructs.

A **variable** is defined by the user to hold a specified value of a particular data type. This value is mutable; it can change at runtime.

A **constant** holds a value that cannot be changed; the compiler ensures that this value is immutable and does not compile any code that could change it. You should use constants in your code instead of direct values because they will make it simpler to maintenance of your code base over time. When you declare all values that do not change as constants, this optimizes your compiled code.

**PL/SQL Data Types**

In addition to the SQL data types such as VARCHAR2, DATE, NUMBER, and so on, Oracle Database supports data types that you can use only through PL/SQL. These data types include BOOLEAN, composite data types such as RECORD, reference types such as REF CURSOR and INDEX BY TABLE, and numerous specialized types that represent numbers, characters, and date elements. One numeric type, PLS\_INTEGER, is especially useful because it performs binary integer arithmetic and has significant performance benefits. Note that these PL/SQL types cannot be used at the level of the schema (and therefore, in tables), but only for types and processes that are defined within a package.

**Using Variables and Constants**

Variables and constants can have any SQL or PL/SQL data type, and are declared in the declaration block of a subprogram. By default, any variable that is declared has a value of NULL. When defining a constant, you must use the CONSTANT clause, and assign a value immediately.

**Using Comments**

In PL/SQL, in-line comments start with a double hyphen, --, and extend to the end of the line. Multi-line comments must start with a slash and asterisk, /\*, and terminate with an asterisk and a slash, \*/.

**Using****Identifiers**

Identifiers name PL/SQL program units such as constants, variables, and subprograms. All identifiers must have at most 30 characters, and must start with a letter that is followed by any combination of letters, numerals, and the signs '$', '\_', and '#'. Other characters cannot be used in identifiers.

Note that because PL/SQL is not case-sensitive except in managing string and character literals, you can use uppercase and lowercase letters interchangeably. This means that an identifier last\_name is equivalent to LAST\_NAME. Declaring the second identifier generates an error.

You should use meaningful names for your variables and constants, and use a good naming convention. For example, you could start each constant name with 'cons\_'. Also, remember not to use reserved words as identifiers.

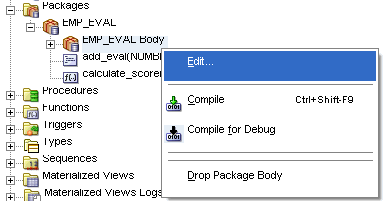
**Declaring Variables and Constants**

You will update the new function of the emp\_eval package, calculate\_score, which calculates the final score for the employee evaluation by combing all weighted scores in different categories.

**To declare variables and constants:**

1. In the Connections navigation hierarchy, click the plus sign (**+**) beside Packages to expand the group.
2. Click the 'plus' beside emp\_eval to expand the package.
3. Right-click **EMP\_EVAL Body**.
4. Select **Edit**.

emp\_eval Body pane appears.



1. In the emp\_eval Body pane, modify function calculate\_score by adding variables and constants, as shown by the following code. New code is bold font.
2. FUNCTION calculate\_score(evaluation\_id IN NUMBER
3. , performance\_id IN NUMBER)
4. RETURN NUMBER AS
5. **n\_score NUMBER(1,0); -- a variable**
6. **n\_weight NUMBER; -- a variable**
7. **max\_score CONSTANT NUMBER(1,0) := 9; -- a constant limit check**
8. **max\_weight CONSTANT NUMBER(8,8) := 1; -- a constant limit check**
9. BEGIN
10. RETURN NULL;
11. END calculate\_score;
12. Use the key combination '**CTRL**'+'**S**' to save the updated package body.

The following message appears in the Messages-Log pane:

EMP\_EVAL Body Compiled

**Declaring Variables with Structure Identical to Database Columns**

In ["Declaring Variables and Constants"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHJDFCH), you modified function calculate\_score by adding two variables, n\_score and n\_weight. These variables will represent values from tables in the database: n\_score is stored in the scores table, and n\_weight is stored in theperformance\_parts table. The data types you used for these variables match the column data type definitions in the tables.

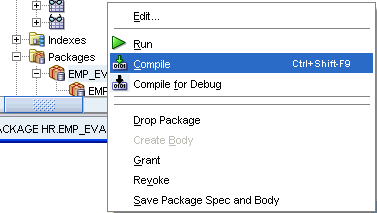
Over time, applications evolve and the column definitions may change; this may invalidate the calculate\_score function. For easier code maintenance, you should use special qualifiers that declare variables with data types that match the definitions of the appropriate columns and rows. These qualifiers are %TYPE and %ROWTYPE.

* The %TYPE attribute supplies the data type of a table column or another variable. This has the advantages of guaranteeing the correct data type assignment, and the correct implementation of the function at runtime if the data type of the table column changes.
* The %ROWTYPE attribute supplies the definition of a row in a table to a RECORD variable. Columns in a table row and the corresponding fields in a RECORD have the same names and data types. The advantages of using %ROWTYPE are the same as for%TYPE. See ["Using Composite Data Structures; Records"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHHCJDI) for a demonstration.

The following task shows how to use the %TYPE attribute in a function. You will edit the function calculate\_score to assign to variablesn\_score and n\_weight the data types that match the columns of the source tables. Note that the constants max\_score and max\_weight will be used to check equivalence to table values, so they too must match the table types.

**To use the %TYPE attribute:**

1. In the emp\_eval Body pane, modify function calculate\_score by changing the definition of the variables, as shown by the following code. New code is bold font.
2. FUNCTION calculate\_score(evaluation\_id IN **scores.evaluation\_id%TYPE**
3. , performance\_id IN **scores.performance\_id%TYPE**)
4. RETURN NUMBER AS
5. n\_score **scores.score%TYPE; -- from SCORES**
6. n\_weight **performance\_parts.weight%TYPE; -- from PERFORMANCE\_PARTS**
7. max\_score  **CONSTANT scores.score%TYPE := 9; -- a constant limit check**
8. max\_weight **CONSTANT performance\_parts.weight%TYPE := 1;**
9. **-- a constant limit check**
10. BEGIN
11. RETURN NULL;
12. END calculate\_score;
13. In the emp\_eval package specification, change the declaration of the function calculate\_score.
14. FUNCTION calculate\_score(evaluation\_id IN **scores.evaluation\_id%TYPE**
15. , performance\_id IN **scores.performance\_id%TYPE**)
16. RETURN NUMBER;
17. In the Connections navigation hierarchy, right-click the emp\_eval package, and select **Compile**. Alternatively, use the**Ctrl+Shift+F9** keyboard shortcut.

  
The following message appears in the Messages-Log pane:

EMP\_EVAL Body Compiled

**To use the %ROWTYPE attribute:**

Look at the code used in the eval\_department procedure in["Using Explicit Cursors"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHDCIIC) .

**Assigning Values to Variables**

You can assign values to a variable in three general ways: through the assignment operator, by selecting a value into the variable, or by binding a variable. This section covers the first two methods. Variable binding is described in 2 Day + guides for Application Express, Java, .NET, and PHP.

**Assigning Values with the Assignment Operator**

You can assign values to a variable both in the declaration and the body of a subprogram.

The following code shows the standard declaration of variables and constants. In procedures and functions, the declaration block does not use the DECLARE keyword; instead, it follows the AS keyword of the subprogram definition.

***Example 4-2 Assigning variable values in a declaration***

In the emp\_eval Body pane, modify function calculate\_score by adding a new variable running\_total. The value of running\_total is also the new return value of the function. You will set the initial value of the return variable to 0. Note that running\_total is declared as a general NUMBER because it will hold a product of two NUMBERs with different precision and scale. New code is 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; -- from SCORES

n\_weight performance\_parts.weight%TYPE; -- from PERFORMANCE\_PARTS

**running\_total NUMBER := 0; -- used in calculations**

max\_score CONSTANT scores.score%TYPE := 9; -- a constant limit check

max\_weight CONSTANT performance\_parts.weight%TYPE:= 1;

-- a constant limit check

BEGIN

RETURN **running\_total**;

END calculate\_score;

Compile the emp\_eval Body.

You can also assign values to variables within the body of a subprogram. You will edit the function calculate\_score by using therunning\_total variable inside the body of the function to hold a value of an expression.

***Example 4-3 Assigning variable values in the body of a function***

In the emp\_eval Body pane, modify function calculate\_score by assigning to the running\_total variable the value of an expression, as shown by the following code. New code is 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; -- from SCORES

n\_weight performance\_parts.weight%TYPE; -- from PERFORMANCE\_PARTS

running\_total NUMBER :=0; -- used in calculations

max\_score CONSTANT scores.score%TYPE := 9; -- a constant limit check

max\_weight CONSTANT performance\_parts.weight%TYPE:= 1;

-- a constant limit check

BEGIN

**running\_total := max\_score \* max\_weight;**

RETURN running\_total;

END calculate\_score;

Compile and save emp\_eval Body.

**Assigning Values from the Database**

The simplest possible assignment of a value is to use the assignment operator (**:=**) as you did for the variable running\_total in ["Assigning Values with the Assignment Operator"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHIBCCJ).

However, the purpose of function calculate\_score is to perform a calculation based on values stored in database tables. To use existing database values in a procedure, function, or package, you must assign these values to a variable by using a SELECT INTO statement. You can then use the variable in subsequent computations.

***Example 4-4 Assigning to a variable a values from the database***

In the emp\_eval Body pane, modify function calculate\_score by assigning the table values to the variables n\_score and n\_weight, and then assigning their product to the running\_total variable, as shown by the following code. New code is 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; -- from SCORES

n\_weight performance\_parts.weight%TYPE; -- from PERFORMANCE\_PARTS

running\_total NUMBER := 0; -- used in calculations

max\_score CONSTANT scores.score%TYPE := 9; -- a constant limit check

max\_weight CONSTANT performance\_parts.weight%TYPE:= 1;

-- a constant limit check

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;

Compile and save emp\_eval Body.

Similarly, add a new add\_eval procedure for inserting new records into the evaluations table, based on the content of the corresponding row in the employees table. Note that add\_eval is using the sequence evaluations\_seq.

***Example 4-5 Creating a new******table row with values from another table***

In the emp\_eval Body pane, above the line END emp\_eval, add procedure add\_eval, which uses some columns from the employees table to insert rows into the evaluations table. Note also that you will create the local function add\_eval in the body of the emp\_eval package, but not declare it in the package specification. This means that add\_eval may be invoked only within the emp\_eval package, by another subprogram.

PROCEDURE add\_eval(employee\_id IN employees.employee\_id%TYPE, today IN DATE) AS

-- placeholders for variables

job\_id employees.job\_id%TYPE;

manager\_id employees.manager\_id%TYPE;

department\_id employees.department\_id%TYPE;

BEGIN

-- extracting values from employees for later insertion into evaluations

SELECT e.job\_id INTO job\_id FROM employees e

WHERE employee\_id = e.employee\_id;

SELECT e.manager\_id INTO manager\_id FROM employees e

WHERE employee\_id = e.employee\_id;

SELECT e.department\_id INTO department\_id FROM employees e

WHERE employee\_id = e.employee\_id;

-- inserting a new row of values into evaluations table

INSERT INTO evaluations VALUES (

evaluations\_seq.NEXTVAL, -- evaluation\_id

employee\_id, -- employee\_id

today, -- evaluation\_date

job\_id, -- job\_id

manager\_id, -- manager\_id

department\_id, -- department\_id

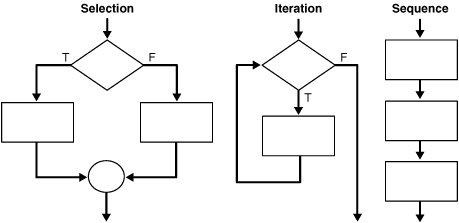
0); -- total\_score

END add\_eval;

Compile and save emp\_eval Body.

**Controlling Program Flow**

Control structures are the most powerful feature of the PL/SQL extension to SQL. They let you manipulate data and process it using conditional selection, iterative control, and sequential statements. Conditional selection is a situation where you may have different types of data values, and may need to perform different processing steps. Iterative control is a situation where you need to perform repetitive process steps on similar data. In general, all the lines of code in your programs run sequentially; sequential control means that you are choosing to execute an alternate labeled programming branch (GOTO statement).

  
This section will only cover conditional selection and iterative program flow structures, such as IF...THEN...ELSE, CASE, FOR...LOOP,WHILE...LOOP, and LOOP...EXIT WHEN.

**Using Conditional Selection Control**

Conditional selection structures test an expression that evaluates to a BOOLEAN value TRUE or FALSE. Depending on the value, control structures execute the assigned sequence of statements. There are two general selection control mechanisms: IF...THEN...ELSE and its variations, and the CASE statement.

**Using IF...THEN...ELSE Selection Control**

The IF...THEN...ELSE statement runs a sequence of statements conditionally. If the test condition evaluates to TRUE, the program runs statements in the THEN clause. If the condition evaluates to FALSE, the program runs the statements in the ELSE clause. You can also use this structure for testing multiple conditions if you include the ELSIF keyword. The general form of the IF...THEN...[ELSIF]...ELSEstatement follows:

IF *condition\_1* THEN

...;

ELSIF *condition\_2* THEN -- optional

...;

ELSE -- optional

...;

END IF;

For example, the sample company could have a rule that an employee evaluation should be done twice a year (December 31 and June 30) in the first ten years of employment, but only once a year (December 31) subsequently. You could implement this rule in aneval\_frequency function that determines how many times in each year an evaluation should be performed by using the IF...THEN...ELSEclause on the value of the hire\_date column.

The function eval\_frequency uses the employees.hire\_date value to determine if evaluations should be performed once each year (over 10 years employment) or twice each year.

Note also that you will create the function eval\_frequency in the body of the emp\_eval package, but not declare it in the package specification. This means that eval\_frequency may be invoked only within the emp\_eval package, by another subprogram.

***Example 4-6 Using the IF... THEN...ELSE Selection Control***

In the emp\_eval Body pane, add eval\_frequency function immediately before the END emp\_eval; statement, as shown by the following code. The control structures are in bold font.

FUNCTION eval\_frequency (employee\_id IN employees.employee\_id%TYPE)

RETURN PLS\_INTEGER AS

hire\_date employees.hire\_date%TYPE; -- start of employment

today employees.hire\_date%TYPE; -- today's date

eval\_freq PLS\_INTEGER; -- frequency of evaluations

BEGIN

SELECT SYSDATE INTO today FROM DUAL; -- set today's date

SELECT e.hire\_date INTO hire\_date -- determine when employee started

FROM employees e

WHERE employee\_id = e.employee\_id;

**IF**((hire\_date + (INTERVAL '120' MONTH)) < today) **THEN**

eval\_freq := 1;

**ELSE**

eval\_freq := 2;

**END IF**;

RETURN eval\_freq;

END eval\_frequency;

Compile and save emp\_eval Body.

**Using CASE...WHEN Selection Control**

The CASE...WHEN construct is a good alternative to nested IF...THEN statements if the variable that determines the course of action has several possible values. The CASE evaluates a condition, and performs a different action for each possible value. Whenever possible, use the CASE...WHEN statement instead of IF...THEN, both for readability and efficiency. The general form of the CASE...WHEN construct follows:

CASE *condition*

WHEN *value\_1* THEN *expression\_1*;

WHEN *value\_2* THEN *expression\_2*;

...

ELSE *expression\_default*;

END CASE;

Suppose that in the make\_evaluation function from ["Using IF...THEN...ELSE Selection Control"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHIABFA), you wanted to notify the hr user if a long-time employee who holds one of a select positions should be considered for a salary raise. Depending on the value of employees.job\_id, the program logic should notify the user of the suggested salary raise.

Note that you will use the DBMS\_OUTPUT.PUT\_LINE procedure, described in [*Oracle Database PL/SQL Packages and Types Reference*](http://docs.oracle.com/cd/B28359_01/appdev.111/b28419/d_output.htm#ARPLS036).

***Example 4-7 Using******CASE...WHEN Conditional Control***

In the emp\_eval Body pane, edit eval\_frequency function to add a job\_id variable and a CASE statement that is based on the value of thejob\_id, as shown by the following code. New code is in bold font.

FUNCTION eval\_frequency (employee\_id IN employees.employee\_id%TYPE)

RETURN PLS\_INTEGER AS

hire\_date employees.hire\_date%TYPE; -- start of employment

today employees.hire\_date%TYPE; -- today's date

eval\_freq PLS\_INTEGER; -- frequency of evaluations

**job\_id employees.job\_id%TYPE; -- category of the job**

BEGIN

SELECT SYSDATE INTO today FROM DUAL; -- set today's date

SELECT e.hire\_date INTO hire\_date -- determine when employee started

FROM employees e

WHERE employee\_id = e.employee\_id;

IF((hire\_date + (INTERVAL '120' MONTH)) < today) THEN

eval\_freq := 1;

**/\* Suggesting salary increase based on position \*/**

**SELECT e.job\_id INTO job\_id FROM employees e**

**WHERE employee\_id = e.employee\_id;**

**CASE job\_id**

**WHEN 'PU\_CLERK' THEN DBMS\_OUTPUT.PUT\_LINE(**

**'Consider 8% salary increase for employee number ' || employee\_id);**

**WHEN 'SH\_CLERK' THEN DBMS\_OUTPUT.PUT\_LINE(**

**'Consider 7% salary increase for employee number ' || employee\_id);**

**WHEN 'ST\_CLERK' THEN DBMS\_OUTPUT.PUT\_LINE(**

**'Consider 6% salary increase for employee number ' || employee\_id);**

**WHEN 'HR\_REP' THEN DBMS\_OUTPUT.PUT\_LINE(**

**'Consider 5% salary increase for employee number ' || employee\_id);**

**WHEN 'PR\_REP' THEN DBMS\_OUTPUT.PUT\_LINE(**

**'Consider 5% salary increase for employee number ' || employee\_id);**

**WHEN 'MK\_REP' THEN DBMS\_OUTPUT.PUT\_LINE(**

**'Consider 4% salary increase for employee number ' || employee\_id);**

**ELSE DBMS\_OUTPUT.PUT\_LINE(**

**'Nothing to do for employee #' || employee\_id);**

**END CASE;**

ELSE

eval\_freq := 2;

END IF;

RETURN eval\_freq;

END eval\_frequency;

Compile and save emp\_eval Body.

**Using Iterative Control**

Iteration structures, or loops, execute a sequence of statements repeatedly. There are three basic types of loops, the FOR...LOOP, theWHILE...LOOP, and the LOOP...EXIT WHEN.

**Using the FOR...LOOP**

The FOR...LOOP repeats a sequence of steps a defined number of times and uses a counter variable that must be in the defined range of integers to run the loop. The loop counter is implicitly declared in the FOR...LOOP statement, and implicitly incremented every time the loop runs. Note that the value of the loop counter can be used within the body of the loop, but it cannot be changed programmatically. The FOR...LOOP statement has the following form:

FOR *counter* IN *integer\_1*..*integer\_2* LOOP

...

END LOOP;

Suppose that in addition to recommending that some employees receive a raise, as described in ["Using CASE...WHEN Selection Control"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHCFABG), function eval\_frequency prints how the salary for the employee would change over a set number of years if this increase in salary continued.

Note that you will use the DBMS\_OUTPUT.PUT procedure, described in [*Oracle Database PL/SQL Packages and Types Reference*](http://docs.oracle.com/cd/B28359_01/appdev.111/b28419/d_output.htm#ARPLS036).

***Example 4-8 Using FOR...LOOP iterative control***

In the emp\_eval Body pane, edit eval\_frequency function so that it uses the proposed salary increase (sal\_raise) that is assigned in theCASE block to print the proposed salary over a number of years, starting with the current salary, salary. The new code is in bold font.

FUNCTION eval\_frequency (employee\_id IN employees.employee\_id%TYPE)

RETURN PLS\_INTEGER AS

hire\_date employees.hire\_date%TYPE; -- start of employment

today employees.hire\_date%TYPE; -- today's date

eval\_freq PLS\_INTEGER; -- frequency of evaluations

job\_id employees.job\_id%TYPE; -- category of the job

**salary employees.salary%TYPE; -- current salary**

**sal\_raise NUMBER(3,3) := 0; -- proposed % salary increase**

BEGIN

SELECT SYSDATE INTO today FROM DUAL; -- set today's date

SELECT e.hire\_date INTO hire\_date -- determine when employee started

FROM employees e

WHERE employee\_id = e.employee\_id;

IF((hire\_date + (INTERVAL '120' MONTH)) < today) THEN

eval\_freq := 1;

/\* Suggesting salary increase based on position \*/

SELECT e.job\_id INTO job\_id FROM employees e

WHERE employee\_id = e.employee\_id;

**SELECT e.salary INTO salary FROM employees** e

**WHERE employee\_id = e.employee\_id;**

**CASE job\_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; -- job type does not match ones that should consider increases**

**END CASE;**

**/\* If a salary raise is not zero, print the salary schedule \*/**

**IF (sal\_raise != 0) THEN -- start code for salary schedule printout**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('If the salary ' || salary || ' increases by ' ||**

**ROUND((sal\_raise \* 100),0) ||**

'% each year over 5 years, it would be ');

**FOR loop\_c IN 1..5 LOOP**

**salary := salary \* (1 + sal\_raise);**

**DBMS\_OUTPUT.PUT (ROUND(salary,2) ||', ');**

**END LOOP;**

**DBMS\_OUTPUT.PUT\_LINE('in successive years.');**

**END;**

**END IF;**

ELSE

eval\_freq := 2;

END IF;

RETURN eval\_freq;

END eval\_frequency;

Compile the emp\_eval Body.

**Using the****WHILE...LOOP**

The WHILE...LOOP repeats as long as a condition holds TRUE. The condition evaluates at the top of each loop and if TRUE, the statements in the body of the loop run. If the condition is FALSE or NULL, the control passes to the next statement after the loop. The general form of theWHILE...LOOP control structure follows.

WHILE *condition* LOOP

...

END LOOP;

Note that the WHILE...LOOP may run indefinatelly, so use it with care.

Suppose that the EVAL\_FREQUENCY function in ["Using the FOR...LOOP"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHJFEHH) uses the WHILE...LOOP instead of the FOR...LOOP, and terminates after the proposed salary reaches the upper salary limit for the job\_id.

***Example 4-9 Using WHILE...LOOP Iterative Control***

In the emp\_eval Body pane, edit eval\_frequency function so that it uses the proposed salary increase (sal\_raise) that is assigned in theCASE block to print the proposed salary over a number of years and stops when it reaches the maximum level possible for the job\_id. The new code is in bold font.

FUNCTION eval\_frequency (employee\_id IN employees.employee\_id%TYPE)

RETURN PLS\_INTEGER AS

hire\_date employees.hire\_date%TYPE; -- start of employment

today employees.hire\_date%TYPE; -- today's date

eval\_freq PLS\_INTEGER; -- frequency of evaluations

job\_id employees.job\_id%TYPE; -- category of the job

salary employees.salary%TYPE; -- current salary

sal\_raise NUMBER(3,3) := 0; -- proposed % salary increase

**sal\_max jobs.max\_salary%TYPE; -- maximum salary for a job**

BEGIN

SELECT SYSDATE INTO today FROM DUAL; -- set today's date

SELECT e.hire\_date INTO hire\_date -- determine when employee started

FROM employees e

WHERE employee\_id = e.employee\_id;

IF((hire\_date + (INTERVAL '120' MONTH)) < today) THEN

eval\_freq := 1;

/\* Suggesting salary increase based on position \*/

SELECT e.job\_id INTO job\_id FROM employees e

WHERE employee\_id = e.employee\_id;

SELECT e.salary INTO salary FROM employees e

WHERE employee\_id = e.employee\_id;

**SELECT j.max\_salary INTO sal\_max FROM jobs j**

**WHERE job\_id = j.job\_id;**

CASE job\_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 a salary raise is not zero, print the salary schedule \*/

IF (sal\_raise != 0) THEN -- start code for salary schedule printout

BEGIN

DBMS\_OUTPUT.PUT\_LINE('If the salary ' || salary || ' increases by ' ||

ROUND((sal\_raise \* 100),0) ||

**'% each year, it would be ');**

**WHILE salary <= sal\_max LOOP**

salary := salary \* (1 + sal\_raise);

DBMS\_OUTPUT.PUT (ROUND(salary,2) ||', ');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('in successive years.');

END;

END IF;

ELSE

eval\_freq := 2;

END IF;

RETURN eval\_freq;

END eval\_frequency;

**Using the****LOOP...EXIT WHEN**

The LOOP...EXIT WHEN structure enables you to exit the loop if further processing is undesirable. If the EXIT WHEN condition evaluates toTRUE, the loop exits and control passes to the next statement.

The eval\_frequency function in ["Using the WHILE...LOOP"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHFHAGC) uses the WHILE...LOOP. Note that the last computed value may (and typically does) exceed the maximum possible value for a salary in the last iteration of the loop. If you use the LOOP\_EXIT WHEN construct instead of the WHILE...LOOP, you can have finer control for terminating the loop.

***Example 4-10 Using LOOP...EXIT WHEN Iterative Control***

In the emp\_eval Body pane, edit eval\_frequency function so that it uses the proposed salary increase (sal\_raise) that is assigned in theCASE block to print the proposed salary over a number of years and stops when it reaches the maximum level possible for the job\_id. The new code is in bold font.

FUNCTION eval\_frequency (employee\_id IN employees.employee\_id%TYPE)

RETURN PLS\_INTEGER AS

hire\_date employees.hire\_date%TYPE; -- start of employment

today employees.hire\_date%TYPE; -- today's date

eval\_freq PLS\_INTEGER; -- frequency of evaluations

job\_id employees.job\_id%TYPE; -- category of the job

salary employees.salary%TYPE; -- current salary

sal\_raise NUMBER(3,3) := 0; -- proposed % salary increase

sal\_max jobs.max\_salary%TYPE; -- maximum salary for a job

BEGIN

SELECT SYSDATE INTO today FROM DUAL; -- set today's date

SELECT e.hire\_date INTO hire\_date -- determine when employee started

FROM employees e

WHERE employee\_id = e.employee\_id;

IF((hire\_date + (INTERVAL '120' MONTH)) < today) THEN

eval\_freq := 1;

/\* Suggesting salary increase based on position \*/

SELECT e.job\_id INTO job\_id FROM employees e

WHERE employee\_id = e.employee\_id;

SELECT e.salary INTO salary FROM employees e

WHERE employee\_id = e.employee\_id;

SELECT j.max\_salary INTO sal\_max FROM jobs j

WHERE job\_id = j.job\_id;

CASE job\_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 a salary raise is not zero, print the salary schedule \*/

IF (sal\_raise != 0) THEN -- start code for salary schedule printout

BEGIN

DBMS\_OUTPUT.PUT\_LINE('If the salary ' || salary || ' increases by ' ||

ROUND((sal\_raise \* 100),0) ||

'% each year, it would be ');

**LOOP**

salary := salary \* (1 + sal\_raise);

**EXIT WHEN salary > sal\_max;**

DBMS\_OUTPUT.PUT (ROUND(salary,2) ||', ');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('in successive years.');

END;

END IF;

ELSE

eval\_freq := 2;

END IF;

RETURN eval\_freq;

END eval\_frequency;

**Using Composite Data Structures; Records**

A composite data structure, or a **record,** is a group of related data items stored in fields, each with its own name and data type. You can think of a record as a variable that can hold a table row, or some columns from a table row. The fields correspond to table columns. The record structure is very efficient for passing related items to a subprogram as a single parameter, and for effectively using related fields from different tables during run time.

You must define a RECORD as a type, and access its fields through the point notation. The general form for defining and using a record follows:

TYPE *record\_name* IS RECORD( -- define record type

*field\_1* *data\_type*, -- define fields in record

...

*field\_n* *data\_type*);

...

*variable\_name record\_name*; -- define variable of new type

...

BEGIN

...

...*variable\_name.field1*...; -- use fields of new variable

...*variable\_name.fieldn*...;

...

END...;

In the eval\_frequency function from ["Using the LOOP...EXIT WHEN"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHDJAGA), you used several related parameters. You can use the RECORDconstruct to combine some of these items into a single parameter.

You will create a type that will contain the upper and lower limits for a job specification.

**To create a RECORD type:**

1. In the Connections navigation hierarchy, click the plus sign (**+**) beside Packages to expand the group.
2. Right-click **EMP\_EVAL**.
3. Select **Edit**.

The emp\_eval pane appears. It shows the specification of the emp\_eval package.

1. In the emp\_eval package specification, immediately before the closing line of the package specification, END emp\_eval, enter the definition of a record type sal\_info, which contains the fields necessary for evaluating salary levels.
2. TYPE sal\_info IS RECORD -- type for salary, limits, raises, and adjustments
3. ( job\_id jobs.job\_id%type
4. , sal\_min jobs.min\_salary%type
5. , sal\_max jobs.max\_salary%type
6. , salary employees.salary%type
7. , sal\_raise NUMBER(3,3) );
8. Compile and save emp\_eval.

The following message appears in the Messages-Log pane:

EMP\_EVAL Compiled

Once you declare a new RECORD type in the package specification, you can use it inside the package body to declare variables of that type. You will create a new procedure, salary\_schedule, and invoke it from the eval\_frequency function using a variable of type sal\_info.

Note that PL/SQL compilation is a single path process; if a subprogram is declared after its client subprogram, PL/SQL compiler throws an error. To work around this situation, you could declare all the subprograms that are not already declared in the package specification at the top of the package body. The definition of the subprogram can be anywhere within the package body. See step [2](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHEHDHE) in the following task on instructions for declaring function eval\_frequency and procedures salary\_schedule and add\_eval.

**To use a****RECORD type:**

1. In the emp\_eval Body pane, add the definition of the salary\_schedule procedure immediately before the END emp\_eval statement, as shown by the following code. Note that this code is similar to the content of the BEGIN...END block in eval\_frequency that executes if the salary raise is nonzero.
2. PROCEDURE salary\_schedule(emp IN sal\_info) AS
3. accumulating\_sal NUMBER; -- accumulator
4. BEGIN
5. DBMS\_OUTPUT.PUT\_LINE('If the salary of ' || emp.salary ||
6. ' increases by ' || ROUND((emp.sal\_raise \* 100),0) ||
7. '% each year, it would be ');
8. accumulating\_sal := emp.salary; -- assign value of sal to accumulator
9. WHILE accumulating\_sal <= emp.sal\_max LOOP
10. accumulating\_sal := accumulating\_sal \* (1 + emp.sal\_raise);
11. DBMS\_OUTPUT.PUT (ROUND( accumulating\_sal,2) ||', ');
12. END LOOP;
13. DBMS\_OUTPUT.PUT\_LINE('in successive years.');
14. END salary\_schedule;
15. In the emp\_eval Body pane, near the top of the emp\_eval body definition, enter declarations for eval\_frequency andsalary\_schedule. New code is in bold font.
16. create or replace
17. PACKAGE BODY emp\_eval AS
18. /\* local subprogram declarations \*/
19. **FUNCTION eval\_frequency (employee\_id employees.employee\_id%TYPE) RETURN NUMBER;**
20. **PROCEDURE salary\_schedule(emp IN sal\_info);**
21. **PROCEDURE add\_eval(employee\_id IN NUMBER, today IN DATE);**
23. /\* subprogram definition \*/
24. PROCEDURE eval\_department (dept\_id IN NUMBER) AS
25. ...
26. In the emp\_eval Body pane, edit eval\_frequency function so that it uses the new sal\_info type as variable emp\_sal, populates its fields, and invokes salary\_schedule. Note that the code that was previously executed if the salary raise was nonzero is no longer part of this function; it has been incorporated into the salary\_schedule procedure. Note also that the declarations at the top of the functions changed. New code is in bold font.
27. FUNCTION eval\_frequency (employee\_id employees.employee\_id%TYPE)
28. RETURN PLS\_INTEGER AS
29. hire\_date employees.hire\_date%TYPE; -- start of employment
30. today employees.hire\_date%TYPE; -- today's date
31. eval\_freq PLS\_INTEGER; -- frequency of evaluations
32. **emp\_sal SAL\_INFO; -- record for fields associated**
33. **-- with salary review**
34. BEGIN
35. SELECT SYSDATE INTO today FROM DUAL; -- set today's date
36. SELECT e.hire\_date INTO hire\_date -- determine when employee started
37. FROM employees e
38. WHERE employee\_id = e.employee\_id;
40. IF((hire\_date + (INTERVAL '120' MONTH)) < today) THEN
41. eval\_freq := 1;
43. **/\* populate emp\_sal \*/**
44. **SELECT e.job\_id INTO emp\_sal.job\_id FROM employees e**
45. **WHERE employee\_id = e.employee\_id;**
46. **SELECT j.min\_salary INTO emp\_sal.sal\_min FROM jobs j**
47. **WHERE emp\_sal.job\_id = j.job\_id;**
48. **SELECT j.max\_salary INTO emp\_sal.sal\_max FROM jobs j**
49. **WHERE emp\_sal.job\_id = j.job\_id;**
50. **SELECT e.salary INTO emp\_sal.salary FROM employees e**
51. **WHERE employee\_id = e.employee\_id;**
52. **emp\_sal.sal\_raise := 0; -- default**
54. CASE **emp\_sal.**job\_id
55. WHEN 'PU\_CLERK' THEN **emp\_sal.**sal\_raise := 0.08;
56. WHEN 'SH\_CLERK' THEN **emp\_sal.**sal\_raise := 0.07;
57. WHEN 'ST\_CLERK' THEN **emp\_sal.**sal\_raise := 0.06;
58. WHEN 'HR\_REP' THEN **emp\_sal.**sal\_raise := 0.05;
59. WHEN 'PR\_REP' THEN **emp\_sal.**sal\_raise := 0.05;
60. WHEN 'MK\_REP' THEN **emp\_sal.**sal\_raise := 0.04;
61. ELSE NULL;
62. END CASE;
64. /\* If a salary raise is not zero, print the salary schedule \*/
65. **IF (emp\_sal.sal\_raise != 0) THEN salary\_schedule(emp\_sal);**
66. END IF;
68. ELSE
69. eval\_freq := 2;
70. END IF;
72. RETURN eval\_freq;
73. END eval\_frequency;
74. Compile and save emp\_eval Body.

The following message appears in the Messages - Log pane:

EMP\_EVAL Body Compiled

**Retrieving Data from a Set Using Cursors and Cursor Variables**

A **cursor** is a type of pointer that is built into PL/SQL for querying the database, retrieving a set of records (a **result set**), and enabling the developer to access these records one row at a time. A cursor is a handle or a name for a private in-memory SQL area that holds a parsed statement and related information. Oracle Database implicitly manages cursors. However, there are a few interfaces that enable you to use cursors explicitly, as a named resource within a program to more effectively parse embedded SQL statements. The two main types of cursors are therefore defined as:

* **Implicit cursors** can be used in PL/SQL without explicit code to process the cursor itself. A result set that is returned by the cursors can be used programmatically, but there is no programmatic control over the cursor itself.
* **Explicit cursors** allow you to programmatically manage the cursor, and give you a detailed level of control over record access in the result set.

Each user session may have many open cursors, up to the limit set by the initialization parameter OPEN\_CURSORS, which is 50 by default. You should ensure that your applications close cursors to conserve system memory. If a cursor cannot be opened because theOPEN\_CURSORS limit is reached, contact the database administrator to alter the OPEN\_CURSORS initialization parameter.

**Using Explicit Cursors**

The implicit cursor, such as in a FOR...LOOP, are generally more efficient than an explicit cursor. However, explicit cursors may be more appropriate for your program, and they also allow you to manage specific in-memory areas as a named resource.

An explicit cursor has the attributes described in the following table:

| **Cursor Attribute** | **Description** |
| --- | --- |
| %NOTFOUND | Returns TRUE or FALSE, based on the results of the last fetch. |
| %FOUND | Returns TRUE or FALSE, based on the results of the last fetch; negation of the %NOTFOUND results. |
| %ROWCOUNT | Returns the number of rows fetched. Can be called at any time after the first fetch. Also returns the number of rows affected from UPDATE and DELETE statements. |
| %ISOPEN | Returns TRUE if a cursor is still open. |

An explicit cursor must be defined as a variable of the same type as the columns it fetches; the data type of the record is derived from the cursor definition. Explicit cursors must be opened and may then retrieve rows within a LOOP...EXIT WHEN structure and then closed. The general form for using cursors follows:

DECLARE

CURSOR *cursor\_name* *type* IS *query\_definition*;

OPEN *cursor\_name*

LOOP

FETCH *record*;

EXIT WHEN *cursor\_name*%NOTFOUND;

...; -- process fetched row

END LOOP;

CLOSE *cursor\_name*;

This is what happens during the life time of a cursor:

* The OPEN statement parses the query identified by the cursor, binds the inputs, and ensures that you can successfully fetch records from the result set.
* The FETCH statement runs the query, and then finds and retrieves the matching rows. You will need to define and use local variables as buffers for the data returned by the cursor, and then process the specific record.
* The CLOSE statement completes cursor processing and closes the cursor. Note that once a cursor is closed you cannot retrieve additional records from the result set.

You can implement procedure eval\_department, which you declared in ["Creating a Package"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHHDBFE), using a cursor for each employee record that matches the query.

***Example 4-11 Using a cursor to retrieve rows form a result set***

The cursor emp\_cursor fetches individual rows from the result set. Depending on the value of the eval\_frequency function for each row and the time of the year that the eval\_department procedure runs, a new evaluation record is created for the employee by invoking theadd\_eval procedure. Note that the buffer variable, emp\_record, is defined as a %ROWTYPE.

In the emp\_eval package specification, edit the declaration of procedure eval\_department:

PROCEDURE eval\_department(department\_id IN employees.department\_id%TYPE);

In the emp\_eval Body pane, edit eval\_department procedure.

PROCEDURE eval\_department(department\_id IN employees.department\_id%TYPE) AS

-- declaring buffer variables for cursor data

emp\_record employees%ROWTYPE;

-- declaring variable to monitor if all employees need evaluations

all\_evals BOOLEAN;

-- today's date

today DATE;

-- declaring the cursor

CURSOR emp\_cursor IS SELECT \* FROM employees e

WHERE department\_id = e.department\_id;

BEGIN

-- determine if all evaluations must be done or just for newer employees;

-- this depends on time of the year

today := SYSDATE;

IF (EXTRACT(MONTH FROM today) < 6) THEN all\_evals := FALSE;

ELSE all\_evals := TRUE;

END IF;

OPEN emp\_cursor;

-- start creating employee evaluations in a specific department

DBMS\_OUTPUT.PUT\_LINE('Determining evaluations necessary in department # ' ||

department\_id);

LOOP

FETCH emp\_cursor INTO emp\_record; -- getting specific record

EXIT WHEN emp\_cursor%NOTFOUND; -- all records are been processed

IF all\_evals THEN

add\_eval(emp\_record.employee\_id, today); -- create evals for all

ELSIF (eval\_frequency(emp\_record.employee\_id) = 2) THEN

add\_eval(emp\_record.employee\_id, today); -- create evals; newer employees

END IF;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Processed ' || emp\_cursor%ROWCOUNT || ' records.');

CLOSE emp\_cursor;

END eval\_department;

Compile the emp\_eval package specification, and then the emp\_eval Body.

The following message appears in the Messages-Log panes:

EMP\_EVAL Body Compiled

**Using Cursor Variables: REF Cursors**

Cursors are static, as they are defined by the queries that create them. In some cases, the queries themselves are created at runtime. A cursor variable, known as a REF CURSOR, is more flexible than a cursor because it is independent of a specific query. It can be opened for a query, can process the result set, and can be re-used for a query that returns the same set of columns. This also makes REF CURSORs ideal for passing results of a query between subprograms.

REF CURSORS can be declared with a return type that specifies the form of the result set (strongly typed), or without a return type to retrieve any result set (weakly-typed). Oracle recommends that you declare a REF CURSOR with a return type as it is less prone to error because of its strong association with correctly formulated queries. If you need a more flexible cursor that may be associated with several interchangeable types, use the predefined type SYS\_REFCURSOR.

The general form for using a REF CURSORs follows.

DECLARE

TYPE *cursor\_type* IS REF CURSOR RETURN *return\_type*;

*cursor\_variable cursor\_type*;

*single\_record return\_type*;

OPEN *cursor\_variable* FOR *query\_definition*;

LOOP

FETCH *record*;

EXIT WHEN *cursor\_name*%NOTFOUND;

...; -- process fetched row

END LOOP;

CLOSE *cursor\_name*;

This is what happens during the life time of a REF CURSOR and a cursor variable:

* The REF CURSOR type [with a return type] is declared.
* The cursor variable that matches the cursor type is declared.
* The variable for processing individual rows of the result set is declared; its type must be the same as the return type of the REF CURSOR type definition.
* The OPEN statement parses the query to the cursor variable.
* The FETCH statement inside the loop runs the query, and retrieves the matching rows into the local variable of the same type as the return type of the REF CURSOR for further processing.
* The CLOSE statement completes cursor processing and closes the REF CURSOR.

In ["Using Explicit Cursors"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHDCIIC), the procedure eval\_department retrieves a result set, processes it using a cursor, closes the cursor, and ends. If you declare the cursor as a REF CURSOR type, you could modify it to process more departments (for example, three consecutive departments) by re-using the cursor.

Note that the fetching loop is part of the new eval\_fetch\_control procedure, that uses the cursor variable as input. This has an additional benefit of separating the processing of the result set from the definition of the query. You could write a procedure (eval\_everyone) that initiates evaluations for all employees in the company, not just on a department basis.

Note also that eval\_department uses a single field of a record to call procedure add\_eval, which runs three separate queries on the same record. This is very inefficient; you will re-write the add\_eval to use the entire record buffer of the REF CURSOR.

**To use a REF CURSOR:**

1. In the emp\_eval specification, add the REF CURSOR type definition, emp\_refcursor\_type. The type is defined at package level for visibility for all subprograms. Also add a declaration for procedure eval\_everyone. The new code is in bold font.
2. create or replace
3. PACKAGE emp\_eval AS
4. PROCEDURE eval\_department (department\_id IN employees.department\_id%TYPE);
5. **PROCEDURE eval\_everyone;**
6. FUNCTION calculate\_score(eval\_id IN scores.evaluation\_id%TYPE
7. , perf\_id IN scores.performance\_id%TYPE)
8. RETURN NUMBER;
9. TYPE SAL\_INFO IS RECORD -- type for salary, limits, raises, and adjustments
10. ( job\_id jobs.job\_id%type
11. , sal\_min jobs.min\_salary%type
12. , sal\_max jobs.max\_salary%type
13. , salary employees.salary%type
14. , sal\_raise NUMBER(3,3));
16. **TYPE emp\_refcursor\_type IS REF CURSOR RETURN employees%ROWTYPE;**
17. **-- the REF CURSOR type for result set fetches**
18. END emp\_eval;
19. In the emp\_eval Body pane, add a forward declaration for procedure eval\_loop\_control and edit the declaration of procedureadd\_eval. New code is in bold font.
20. CREATE OR REPLACE PACKAGE BODY emp\_eval AS
21. /\* local subprogram declarations \*/
22. FUNCTION eval\_frequency (employee\_id IN employees.employee\_id%TYPE)
23. RETURN NUMBER;
24. PROCEDURE salary\_schedule(emp IN sal\_info);
25. **PROCEDURE add\_eval(emp\_record IN employees%ROWTYPE, today IN DATE);**
26. **PROCEDURE eval\_loop\_control(emp\_cursor IN emp\_refcursor\_type);**
27. ...
28. In the emp\_eval Body pane, edit eval\_department procedure to retrieve three separate result sets based on the department, and to call the eval\_loop\_control procedure.
29. PROCEDURE eval\_department(department\_id IN employees.department\_id%TYPE) AS
30. **-- declaring the REF CURSOR**
31. **emp\_cursor emp\_refcursor\_type;**
32. **department\_curr departments.department\_id%TYPE;**
34. BEGIN
35. **department\_curr := department\_id; -- starting with the first department**
36. **FOR loop\_c IN 1..3 LOOP**
37. **OPEN emp\_cursor FOR**
38. **SELECT \***
39. **FROM employees e**
40. **WHERE department\_curr = e.department\_id;**
41. **-- create employee evaluations is specific departments**
42. DBMS\_OUTPUT.PUT\_LINE('Determining necessary evaluations in department #' ||
43. department\_curr);
44. **eval\_loop\_control(emp\_cursor); -- call to process the result set**
45. **DBMS\_OUTPUT.PUT\_LINE('Processed ' || emp\_cursor%ROWCOUNT || ' records.');**
46. **CLOSE emp\_cursor;**
47. **department\_curr := department\_curr + 10;**
48. **END LOOP;**
49. END eval\_department;
50. In the emp\_eval Body pane, edit add\_eval procedure to use the entire retrieved record of employee%ROWTYPE, instead of anemployee\_id. Note that you no longer need any declarations at the beginning of the procedure.
51. PROCEDURE add\_eval(**emp\_record IN employees%ROWTYPE**, today IN DATE) AS
52. BEGIN
53. -- inserting a new row of values into evaluations table
54. INSERT INTO evaluations VALUES (
55. evaluations\_seq.NEXTVAL, -- evaluation\_id
56. **emp\_record.employee\_id, -- employee\_id**
57. today, -- evaluation\_date
58. **emp\_record.job\_id, -- job\_id**
59. **emp\_record.manager\_id, -- manager\_id**
60. **emp\_record.department\_id, -- department\_id**
61. 0); -- total\_score
63. END add\_eval;
64. Towards the end of code in the emp\_eval Body pane, add eval\_loop\_control procedure to fetch the individual records from the result set and to process them. Note that much of this code is from an earlier definition of the eval\_department procedure in ["Using Explicit Cursors"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHDCIIC). New structures are in bold font.
65. **PROCEDURE eval\_loop\_control(emp\_cursor IN emp\_refcursor\_type) AS**
66. -- declaring buffer variable for cursor data
67. emp\_record employees%ROWTYPE;
68. -- declaring variable to monitor if all employees need evaluations
69. all\_evals BOOLEAN;
70. -- today's date
71. today DATE;
72. BEGIN
73. -- determine if all evaluations must be done or just for newer employees;
74. -- this depends on time of the year
75. today := SYSDATE;
77. IF (EXTRACT(MONTH FROM today) < 6) THEN
78. all\_evals := FALSE;
79. ELSE all\_evals := TRUE;
80. END IF;
81. LOOP
82. FETCH emp\_cursor INTO emp\_record; -- getting specific record
83. EXIT WHEN emp\_cursor%NOTFOUND; -- all records are been processed
84. IF all\_evals THEN
85. **add\_eval(emp\_record, today); -- create evaluations for all**
86. ELSIF (eval\_frequency(emp\_record.employee\_id) = 2) THEN
87. **add\_eval(emp\_record, today);** -- create evaluations for newer employees
88. END IF;
89. END LOOP;
90. **END eval\_loop\_control;**
91. In the emp\_eval Body pane, add eval\_everyone procedure, which retrieves a result set that contains all employees in the company. Note that its code is similar to that of procedure eval\_department in Step [3](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHCCIHH).
92. PROCEDURE eval\_everyone AS
93. -- declaring the REF CURSOR type
94. emp\_cursor emp\_refcursor\_type;
95. BEGIN
96. OPEN emp\_cursor FOR SELECT \* FROM employees;
97. -- start creating employee evaluations in a specific department
98. DBMS\_OUTPUT.PUT\_LINE('Determining the number of necessary evaluations');
99. eval\_loop\_control(emp\_cursor); -- call to process the result set
100. DBMS\_OUTPUT.PUT\_LINE('Processed ' || emp\_cursor%ROWCOUNT || ' records.');
101. CLOSE emp\_cursor;
102. END eval\_everyone;
103. In the emp\_eval pane, compile and save emp\_eval specification.

The following message appears in the Messages-Log pane:

EMP\_EVAL Compiled

1. In the emp\_eval body pane, compile and save emp\_eval body.

The following message appears in the Messages-Log pane:

EMP\_EVAL Body Compiled

**Using Collections;****Index-By****Tables**

Another group of user-defined datatypes available in PL/SQL is a **collection**, which is Oracle's version of one-dimensional arrays. A collection is a data structure that can hold a number of rows of data in a single variable. In contrast to a record, which holds only one row of data of different types, the data in a collection must be of the same type. In other programming languages, the types of structures represented by collections are called **arrays**.

Collections are used to maintain lists of information and can significantly improve your application's performance because they allow direct access to their elements. There are three types of collection structures: index-by tables, nested tables, and variable arrays.

* An **index-by****table** is the most flexible and generally best-performing collection type for use inside PL/SQL programs.
* A **nested****table** is appropriate for large collections that an application stores and retrieves in portions.
* A **VARRAY** is appropriate for small collections that the application stores and retrieves in their entirety.

In this discussion, we will limit ourselves to index-by tables.

Index-by tables are also known as associative arrays, or sets of key-value pairs where each key is unique and is used to locate a corresponding value in the array. This key, or index, can be either an integer or a string.

Associative arrays represent data sets of arbitrary size that allow access to individual elements without knowledge of its relative position within the array, and without having to loop through all array elements.

For simple temporary storage of lookup data, associative arrays allow you to store data in memory, without using the disk space and network operations required for SQL tables. Because associative arrays are intended for temporary rather than persistent data storage, you cannot use them with SQL statements such as INSERT and SELECT INTO. You can, however, make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body.

Assigning a value using a key for the first time adds that key to the associative array. Subsequent assignments using the same key update the same entry. It is important to choose a key that is unique, such as a primary key of a database table, a result of a good numeric hash function, or a concatenation of strings that forms a unique string value.

Before declaring an index-by table, you must define its type. In the rest of this section, we will show you how to use an index-by table as part of our application.

We will show an efficient implementation of two types of associative arrays (indexed by PLS\_INTEGER and VARCHAR2) using the following steps:

* Defining a cursor.
* Defining the structure of an index-by table using the cursor's ROWTYPE or TYPE.
* Fetching cursor data into the index-by table using BULK COLLECT.
* Iterating through index-by table and looking up values using the index of a particular element.

**Creating Cursors for Index-by****Tables**

It is very convenient to define a cursor that would fetch the data into the index-by table, and then use its element type to create the index-by table. [Example 4-12](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHCJIFG) shows how to create two cursors, employees\_jobs\_cursor for fetching data from the hr.employees table, andjobs\_cursor for fetching data from the hr.jobs table. Notice that we are not using an ORDER BY clause for the second cursor.

***Example 4-12 Declaring cursors for index-by tables***

CURSOR employees\_jobs\_cursor IS

SELECT e.first\_name, e.last\_name, e.job\_id

FROM hr.employees e

ORDER BY e.job\_id, e.last\_name, e.first\_name;

CURSOR jobs\_cursor IS

SELECT j.job\_id, j.job\_title

FROM hr.jobs j;

**Defining****Index-by****Tables**

Now that you have declared your cursors, you can use the %ROWTYPE attribute to create the index-by PLS\_INTEGER tables employees\_jobsand jobs, as shown in [Example 4-13](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHHDCJD):

***Example 4-13 Creating******index-by PLS\_INTEGER tables based on the cursor structure***

TYPE employees\_jobs\_type IS TABLE OF employees\_jobs\_cursor%ROWTYPE

INDEX BY PLS\_INTEGER;

employees\_jobs employees\_jobs\_type;

TYPE jobs\_type IS TABLE OF jobs\_cursor%ROWTYPE

INDEX BY PLS\_INTEGER;

jobs jobs\_type;

To create a table that is indexed by a VARCHAR2, such as the job\_titles index-by table of job\_id, use the definition of these types from the original table, hr.jobs, as shown in [Example 4-14](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHIDJDE):

***Example 4-14 Creating index-by VARCHAR2 tables***

TYPE job\_titles\_type IS TABLE OF hr.jobs.job\_title%TYPE

INDEX BY hr.jobs.job\_id%TYPE;

job\_titles job\_titles\_type;

**Populating****Index-by PLS\_INTEGER****Tables;****BULK COLLECT**

If your work requires referencing a large quantity of data as local PL/SQL variables, the BULK COLLECT clause is much more efficient than looping through a result set one row at a time. When you query only some columns, you can store all the results for each column in a separate collection variable. When you query all the columns of a table, you can store the entire result set in a collection of records.

With the index-by PLS\_INTEGER employees\_jobs and jobs tables, you can now open the cursor and use BULK COLLECT to retrieve data, as shown in [Example 4-15](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHJHGAI):

***Example 4-15 Populating index-by******PLS\_INTEGER tables through BULK COLLECT***

OPEN employees\_jobs\_cursor;

FETCH employees\_jobs\_cursor BULK COLLECT INTO employees\_jobs;

CLOSE employees\_jobs\_cursor;

OPEN jobs\_cursor;

FETCH jobs\_cursor BULK COLLECT INTO jobs;

CLOSE jobs\_cursor;

**Populating Index-by VARCHAR2****Tables**

Once the jobs table contains data, use the FOR ... LOOP, as shown in [Example 4-16](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHGBFFE), to build the index-by VARCHAR2 table, job\_titles:

***Example 4-16 Populating index-by VARCHAR2 tables***

FOR i IN 1..jobs.COUNT() LOOP

job\_titles(jobs(i).job\_id) := jobs(i).job\_title;

END LOOP;

**Iterating Through an****Index-by****Table**

The structure employees\_jobs is a *dense* index-by table, because it is indexed by a PLS\_INTEGER. You can iterate through it simply by placing your operations within a FOR ... LOOP that counts from 1 through the COUNT() value of the table, as demonstrated in [Example 4-17](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHFJFDG). Note that the line in bold represents a direct look-up of a value in the job\_titles table.

***Example 4-17 Iterating through an index-by******PLS\_INTEGER table***

FOR i IN 1..employees\_jobs.count() LOOP

DBMS\_OUTPUT.PUT\_LINE(

RPAD(employees\_jobs(i).employee\_id, 10)||

RPAD(employees\_jobs(i).first\_name, 15)||

RPAD(employees\_jobs(i).last\_name, 15)||

**job\_titles(employees(i).job\_id));**

END LOOP;

The structure job\_titles is a *sparse* index-by table, indexed by a VARCHAR2. As [Example 4-18](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm#CIHFADJH) demonstrates, you can iterate through it within a WHILE ... END LOOP using a pre-defined counter that is equal to the first key value, and the NEXT() value of the table. You will notice that the elements are naturally sorted in lexical order of the index.

***Example 4-18 Iterating through an index-by******VARCHAR2 table***

DECLARE i hr.jobs.job\_id%TYPE := job\_titles.FIRST();

WHILE i IS NOT NULL LOOP

DBMS\_OUTPUT.PUT\_LINE(

RPAD(job\_titles(i).job\_id, 10)||

job\_titles(i).job\_title);

i := job\_titles.NEXT(i);

END LOOP;

**Handling****Errors and****Exceptions**

Error conditions, known as exceptions, are easy to detect and process within your PL/SQL code. When an error occurs, it raises an exception by stopping normal processing and transferring control to exception-handling code. This code is located at the end of thePL/SQL block. In PL/SQL, the checks and calls to error routines are performed automatically, with each exception having its own exception handler.

Predefined exceptions are raised automatically for certain common error conditions that involve variables or database operations. You can also declare custom exceptions for conditions that are errors with respect to your program, or as wrappers to existing Oracle messages.

**Existing PL/SQL and SQL Exceptions**

Oracle Database will automatically raise an exception if a PL/SQL program violates a known database rule, such as the predefined exception NO\_DATA\_FOUND if a SELECT INTO statement returns no rows. The following table shows some of the common exceptions.

| **Exception** | **Description** |
| --- | --- |
| ACCESS\_INTO\_NULL | A program attempts to assign values to the attributes of an uninitialized object |
| CASE\_NOT\_FOUND | None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_IS\_NULL | A program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| CURSOR\_ALREADY\_OPEN | A program attempts to open a cursor that is already open. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop. |
| DUP\_VAL\_ON\_INDEX | A program attempts to store duplicate values in a column that is constrained by a unique index. |
| INVALID\_CURSOR | A program attempts a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE\_ERROR is raised.) This exception is also raised when the LIMIT clause expression in a bulk FETCH statement does not evaluate to a positive number. |
| LOGIN\_DENIED | A program attempts to logon to Oracle database with a user name or password that is not valid. |
| NO\_DATA\_FOUND | A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table.  Because this exception is used internally by some SQL functions to signal completion, do not rely on this exception being propagated if you raise it within a function that is called as part of a query. |
| NOT\_LOGGED\_ON | A program issues a database call without being connected to Oracle database. |
| ROWTYPE\_MISMATCH | The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. When an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible. |
| SUBSCRIPT\_BEYOND\_COUNT | A program references a nested table or varray element using an index number larger than the number of elements in the collection. |
| SUBSCRIPT\_OUTSIDE\_LIMIT | A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range. |
| TOO\_MANY\_ROWS | A SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL cancels the assignment and raises VALUE\_ERROR. In procedural statements, VALUE\_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID\_NUMBER is raised.) |
| ZERO\_DIVIDE | A program attempts to divide a number by zero. |

***Example 4-19 Handling exceptions***

In the emp\_eval Body pane, edit eval\_department procedure to handle cases where the query does not return a result set. New code is in bold font.

PROCEDURE eval\_department(department\_id IN employees.department\_id%TYPE) AS

-- declaring the REF CURSOR

emp\_cursor emp\_refcursor\_type;

department\_curr departments.department\_id%TYPE;

BEGIN

department\_curr := department\_id; -- starting with the first department

FOR loop\_c IN 1..3 LOOP

OPEN emp\_cursor FOR

SELECT \*

FROM employees e

WHERE department\_curr = e.department\_id;

-- create employee evaluations is specific departments

DBMS\_OUTPUT.PUT\_LINE('Determining necessary evaluations in department #' ||

department\_curr);

eval\_loop\_control(emp\_cursor); -- call to process the result set

DBMS\_OUTPUT.PUT\_LINE('Processed ' || emp\_cursor%ROWCOUNT || ' records.');

CLOSE emp\_cursor;

department\_curr := department\_curr + 10;

END LOOP;

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**DBMS\_OUTPUT.PUT\_LINE ('The query did not return a result set');**

END eval\_department;

Compile and save emp\_eval Body.

**Custom Exceptions**

A package may contain custom exceptions for handling errors. Exceptions are declared in the program, in any declarative region, depending on how it is used: a subprogram, a package body, or a package specification.

An exception declaration has the following form:

*exception\_name* EXCEPTION;

To raise custom exceptions programmatically, based on incorrect values, you need to use the following form:

IF *condition* THEN

RAISE *exception\_name*;

To trap unexpected Oracle errors, you must include the exception handling instructions in your code, typically as the last block within the body of your subprogram or package. You should name the specific exceptions you are handling (both standard and custom), and use theOTHERS handler to trap unexpected errors. An exception body has the following form:

EXCEPTION

WHEN *exception\_name\_1* THEN

...;

DBMS\_OUTPUT.PUT\_LINE(*message\_1*);

...

WHEN OTHERS THEN

...

DBMS\_OUTPUT.PUT\_LINE(*message\_others*);

Alternatively, you may design your program to continue running after an exception is raised. You must then enclose the code that may generate an exception in a BEGIN...END block with its own exception handler. For example, code that traps the exception within a loop structure can handle the exception for an element that raises an error, and then continue with the next iteration of the loop.

In the following task, you will redesign the function calculate\_score to declare, raise and trap two possible exceptions, weight\_wrong andscore\_wrong.

***Example 4-20 Handling custom exceptions***

In the emp\_eval Body pane, edit calculate\_score function. New code is in bold font.

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; -- from SCORES

n\_weight performance\_parts.weight%TYPE; -- from PERFORMANCE\_PARTS

running\_total NUMBER := 0; -- used in calculations

max\_score CONSTANT scores.score%TYPE := 9; -- a constant limit check

max\_weight CONSTANT performance\_parts.weight%TYPE:= 1;

-- a constant limit check

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 -- check that weight is valid**

**IF n\_weight > max\_weight OR n\_weight < 0 THEN**

**RAISE weight\_wrong;**

**END IF;**

**END;**

**BEGIN -- check that score is valid**

**IF n\_score > max\_score OR n\_score < 0 THEN**

**RAISE score\_wrong;**

**END IF;**

**END;**

-- calculate the score

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;

Compile and save emp\_eval Body