Oracle Database 11*g*: Develop PL/SQL Program Units

Volume II • Student Guide

D49986GC12
Edition 1.2
April 2009
D59430

Author

Lauran K. Serhal

Technical Contributors and Reviewers

Don Bates
Claire Bennett
Zarko Cesljas
Purjanti Chang
Ashita Dhir
Peter Driver
Gerlinde Frenzen
Steve Friedberg
Nancy Greenberg
Thomas Hoogerwerf
Akira Kinutani
Chaitanya Koratamaddi
Timothy Leblanc
Bryn Llewellyn
Lakshmi Narapareddi
Essi Parast
Alan Paulson
Manish Pawar
Srinivas Putrevu
Bryan Roberts
Grant Spencer
Tulika Srivastava
Glenn Stokol
Jenny Tsai-Smith
Lex Van Der Werff
Ted Witiuk

Graphic Designer

Asha Thampy

Editors

Nita Pavitran
Aju Kumar

Publisher

Sheryl Domingue
Syed Ali

Contents

Appendix D: Review of PL/SQL

Appendix F: Studies for Implementing Triggers

Appendix G: Using the `DBMS_SCHEDULER` and `HTP` Packages

Appendix H: Review of JDeveloper

Appendix A
Practices and Solutions

Table of Contents

Practice 1: Getting Started

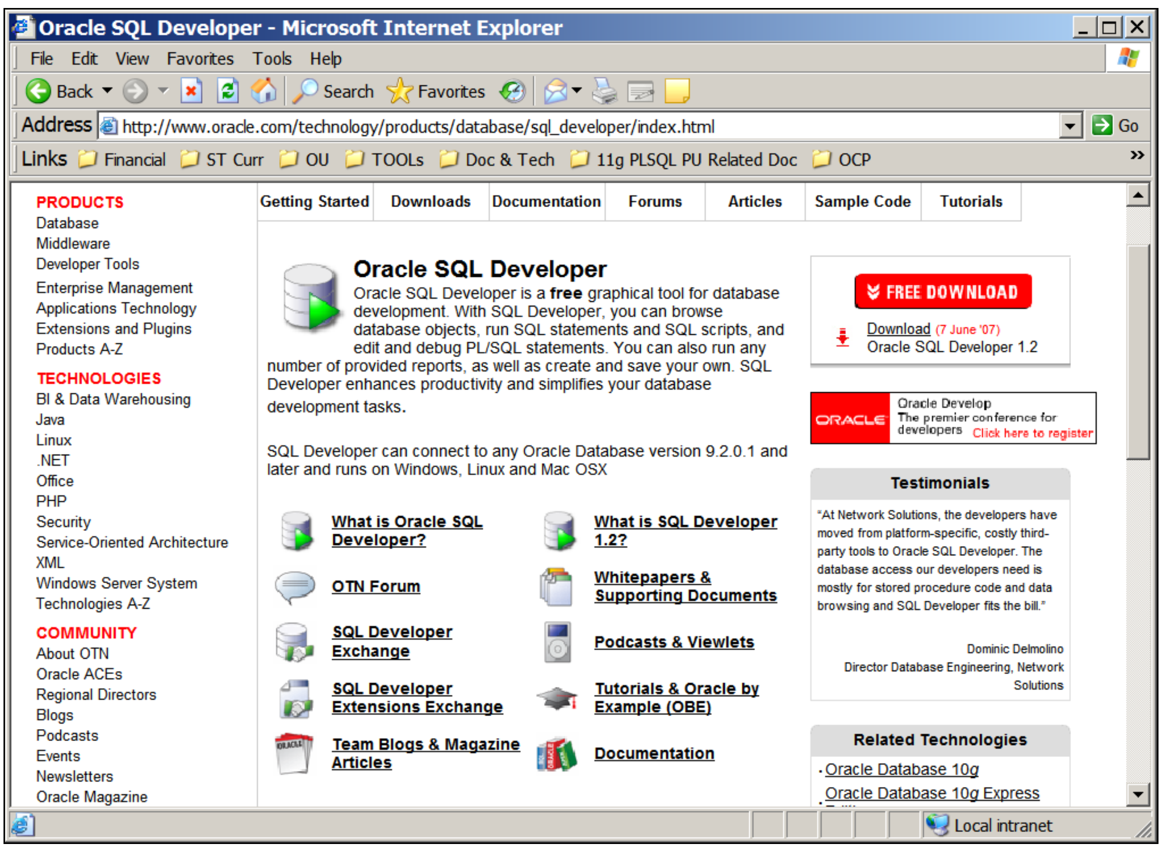
In this practice, you review the available SQL Developer resources. You also learn about your user account that you will use in this course. You then start SQL Developer, create a new database connection, and browse your `HR` tables. You also set some SQL Developer preferences, execute SQL statements, and execute an anonymous PL/SQL block using SQL Worksheet. Finally, you access and bookmark the Oracle Database 11*g* documentation and other useful Web sites that you can use in this course.

Identifying the Available SQL Developer Resources

1) Familiarize yourself with Oracle SQL Developer as needed using Appendix C: Using SQL Developer.

2) Access the online SQL Developer Home Page available online at:
http://www.oracle.com/technology/products/database/sql_developer/index.html

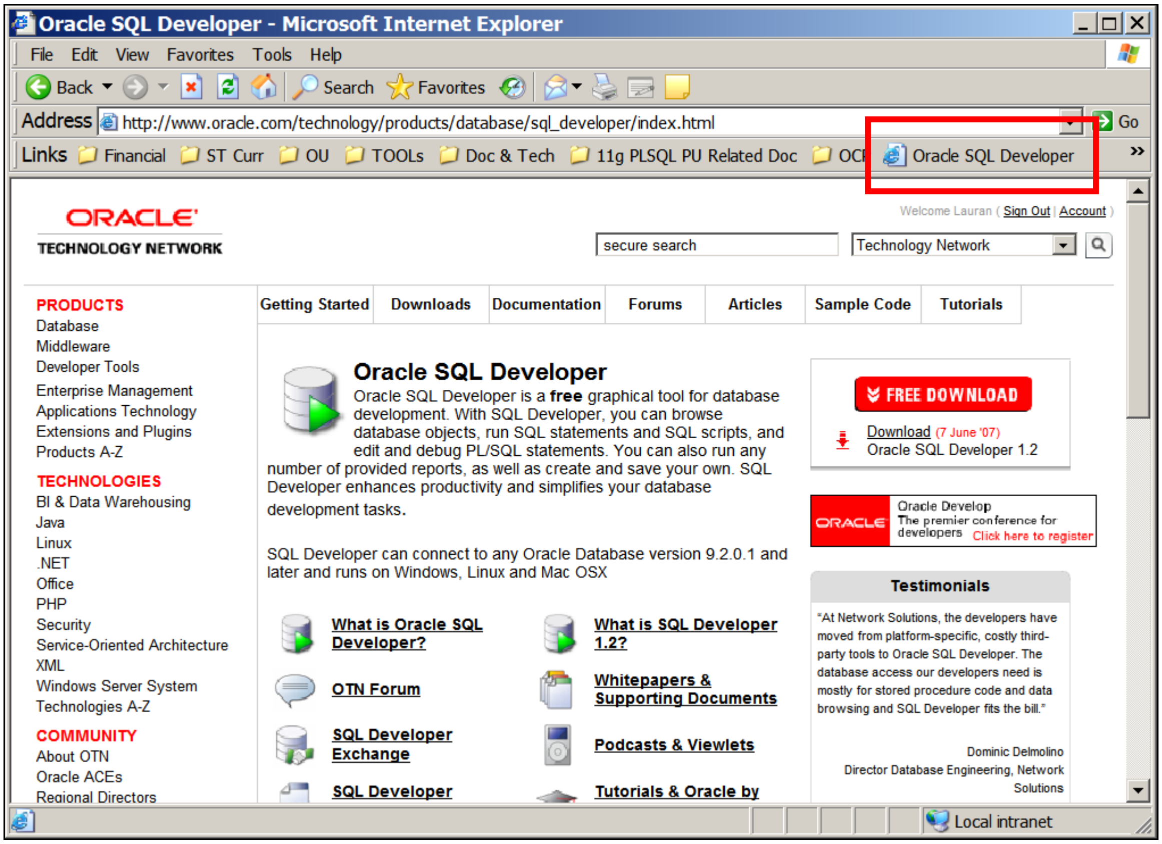
The SQL Developer Home page is displayed as follows:



Practice 1: Getting Started (continued)

3) Bookmark the page for easier future access.

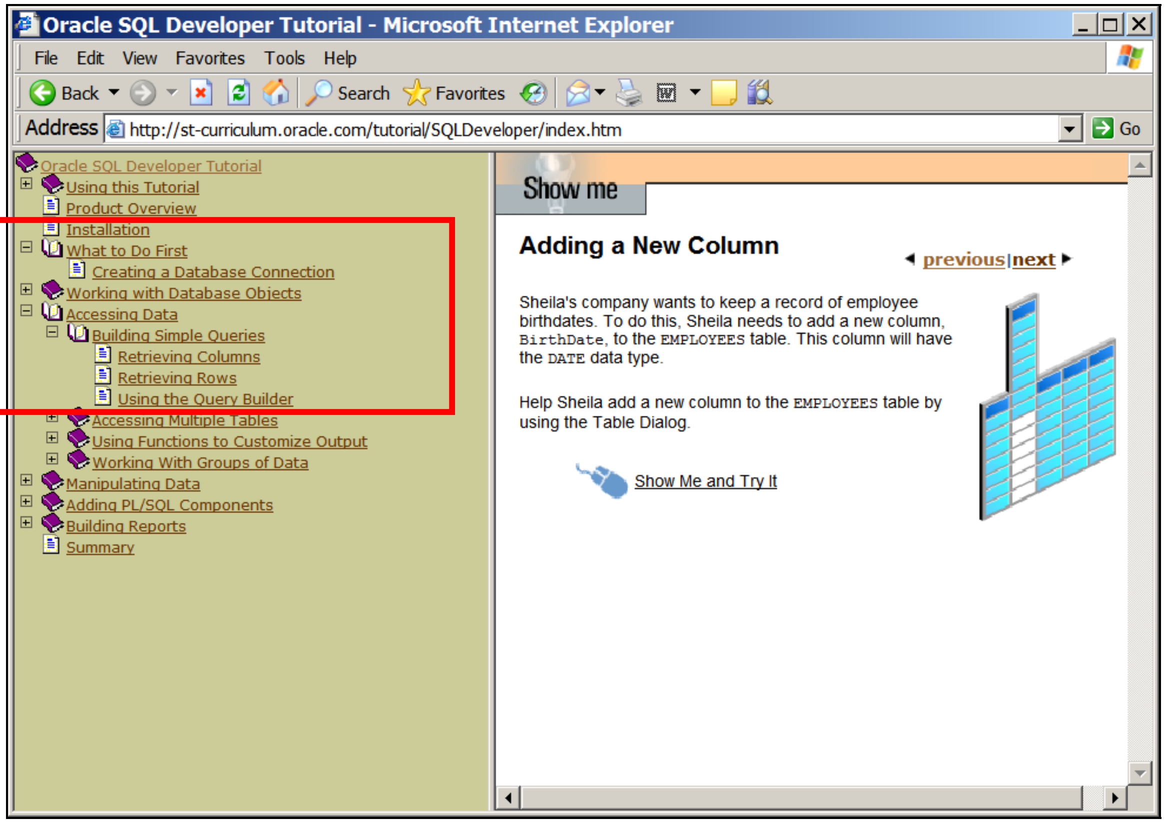
From the Windows Internet Explorer Address toolbar, click and drag the Explorer icon onto the Links toolbar. The link is added to your Links toolbar as follows:



4) Access the SQL Developer tutorial available online at:
http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm

Access the SQL Developer tutorial using the preceding URL. The following page is displayed:

Practice 1: Getting Started (continued)



5) Preview and experiment with the available links and demos in the tutorial as needed, especially the “Creating a Database Connection” and “Accessing Data” links.

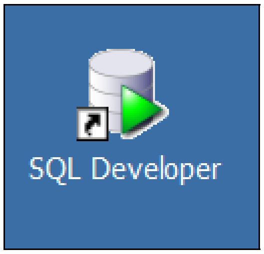
To review the section on creating a database connection, click the plus “+” sign next to the “What to Do First” link to display the “Creating a Database Connection” link. To review the Creating a Database Connection topic, click the topic’s link. To review the section on accessing data, click the plus “+” sign next to the “Accessing Data” link to display the list of available topics. To review any of the topics, click the topic’s link.

Identifying the Available SQL Developer Resources

1) Start up SQL Developer using the user ID and password that are provided to you by the instructor such as `oraxx` where *`xx`* is the number assigned to your PC.

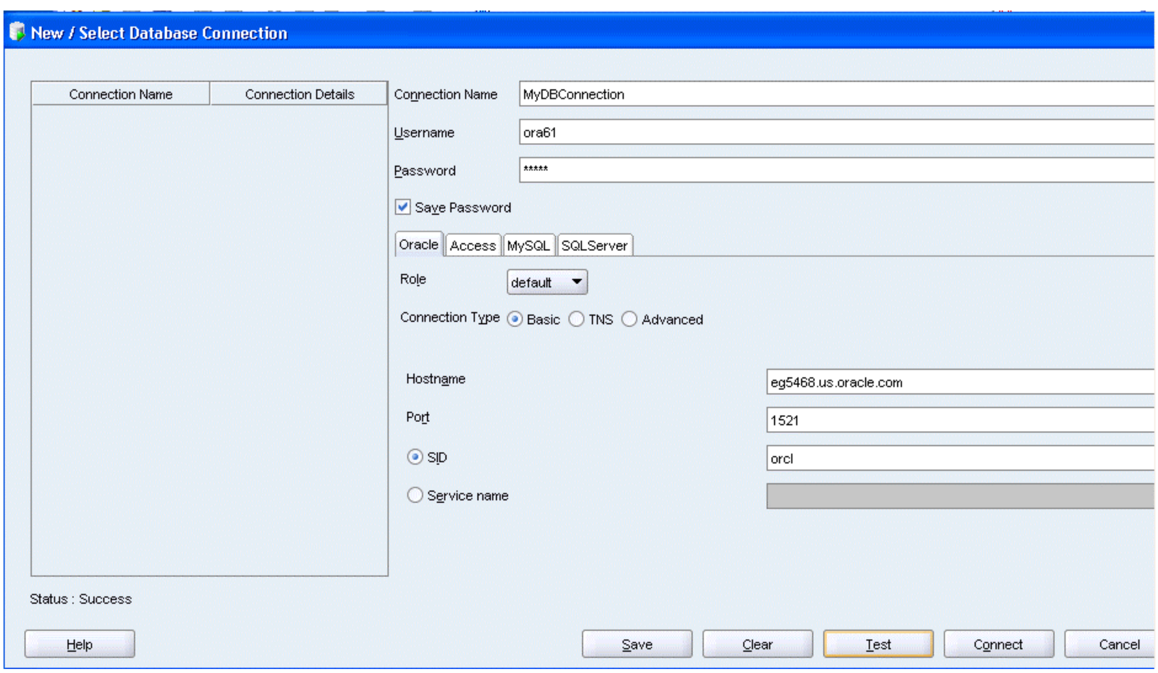
Click the SQL Developer icon on your desktop.

Practice 1: Getting Started (continued)



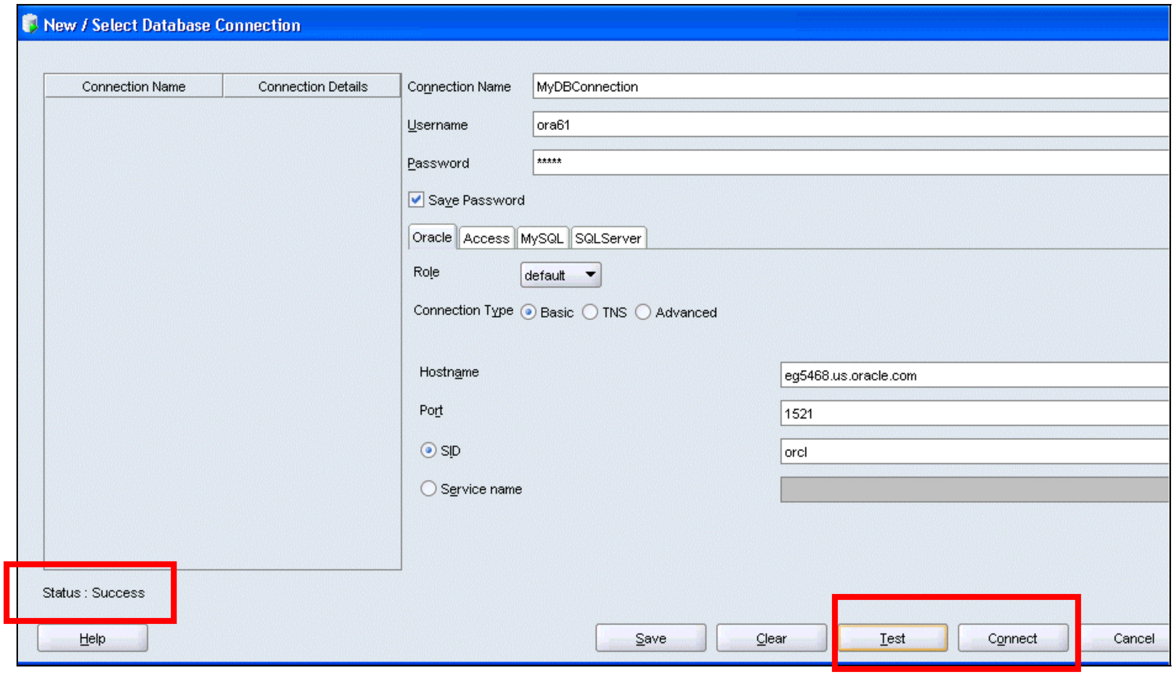
2) Create a database connection using the following information:
   a) Connection Name: `MyDBConnection`
   b) Username: `ora`*`xx`* where *xx* is the number assigned to your PC by the instructor
   c) Password: `ora`*`xx`* where *xx* is the number assigned to your PC by the instructor
   d) Hostname: Enter the host name for your PC
   e) Port: 1521
   f) SID: `ORCL`

Right-click the Connections icon on the Connections tabbed page, and then select the New Database Connection option from the shortcut menu. The New/Select Database Connection window is displayed. Use the preceding information provided to create the new database connection.
Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Substitute the username, password, host name, and service name with the appropriate information as provided by your instructor. The following is a sample of the newly created database connection for student ora61:



Practice 1: Getting Started (continued)

3) Test the new connection. If the Status is Success, connect to the database using this new connection:
   a) Double-click the MyDBConnection icon on the Connections tabbed page.
   b) Click the Test button in the New/Select Database Connection window. If the status is Success, click the Connect button.

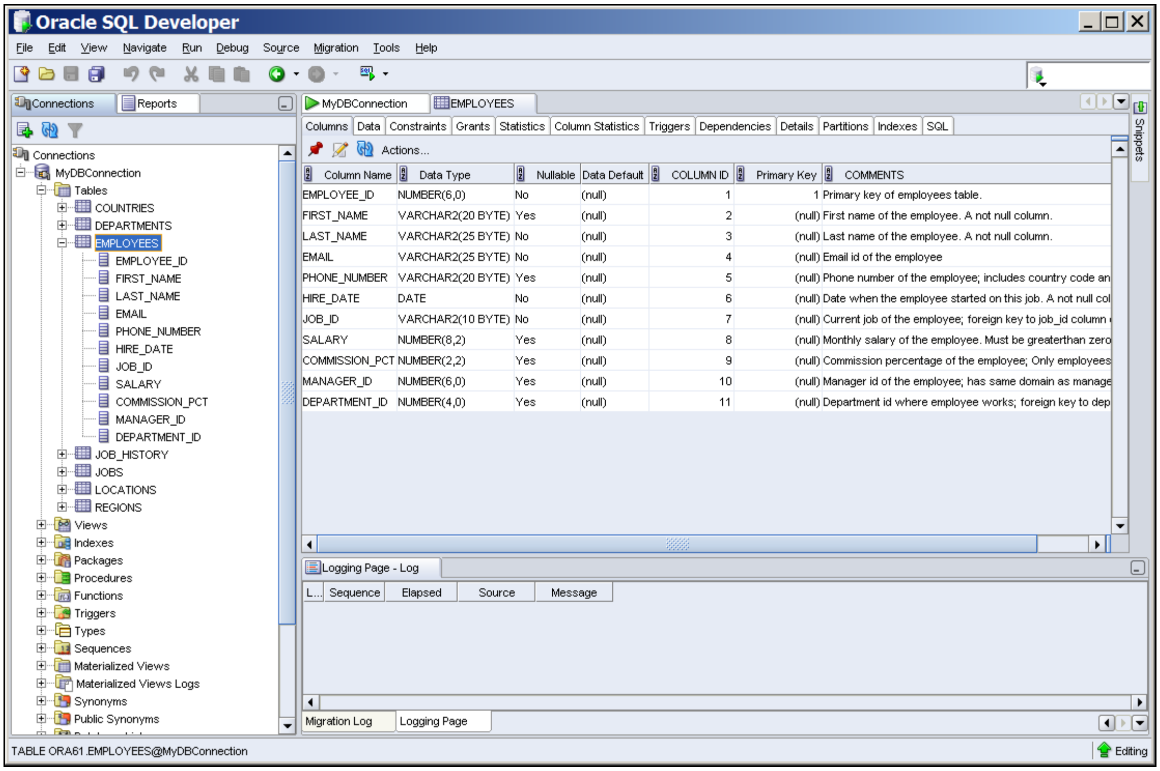


Browsing Your HR Schema Tables

1) Browse the structure of the `EMPLOYEES` table and display its data.
   a) Expand the MyDBConnection connection by clicking the plus sign next to it.
   b) Expand the Tables icon by clicking the plus sign next to it.
   c) Display the structure of the `EMPLOYEES` table.

   Double-click the `EMPLOYEES` table. The `Columns` tab displays the columns in the `EMPLOYEES` table as follows:

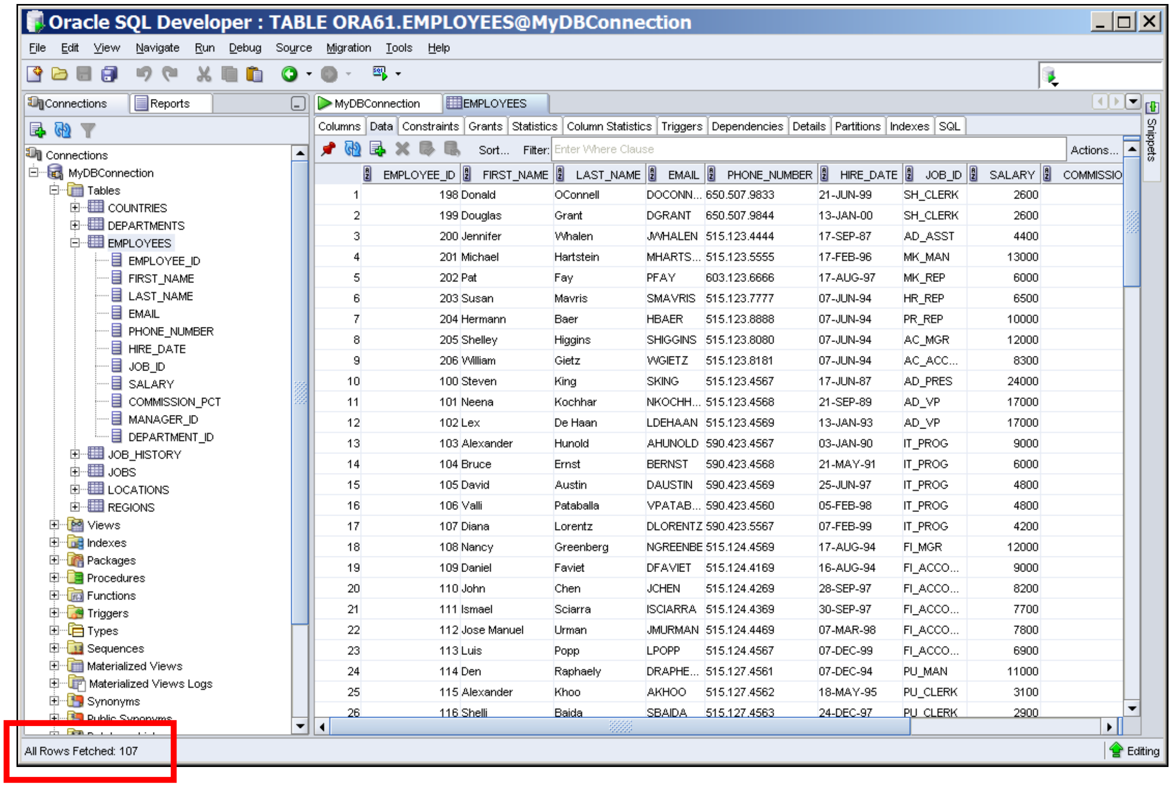
Practice 1: Getting Started (continued)



2) Browse the `EMPLOYEES` table and display its data.

To display the employees' data, click the Data tab. The `EMPLOYEES` table data is displayed as follows:

Practice 1: Getting Started (continued)



3) Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than $10,000. Use both the Execute Statement (F9) and the Run Script icon (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements in the appropriate tabs.

   Note: Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides the description and data for all the tables in the `HR` schema that you will use in this course.

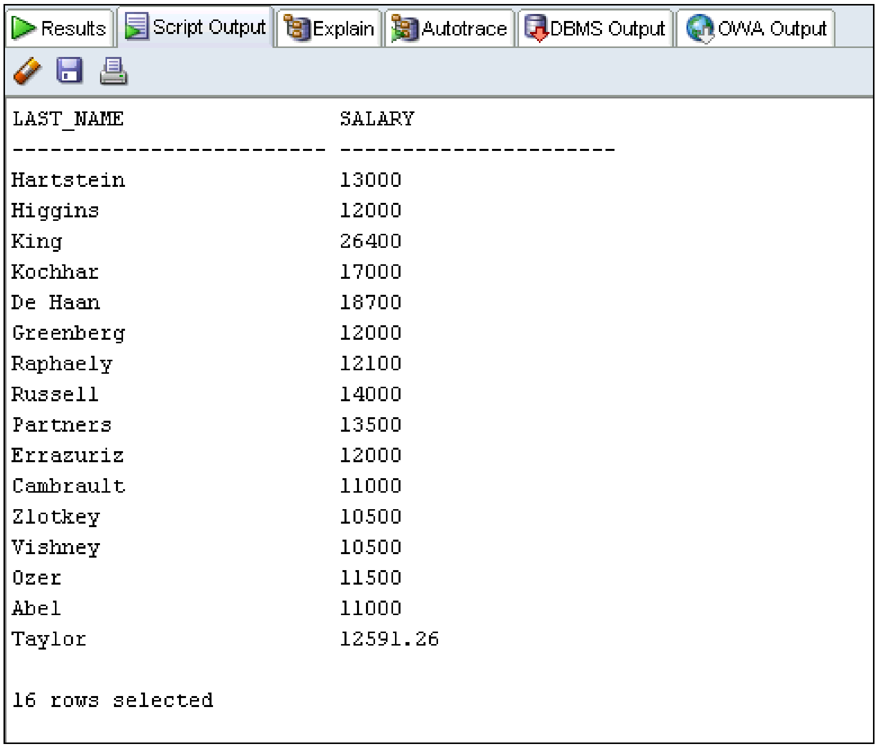
   Display the SQL Worksheet using any of the following two methods:

   1. **Select Tools > SQL Worksheet or click the Open SQL Worksheet icon. The Select Connection window is displayed.**
   2. **Select the new MyDBConnection from the Connection drop-down list (if not already selected), and then click OK.**

   Open the `sol_01_03.sql` file from the `D:\labs\PLPU` folder as follows: Right-click the SQL Worksheet area, and then select Open File. Navigate to the `solns` folder, select the `sol_01_03.sql` file, and then click Open. Click the Execute Statement (F9) icon (while making sure the cursor is on any of the `SELECT` statement lines) on the SQL Worksheet toolbar to execute the statement. The code and the result are displayed as follows:

Practice 1: Getting Started (continued)

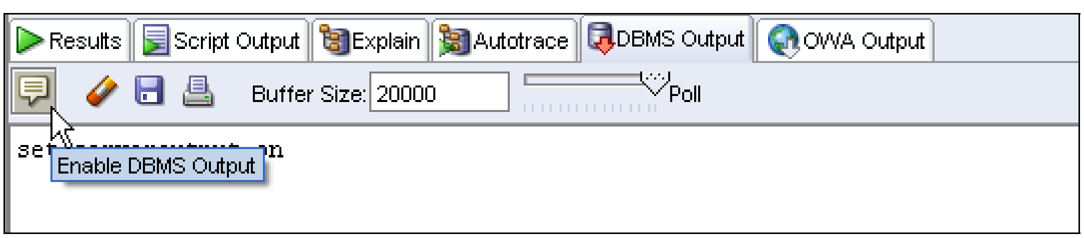
```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE SALARY > 10000;
```



4) Create and execute a simple anonymous block that outputs "Hello World."

a) Enable `SET SERVEROUTPUT ON` to display the output of the `DBMS_OUTPUT` package statements.

Click the DBMS_OUTPUT tab, and then click the Enable DBMS Output icon as follows:

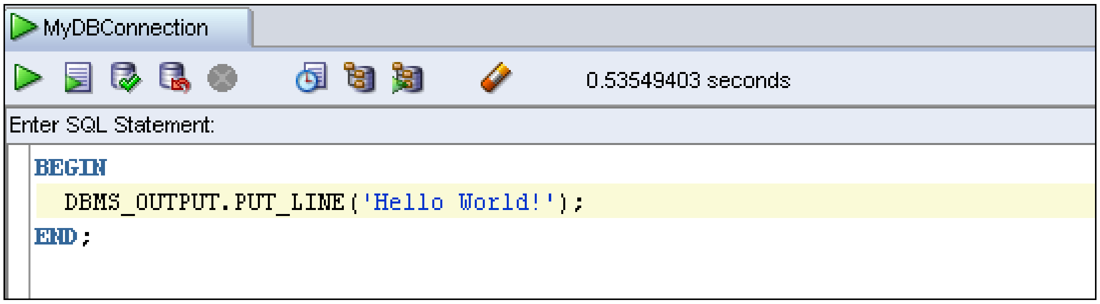


b) Use the SQL Worksheet area to enter the code for your anonymous block.

Enter the following code in the SQL Worksheet area as shown below. Alternatively, open the `sol_01_04.sql` file from the `D:\labs\PLPU` folder as follows: Right-click the SQL Worksheet area, and then select Open

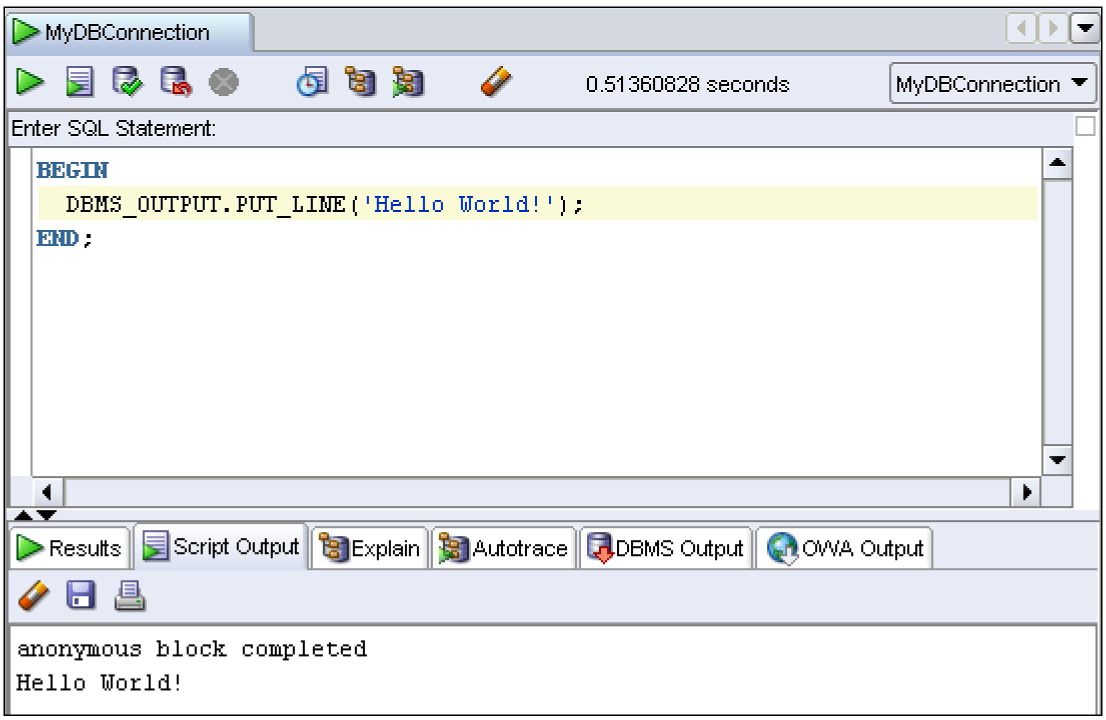
Practice 1: Getting Started (continued)

File. Navigate to the `solns` folder, select the `sol_01_04.sql` file, and then click Open. The code is displayed as follows:



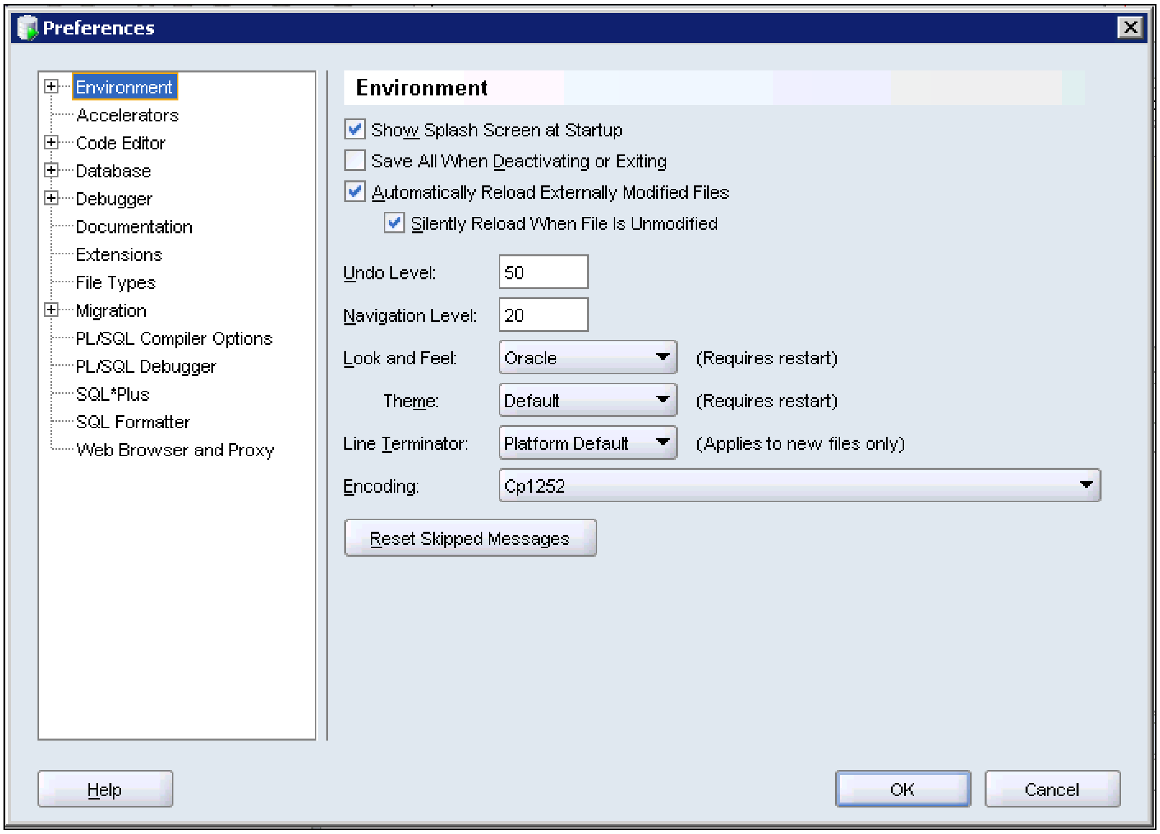
c) Click the Run Script (F5) icon to run the anonymous block.

The Script Output tab displays the output of the anonymous block as follows:



Setting Some SQL Developer Preferences

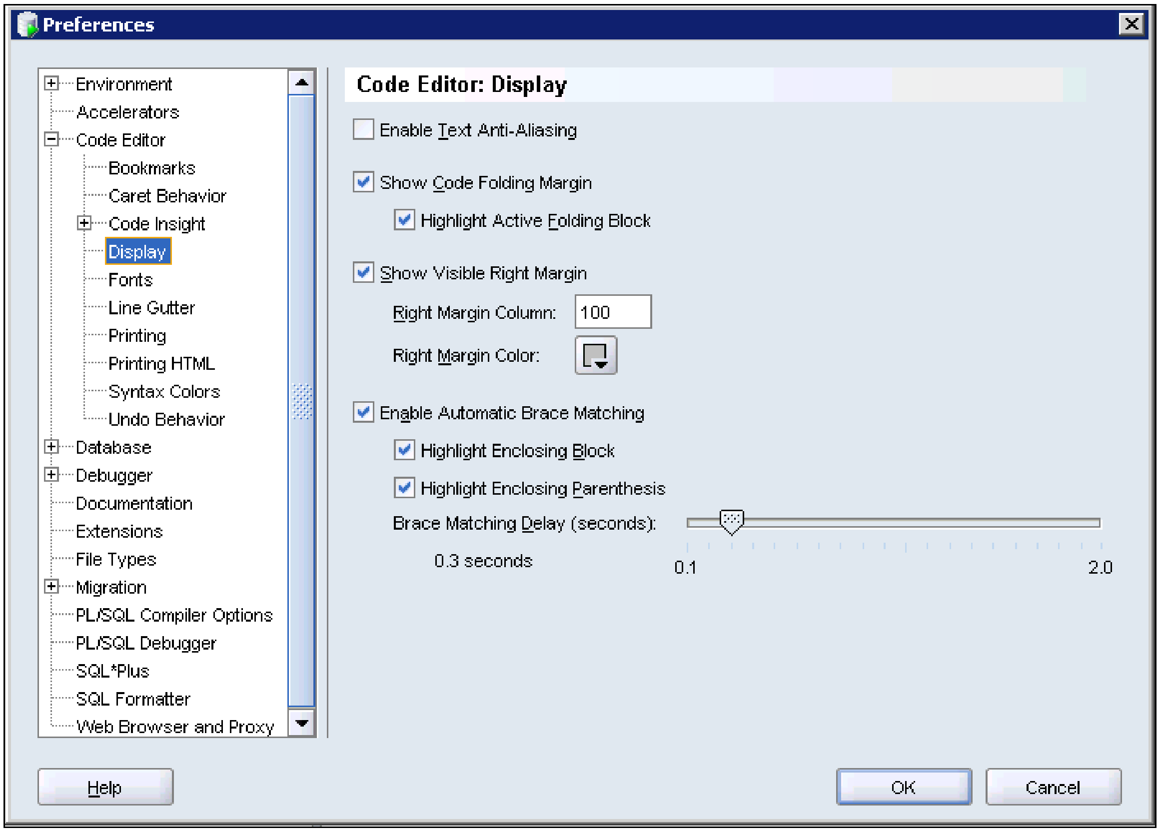
1) In the SQL Developer menu, navigate to Tools > Preferences. The Preferences window is displayed.



2) Expand the Code Editor option, and then click the Display option to display the "Code Editor: Display" section. The "Code Editor: Display" section contains general options for the appearance and behavior of the code editor.

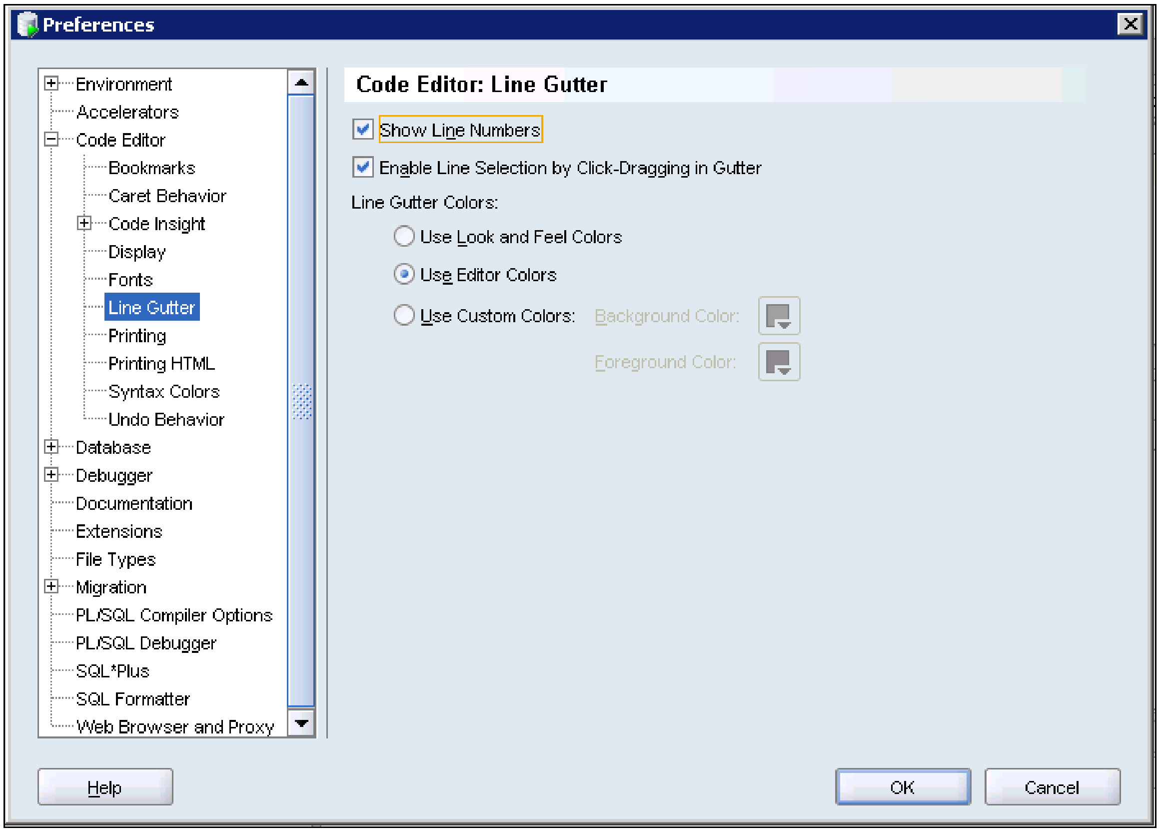
   a) Enter `100` in the Right Margin Column text box in the Show Visible Right Margin section. This renders a right margin that you can set to control the length of lines of code.

Practice 1: Getting Started (continued)



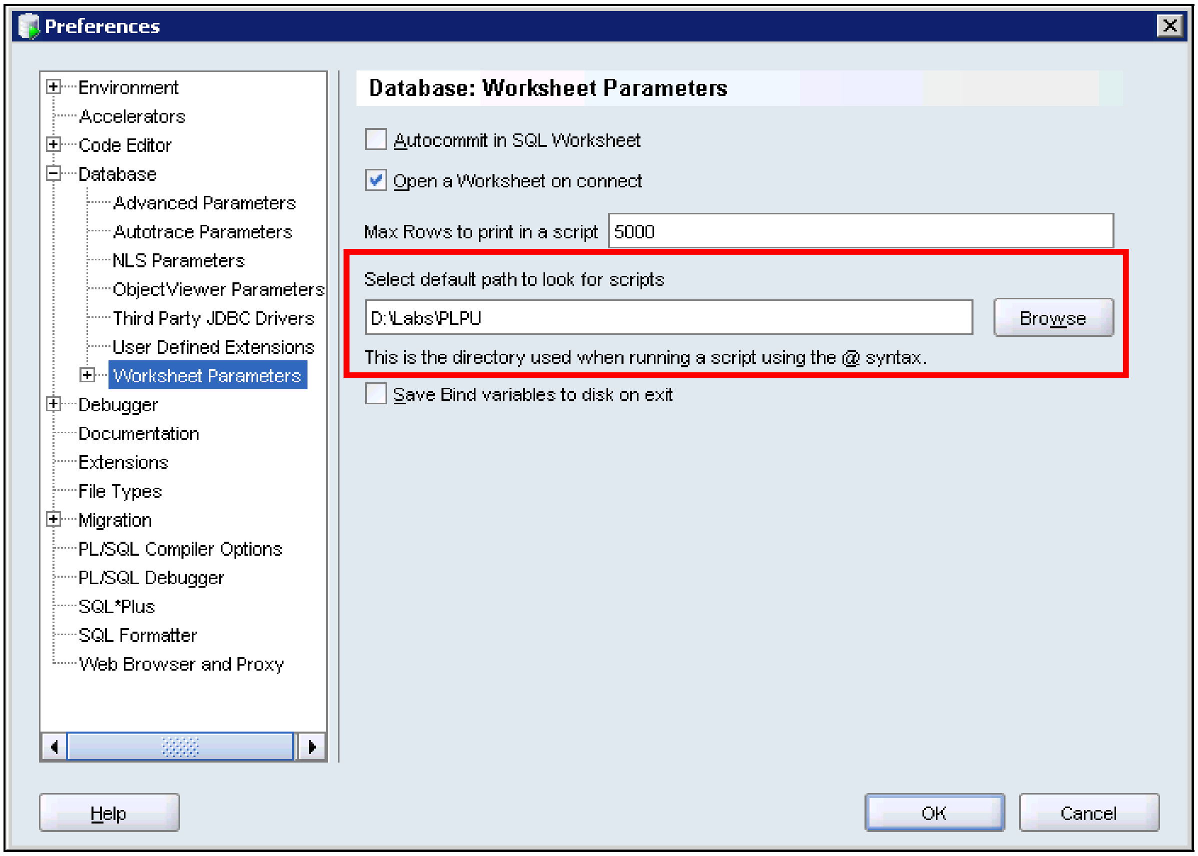
b) Click the Line Gutter option. The Line Gutter option specifies options for the line gutter (left margin of the code editor). Select the Show Line Numbers check box to display the code line numbers.

Practice 1: Getting Started (continued)



3) Click the Worksheet Parameters option under the Database option. In the "Select default path to look for scripts" text box, specify the `D:\labs\PLPU` folder. This folder contains the solutions scripts, code examples scripts, and any labs or demos used in this course.

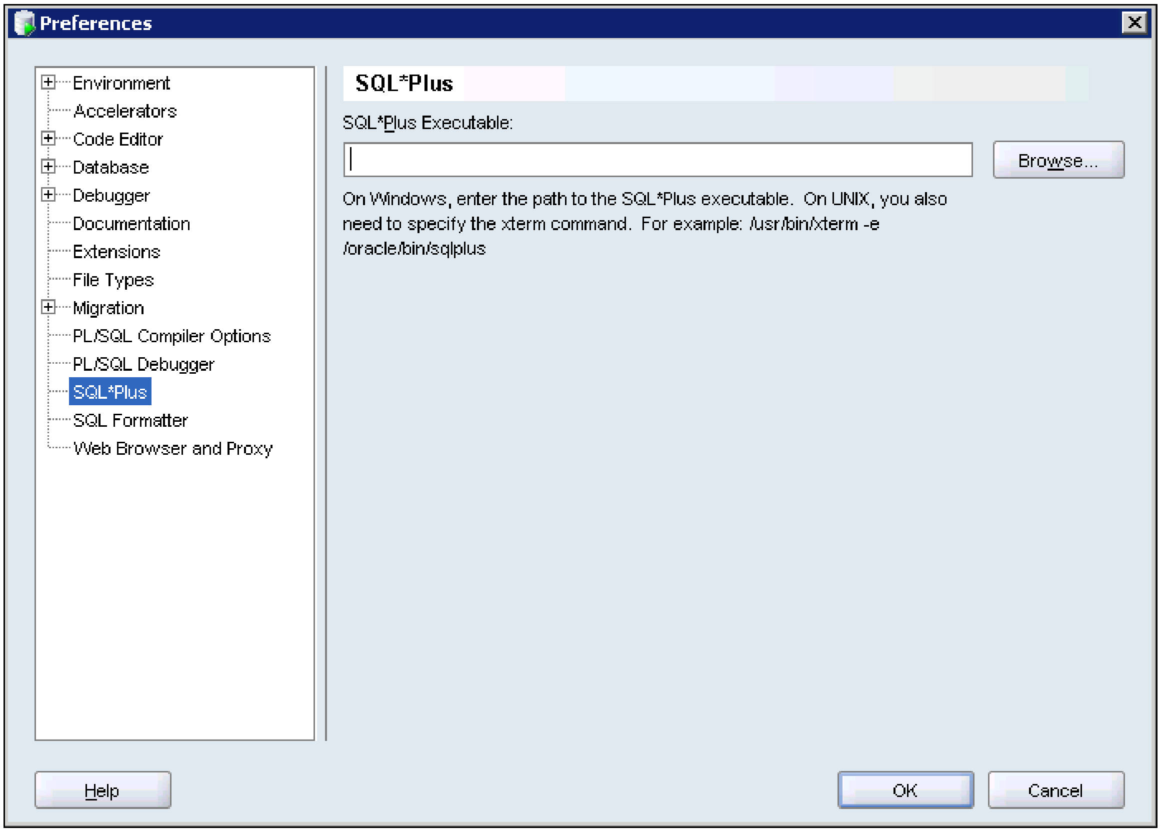
Practice 1: Getting Started (continued)



4) Configure SQL Developer so that you can access SQL*Plus from within SQL Developer.

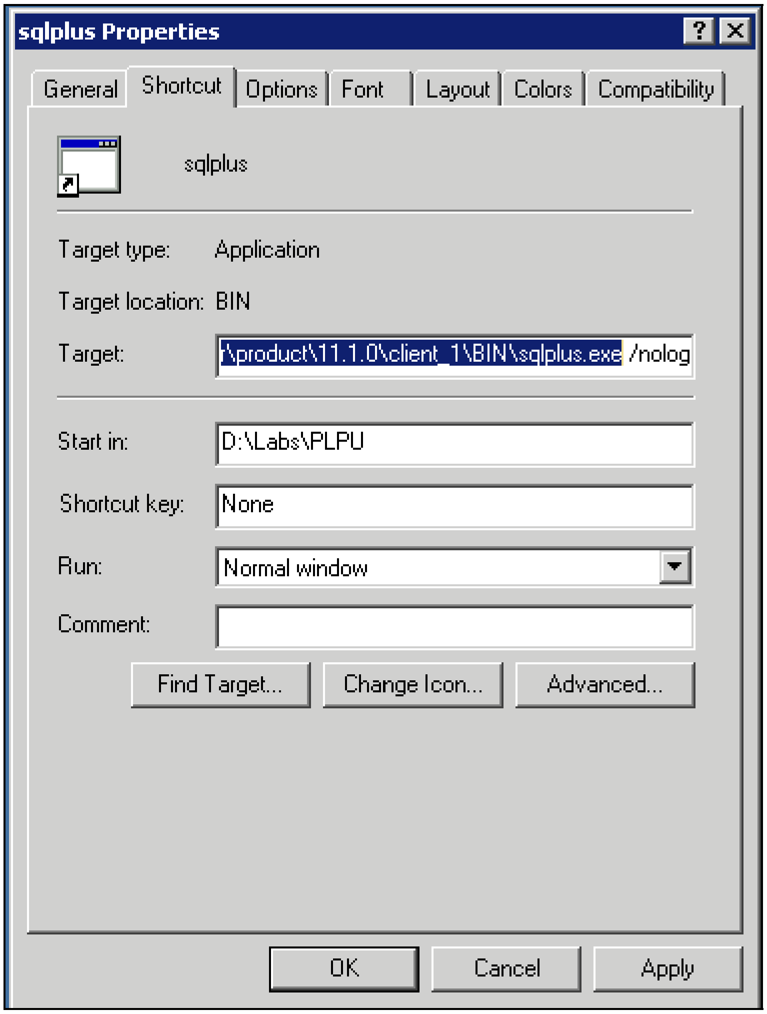
   a) In the Preferences window, click the SQL*Plus option.

Practice 1: Getting Started (continued)



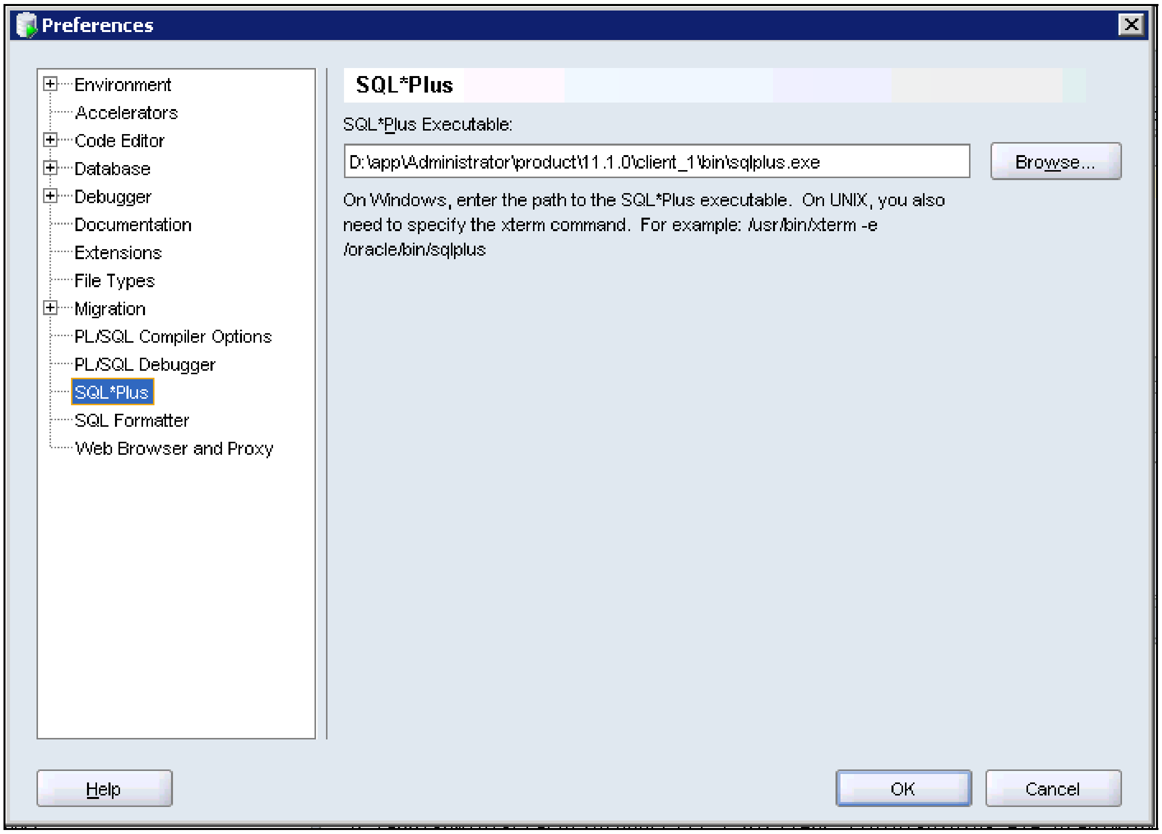
b) In the SQL*Plus Executable text box, enter the path for the SQL*Plus executable. **Note:** To find the path for SQL*Plus: Right-click the SQL*Plus icon on your desktop, select Properties from the shortcut menu, and then copy the SQL*Plus path from the Target text box but do not include the `/nolog` at the end of the Target path.

Practice 1: Getting Started (continued)



c) Paste the SQL*Plus path in the SQL*Plus Executable text box.

Practice 1: Getting Started (continued)



d) Click OK to accept your changes and to exit the Preferences window.

5) Test accessing SQL*Plus from within SQL Developer, and change the default background and text colors.

a) Click your Database Connection name in the Connections tab.

b) Select SQL*Plus from the Tools menu. The SQL*Plus command window is displayed.

c) Enter your password.

d) Change the default screen background and text colors. Click the C:\ icon on the SQL*Plus command window title bar, and then select Properties from the pop-up menu.

e) In the Colors tab, select the Screen Background option, and then click the white color sample from the available color palettes.

f) Select the Screen Text option, and then click the black color sample from the available color palettes.

g) Click OK. The Apply Properties window is displayed. Select the “Save properties for future windows with same title” option, and then click OK.

h) Issue the following simple SQL command to test SQL*Plus:

```
SELECT *
FROM employees;
```

Practice 1: Getting Started (continued)

6) Familiarize yourself with the labs folder on the D:\ drive:
   a) Right-click the SQL Worksheet area, and then select Open File from the shortcut menu. The Open window is displayed.
   b) Ensure that the path that you set in a previous step is the default path that is displayed in the Open window.
   c) How many subfolders do you see in the labs folder?
   d) Navigate through the folders, and open a script file without executing the code.
   e) Clear the displayed code in the SQL Worksheet area.

Accessing the Oracle Database 11*g* Release 1 Online Documentation Library

1) Access the Oracle Database 11*g* Release 1 documentation Web page at: http://www.oracle.com/pls/db111/homepage
2) Bookmark the page for easier future access.
3) Display the complete list of books available for Oracle Database 11*g* Release 1.
4) Make a note of the following documentation references that you will use in this course as needed:
   a) Advanced Application Developer's Guide
   b) New Features Guide
   c) PL/SQL Language Reference
   d) Oracle Database Reference
   e) Oracle Database Concepts
   f) SQL Developer User's Guide
   g) SQL Language Reference Guide
   *h) SQL*Plus User's Guide and Reference*

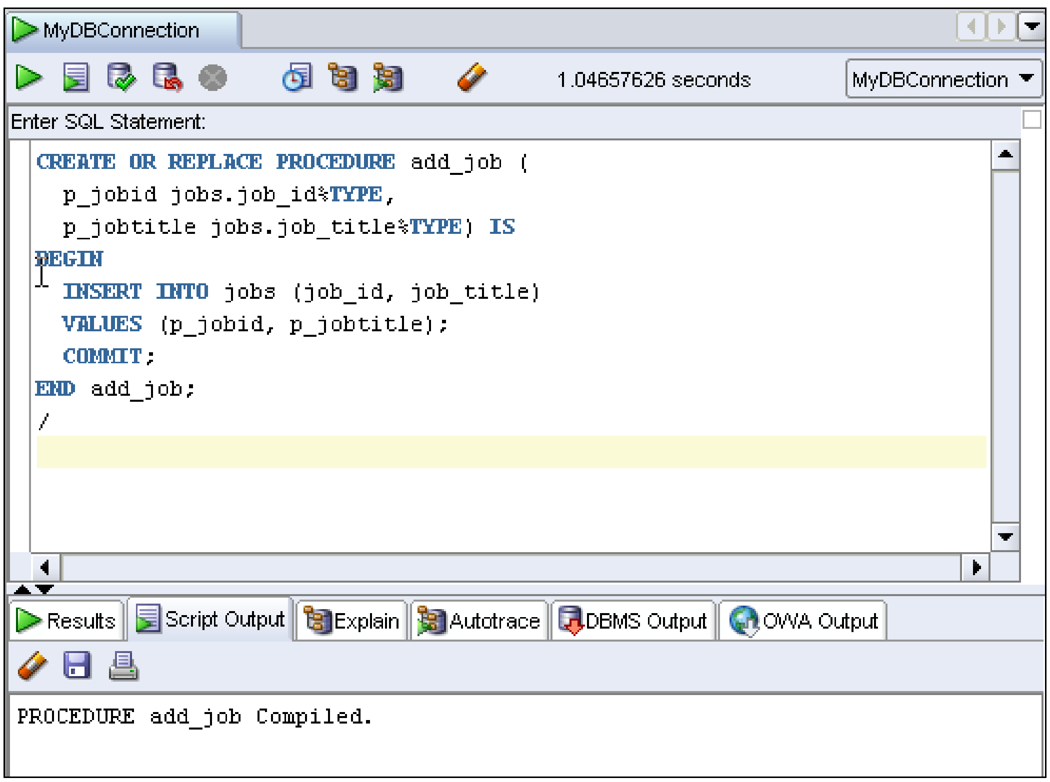
Practice 2: Creating, Compiling, and Calling Procedures

In this practice, you create, compile, and invoke procedures that issue DML and query commands. You also learn how to handle exceptions in procedures.

1) Create, compile, and invoke the `ADD_JOB` procedure and review the results.

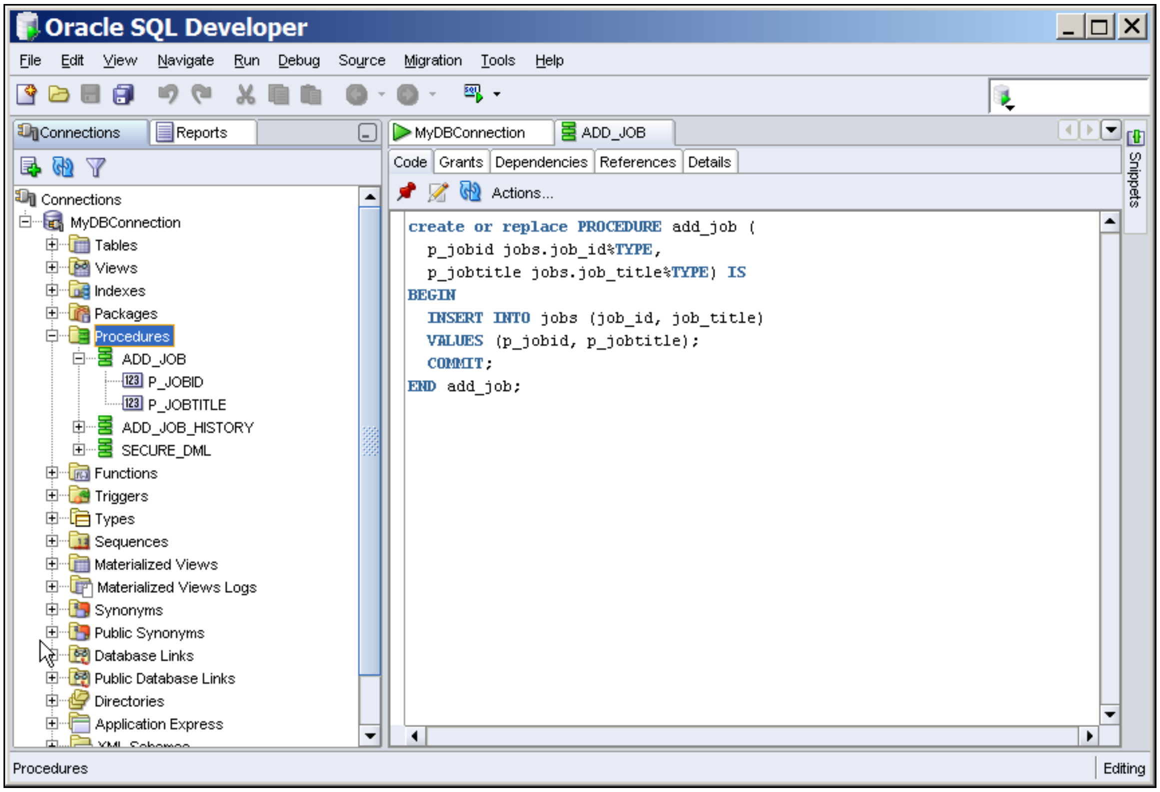
   a) Create a procedure called `ADD_JOB` to insert a new job into the `JOBS` table. Provide the ID and job title using two parameters.
   Note: You can create the procedure (and other objects) by entering the code in the SQL Worksheet area, and then click the Run Script (F5) icon. This creates and compiles the procedure. To find out whether or not the procedure has any errors, click the procedure name in the procedure node, and then select Compile from the pop-up menu.

   Open the `sol_02_01_a.sql` file from the `D:\labs\PLPU` folder as follows: Right-click the SQL Worksheet area, and then select Open File. Navigate to the `solns` folder, select the `sol_02_01_a.sql` file, and then click Open. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the procedure. The code and the result are displayed as follows:



Practice 2: Creating, Compiling, and Calling Procedures (continued)

To view the newly created procedure, click the Procedures node in the Object Navigator, right-click, and then select Refresh from the shortcut menu. The new procedure is displayed as follows:

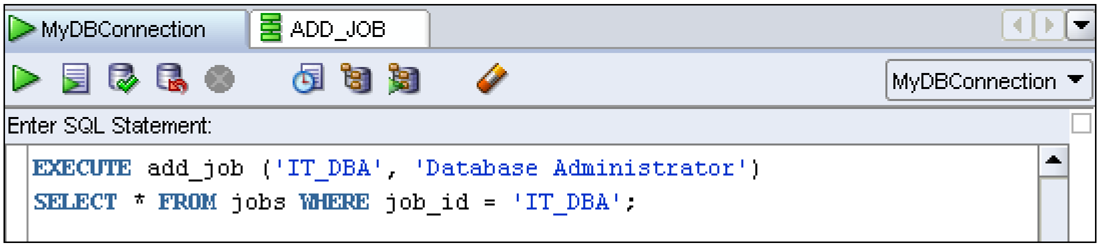


b) Compile the code, and then invoke the procedure with `IT_DBA` as the job ID and `Database Administrator` as the job title. Query the `JOBS` table and view the results.

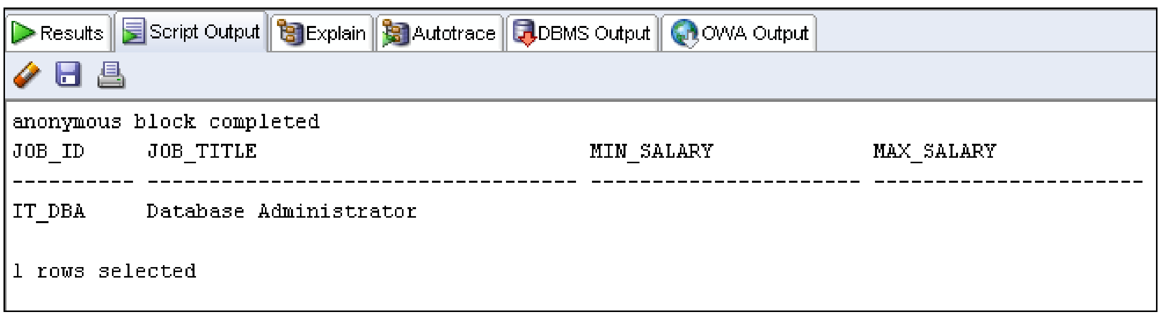
Right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the procedure's name in the Object Navigator, and then select Compile from the shortcut menu. The procedure is compiled.

Practice 2: Creating, Compiling, and Calling Procedures (continued)

To invoke the procedure and then query the `JOBS` table, load the `sol_02_01_b.sql` file from the `D:\labs\PLPU\solns` folder. The code is displayed in the SQL Worksheet as follows:

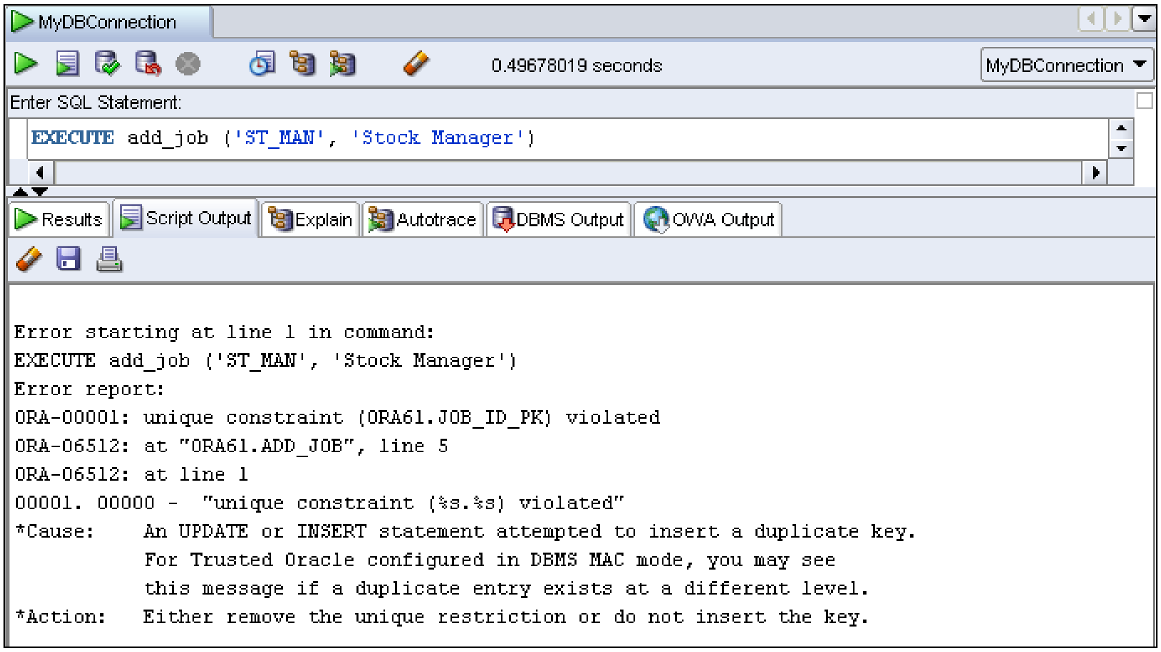


To invoke the procedure, click the Run Script (F5) icon on the SQL Worksheet toolbar. The results are displayed as follows:



c) Invoke your procedure again, passing a job ID of `ST_MAN` and a job title of `Stock Manager`. What happens and why?

An exception occurs because there is a Unique key integrity constraint on the `JOB_ID` column.

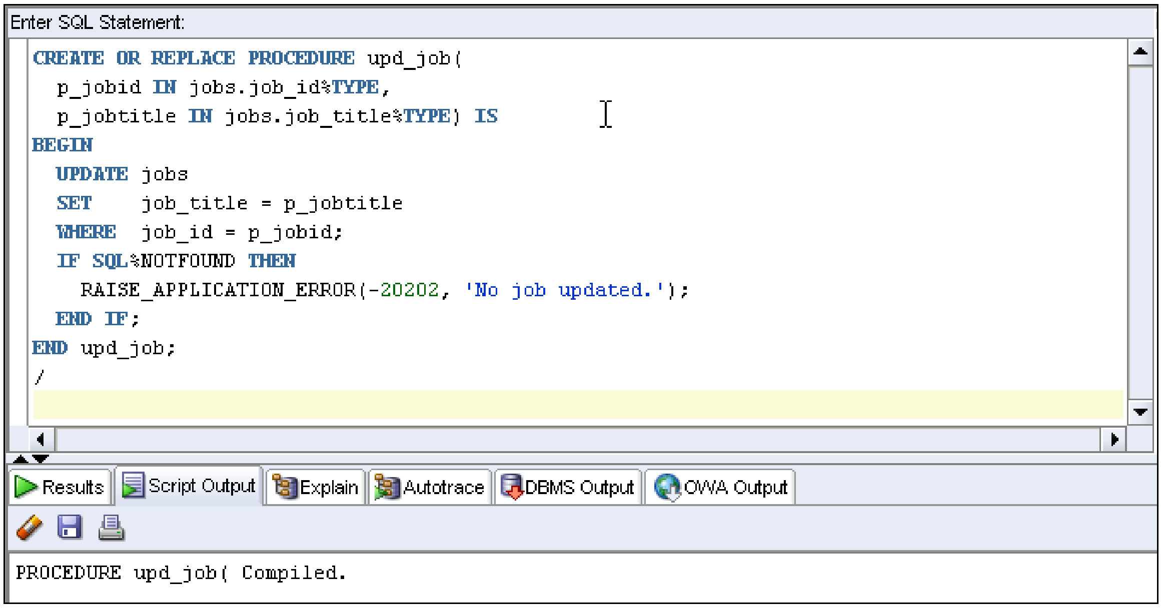


2) Create a procedure called `UPD_JOB` to modify a job in the `JOBS` table.

a) Create a procedure called `UPD_JOB` to update the job title. Provide the job ID and a new title using two parameters. Include the necessary exception handling if no update occurs.

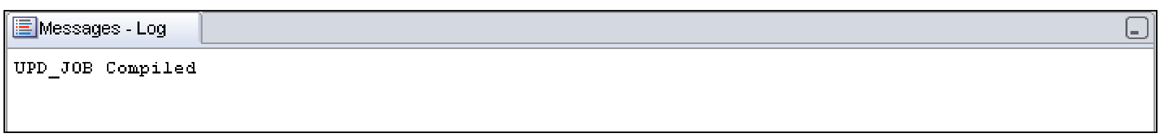
Open the `sol_02_02_a.sql` file from the `D:\labs\PLPU\solns` folder as follows: Right-click the SQL Worksheet area, and then select Open File. Navigate to the `solns` folder, select the `sol_02_02_a.sql` file, and then click Open. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the procedure. The code is displayed in the SQL Worksheet area as follows:

Practice 2: Creating, Compiling, and Calling Procedures (continued)



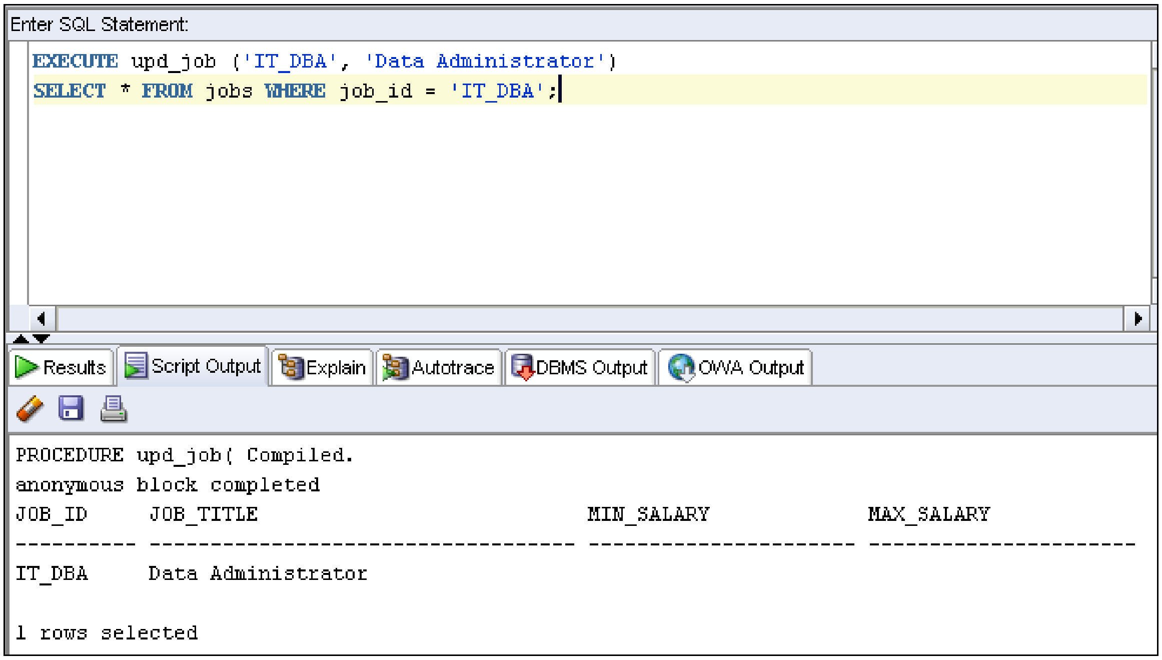
b) Compile the procedure. Invoke the procedure to change the job title of the job ID `IT_DBA` to `Data Administrator`. Query the `JOBS` table and view the results.

Right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the procedure's name in the Object Navigator, and then select Compile from the shortcut menu. The procedure is compiled.

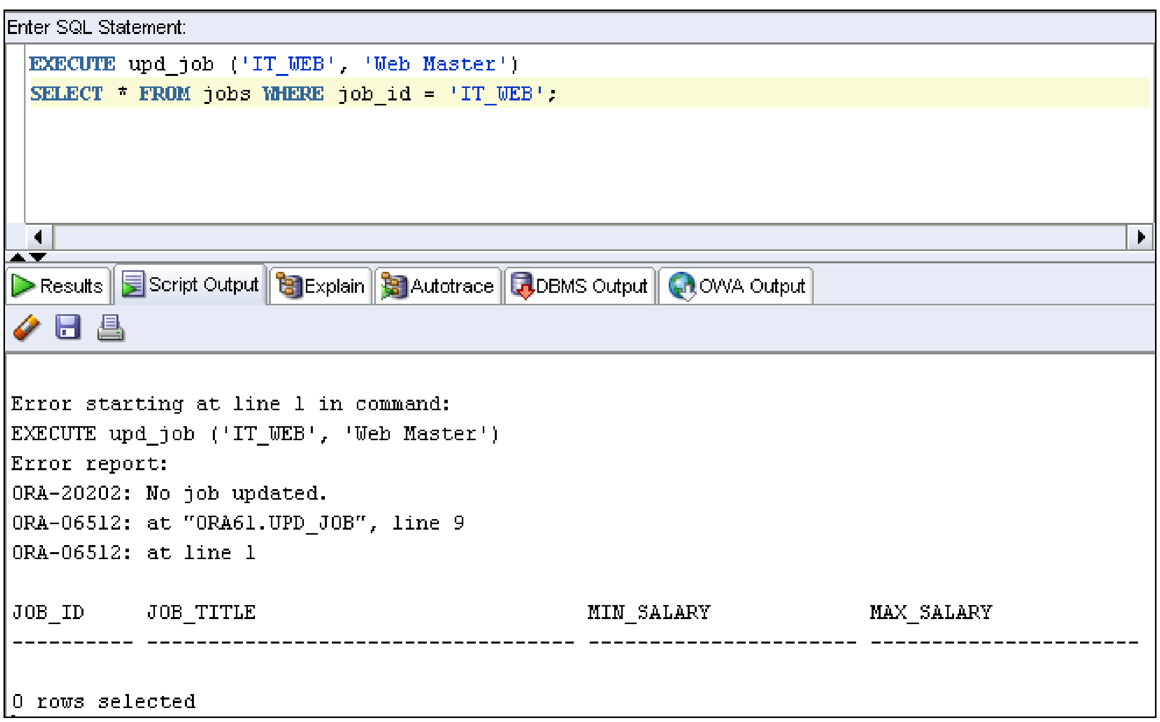


To invoke the procedure and then query the `JOBS` table, load the `sol_02_02_b.sql` file from the `D:\labs\PLPU\solns` folder. The code is displayed in the SQL Worksheet. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

Practice 2: Creating, Compiling, and Calling Procedures (continued)



c) Test the exception-handling section of the procedure by trying to update a job that does not exist. You can use the job ID `IT_WEB` and the job title `Web Master`.

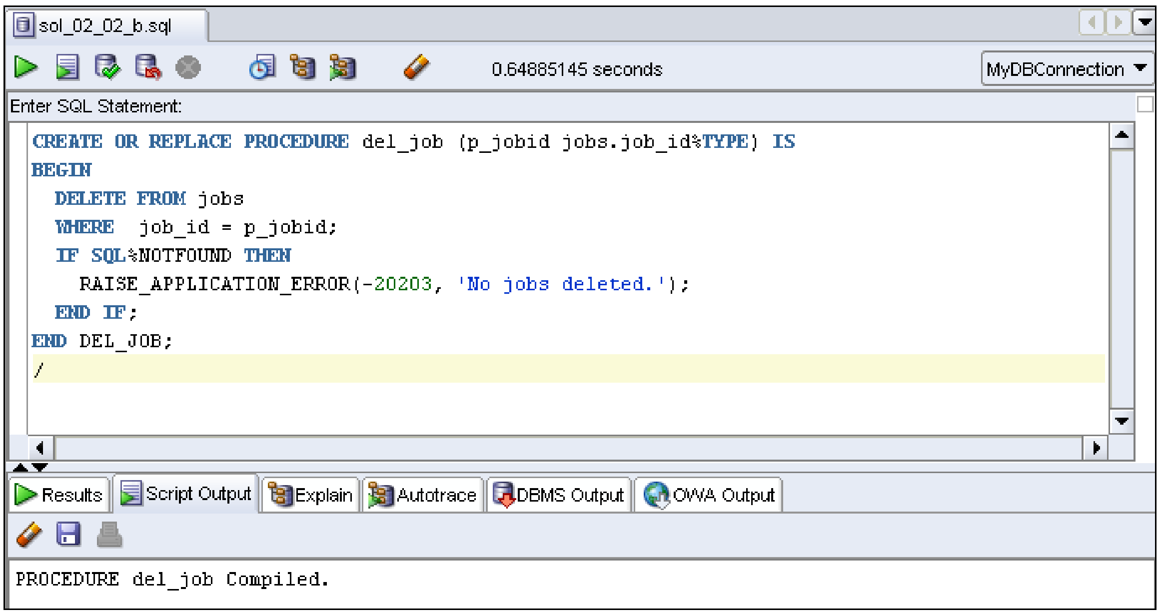


3) Create a procedure called `DEL_JOB` to delete a job from the `JOBS` table.

Practice 2: Creating, Compiling, and Calling Procedures (continued)

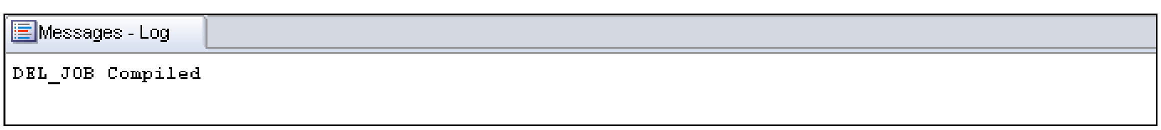
a) Create a procedure called `DEL_JOB` to delete a job. Include the necessary exception-handling code if no job is deleted.

Open the `sol_02_03_a.sql` file from the `D:\labs\PLPU` folder as follows: Right-click the SQL Worksheet area, and then select Open File. Navigate to the `solns` folder, select the `sol_02_03_a.sql` file, and then click OK. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the procedure. The code and the result are displayed as follows:



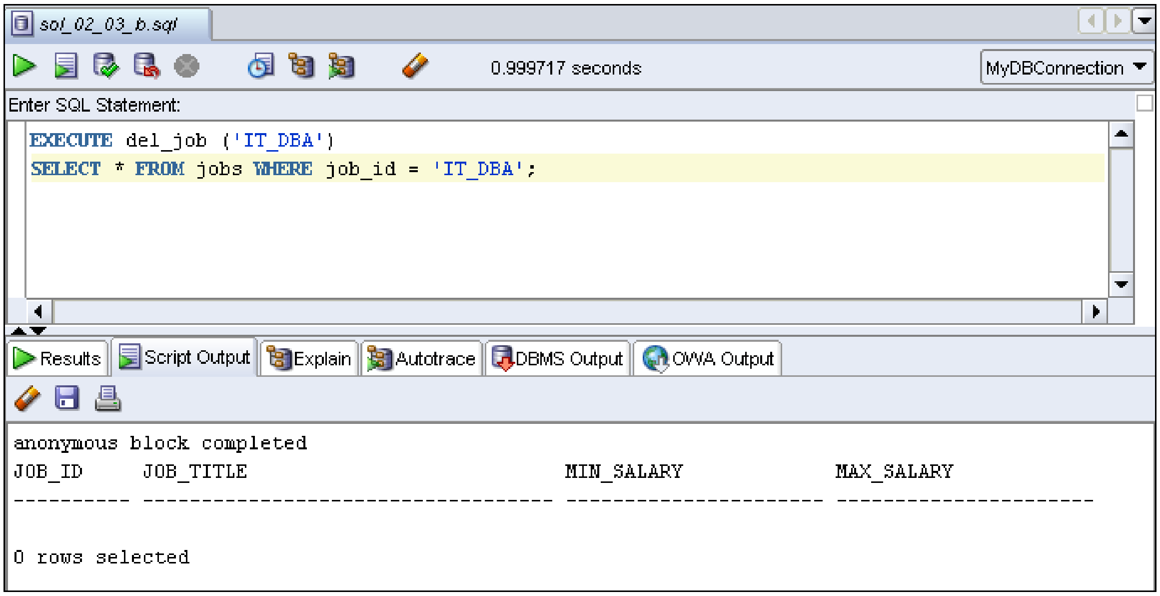
b) Compile the code; invoke the procedure using the job ID `IT_DBA`. Query the `JOBS` table and view the results.

If the newly created procedure is not displayed in the Object Navigator, right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the procedure's name in the Object Navigator, and then select Compile from the shortcut menu. The procedure is compiled.



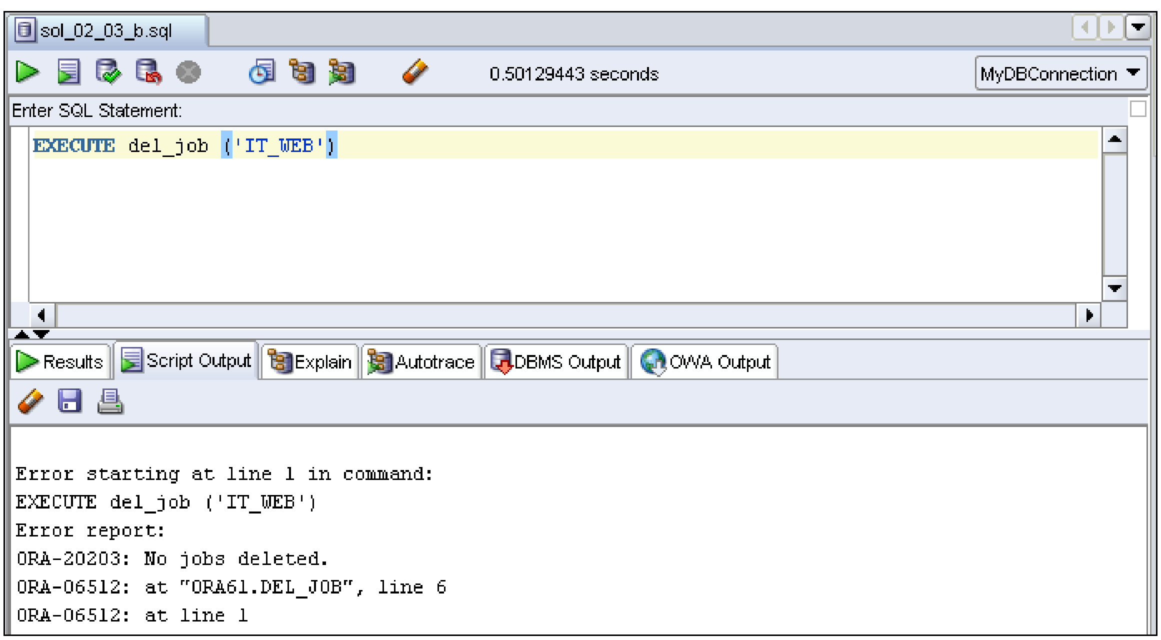
To invoke the procedure and then query the `JOBS` table, load the `sol_02_03_b.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

Practice 2: Creating, Compiling, and Calling Procedures (continued)



c) Test the exception-handling section of the procedure by trying to delete a job that does not exist. Use `IT_WEB` as the job ID. You should get the message that you included in the exception-handling section of the procedure as the output.

To invoke the procedure and then query the `JOBS` table, load the `sol_02_03_c.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

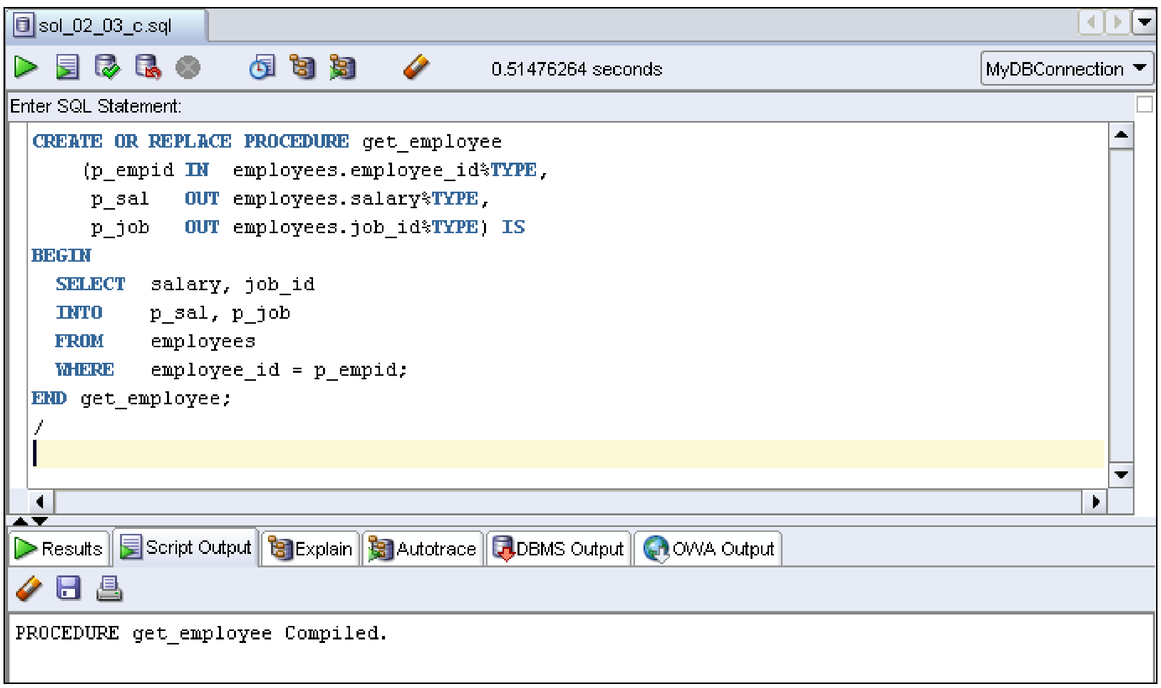


Practice 2: Creating, Compiling, and Calling Procedures (continued)

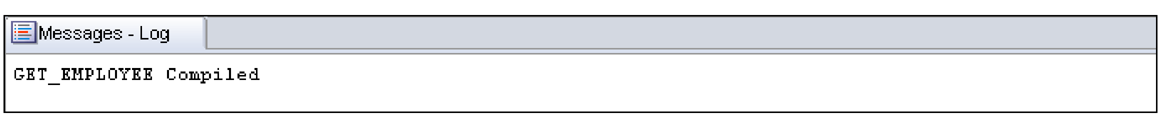
4) Create a procedure called `GET_EMPLOYEE` to query the `EMPLOYEES` table, retrieving the salary and job ID for an employee when provided with the employee ID.

   a) Create a procedure that returns a value from the `SALARY` and `JOB_ID` columns for a specified employee ID. Compile the code and remove syntax errors, if any.

   Open the `sol_02_04_a.sql` file from the `D:\labs\PLPU\solns` folder as follows: Right-click the SQL Worksheet area, and then select Open File. Navigate to the `solns` folder, select the `sol_02_04_a.sql` file, and then click OK. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the procedure. The code and the result are displayed as follows:



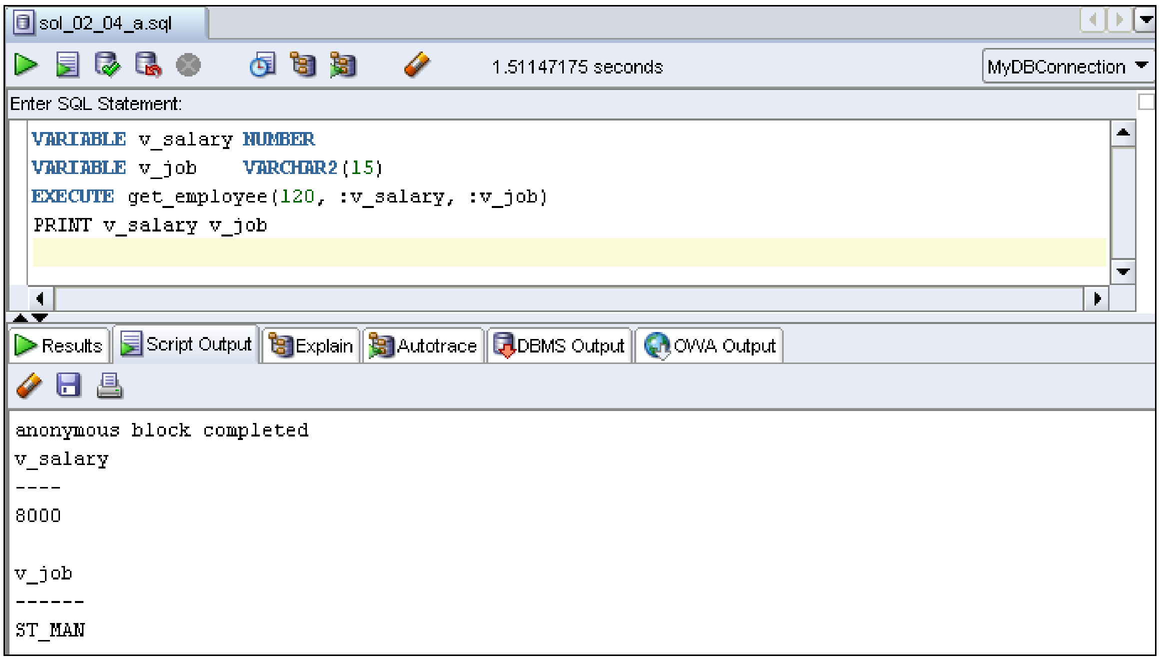
   If the newly created procedure is not displayed in the Object Navigator, right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the procedure's name in the Object Navigator, and then select Compile from the shortcut menu. The procedure is compiled.



Practice 2: Creating, Compiling, and Calling Procedures (continued)

b) Execute the procedure using host variables for the two `OUT` parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

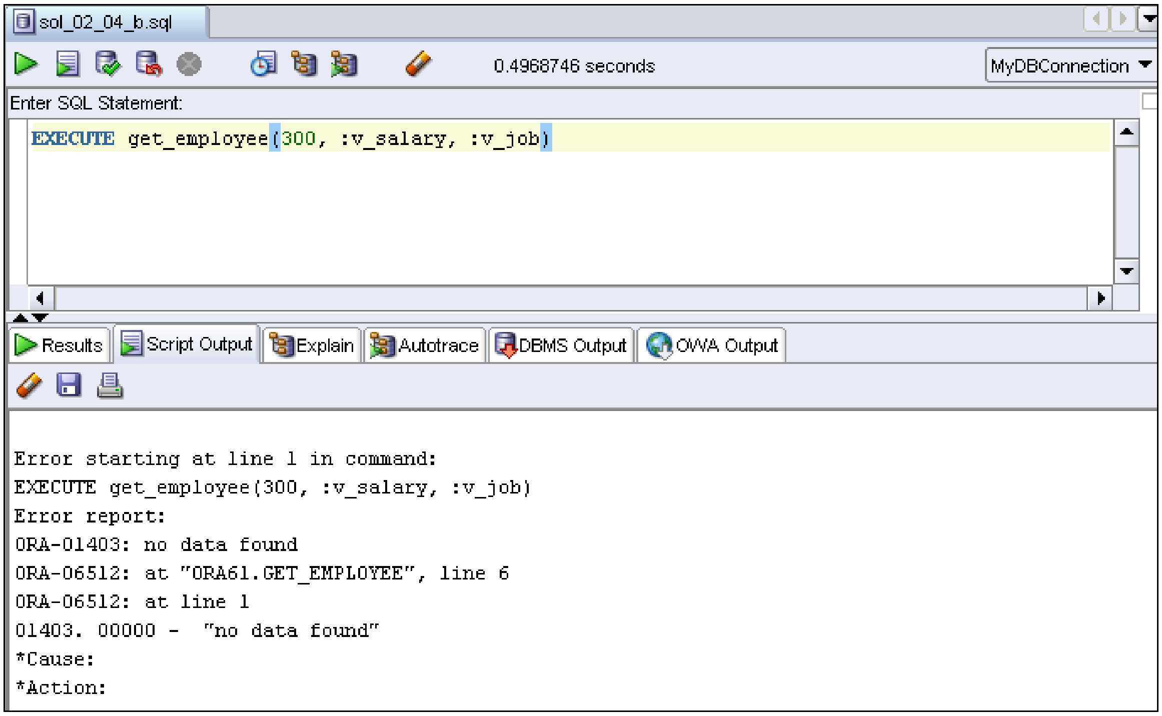
Open the `sol_02_04_b.sql` file from the `D:\labs\PLPU\solns` folder as follows: Right-click the SQL Worksheet area, and then select Open File. Navigate to the `solns` folder, select the `sol_02_04_b.sql` file, and then click OK. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:



c) Invoke the procedure again, passing an `EMPLOYEE_ID` of `300`. What happens and why?

There is no employee in the `EMPLOYEES` table with an `EMPLOYEE_ID` of 300. The `SELECT` statement retrieved no data from the database, resulting in a fatal PL/SQL error: `NO_DATA_FOUND` as follows:

Practice 2: Creating, Compiling, and Calling Procedures (continued)



Practice 3: Creating Functions

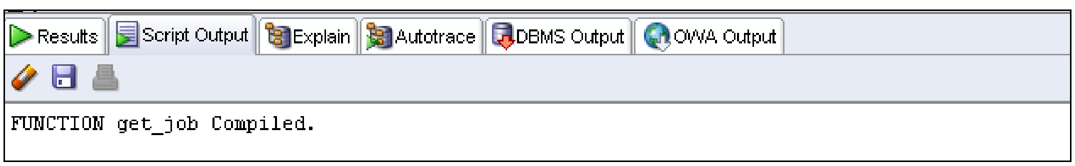
In this practice/task, you create and invoke stored functions.

1) Create and invoke the `GET_JOB` function to return a job title.

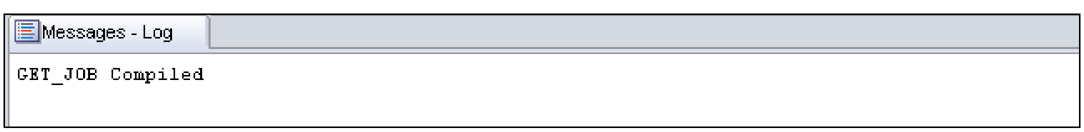
   a) Create and compile a function called `GET_JOB` to return a job title.

   Open the `sol_03_1_a.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

```
CREATE OR REPLACE FUNCTION get_job (p_jobid IN
jobs.job_id%type)
 RETURN jobs.job_title%type IS
  v_title jobs.job_title%type;
BEGIN
  SELECT job_title
  INTO v_title
  FROM jobs
  WHERE job_id = p_jobid;
  RETURN v_title;
END get_job;
/
```



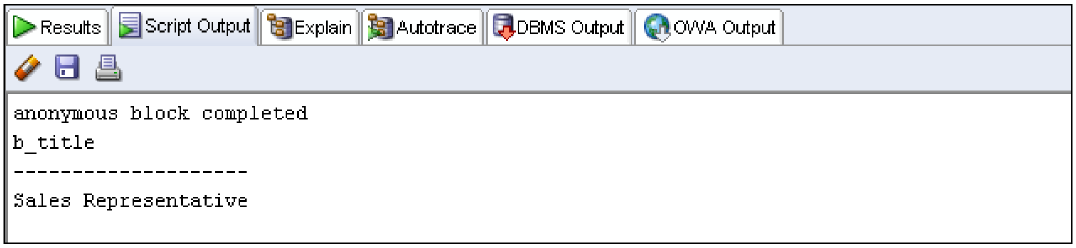
   If the newly created function is not displayed in the Object Navigator, right-click the Functions node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the function's name in the Object Navigator, and then select Compile from the shortcut menu. The function is compiled.



   b) Create a `VARCHAR2` host variable called `b_title`, allowing a length of 35 characters. Invoke the function with job ID `SA_REP` to return the value in the host variable, and then print the host variable to view the result.

   Open the `sol_03_01_b.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

```
VARIABLE b_title VARCHAR2(35)
EXECUTE :b_title := get_job ('SA_REP');
PRINT b_title
```



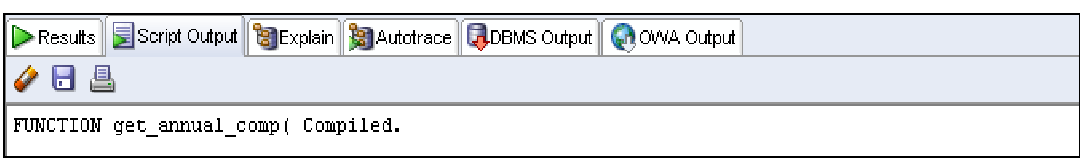
2) Create a function called GET_ANNUAL_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.

   a) Create the GET_ANNUAL_COMP function, which accepts parameter values for the monthly salary and commission. Either or both values passed can be NULL, but the function should still return a non-NULL annual salary. Use the following basic formula to calculate the annual salary:

      (salary*12) + (commission_pct*salary*12)

   Open the sol_03_02_a.sql file from the D:\labs\PLPU\solns folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

```
CREATE OR REPLACE FUNCTION get_annual_comp(
  p_sal  IN employees.salary%TYPE,
  p_comm IN employees.commission_pct%TYPE)
 RETURN NUMBER IS
BEGIN
  RETURN (NVL(p_sal,0) * 12 + (NVL(p_comm,0) * nvl(p_sal,0)
* 12));
END get_annual_comp;
/
```



   If the newly created function is not displayed in the Object Navigator, right-click the Functions node in the Object Navigator, and then select Refresh from the shortcut menu. To compile the function, right-click the function's name, and then select Compile from the shortcut menu.

Practice 3: Creating Functions (continued)

```
Messages - Log
GET_ANNUAL_COMP Compiled
```

b) Use the function in a SELECT statement against the EMPLOYEES table for employees in department 30.

Open the `sol_03_02_b.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

```
SELECT employee_id, last_name,
       get_annual_comp(salary,commission_pct) "Annual
Compensation"
FROM   employees
WHERE department_id=30
/
```

```
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

EMPLOYEE_ID            LAST_NAME                 Annual Compensation
---------------------- ------------------------- ----------------------
114                    Raphaely                  132000
115                    Khoo                      37200
116                    Baida                     34800
117                    Tobias                    33600
118                    Himuro                    31200
119                    Colmenares                30000

6 rows selected
```

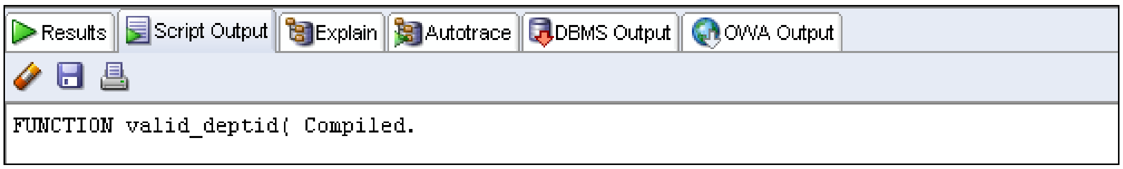
3) Create a procedure, ADD_EMPLOYEE, to insert a new employee into the EMPLOYEES table. The procedure should call a VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.

a) Create a function called VALID_DEPTID to validate a specified department ID and return a BOOLEAN value of TRUE if the department exists.

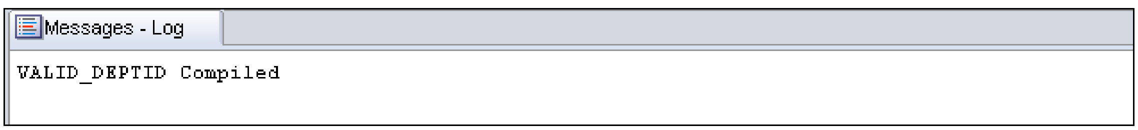
Open the `sol_03_03_a.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

```
CREATE OR REPLACE FUNCTION valid_deptid(
  p_deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN IS
  v_dummy  PLS_INTEGER;

BEGIN
  SELECT  1
```

```
  INTO    v_dummy
  FROM    departments
  WHERE   department_id = p_deptid;
  RETURN  TRUE;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;
/
```



If the newly created function is not displayed in the Object Navigator, right-click the Functions node in the Object Navigator, and then select Refresh from the shortcut menu. To compile the function, right-click the function's name, and then select Compile from the shortcut menu.



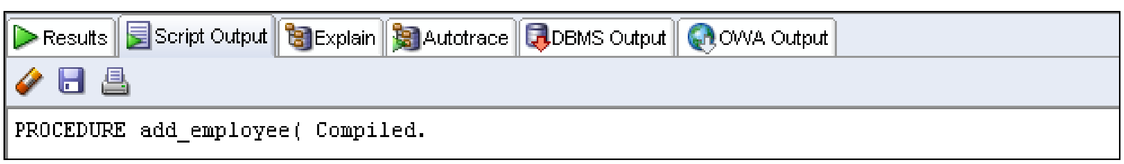
b) Create the `ADD_EMPLOYEE` procedure to add an employee to the `EMPLOYEES` table. The row should be added to the `EMPLOYEES` table if the `VALID_DEPTID` function returns `TRUE`; otherwise, alert the user with an appropriate message. Provide the following parameters:

- `first_name`
- `last_name`
- `email`
- `job`: Use `'SA_REP'` as the default.
- `mgr`: Use 145 as the default.
- `sal`: Use 1000 as the default.
- `comm`: Use 0 as the default.
- `deptid`: Use 30 as the default.
- Use the `EMPLOYEES_SEQ` sequence to set the `employee_id` column.
- Set the `hire_date` column to `TRUNC(SYSDATE)`.

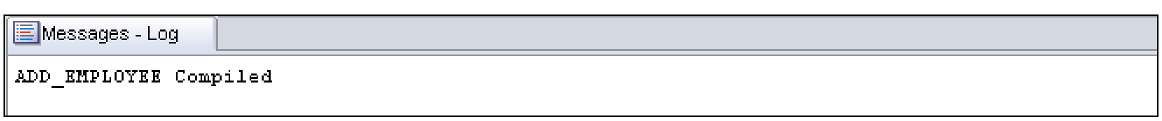
Practice 3: Creating Functions (continued)

Open the `sol_03_03_b.sql` file from the `D:\labs\PLPU\solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the procedure. The code and the result are displayed as follows:

```
CREATE OR REPLACE PROCEDURE add_employee(
   p_first_name employees.first_name%TYPE,
   p_last_name  employees.last_name%TYPE,
   p_email      employees.email%TYPE,
   p_job        employees.job_id%TYPE        DEFAULT 'SA_REP',
   p_mgr        employees.manager_id%TYPE     DEFAULT 145,
   p_sal        employees.salary%TYPE         DEFAULT 1000,
   p_comm       employees.commission_pct%TYPE DEFAULT 0,
   p_deptid     employees.department_id%TYPE  DEFAULT 30) IS
BEGIN
 IF valid_deptid(p_deptid) THEN
   INSERT INTO employees(employee_id, first_name, last_name,
email,
     job_id, manager_id, hire_date, salary, commission_pct,
department_id)
   VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
p_email,
     p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm, p_deptid);
 ELSE
   RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
 END IF;
END add_employee;
/
```



If the newly created procedure is not displayed in the Object Navigator, right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. To compile the procedure, right-click the procedure's name, and then select Compile from the shortcut menu.



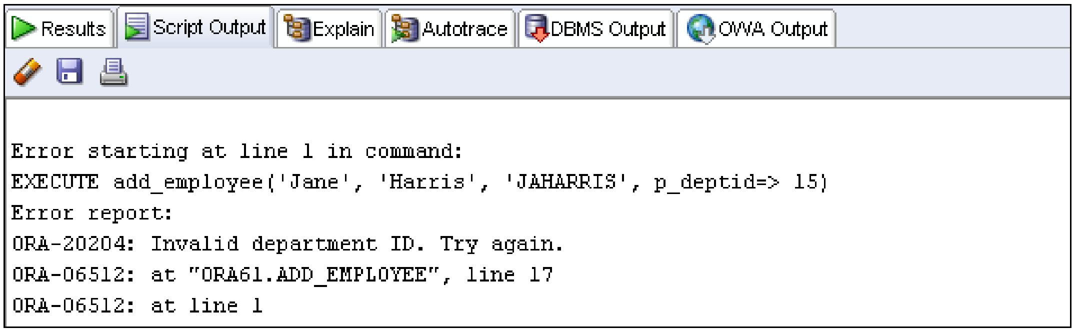
c) Call `ADD_EMPLOYEE` for the name `'Jane Harris'` in department `15`, leaving other parameters with their default values. What is the result?

Open the `sol_03_03_c.sql` file from the `D:\labs\PLPU\solns` folder, or enter the following code in the SQL Worksheet area. Click the Run Script

Practice 3: Creating Functions (continued)

(F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

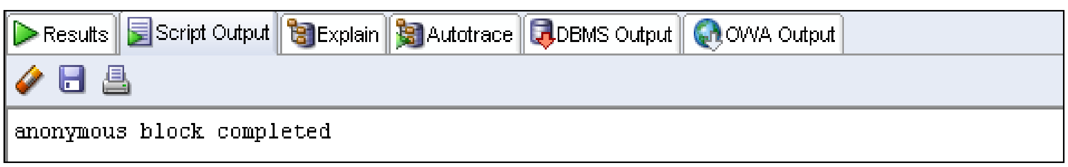
```
EXECUTE add_employee('Jane', 'Harris', 'JAHARRIS',
p_deptid=> 15)
```



d) Add another employee named Joe Harris in department 80, leaving the remaining parameters with their default values. What is the result?

Open the `sol_03_03_d.sql` file from the `D:\labs\PLPU\solns` folder, or enter the following code in the SQL Worksheet area, and then click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

```
EXECUTE add_employee('Joe', 'Harris', 'JAHARRIS',
p_deptid=> 80)
```



Practice 4: Creating and Using Packages

In this practice, you create package specifications and package bodies. You then invoke the constructs in the packages by using sample data.

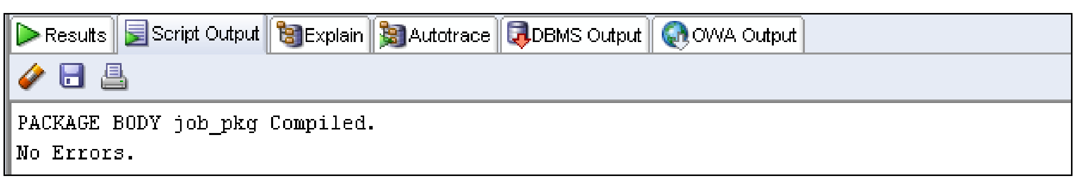
1) Create a package specification and body called `JOB_PKG`, containing a copy of your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures as well as your `GET_JOB` function.

   Note: Use the code from your previously saved procedures and functions when creating the package. You can copy the code in a procedure or function, and then paste the code into the appropriate section of the package.

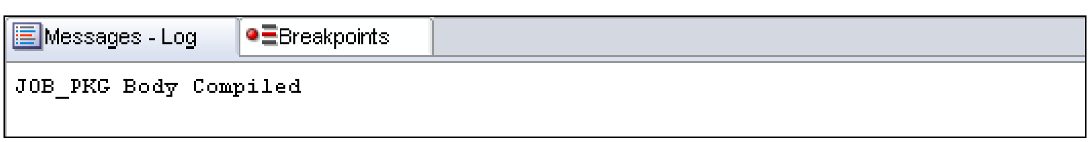
   a) Create the package specification including the procedures and function headings as public constructs.

      Open the `sol_04_01_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The code and the result are displayed as follows:

```
CREATE OR REPLACE PACKAGE job_pkg IS
  PROCEDURE add_job (p_jobid jobs.job_id%TYPE, p_jobtitle
jobs.job_title%TYPE);
  PROCEDURE del_job (p_jobid jobs.job_id%TYPE);
  FUNCTION get_job (p_jobid IN jobs.job_id%type) RETURN
jobs.job_title%type;
  PROCEDURE upd_job(p_jobid IN jobs.job_id%TYPE, p_jobtitle
IN jobs.job_title%TYPE);
END job_pkg;
/
SHOW ERRORS
```



To compile the new package body, right-click the package's body name in the Object Navigation tree, and then select Compile from the shortcut menu. The package body is compiled as shown below:

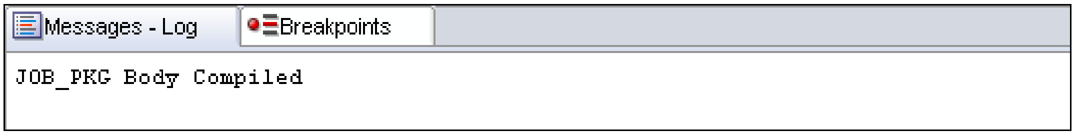


   b) Create the package body with the implementations for each of the subprograms.

Open the `sol_04_01_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package body. The code and the result are displayed as follows:

```
CREATE OR REPLACE PACKAGE BODY job_pkg IS
  PROCEDURE add_job (
    p_jobid jobs.job_id%TYPE,
    p_jobtitle jobs.job_title%TYPE) IS
  BEGIN
    INSERT INTO jobs (job_id, job_title)
    VALUES (p_jobid, p_jobtitle);
    COMMIT;
  END add_job;

  PROCEDURE del_job (p_jobid jobs.job_id%TYPE) IS
    BEGIN
      DELETE FROM jobs
      WHERE job_id = p_jobid;
      IF SQL%NOTFOUND THEN
        RAISE_APPLICATION_ERROR(-20203, 'No jobs
deleted.');
      END IF;
    END DEL_JOB;

  FUNCTION get_job (p_jobid IN jobs.job_id%type)
    RETURN jobs.job_title%type IS
    v_title jobs.job_title%type;
    BEGIN
      SELECT job_title
      INTO v_title
      FROM jobs
      WHERE job_id = p_jobid;
      RETURN v_title;
    END get_job;

  PROCEDURE upd_job(
    p_jobid IN jobs.job_id%TYPE,
    p_jobtitle IN jobs.job_title%TYPE) IS
    BEGIN
      UPDATE jobs
      SET job_title = p_jobtitle
      WHERE job_id = p_jobid;
      IF SQL%NOTFOUND THEN
        RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
      END IF;
    END upd_job;

END job_pkg;
```

```
/

SHOW ERRORS
```



c) Delete the following stand-alone procedures and function you just packaged using the Procedures and Functions nodes in the Object Navigation tree:

i) The `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures

ii) The `GET_JOB` function

To delete a procedure or a function, right-click the procedure's name or function's name in the Object Navigation tree, and then select Drop from the pop-up menu. The Drop window is displayed. Click Apply to drop the procedure or function. A confirmation window is displayed.

d) Invoke your `ADD_JOB` package procedure by passing the values `IT_SYSAN` and `SYSTEMS ANALYST` as parameters.
Open the `sol_04_01_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
EXECUTE job_pkg.add_job('IT_SYSAN', 'Systems Analyst')
```

e) Query the `JOBS` table to see the result.

Open the `sol_04_01_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon or the Execute Statement (F9) on the SQL Worksheet toolbar to query the `JOBS` table. The code and the result (using the Run Script icon) are displayed as follows:

```
SELECT *
FROM jobs
WHERE job_id = 'IT_SYSAN';
```

Practice 4: Creating and Using Packages (continued)


```
JOB_ID     JOB_TITLE                           MIN_SALARY             MAX_SALARY
---------- ----------------------------------- ---------------------- -------------
IT_SYSAN   Systems Analyst

1 rows selected
```

2) Create and invoke a package that contains private and public constructs.

   a) Create a package specification and a package body called `EMP_PKG` that contains the following procedures and function that you created earlier:

      i) `ADD_EMPLOYEE` procedure as a public construct

      ii) `GET_EMPLOYEE` procedure as a public construct

      iii) `VALID_DEPTID` function as a private construct

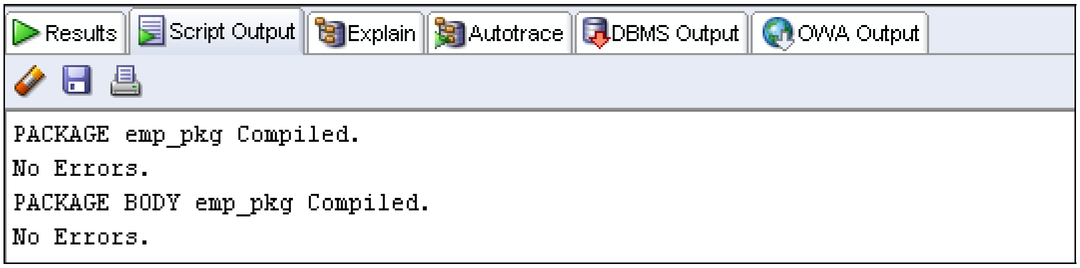
      Open the `sol_04_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);
PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
```

Practice 4: Creating and Using Packages (continued)

```
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;

END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary,
commission_pct, department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid
department ID. Try again.');
    END IF;
  END add_employee;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;
END emp_pkg;
/
SHOW ERRORS
```

Practice 4: Creating and Using Packages (continued)

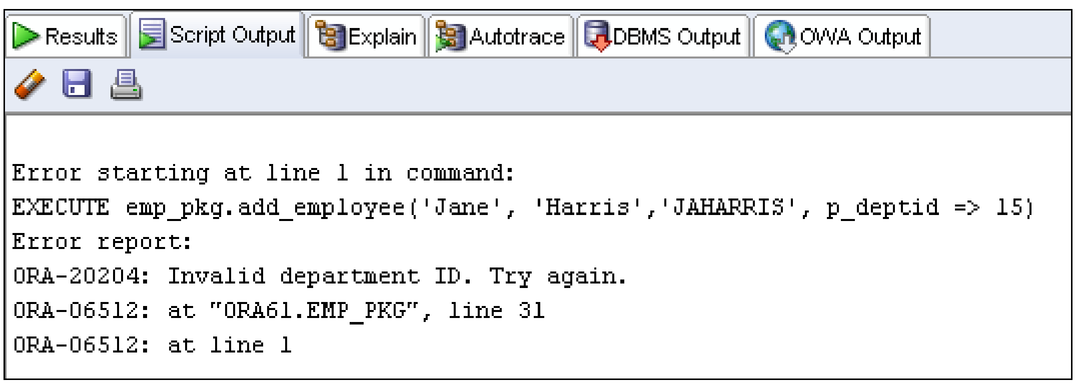



```
PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.
```

b) Invoke the EMP_PKG.ADD_EMPLOYEE procedure, using department ID 15 for employee Jane Harris with the email ID JAHARRIS. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.

Open the `sol_04_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
EXECUTE emp_pkg.add_employee('Jane', 'Harris','JAHARRIS',
p_deptid => 15)
```

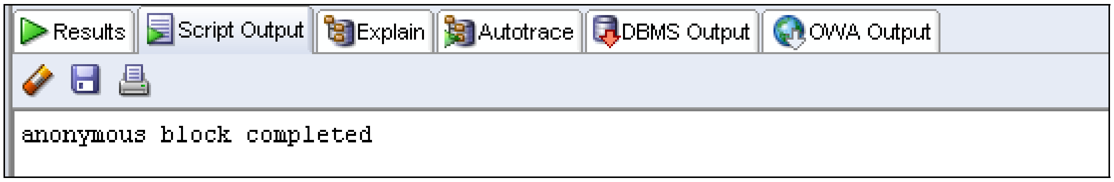



```
Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('Jane', 'Harris','JAHARRIS', p_deptid => 15)
Error report:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA61.EMP_PKG", line 31
ORA-06512: at line 1
```

c) Invoke the ADD_EMPLOYEE package procedure by using department ID 80 for employee David Smith with the email ID DASMITH.

Open the `sol_04_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

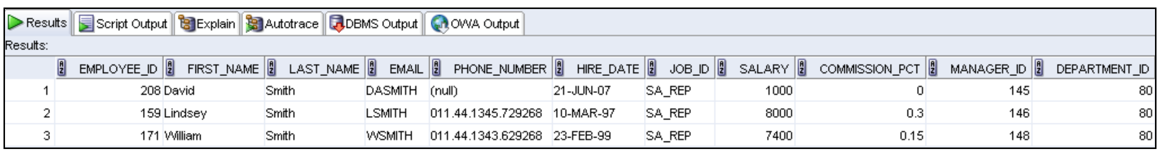
```
EXECUTE emp_pkg.add_employee('David', 'Smith','DASMITH',
p_deptid => 80)
```



d) Query the `EMPLOYEES` table to verify that the new employee was added.

Open the `sol_04_02_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon or the Execute Statement (F9) on the SQL Worksheet toolbar to query the `EMPLOYEES` table. The code and the result (Execute Statement icon) are displayed as follows:

```
SELECT *
FROM employees
WHERE last_name = 'Smith';
```



	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	208	David	Smith	DASMITH	(null)	21-JUN-07	SA_REP	1000	0	145	80
2	159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	0.3	146	80
3	171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	0.15	148	80

Practice 5: Working with Packages

In this practice, you modify an existing package to contain overloaded subprograms and you use forward declarations. You also create a package initialization block within a package body to populate a PL/SQL table.

1) Modify the code for the `EMP_PKG` package that you created in Practice 4 step 2, and overload the `ADD_EMPLOYEE` procedure.

   a) In the package specification, add a new procedure called `ADD_EMPLOYEE` that accepts the following three parameters:

      i) First name

      ii) Last name

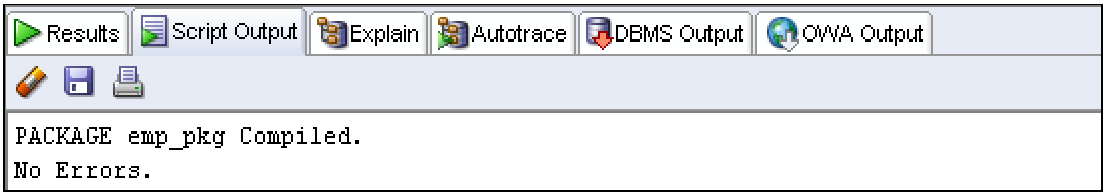
      iii) Department ID

      Open the `sol_05_01_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the highlighted part (code in bold-face letters) in the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

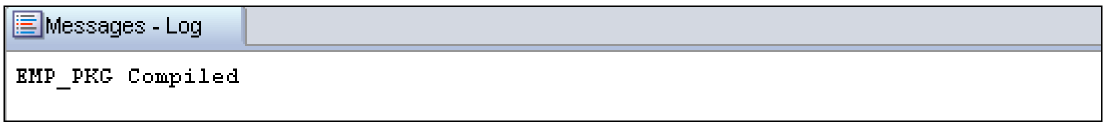
```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

/* New overloaded add_employee */

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS
```

Practice 5: Working with Packages (continued)

b) Click Run Script to create the package. Compile the package.



To compile the package, right-click the package's name in the Object Navigator tree, and then select Compile from the shortcut menu. The package is compiled as shown below:



c) Implement the new `ADD_EMPLOYEE` procedure in the package body as follows:

i) Format the email address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name.

ii) The procedure should call the existing `ADD_EMPLOYEE` procedure to perform the actual `INSERT` operation using its parameters and formatted email to supply the values.

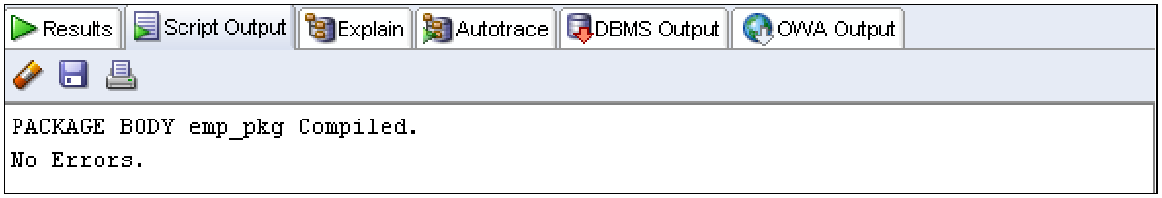
iii) Click Run Script to create the package. Compile the package.

Open the `sol_05_01_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the newly added and highlighted part (code in bold-face letters) in the following code box in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

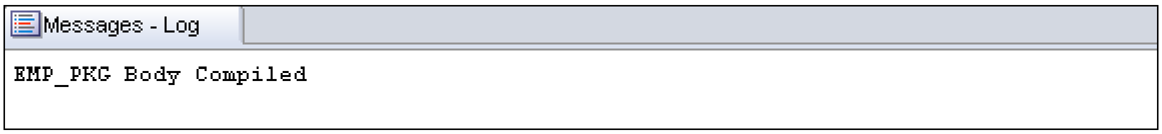
```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
```

```
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS


BEGIN
  IF valid_deptid(p_deptid) THEN
    INSERT INTO employees(employee_id, first_name, last_name,
      email, job_id, manager_id, hire_date, salary,
      commission_pct, department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
      p_email, p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
      p_deptid);
  ELSE
    RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try
     again.');
  END IF;
  END add_employee;

/* New overloaded add_employee procedure */

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
                     1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid =>
                 p_deptid);
  END;

/* End declaration of the overloaded add_employee procedure */

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;
END emp_pkg;
/
```

```
SHOW ERRORS
```



To compile the package, right-click the package's body (or the entire package) name in the Object Navigator tree, and then select Compile from the shortcut menu. The package body is compiled as shown below:



d) Invoke the new `ADD_EMPLOYEE` procedure using the name `Samuel Joplin` to be added to department 30.

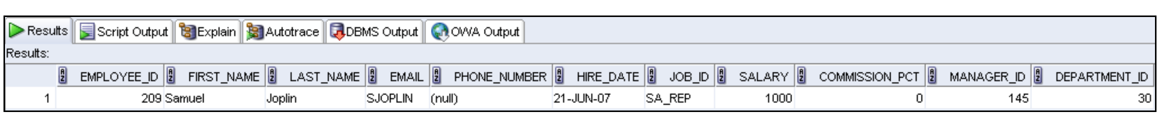
Open the `sol_05_01_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
EXECUTE emp_pkg.add_employee('Samuel', 'Joplin', 30)
```

e) Confirm that the new employee was added to the `EMPLOYEES` table.

Open the `sol_05_01_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to execute the query. The code and the result are displayed as follows:

```
SELECT *
FROM employees
WHERE last_name = 'Joplin';
```


Results:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	209	Samuel	Joplin	SJOPLIN	(null)	21-JUN-07	SA_REP	1000	0	145	30

Practice 5: Working with Packages (continued)

2) In the EMP_PKG package, create two overloaded functions called GET_EMPLOYEE:

   a) In the package specification, add the following functions:

      i) The GET_EMPLOYEE function that accepts the parameter called p_emp_id based on the employees.employee_id%TYPE type. This function should return EMPLOYEES%ROWTYPE.

      ii) The GET_EMPLOYEE function that accepts the parameter called p_family_name of type employees.last_name%TYPE. This function should return EMPLOYEES%ROWTYPE.

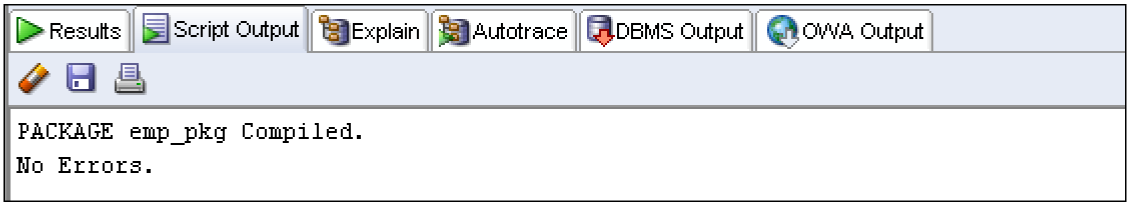
      Open the sol_05_02_a.sql file in the D:\labs\PLPU\solns folder, or copy and paste the newly added and highlighted code (code in bold-face letters) in the following code box in the SQL Worksheet area.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

/* New overloaded get_employees functions specs starts here: */

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype;

/* New overloaded get_employees functions specs ends here. */

END emp_pkg;
/
SHOW ERRORS
```

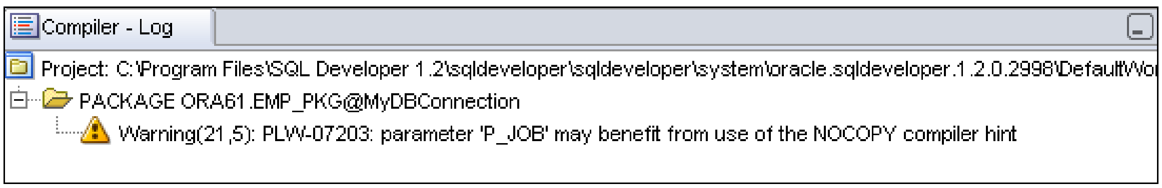
Practice 5: Working with Packages (continued)

b) Click Run Script to re-create and compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package's specification. The result is shown below:



To compile the package specification, right-click the package's specification (or the entire package) name in the Object Navigator tree, and then select Compile from the shortcut menu. The warning is expected and is for informational purposes only.



c) In the package body:

i) Implement the first `GET_EMPLOYEE` function to query an employee using the employee's ID.

ii) Implement the second `GET_EMPLOYEE` function to use the equality operator on the value supplied in the `p_family_name` parameter.

Open the `sol_05_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The newly added functions are highlighted in the following code box.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);
```

Practice 5: Working with Packages (continued)

```
PROCEDURE get_employee(


    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

/* New overloaded get_employees functions specs starts here: */

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype;

/* New overloaded get_employees functions specs ends here. */

END emp_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name,
        email, job_id, manager_id, hire_date, salary,
        commission_pct, department_id)
```

Practice 5: Working with Packages (continued)

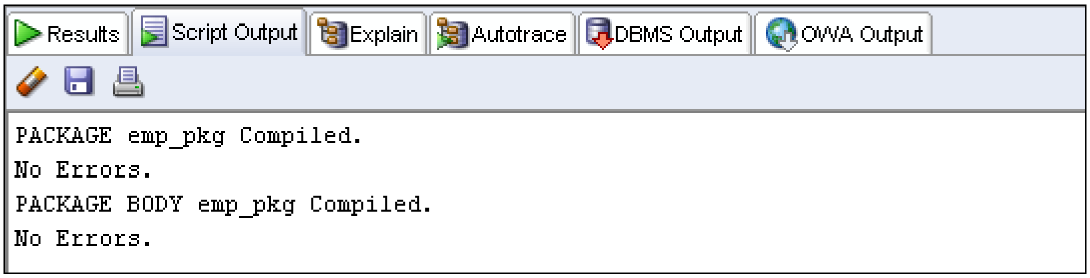
```
      VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
        p_email, p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
        p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
                              Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid =>
p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

/* New get_employee function declaration starts here */

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
```

Practice 5: Working with Packages (continued)

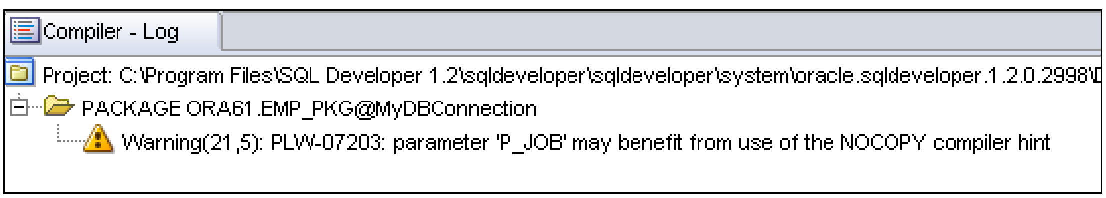
```
  END;

/* New overloaded get_employee function declaration ends here */

END emp_pkg;
/
SHOW ERRORS
```

d) Click Run Script to re-create the package. Compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package. The result is shown below:



To compile the package, right-click the package's name in the Object Navigator tree, and then select Compile from the shortcut menu. If you get a warning message, that is all right and is meant for informational purposes only.



e) Add a utility procedure `PRINT_EMPLOYEE` to the `EMP_PKG` package as follows:

i) The procedure accepts an `EMPLOYEES%ROWTYPE` as a parameter.

ii) The procedure displays the following for an employee on one line, using the `DBMS_OUTPUT` package:

- `department_id`
- `employee_id`
- `first_name`
- `last_name`
- `job_id`
- `salary`

Open the `sol_05_02_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The newly added code is highlighted in the following code box.

Practice 5: Working with Packages (continued)

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

/* New print_employee print_employee procedure spec */

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
```

Practice 5: Working with Packages (continued)

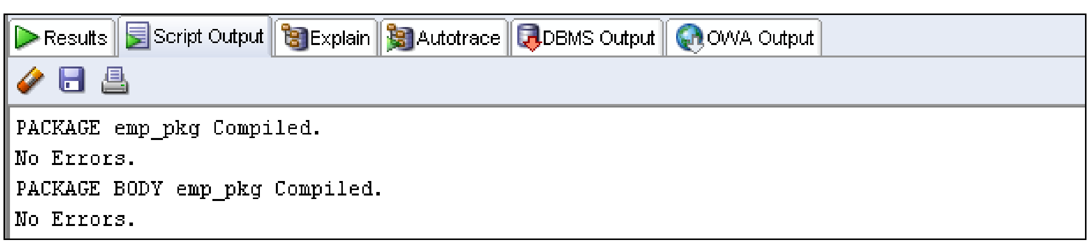
```
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;
```

Practice 5: Working with Packages (continued)

```
FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

/* New print_employees procedure declaration. */

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;

END emp_pkg;
/
SHOW ERRORS
```

f) Click Run Script (F5) to create the package. Compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package.



Practice 5: Working with Packages (continued)

To compile the package, right-click the package's name in the Object Navigator tree, and then select Compile from the shortcut menu.

Messages - Log

```
EMP_PKG Compiled
```

g) Use an anonymous block to invoke the `EMP_PKG.GET_EMPLOYEE` function with an employee ID of `100` and family name of `'Joplin'`. Use the `PRINT_EMPLOYEE` procedure to display the results for each row returned.

Open the `sol_05_02_g.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Make sure that `SET SERVEROUTPUT ON` is enabled by using the DBMS Output tab.

```
BEGIN
  emp_pkg.print_employee(emp_pkg.get_employee(100));
  emp_pkg.print_employee(emp_pkg.get_employee('Joplin'));
END;
/
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
anonymous block completed
90 100 Steven King AD_PRES 24000
30 209 Samuel Joplin SA_REP 1000
```

3) Because the company does not frequently change its departmental data, you can improve performance of your `EMP_PKG` by adding a public procedure, `INIT_DEPARTMENTS`, to populate a private PL/SQL table of valid department IDs. Modify the `VALID_DEPTID` function to use the private PL/SQL table contents to validate department ID values.

Note: The `sol_05_03.sql` solution file script contains the code for steps a, b, and c.

a) In the package specification, create a procedure called `INIT_DEPARTMENTS` with no parameters by adding the following to the package specification section before the `PRINT_EMPLOYEES` specification:

```
PROCEDURE init_departments;
```

b) In the package body, implement the `INIT_DEPARTMENTS` procedure to store all department IDs in a private PL/SQL index-by table named `valid_departments` containing `BOOLEAN` values.

i) Declare the `valid_departments` variable and its type definition `boolean_tab_type` before all procedures in the body. Enter the following at the beginning of the package body:

```
TYPE boolean_tab_type IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;
```

ii) Use the `department_id` column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of `TRUE`. Enter the `INIT_DEPARTMENTS` procedure declaration at the end of the package body (right after the `print_employees` procedure) as follows:

```
PROCEDURE init_departments IS
BEGIN
  FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;
```

c) In the body, create an initialization block that calls the `INIT_DEPARTMENTS` procedure to initialize the table as follows:

```
BEGIN
  init_departments;
END;
```

Open the `sol_05_03.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The newly added code is highlighted in the following code box.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
```

```
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

/* New procedure init_departments spec */

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS


-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS

/* New type */

TYPE boolean_tab_type IS TABLE OF BOOLEAN
          INDEX BY BINARY_INTEGER;
  valid_departments boolean_tab_type;


FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
```

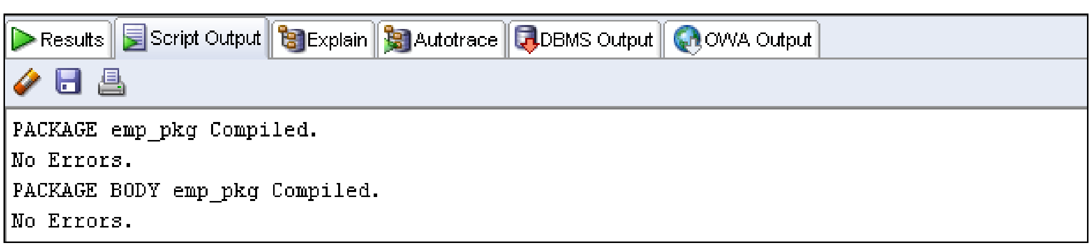
```
    IF valid_deptid(p_deptid) THEN

    INSERT INTO employees(employee_id, first_name, last_name,
       email, job_id, manager_id, hire_date, salary,
       commission_pct, department_id)
   VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
      p_email, p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
      p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
                              Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
```

Practice 5: Working with Packages (continued)

```
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         P_rec_emp.employee_id||' '||
                         P_rec_emp.first_name||' '||
                         P_rec_emp.last_name||' '||
                         P_rec_emp.job_id||' '||
                         P_rec_emp.salary);
  END;

/* New init_departments procedure declaration. */

PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

/* call the new init_departments procedure. */

BEGIN
  init_departments;
END emp_pkg;

/
SHOW ERRORS
```

d) Click Run Script (F5) to create the package. Compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package.



Practice 5: Working with Packages (continued)

To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile from the shortcut menu.

4) Change the `VALID_DEPTID` validation processing function to use the private PL/SQL table of department IDs.

   a) Modify the `VALID_DEPTID` function to perform its validation by using the PL/SQL table of department ID values. Click Run Script (F5) to create the package. Compile the package.

      Open the `sol_05_04_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The newly added code is highlighted in the following code box.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
      employees.last_name%type)
    return employees%rowtype;

/* New procedure init_departments spec */

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
```

Practice 5: Working with Packages (continued)

```
/
SHOW ERRORS



-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS

TYPE boolean_tab_type IS TABLE OF BOOLEAN
     INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name,
        last_name, email, job_id, manager_id, hire_date,
        salary, commission_pct, department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
        p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
                               Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
```

```
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;

/* New init_departments procedure declaration. */

PROCEDURE init_departments IS
  BEGIN
```

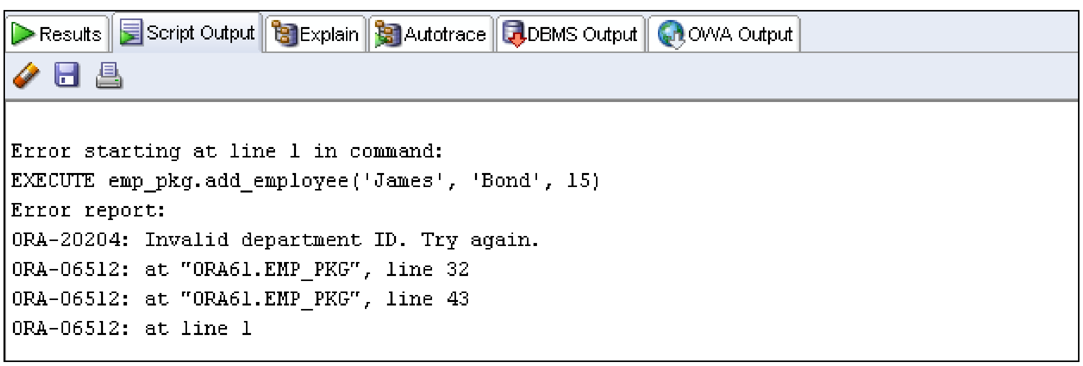
```
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

/* call the new init_departments procedure. */

BEGIN
  init_departments;
END emp_pkg;

/
SHOW ERRORS
```

b) Test your code by calling `ADD_EMPLOYEE` using the name `James Bond` in department `15`. What happens?

Open the `sol_05_04_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area.

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
```

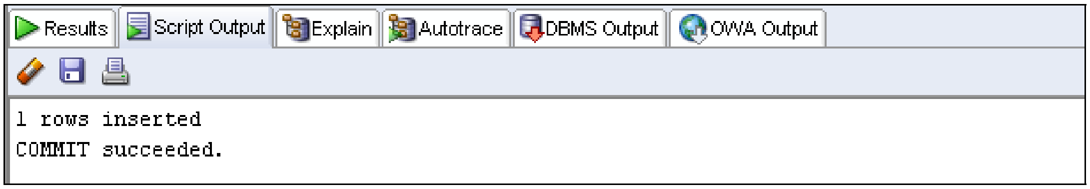
Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package. The insert operation to add the employee fails with an exception because department 15 does not exist.



c) Insert a new department. Specify 15 for the department ID and `'Security'` for the department name. Commit and verify the changes.

Open the `sol_05_04_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The result is shown below:

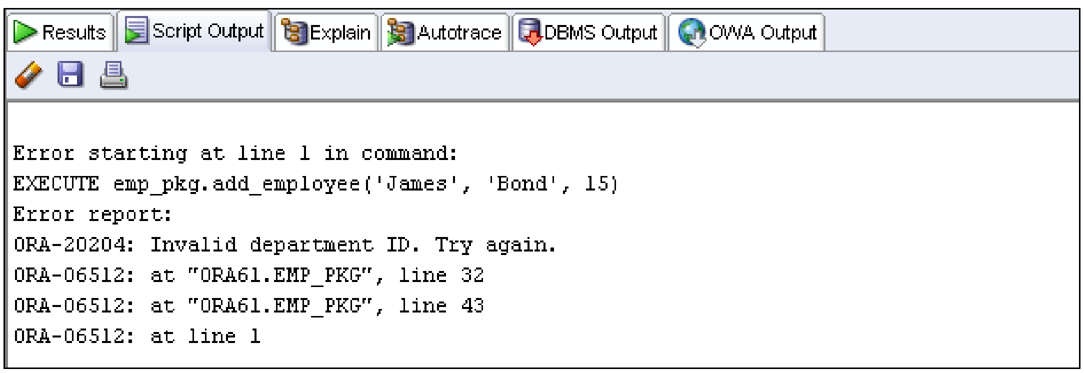
```
INSERT INTO departments (department_id, department_name)
VALUES (15, 'Security');
COMMIT;
```



d) Test your code again, by calling `ADD_EMPLOYEE` using the name `James Bond` in department 15. What happens?

Open the `sol_05_04_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The result is shown below:

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
```

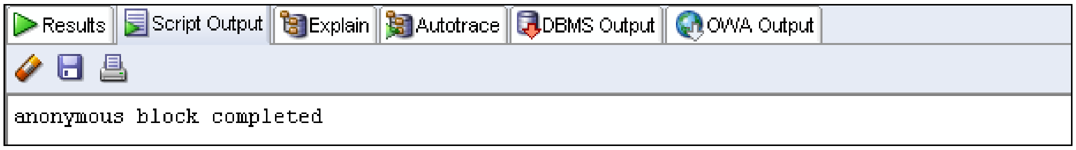


The insert operation to add the employee fails with an exception. Department 15 does not exist as an entry in the PL/SQL index-by-table package state variable.

e) Execute the `EMP_PKG.INIT_DEPARTMENTS` procedure to update the internal PL/SQL table with the latest departmental data.

Open the `sol_05_04_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The result is shown below:

```
EXECUTE EMP_PKG.INIT_DEPARTMENTS
```

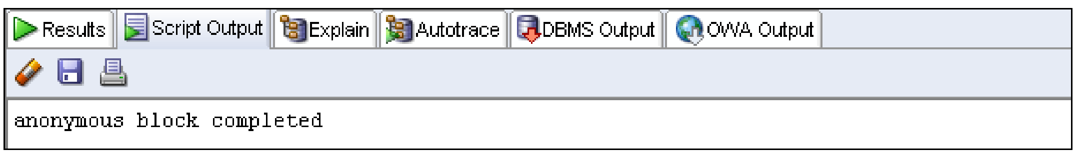


f) Test your code by calling `ADD_EMPLOYEE` using the employee name `James Bond`, who works in department `15`. What happens?

Open the `sol_05_04_f.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The result is shown below.

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
```

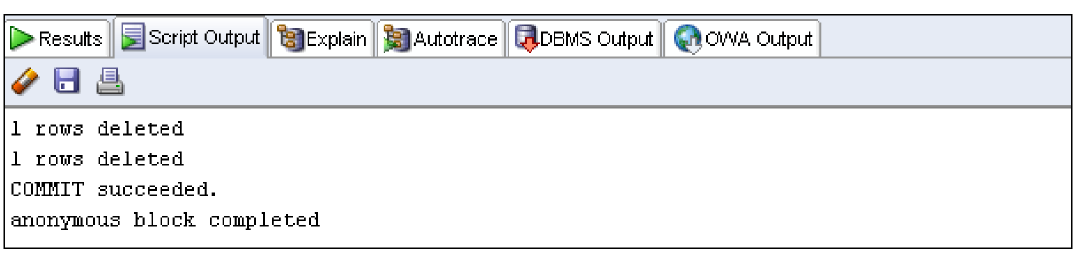
The row is finally inserted because the department 15 record exists in the database and the package's PL/SQL index-by table, due to invoking `EMP_PKG.INIT_DEPARTMENTS`, which refreshes the package state data.



g) Delete employee `James Bond` and department `15` from their respective tables, commit the changes, and refresh the department data by invoking the `EMP_PKG.INIT_DEPARTMENTS` procedure.

Open the `sol_05_04_g.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The result is shown below.

```
DELETE FROM employees
WHERE first_name = 'James' AND last_name = 'Bond';
DELETE FROM departments WHERE department_id = 15;
COMMIT;
EXECUTE EMP_PKG.INIT_DEPARTMENTS
```



Practice 5: Working with Packages (continued)

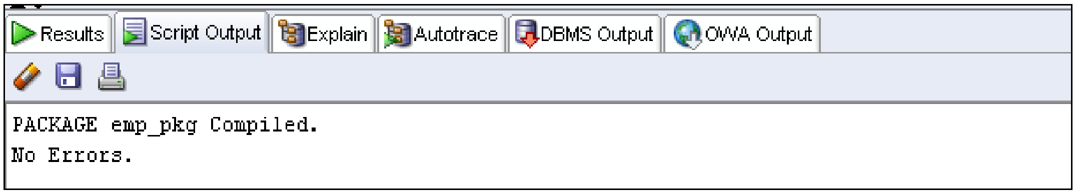
5) Reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.

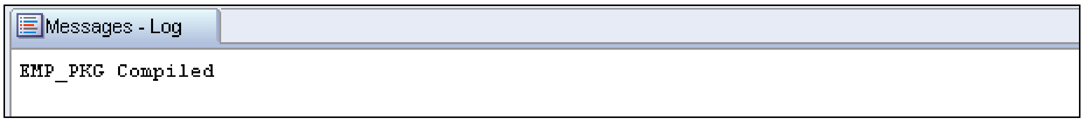
   a) Edit the package specification and reorganize subprograms alphabetically. Click Run Script to re-create the package specification. Compile the package specification. What happens?

      Open the `sol_05_05_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package. The result is shown below. The package's specification subprograms are already in an alphabetical order. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE emp_pkg IS

/* the package spec is already in an alphabetical order. */

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id
employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);
```

```
END emp_pkg;
/
SHOW ERRORS
```





b) Edit the package body and reorganize all subprograms alphabetically. Click Run Script to re-create the package specification. Re-compile the package specification. What happens?

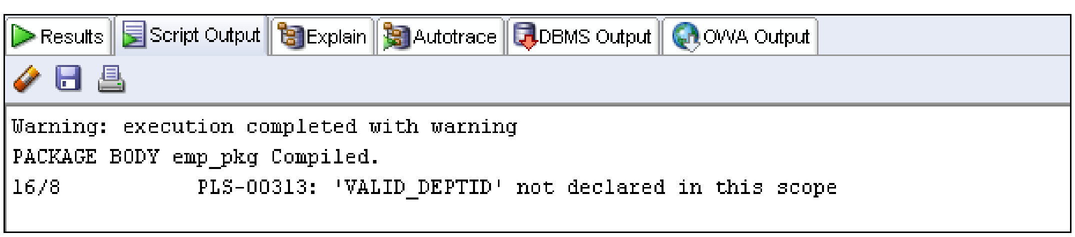
Open the `sol_05_05_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package. The result is shown below. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

```
-- Package BODY
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
     INDEX BY BINARY_INTEGER;
  valid_departments boolean_tab_type;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary,
commission_pct, department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
```

```
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department
ID. Try again.');
    END IF;
  END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email,
p_deptid => p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

  FUNCTION get_employee(p_emp_id
employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;
```

```
  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

  PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;


BEGIN
  init_departments;
END emp_pkg;

/
SHOW ERRORS
```

The package does not compile successfully because the `VALID_DEPTID` function is referenced before it is declared.



c) Correct the compilation error using a forward declaration in the body for the appropriate subprogram reference. Click Run Script to re-create the package, and then recompile the package. What happens?

Open the `sol_05_05_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The function's forward declaration is highlighted in the code box below. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package. The result is shown below. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

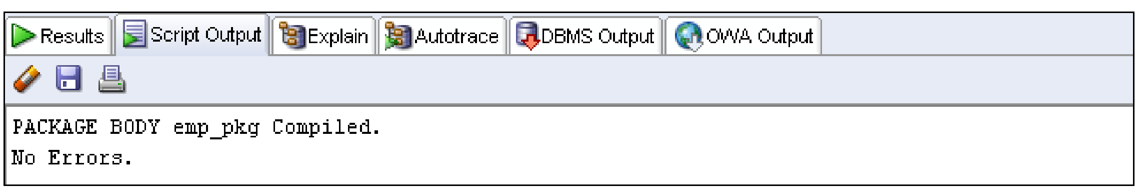
```
-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
      INDEX BY BINARY_INTEGER;
  valid_departments boolean_tab_type;

/* forward declaration of valid_deptid */

  FUNCTION valid_deptid(p_deptid IN
     departments.department_id%TYPE)
   RETURN BOOLEAN;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN /* valid_deptid function 
referneced */
      INSERT INTO employees(employee_id, first_name, 
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct, 
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name, 
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm, 
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. 
Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
```

```
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

/* New alphabetical location of function init_departments. */

PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
```

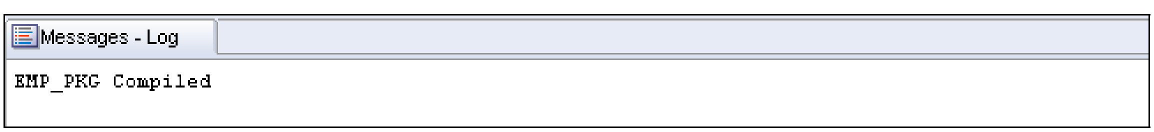
Practice 5: Working with Packages (continued)

```
                              p_rec_emp.employee_id||' '||
                              p_rec_emp.first_name||' '||
                              p_rec_emp.last_name||' '||
                              p_rec_emp.job_id||' '||
                              p_rec_emp.salary);
  END;

/* New alphabetical location of function valid_deptid. */

FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;


BEGIN
  init_departments;
END emp_pkg;

/
SHOW ERRORS
```

A forward declaration for the `VALID_DEPTID` function enables the package body to compile successfully as shown below:



To compile the package, click the package's name in the Object Navigation tree, and then select Compile from the pop-up menu.



Practice 6: Using the `UTL_FILE` Package

In this practice, you use the UTL_FILE package to generate a text file report of employees in each department.

1) Create a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their departments.
   a) Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.

      Note: Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.

      Open the `sol_06_01_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the procedure. The result is shown below. To compile the procedure, right-click the procedure's name in the Object Navigation tree, and then select Compile.

```
-- Verify with your instructor that the database initSID.ora
-- file has the directory path you are going to use with this
-- procedure.
-- For example, there should be an entry such as:
-- UTL_FILE_DIR = /home1/teachX/UTL_FILE in your initSID.ora
-- (or the SPFILE)
-- HOWEVER: The course has a directory alias provided called
-- "UTL_FILE" that is associated with an appropriate
-- directory. Use the directory alias name in quotes for the
-- first parameter to create a file in the appropriate
-- directory.

CREATE OR REPLACE PROCEDURE employee_report(
  p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
  f UTL_FILE.FILE_TYPE;
  CURSOR cur_avg IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                    FROM  employees inner
                    GROUP BY outer.department_id)
    ORDER BY department_id;
BEGIN
  f := UTL_FILE.FOPEN(p_dir, p_filename,'W');
```

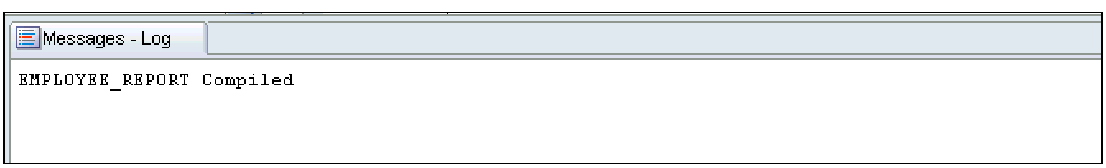
Practice 6: Using the UTL_FILE Package (continued)

```
 UTL_FILE.PUT_LINE(f, 'Employees who earn more than average
   salary: ');
  UTL_FILE.PUT_LINE(f, 'REPORT GENERATED ON ' ||SYSDATE);
  UTL_FILE.NEW_LINE(f);
  FOR emp IN cur_avg
  LOOP

    UTL_FILE.PUT_LINE(f,
    RPAD(emp.last_name, 30) || ' ' ||
    LPAD(NVL(TO_CHAR(emp.department_id,'9999'),'-'), 5) || ' '
||
    LPAD(TO_CHAR(emp.salary, '$99,999.00'), 12));
  END LOOP;
  UTL_FILE.NEW_LINE(f);
  UTL_FILE.PUT_LINE(f, '*** END OF REPORT ***');
  UTL_FILE.FCLOSE(f);
END employee_report;
/
```

b) Click Run Script (F5) to create the procedure. Compile the procedure.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the procedure.

To compile the procedure, right-click the procedure's name in the Object Navigator tree, and then select Compile from the shortcut menu.



2) Invoke the program, using the second parameter with a name such as `sal_rpt`*`xx`*`.txt`, where *xx* represents your user number (for example, 61, 62, …, 80, and so on).

Open the `sol_06_02.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to execute the procedure. The result is shown below. To compile the procedure, right-click the package's name in the Object Navigation tree, and then select Compile from the shortcut menu.

```
-- For example, if you are student ora61, use 61 as a prefix

EXECUTE employee_report('UTL_FILE','sal_rpt61.txt')
```

3) Transfer the generated output text file from the host to your desktop client as follows:

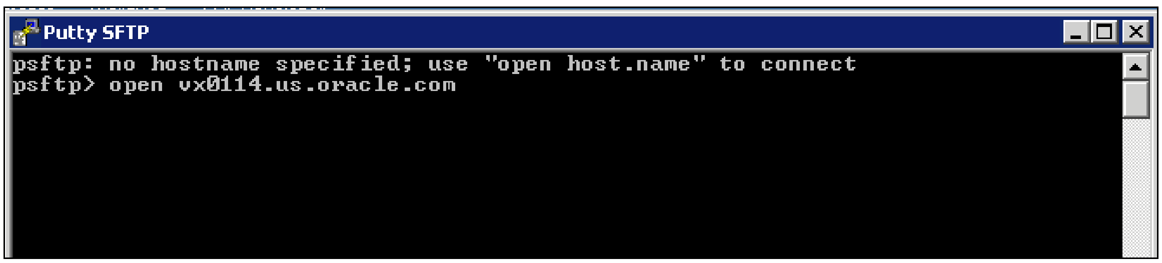
   a) Double-click the **Putty-SFTP** icon on your desktop. The Putty SFTP command window is displayed.

   b) At the **psftp>** prompt, enter the following command substituting the *host_name* with the host name provided to you by your instructor:

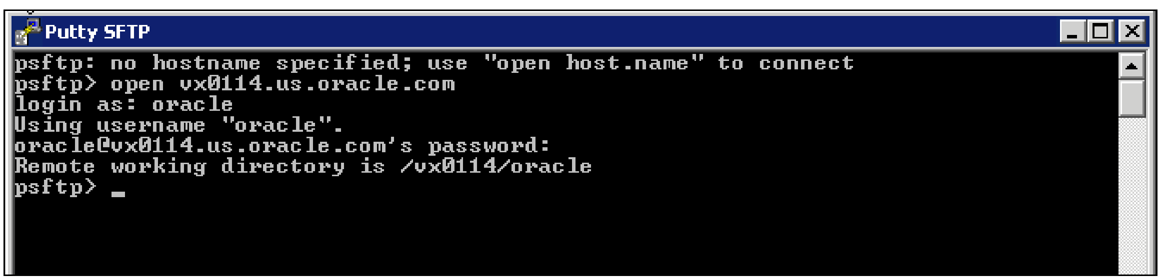
   `open` *`host_name`*

   For example, if you are connecting to a host named `vx0114.us.oracle.com`, enter the following at the prompt:

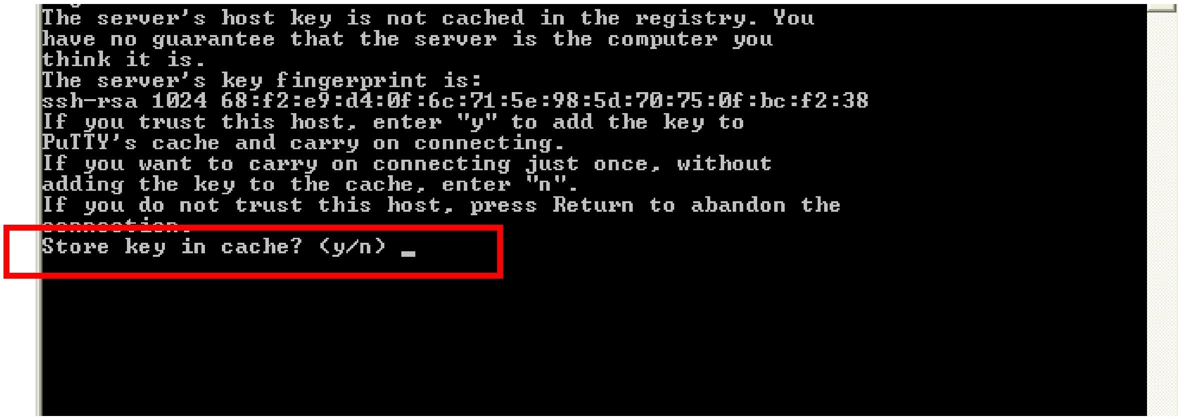
   ```
   open vx0114.us.oracle.com
   ```



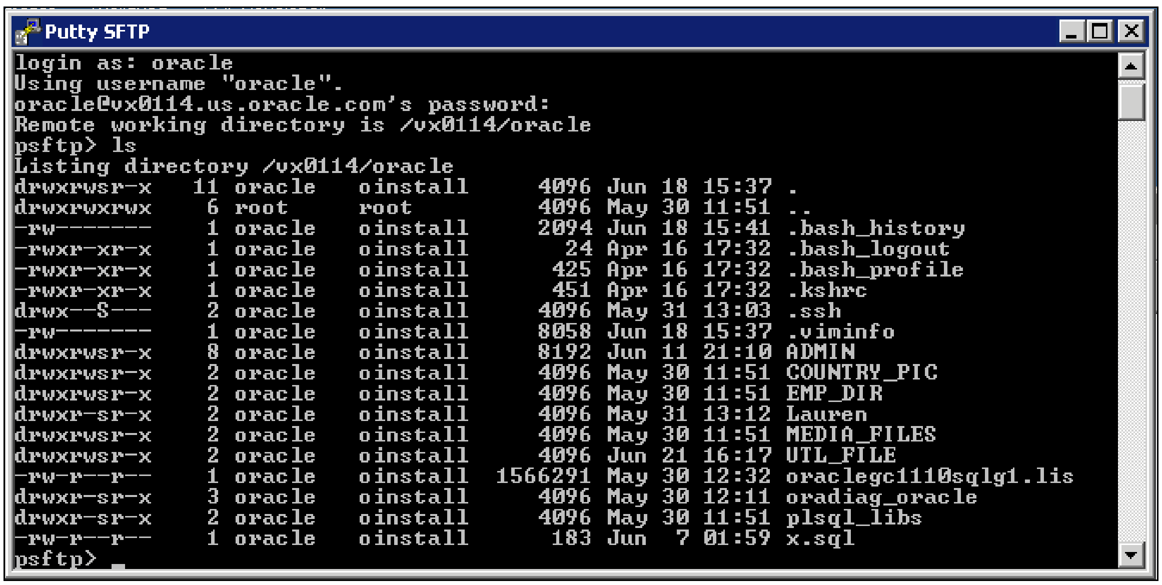
   c) Enter `oracle` as both your username and password.



   Note: After you enter the username, if you get a message about the host key not being cached in as shown in the following screen capture, enter `y` at the following prompt: "Store key in cache? <y/n>_"



d) To display the list of folders and files in the current directory, issue the `ls` command.



e) Change your directory to `UTL_FILE` using the `cd UTL_FILE` command as follows:

```
Putty SFTP
oracle@vx0114.us.oracle.com's password:
Remote working directory is /vx0114/oracle
psftp> ls
Listing directory /vx0114/oracle
drwxrwsr-x   11 oracle   oinstall     4096 Jun 18 15:37 .
drwxrwxrwx    6 root     root         4096 May 30 11:51 ..
-rw-------    1 oracle   oinstall     2094 Jun 18 15:41 .bash_history
-rwxr-xr-x    1 oracle   oinstall       24 Apr 16 17:32 .bash_logout
-rwxr-xr-x    1 oracle   oinstall      425 Apr 16 17:32 .bash_profile
-rwxr-xr-x    1 oracle   oinstall      451 Apr 16 17:32 .kshrc
drwx--S---    2 oracle   oinstall     4096 May 31 13:03 .ssh
-rw-------    1 oracle   oinstall     8058 Jun 18 15:37 .viminfo
drwxrwsr-x    8 oracle   oinstall     8192 Jun 11 21:10 ADMIN
drwxrwsr-x    2 oracle   oinstall     4096 May 30 11:51 COUNTRY_PIC
drwxrwsr-x    2 oracle   oinstall     4096 May 30 11:51 EMP_DIR
drwxr-sr-x    2 oracle   oinstall     4096 May 31 13:12 Lauren
drwxrwsr-x    2 oracle   oinstall     4096 May 30 11:51 MEDIA_FILES
drwxrwsr-x    2 oracle   oinstall     4096 Jun 21 16:17 UTL_FILE
-rw-r--r--    1 oracle   oinstall  1566291 May 30 12:32 oraclegc1110sqlg1.lis
drwxr-sr-x    3 oracle   oinstall     4096 May 30 12:11 oradiag_oracle
drwxr-sr-x    2 oracle   oinstall     4096 May 30 11:51 plsql_libs
-rw-r--r--    1 oracle   oinstall      183 Jun  7 01:59 x.sql
psftp> cd UTL_FILE
Remote directory is now /vx0114/oracle/UTL_FILE
psftp>
```

f) List the contents of the current directory using the `ls` command as follows:

```
Putty SFTP
-rw-------    1 oracle   oinstall     8058 Jun 18 15:37 .viminfo
drwxrwsr-x    8 oracle   oinstall     8192 Jun 11 21:10 ADMIN
drwxrwsr-x    2 oracle   oinstall     4096 May 30 11:51 COUNTRY_PIC
drwxrwsr-x    2 oracle   oinstall     4096 May 30 11:51 EMP_DIR
drwxr-sr-x    2 oracle   oinstall     4096 May 31 13:12 Lauren
drwxrwsr-x    2 oracle   oinstall     4096 May 30 11:51 MEDIA_FILES
drwxrwsr-x    2 oracle   oinstall     4096 Jun 21 16:17 UTL_FILE
-rw-r--r--    1 oracle   oinstall  1566291 May 30 12:32 oraclegc1110sqlg1.lis
drwxr-sr-x    3 oracle   oinstall     4096 May 30 12:11 oradiag_oracle
drwxr-sr-x    2 oracle   oinstall     4096 May 30 11:51 plsql_libs
-rw-r--r--    1 oracle   oinstall      183 Jun  7 01:59 x.sql
psftp> cd UTL_FILE
Remote directory is now /vx0114/oracle/UTL_FILE
psftp> ls
Listing directory /vx0114/oracle/UTL_FILE
drwxrwsr-x    2 oracle   oinstall     4096 Jun 21 16:17 .
drwxrwsr-x   11 oracle   oinstall     4096 Jun 18 15:37 ..
-rw-r--r--    1 oracle   oinstall     3780 May 31  2006 instructor.txt
-rw-r--r--    1 oracle   oinstall     2700 Jun 21 16:17 sal_rpt61.txt
psftp> _
```

Note the generated output file, `sal_rpt61.txt` (your file will have a different prefixed number that corresponds to your db account #).

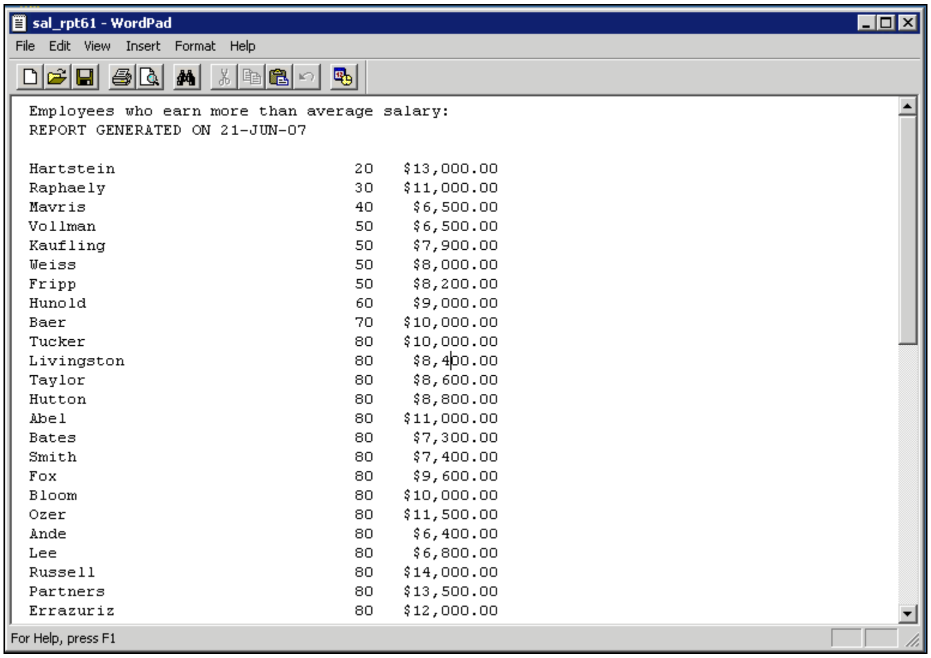
g) Transfer the output file from the host to your client machine by issuing the following command:

```
get sal_rpt61.txt
```

h) Exit **Putty-SFTP** by entering `bye` at the command line or by clicking the close control on title bar.

i) Open the transferred file, such as `sal_rpt61.txt`, which you can find in the `D:\Other\putty` folder using WordPad. The report is displayed as follows:

Practice 6: Using the UTL_FILE Package (continued)




```
Employees who earn more than average salary:
REPORT GENERATED ON 21-JUN-07

Hartstein                          20   $13,000.00
Raphaely                           30   $11,000.00
Mavris                             40    $6,500.00
Vollman                            50    $6,500.00
Kaufling                           50    $7,900.00
Weiss                              50    $8,000.00
Fripp                              50    $8,200.00
Hunold                             60    $9,000.00
Baer                               70   $10,000.00
Tucker                             80   $10,000.00
Livingston                         80    $8,400.00
Taylor                             80    $8,600.00
Hutton                             80    $8,800.00
Abel                               80   $11,000.00
Bates                              80    $7,300.00
Smith                              80    $7,400.00
Fox                                80    $9,600.00
Bloom                              80   $10,000.00
Ozer                               80   $11,500.00
Ande                               80    $6,400.00
Lee                                80    $6,800.00
Russell                            80   $14,000.00
Partners                           80   $13,500.00
Errazuriz                          80   $12,000.00
```

Practice 7: Using Native Dynamic SQL

In this practice, you create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. In addition, you create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an `INVALID` status in the `USER_OBJECTS` table....

1) Create a package called `TABLE_PKG` that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. The subprograms should manage optional default parameters with `NULL` values.

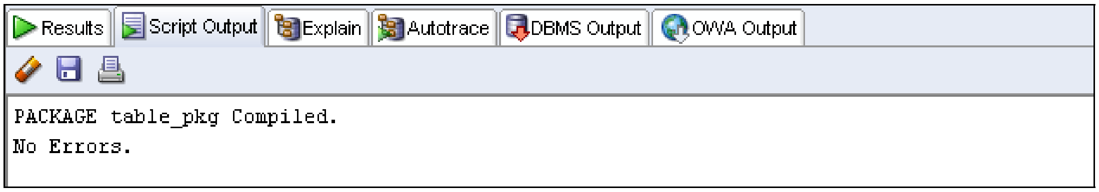
   a) Create a package specification with the following procedures:

```
PROCEDURE make(p_table_name VARCHAR2, p_col_specs VARCHAR2)
PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
  VARCHAR2, p_cols VARCHAR2 := NULL)
PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
  VARCHAR2, p_conditions VARCHAR2 := NULL)
PROCEDURE del_row(p_table_name VARCHAR2,
    p_conditions VARCHAR2 := NULL);
PROCEDURE remove(p_table_name VARCHAR2)
```

   Open the `sol_07_01_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The result is shown below. To compile the package's specification, right-click the package's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE table_pkg IS
  PROCEDURE make(p_table_name VARCHAR2, p_col_specs
      VARCHAR2);
  PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
      VARCHAR2, p_cols VARCHAR2 := NULL);
  PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
      VARCHAR2, p_conditions VARCHAR2 := NULL);
  PROCEDURE del_row(p_table_name VARCHAR2, p_conditions
      VARCHAR2 := NULL);
  PROCEDURE remove(p_table_name VARCHAR2);
END table_pkg;
/
SHOW ERRORS
```

Practice 7: Using Native Dynamic SQL (continued)



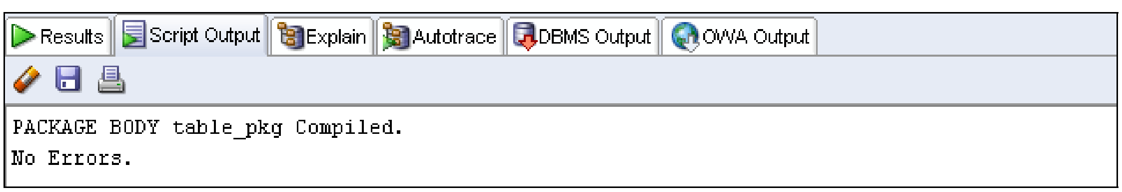
b) Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the remove procedure. This procedure should be written using the `DBMS_SQL` package.

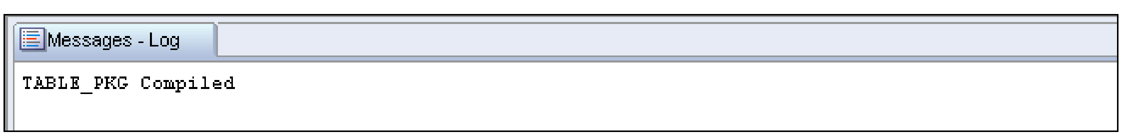
Open the `sol_07_01_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The result is shown below. To compile the package's specification, right-click the package's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE BODY table_pkg IS
  PROCEDURE execute(p_stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_stmt);
    EXECUTE IMMEDIATE p_stmt;
  END;

  PROCEDURE make(p_table_name VARCHAR2, p_col_specs VARCHAR2)
  IS
    v_stmt VARCHAR2(200) := 'CREATE TABLE '|| p_table_name ||
                           ' (' || p_col_specs || ')';
  BEGIN
    execute(v_stmt);
  END;

  PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
                    VARCHAR2, p_cols VARCHAR2 := NULL) IS
    v_stmt VARCHAR2(200) := 'INSERT INTO '|| p_table_name;
  BEGIN
    IF p_cols IS NOT NULL THEN
       v_stmt := v_stmt || ' (' || p_cols || ')';
    END IF;
    v_stmt := v_stmt || ' VALUES (' || p_col_values || ')';
    execute(v_stmt);
  END;

  PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
                   VARCHAR2, p_conditions VARCHAR2 := NULL) IS
```

Practice 7: Using Native Dynamic SQL (continued)

```
   v_stmt VARCHAR2(200) := 'UPDATE '|| p_table_name || ' SET '
|| p_set_values;
  BEGIN
    IF p_conditions IS NOT NULL THEN
       v_stmt := v_stmt || ' WHERE ' || p_conditions;
    END IF;
    execute(v_stmt);
  END;

  PROCEDURE del_row(p_table_name VARCHAR2, p_conditions
                    VARCHAR2 := NULL) IS
    v_stmt VARCHAR2(200) := 'DELETE FROM '|| p_table_name;
  BEGIN
    IF p_conditions IS NOT NULL THEN
       v_stmt := v_stmt || ' WHERE ' || p_conditions;
    END IF;
    execute(v_stmt);
  END;

  PROCEDURE remove(p_table_name VARCHAR2) IS
    cur_id INTEGER;
    v_stmt VARCHAR2(100) := 'DROP TABLE '||p_table_name;
  BEGIN
    cur_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_OUTPUT.PUT_LINE(v_stmt);
    DBMS_SQL.PARSE(cur_id, v_stmt, DBMS_SQL.NATIVE);
    -- Parse executes DDL statements,no EXECUTE is required.
    DBMS_SQL.CLOSE_CURSOR(cur_id);
  END;

END table_pkg;
/
SHOW ERRORS
```



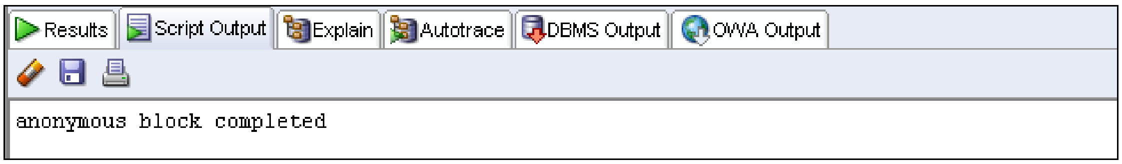


c) Execute the MAKE package procedure to create a table as follows:

```
make('my_contacts', 'id number(4), name
varchar2(40)');
```

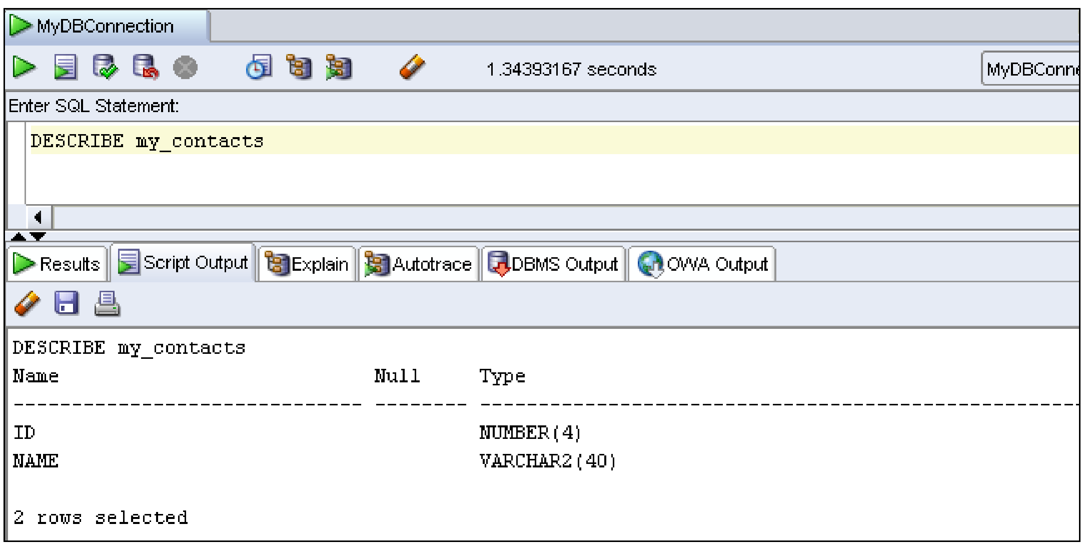
Open the `sol_07_01_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The code and the results are shown below. To compile the package's specification, right-click the package's name in the Object Navigation tree, and then select Compile.

```
EXECUTE table_pkg.make('my_contacts', 'id number(4), name
varchar2(40)')
```



d) Describe the `MY_CONTACTS` table structure.

The code and the results are shown below.



e) Execute the `ADD_ROW` package procedure to add the following rows:

```
add_row('my_contacts','1,''Lauran Serhal''','id, name');
add_row('my_contacts','2,''Nancy''','id, name');
add_row('my_contacts','3,''Sunitha Patel''','id,name');
add_row('my_contacts','4,''Valli Pataballa''','id,name');
```

Open the `sol_07_01_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to execute the script. The result is shown below. To compile the package's specification, right-click the package's name in the Object Navigation tree, and then select Compile.

Practice 7: Using Native Dynamic SQL (continued)

```
0.52349108 seconds
Enter SQL Statement:
BEGIN
  table_pkg.add_row('my_contacts','1,''Lauran Serhal''','id, name');
  table_pkg.add_row('my_contacts','2,''Nancy''','id, name');
  table_pkg.add_row('my_contacts','3,''Sunitha Patel''','id,name');
  table_pkg.add_row('my_contacts','4,''Valli Pataballa''','id,name');
END;
/
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
anonymous block completed
INSERT INTO my_contacts (id, name) VALUES (1,'Lauran Serhal')
INSERT INTO my_contacts (id, name) VALUES (2,'Nancy')
INSERT INTO my_contacts (id,name) VALUES (3,'Sunitha Patel')
INSERT INTO my_contacts (id,name) VALUES (4,'Valli Pataballa')
```

f) Query the `MY_CONTACTS` table contents to verify the additions.

The code and result are shown below.

```
Enter SQL Statement:
SELECT *
FROM my_contacts;
```

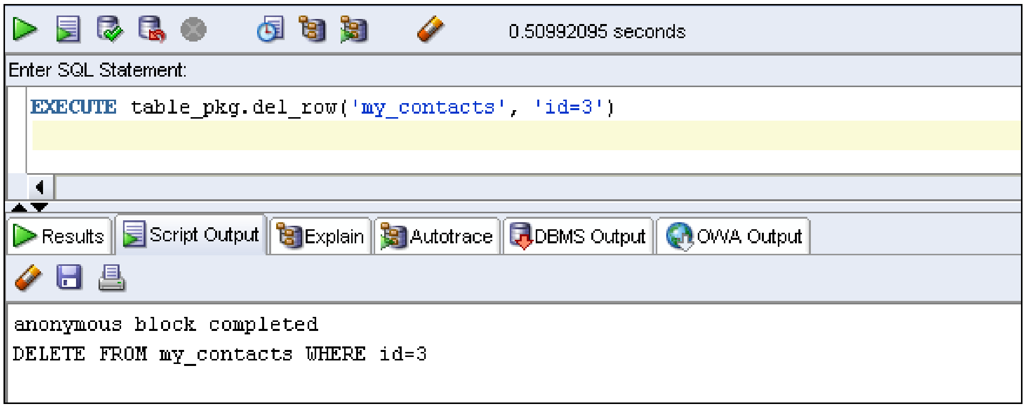
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
ID                     NAME
---------------------- -----------------------------------------
1                      Lauran Serhal
2                      Nancy
3                      Sunitha Patel
4                      Valli Pataballa

4 rows selected
```

g) Execute the `DEL_ROW` package procedure to delete a contact with ID value `3`.

The code and result are shown below.

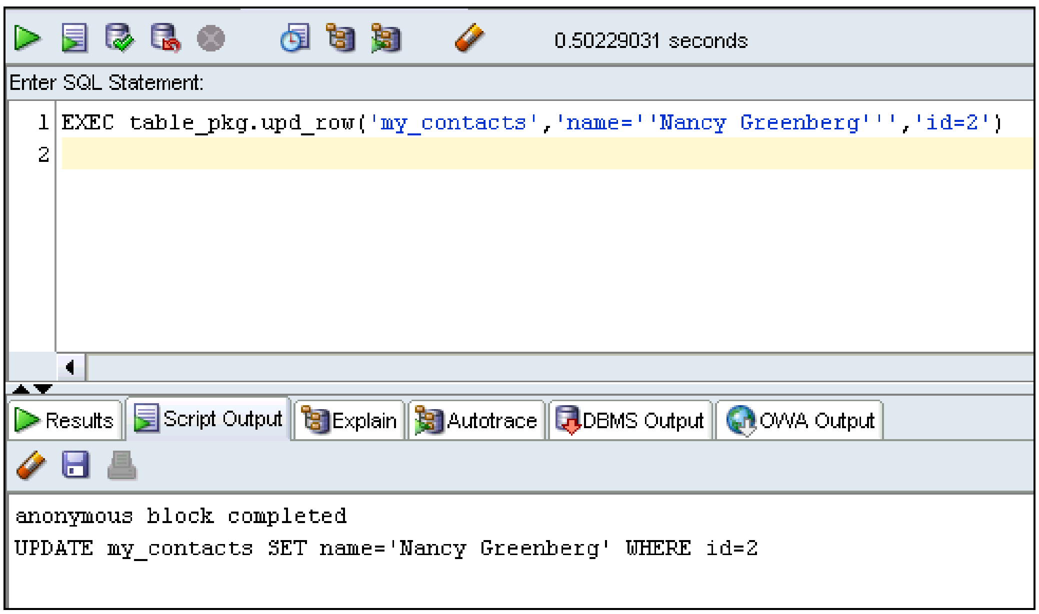
Practice 7: Using Native Dynamic SQL (continued)



h) Execute the `UPD_ROW` procedure with the following row data:

```
upd_row('my_contacts','name=''Nancy Greenberg''','id=2');
```

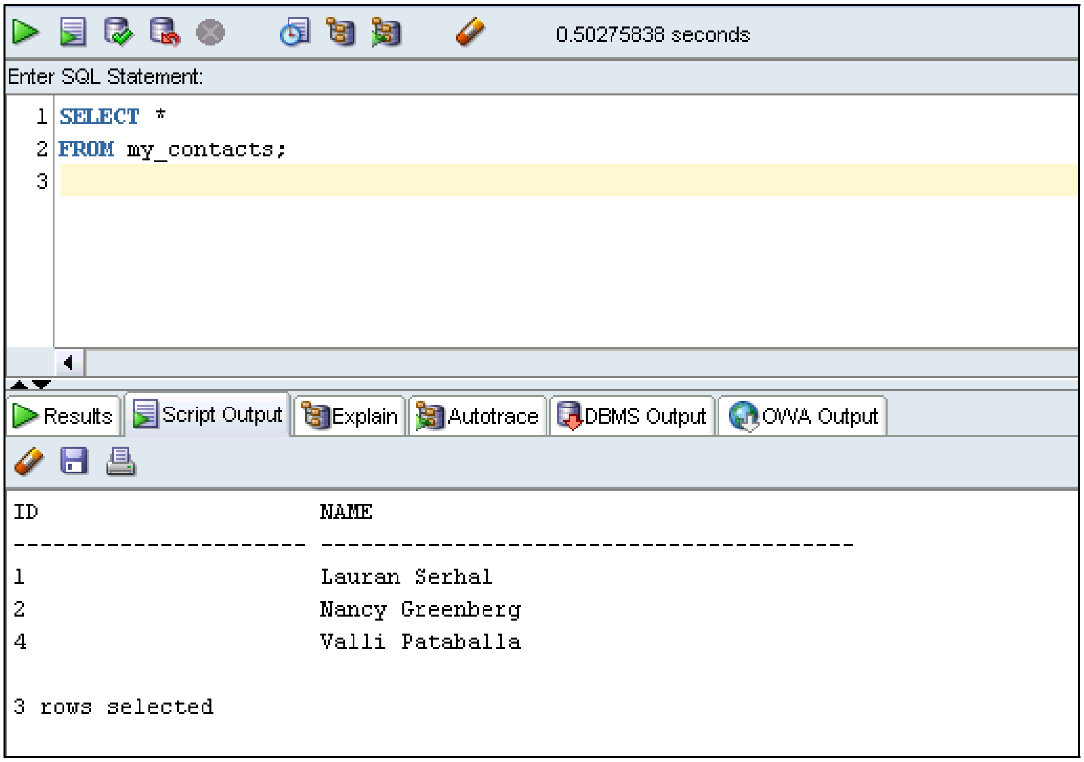
The code and result are shown below.



i) Query the `MY_CONTACTS` table contents to verify the changes.

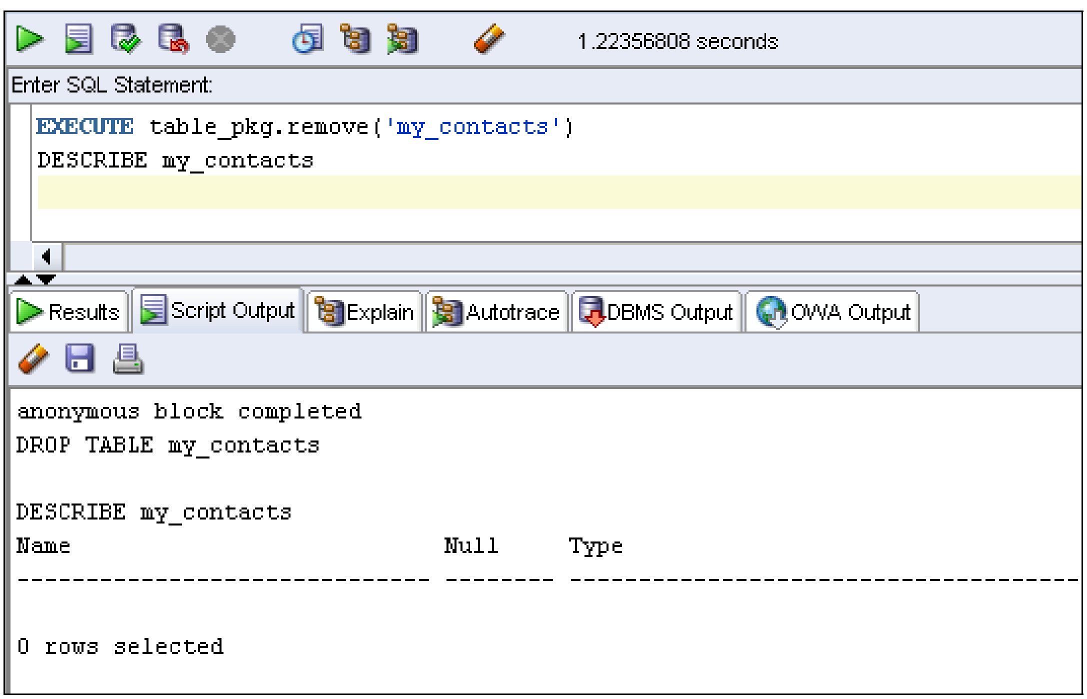
The code and result are shown below.

Practice 7: Using Native Dynamic SQL (continued)



j) Drop the table by using the remove procedure and describe the `MY_CONTACTS` table.

The code and result are shown below.



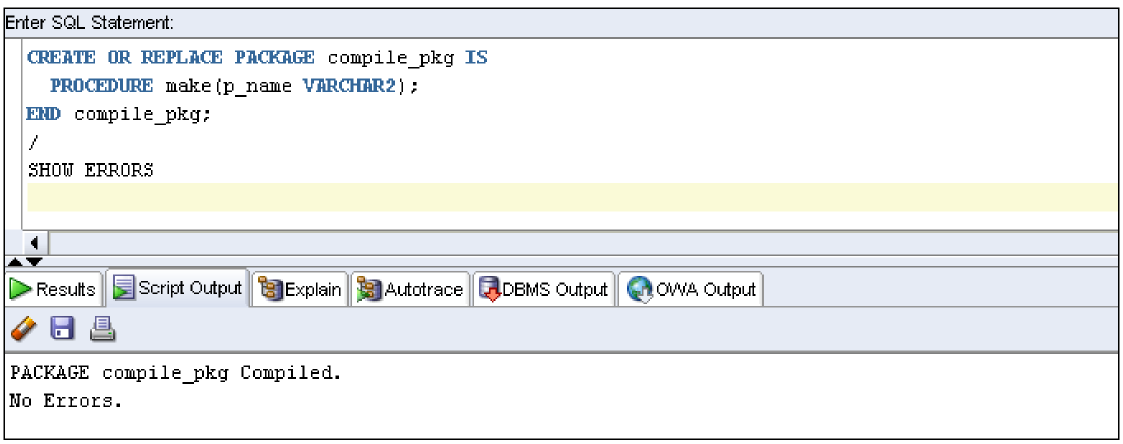
Practice 7: Using Native Dynamic SQL (continued)

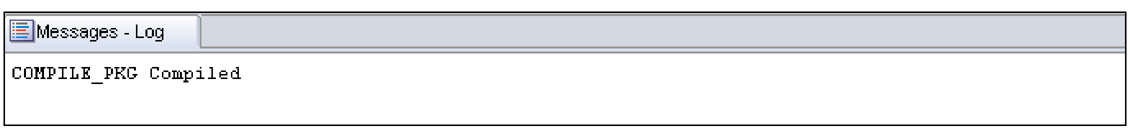
2) Create a `COMPILE_PKG` package that compiles the PL/SQL code in your schema.

   a) In the specification, create a package procedure called `MAKE` that accepts the name of a PL/SQL program unit to be compiled.

      Open the `sol_07_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The code and the results are shown below. To compile the package's specification, right-click the package's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  PROCEDURE make(p_name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS
```





   b) In the package body, include the following:

      i) The `EXECUTE` procedure used in the `TABLE_PKG` procedure in step 1 of this practice.

      ii) A private function named `GET_TYPE` to determine the PL/SQL object type from the data dictionary.

         - The function returns the type name (use `PACKAGE` for a package with a body) if the object exists; otherwise, it should return a `NULL`.

- In the WHERE clause condition, add the following to the condition to ensure that only one row is returned if the name represents a PACKAGE, which may also have a PACKAGE BODY. In this case, you can only compile the complete package, but not the specification or body as separate components:

  ```
  rownum = 1
  ```

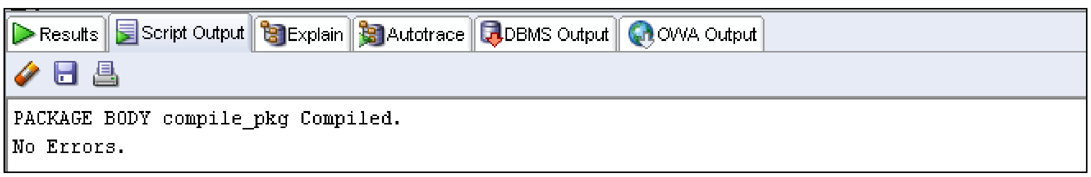
iii) Create the MAKE procedure by using the following information:

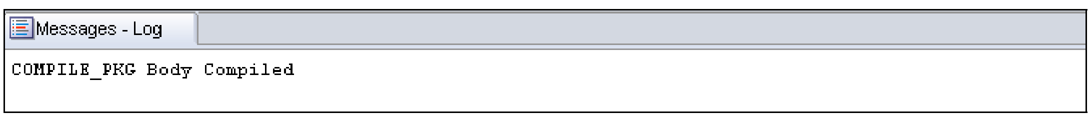
- The MAKE procedure accepts one argument, name, which represents the object name.
- The MAKE procedure should call the GET_TYPE function. If the object exists, MAKE dynamically compiles it with the ALTER statement.

Open the `sol_07_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package body. The code and the results are shown below. To compile the package's body, right-click the package's name or body in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE BODY compile_pkg IS

  PROCEDURE execute(p_stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_stmt);
    EXECUTE IMMEDIATE p_stmt;
  END;

  FUNCTION get_type(p_name VARCHAR2) RETURN VARCHAR2 IS
    v_proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
     */
    SELECT object_type INTO v_proc_type
    FROM user_objects
    WHERE object_name = UPPER(p_name)
    AND ROWNUM = 1;
    RETURN v_proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;
```

```
  PROCEDURE make(p_name VARCHAR2) IS
    v_stmt       VARCHAR2(100);
    v_proc_type  VARCHAR2(30) := get_type(p_name);
  BEGIN
    IF v_proc_type IS NOT NULL THEN
      v_stmt := 'ALTER '|| v_proc_type ||' '|| p_name ||'
COMPILE';
      execute(v_stmt);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Subprogram '''|| p_name ||''' does not exist');
    END IF;
  END make;
END compile_pkg;
/
SHOW ERRORS
```





c) Use the `COMPILE_PKG.MAKE` procedure to compile the following:

   i) The `EMPLOYEE_REPORT` procedure

   ii) The `EMP_PKG` package

   iii) A nonexistent object called `EMP_DATA`

   Open the `sol_07_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to execute the package's procedure. The code and the results are shown below.

```
EXECUTE compile_pkg.make('employee_report')
EXECUTE compile_pkg.make('emp_pkg')
EXECUTE compile_pkg.make('emp_data')
```

Practice 7: Using Native Dynamic SQL (continued)

```
anonymous block completed
ALTER PROCEDURE employee_report COMPILE

anonymous block completed
ALTER PACKAGE emp_pkg COMPILE


Error starting at line 3 in command:
EXECUTE compile_pkg.make('emp_data')
Error report:
ORA-20001: Subprogram 'emp_data' does not exist
ORA-06512: at "ORA61.COMPILE_PKG", line 39
ORA-06512: at line 1
```

Practice 8: Using Bulk Binding and Autonomous Transactions

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table. In addition, you create the `add_employee` procedure that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the `add_employee` procedure is called, whether it successfully adds a record or not.

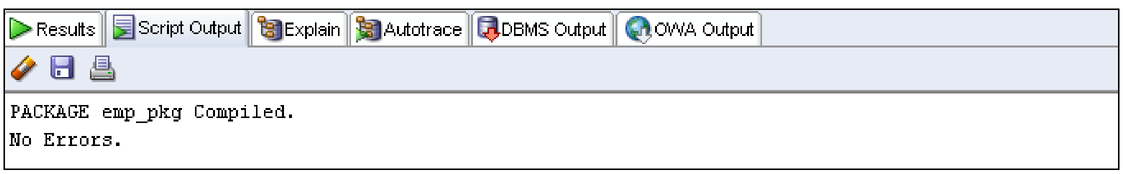
1) Update the `EMP_PKG` package with a new procedure to query employees in a specified department.
   a) In the package specification:
      i) Declare a `get_employees` procedure with a parameter called `dept_id`, which is based on the `employees.department_id` column type
      ii) Define an index-by PL/SQL type as a `TABLE OF EMPLOYEES%ROWTYPE`

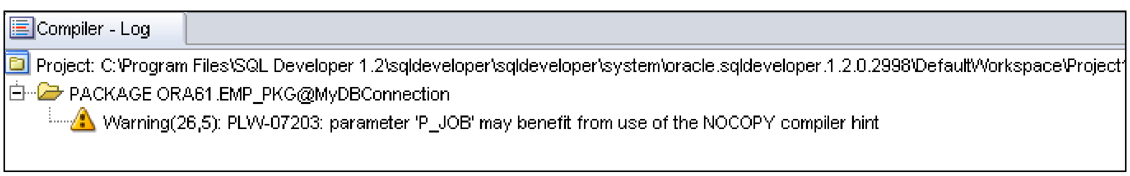
   Open the `sol_08_01_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The code and the results are shown below. The newly added code is highlighted in bold letters in the code box below. To compile the package's specification, right-click the package's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE emp_pkg IS

  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type);

  PROCEDURE init_departments;

  PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS
```





b) In the package body:

i) Define a private variable called `emp_table` based on the type defined in the specification to hold employee records

ii) Implement the `get_employees` procedure to bulk fetch the data into the table.

Open the `sol_08_01_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package body. The code and the results are shown below. The newly added code is highlighted in bold letters in the code box below. To compile the package's body, right-click the package's (or body) name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

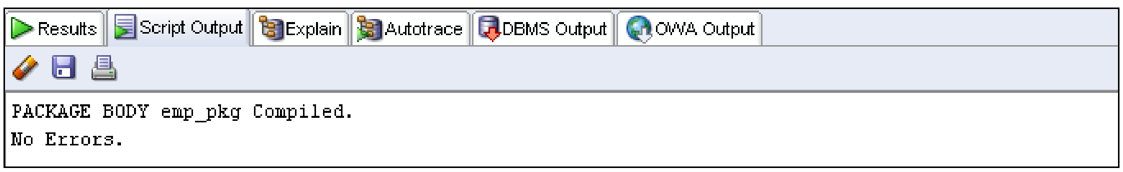
```
     INDEX BY BINARY_INTEGER;
  valid_departments boolean_tab_type;
  emp_table         emp_tab_type;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN

      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
```

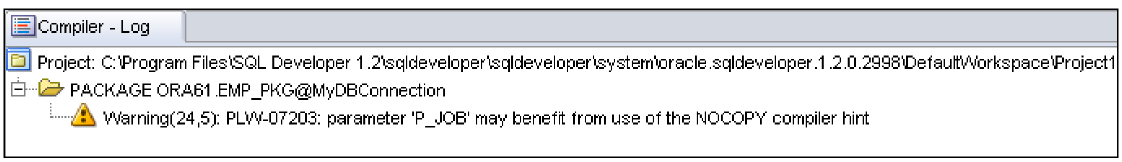
Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
  END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS

rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
END;

/* New get_employees procedure. */

PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
  BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
  END;

PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

BEGIN
  init_departments;

END emp_pkg;

/
SHOW ERRORS
```





c) Create a new procedure in the specification and body, called `show_employees`, that does not take arguments. The procedure displays the contents of the private PL/SQL table variable (if any data exists). Use the `print_employee` procedure that you created in an earlier practice. To view the results, click the Enable DBMS Output icon in the DBMS Output tab in SQL Developer, if you have not already done so.

Open the `sol_08_01_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package with the new procedure. The code and the results are shown below. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

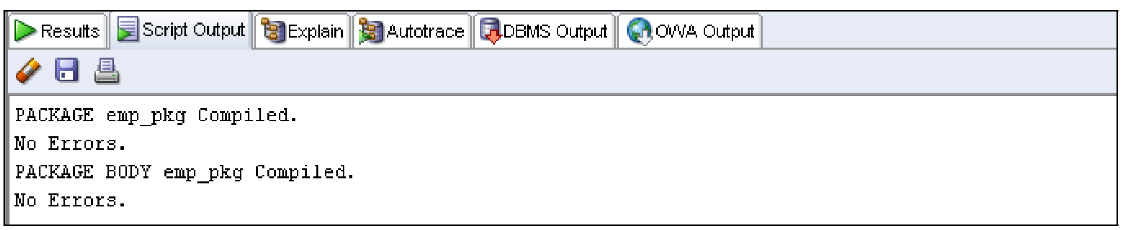
```
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type);

  PROCEDURE init_departments;

  PROCEDURE print_employee(p_rec_emp employees%rowtype);

  PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
     INDEX BY BINARY_INTEGER;

  valid_departments boolean_tab_type;
  emp_table         emp_tab_type;
  FUNCTION valid_deptid(p_deptid IN
   departments.department_id%TYPE)
```

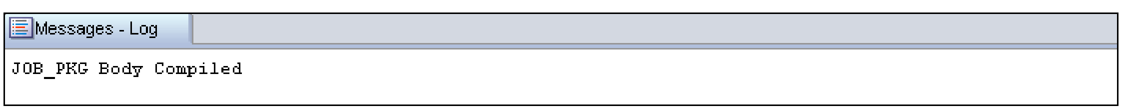
```
    RETURN BOOLEAN;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
```

```
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
  BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
  END;

  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

  PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;



  PROCEDURE show_employees IS
  BEGIN
    IF emp_table IS NOT NULL THEN
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
      DBMS_OUTPUT.PUT_LINE('Employees in Package table');
      FOR i IN 1 .. emp_table.COUNT
      LOOP
        print_employee(emp_table(i));
      END LOOP;
    END IF;
  END show_employees;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
   RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;


BEGIN
  init_departments;
END emp_pkg;

/
SHOW ERRORS
```





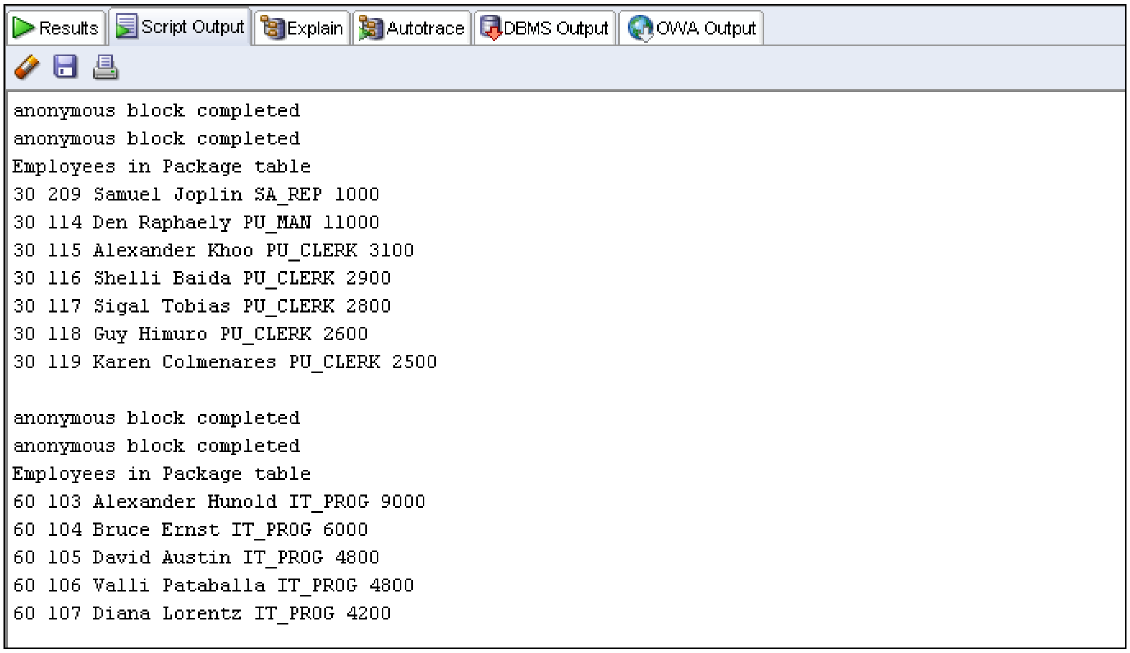
d) Invoke the `emp_pkg.get_employees` procedure for department `30`, and then invoke `emp_pkg.show_employees`. Repeat this for department `60`.

Open the `sol_08_01_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedures. The code and the results are shown below:

```
EXECUTE emp_pkg.get_employees(30)
EXECUTE emp_pkg.show_employees
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
EXECUTE emp_pkg.get_employees(60)
EXECUTE emp_pkg.show_employees
```

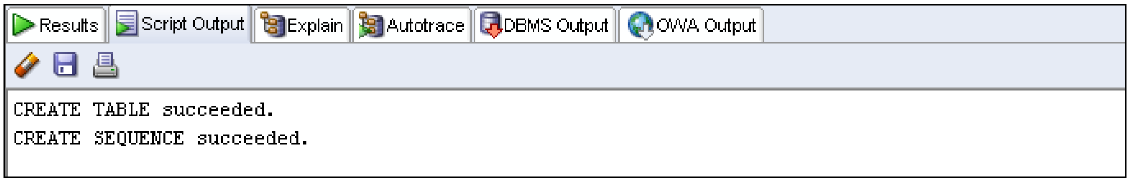


2) Your manager wants to keep a log whenever the `add_employee` procedure in the package is invoked to insert a new employee into the `EMPLOYEES` table.

   a) First, load and execute the `D:\labs\PLPU\solns\sol_08_02_a.sql` script to create a log table called `LOG_NEWEMP`, and a sequence called `log_newemp_seq`.

      Open the `sol_08_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE TABLE log_newemp (
  entry_id  NUMBER(6) CONSTRAINT log_newemp_pk PRIMARY KEY,
  user_id   VARCHAR2(30),
  log_time  DATE,
  name      VARCHAR2(60)
);


CREATE SEQUENCE log_newemp_seq;
```



b) In the `EMP_PKG` package body, modify the `add_employee` procedure, which performs the actual `INSERT` operation. Add a local procedure called `audit_newemp` as follows:

   i) The `audit_newemp` procedure must use an autonomous transaction to insert a log record into the `LOG_NEWEMP` table.

   ii) Store the `USER`, the current time, and the new employee name in the log table row.

   iii) Use `log_newemp_seq` to set the `entry_id` column.

   Note: Remember to perform a `COMMIT` operation in a procedure with an autonomous transaction.

   Open the `sol_08_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The newly added code is highlighted in bold letters in the following code box. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS


  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type);

  PROCEDURE init_departments;

  PROCEDURE print_employee(p_rec_emp employees%rowtype);

  PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
     INDEX BY BINARY_INTEGER;

  valid_departments boolean_tab_type;
  emp_table         emp_tab_type;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
    RETURN BOOLEAN;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS

-- New local procedure
```

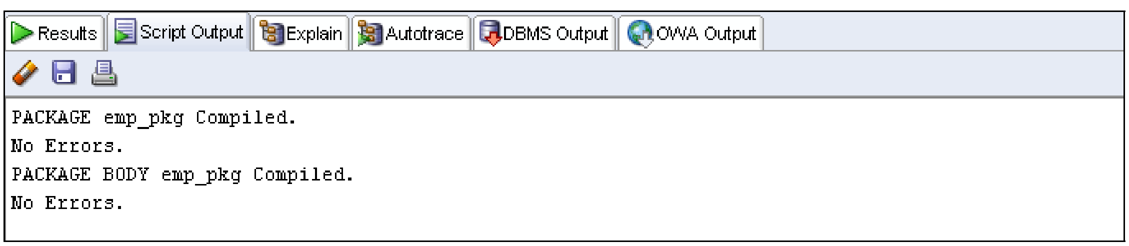
Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

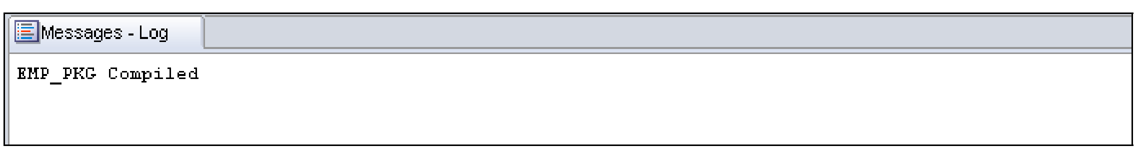
```
    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time,
                              name)
      VALUES (log_newemp_seq.NEXTVAL, user_id,
              sysdate,p_first_name||' '||p_last_name);
      COMMIT;
    END audit_newemp;

  BEGIN -- add_employee
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

/* New get_employees procedure. */

  PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
  BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
  END;

  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

  PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;

  PROCEDURE show_employees IS
  BEGIN
```

```
    IF emp_table IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employees in Package table');
      FOR i IN 1 .. emp_table.COUNT
      LOOP
        print_employee(emp_table(i));
      END LOOP;
    END IF;
  END show_employees;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
   RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;


BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
```





c) Modify the `add_employee` procedure to invoke `audit_emp` before it performs the insert operation.

Open the `sol_08_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. The newly added code is highlighted in bold letters in the following code box. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The

code and the results are shown below. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS


  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type);

  PROCEDURE init_departments;

  PROCEDURE print_employee(p_rec_emp employees%rowtype);

  PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

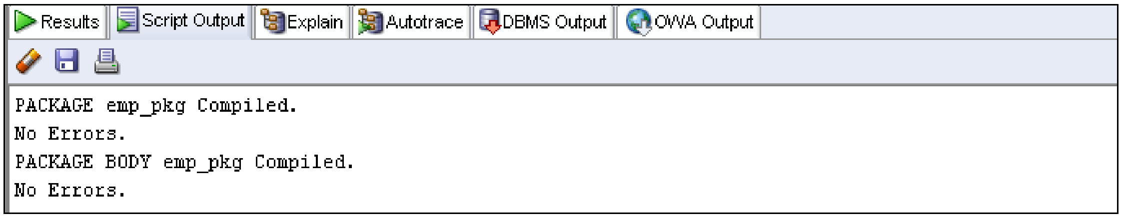
```
-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
     INDEX BY BINARY_INTEGER;

  valid_departments boolean_tab_type;
  emp_table         emp_tab_type;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
    RETURN BOOLEAN;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS

    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time,
name)
      VALUES (log_newemp_seq.NEXTVAL, user_id,
sysdate,p_first_name||' '||p_last_name);
      COMMIT;
    END audit_newemp;

  BEGIN -- add_employee
    IF valid_deptid(p_deptid) THEN
      audit_newemp;
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
  END add_employee;
```

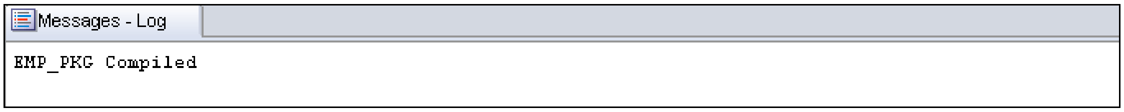
Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;

END;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
  BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
  END;

  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

  PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;

  PROCEDURE show_employees IS
  BEGIN
    IF emp_table IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employees in Package table');
      FOR i IN 1 .. emp_table.COUNT
      LOOP
        print_employee(emp_table(i));
      END LOOP;
    END IF;
  END show_employees;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
   RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN

    RETURN FALSE;
END valid_deptid;
BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
```

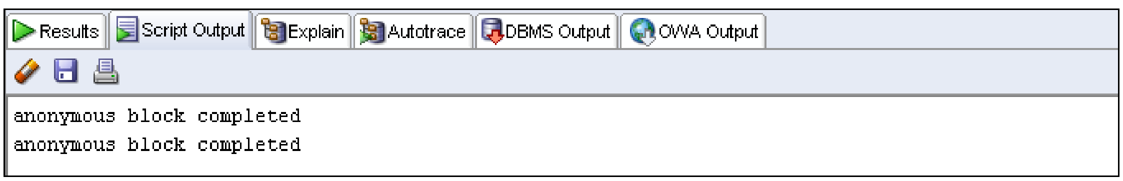




d) Invoke the `add_employee` procedure for these new employees: `Max Smart` in department 20 and `Clark Kent` in department 10. What happens?

Open the `sol_08_02_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE emp_pkg.add_employee('Max', 'Smart', 20)
EXECUTE emp_pkg.add_employee('Clark', 'Kent', 10)
```



Both insert statements complete successfully. The log table has two log records as shown in the next step.

e) Query the two `EMPLOYEES` records added, and the records in the `LOG_NEWEMP` table. How many log records are present?

Open the `sol_08_02_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
select department_id, employee_id, last_name, first_name
from employees
where last_name in ('Kent', 'Smart');

select * from log_newemp;
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)

```
DEPARTMENT_ID          EMPLOYEE_ID            LAST_NAME                 FIRST_NAME
---------------------- ---------------------- ------------------------- --------------------
10                     212                    Kent                      Clark
20                     211                    Smart                     Max

2 rows selected

ENTRY_ID               USER_ID                        LOG_TIME                  NAME
---------------------- ------------------------------ ------------------------- ----------------
1                      ORA61                          22-JUN-07                 Max Smart
2                      ORA61                          22-JUN-07                 Clark Kent

2 rows selected
```

There are two log records, one for `Smart` and another for `Kent`.

f) Execute a `ROLLBACK` statement to undo the insert operations that have not been committed. Use the same queries from step 2 e. as follows:

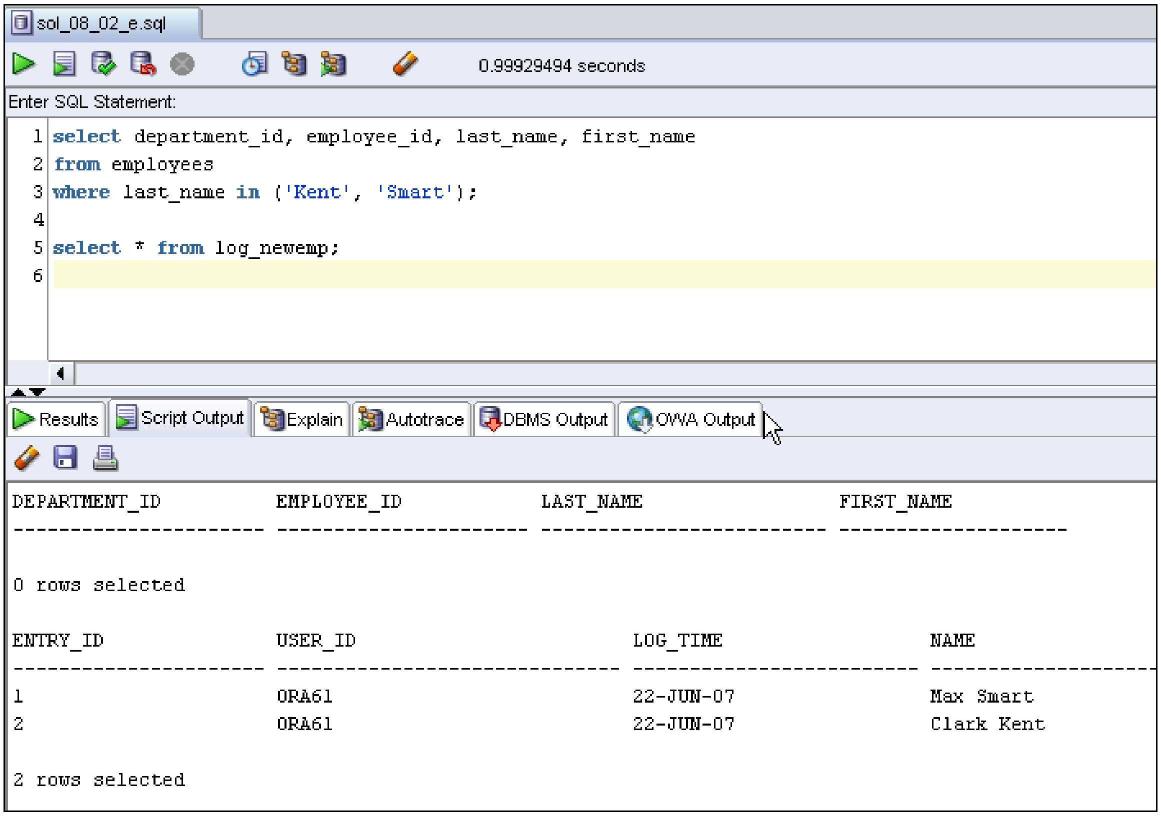
   i) Use the first query to check whether the employee rows for `Smart` and `Kent` have been removed.

   ii) Use the second query to check the log records in the `LOG_NEWEMP` table. How many log records are present? Why?

```
ROLLBACK;
```

```
ROLLBACK succeeded.
```

Practice 8: Using Bulk Binding and Autonomous Transactions (continued)



The two employee records are removed (rolled back). The two log records remain in the log table because they were inserted using an autonomous transaction, which is unaffected by the rollback performed in the main transaction.

Practice 9: Creating Statement and Row Triggers

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

1) The rows in the JOBS table store a minimum and maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.

   a) Create a procedure called CHECK_SALARY as follows:

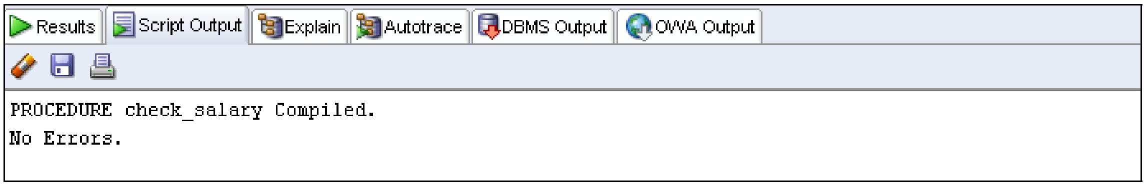
      i) The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.

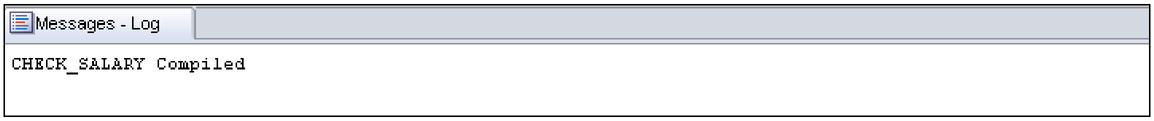
      ii) The procedure uses the job ID to determine the minimum and maximum salary for the specified job.

      iii) If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>". Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.

   Open the sol_09_01_a.sql file in the D:\labs\PLPU\solns folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the procedure, right-click the procedure's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN
  SELECT min_salary, max_salary INTO v_minsal, v_maxsal
  FROM jobs
  WHERE job_id = UPPER(p_the_job);
  IF p_the_salary NOT BETWEEN v_minsal AND v_maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $' ||p_the_salary ||'. '||
      'Salaries for job '|| p_the_job ||
      ' must be between $'|| v_minsal ||' and $' || v_maxsal);
  END IF;
END;
/
SHOW ERRORS
```





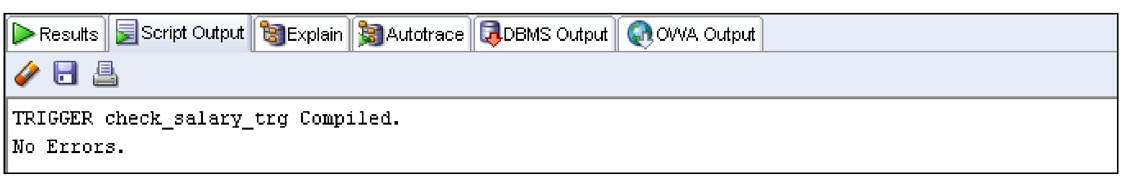
b) Create a trigger called `CHECK_SALARY_TRG` on the `EMPLOYEES` table that fires before an `INSERT` or `UPDATE` operation on each row:

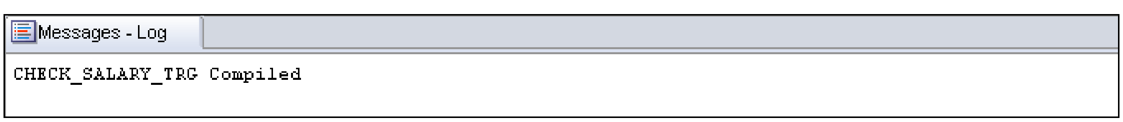
   i) The trigger must call the `CHECK_SALARY` procedure to carry out the business logic.

   ii) The trigger should pass the new job ID and salary to the procedure parameters.

   Open the `sol_09_01_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the trigger, right-click the trigger's name in the Object Navigation tree, and then select Compile.

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees
FOR EACH ROW
BEGIN
  check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```





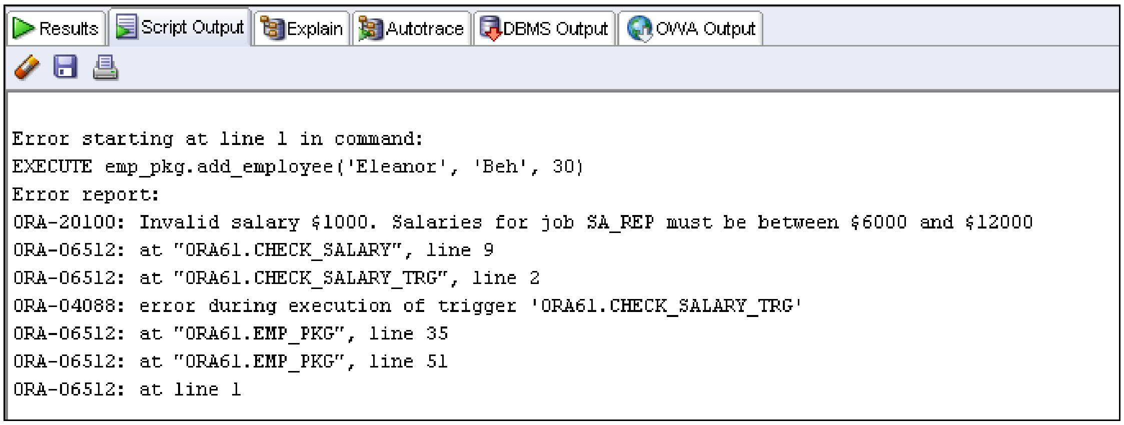
2) Test the `CHECK_SAL_TRG` trigger using the following cases:

Practice 9: Creating Statement and Row Triggers (continued)

a) Using your `EMP_PKG.ADD_EMPLOYEE` procedure, add employee `Eleanor Beh` to department 30. What happens and why?

Open the `sol_09_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)
```




```
Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)
Error report:
ORA-20100: Invalid salary $1000. Salaries for job SA_REP must be between $6000 and $12000
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
ORA-06512: at "ORA61.EMP_PKG", line 35
ORA-06512: at "ORA61.EMP_PKG", line 51
ORA-06512: at line 1
```

The trigger raises an exception because the `EMP_PKG.ADD_EMPLOYEE` procedure invokes an overloaded version of itself that uses the default salary of $1,000 and a default job ID of `SA_REP`. However, the `JOBS` table stores a minimum salary of $ 6,000 for the `SA_REP` type.

b) Update the salary of employee 115 to $2,000. In a separate update operation, change the employee job ID to `HR_REP`. What happens in each case?

Open the `sol_09_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package, right-click the package's name in the Object Navigation tree, and then select Compile.

```
UPDATE employees
  SET salary = 2000
WHERE employee_id = 115;

UPDATE employees
  SET job_id = 'HR_REP'
WHERE employee_id = 115;
```

Practice 9: Creating Statement and Row Triggers (continued)

```
Error starting at line 1 in command:
UPDATE employees
  SET salary = 2000
WHERE employee_id = 115
Error report:
SQL Error: ORA-20100: Invalid salary $2000. Salaries for job PU_CLERK must be between $2500 and $5500
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'


Error starting at line 5 in command:
UPDATE employees
  SET job_id = 'HR_REP'
WHERE employee_id = 115
Error report:
SQL Error: ORA-20100: Invalid salary $3100. Salaries for job HR_REP must be between $4000 and $9000
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
```

The first update statement fails to set the salary to $2,000. The check salary trigger rule fails the update operation because the new salary for employee 115 is less than the minimum allowed for the `PU_CLERK` job ID.

The second update fails to change the employee's job because the current employee's salary of $3,100 is less than the minimum for the new `HR_REP` job ID.

c) Update the salary of employee 115 to $2,800. What happens?

Open the `sol_09_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  SET salary = 2800
WHERE employee_id = 115;
```

```
1 rows updated
```

The update operation is successful because the new salary falls within the acceptable range for the current job ID.

Practice 9: Creating Statement and Row Triggers (continued)

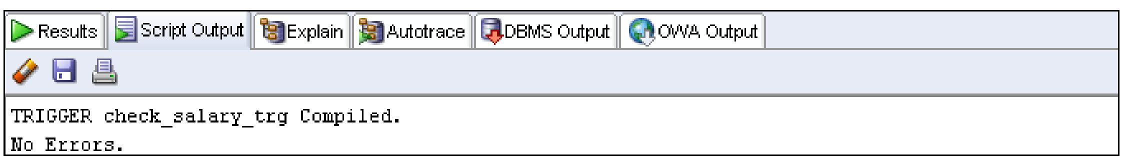
3) Update the `CHECK_SALARY_TRG` trigger to fire only when the job ID or salary values have actually changed.

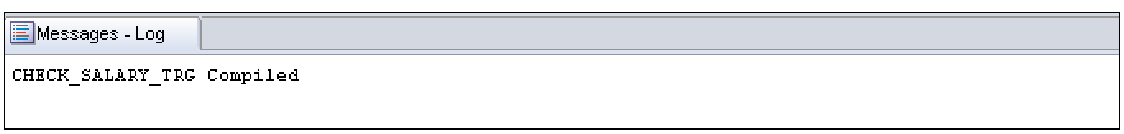
   a) Implement the business rule using a `WHEN` clause to check whether the `JOB_ID` or `SALARY` values have changed.

      Note: Make sure that the condition handles the `NULL` in the `OLD.column_name` values if an `INSERT` operation is performed; otherwise, an insert operation will fail.

      Open the `sol_09_03_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the trigger, right-click the trigger's name in the Object Navigation tree, and then click Compile.

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees FOR EACH ROW
WHEN (new.job_id <> NVL(old.job_id,'?') OR
      new.salary <> NVL(old.salary,0))
BEGIN
  check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```



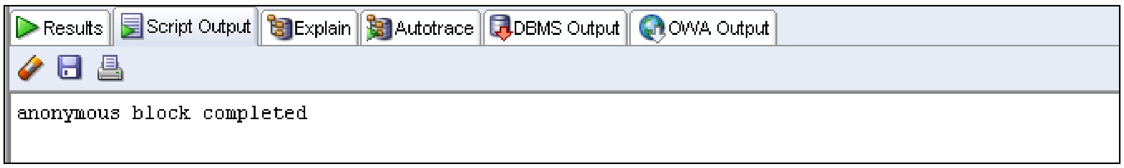


   b) Test the trigger by executing the `EMP_PKG.ADD_EMPLOYEE` procedure with the following parameter values:

      - `p_first_name: 'Eleanor'`
      - `p_last name: 'Beh'`
      - `p_Email: 'EBEH'`
      - `p_Job: 'IT_PROG'`
      - `p_Sal: 5000`

Open the `sol_09_03_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

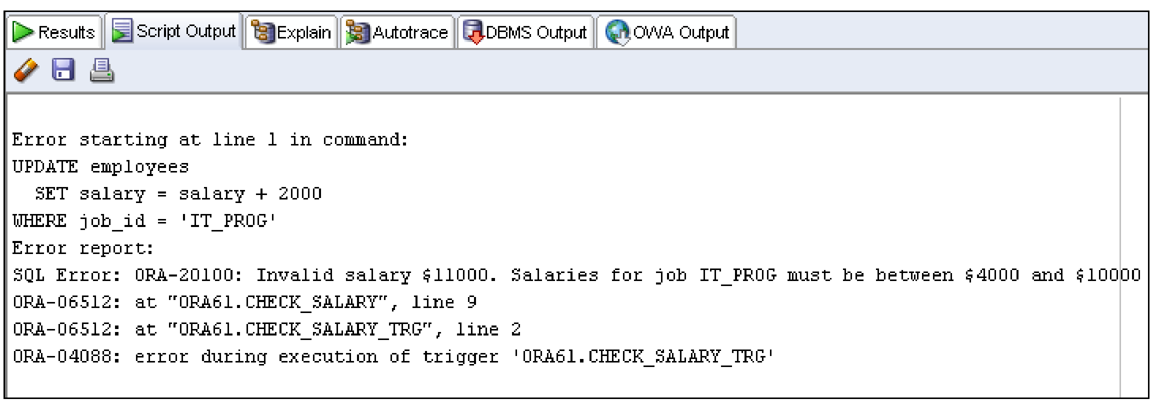
```
BEGIN
  emp_pkg.add_employee('Eleanor', 'Beh', 'EBEH',
                       job => 'IT_PROG', sal => 5000);
END;
/
```



c) Update employees with the `IT_PROG` job by incrementing their salary by $2,000. What happens?

Open the `sol_09_03_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  SET salary = salary + 2000
WHERE job_id = 'IT_PROG';
```

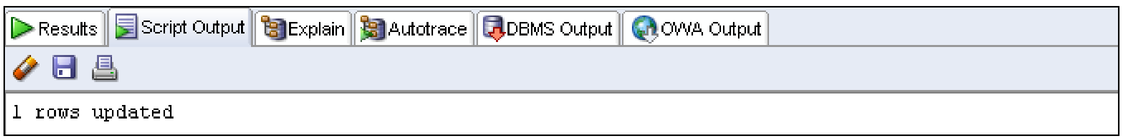


An employee's salary in the specified job type exceeds the maximum salary for that job type. No employee salaries in the `IT_PROG` job type are updated.

d) Update the salary to $9,000 for `Eleanor Beh`.

Open the `sol_09_03_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  SET salary = 9000
WHERE employee_id = (SELECT employee_id
                     FROM employees
                     WHERE last_name = 'Beh');
```



Hint: Use an `UPDATE` statement with a subquery in the `WHERE` clause. What happens?

e) Change the job of `Eleanor Beh` to `ST_MAN` using another `UPDATE` statement with a subquery. What happens?

Open the `sol_09_03_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  set job_id = 'ST_MAN'
WHERE employee_id = (SELECT employee_id
                     FROM employees
                     WHERE last_name = 'Beh');
```

```
Error starting at line 1 in command:
UPDATE employees
  set job_id = 'ST_MAN'
WHERE employee_id = (SELECT employee_id
                     FROM employees
                     WHERE last_name = 'Beh')
Error report:
SQL Error: ORA-20100: Invalid salary $9000. Salaries for job ST_MAN must be between $5500 and $8500
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
```

The maximum salary of the new job type is less than the employee's current salary; therefore, the update operation fails.

4) You are asked to prevent employees from being deleted during business hours.

   a) Write a statement trigger called `DELETE_EMP_TRG` on the `EMPLOYEES` table to prevent rows from being deleted during weekday business hours, which are from 9:00 AM to 6:00 PM.

   Open the `sol_09_04_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the trigger, right-click the trigger's name in the Object Navigation tree, and then click Compile.

```
CREATE OR REPLACE TRIGGER delete_emp_trg
BEFORE DELETE ON employees
DECLARE
  the_day VARCHAR2(3)  := TO_CHAR(SYSDATE, 'DY');
  the_hour PLS_INTEGER := TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'));
BEGIN
   IF (the_hour BETWEEN 9 AND 18) AND (the_day NOT IN
('SAT','SUN')) THEN
     RAISE_APPLICATION_ERROR(-20150,
      'Employee records cannot be deleted during the business
       hours of 9AM and 6PM');
   END IF;
END;
/
SHOW ERRORS
```

```
TRIGGER delete_emp_trg Compiled.
No Errors.
```

Practice 9: Creating Statement and Row Triggers (continued)

Messages - Log

```
DELETE_EMP_TRG Compiled
```

b) Attempt to delete employees with `JOB_ID` of `SA_REP` who are not assigned to a department.

Hint: This is employee Grant with ID 178.

Open the `sol_09_04_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the trigger, right-click the trigger's name in the Object Navigation tree, and then click Compile.

```
DELETE FROM employees
WHERE job_id = 'SA_REP'
  AND   department_id IS NULL;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
Error starting at line 1 in command:
DELETE FROM employees
  WHERE job_id = 'SA_REP'
  AND   department_id IS NULL
Error report:
SQL Error: ORA-20150: Employee records cannot be deleted during the business hours of 9AM and 6PM
ORA-06512: at "ORA61.DELETE_EMP_TRG", line 6
ORA-04088: error during execution of trigger 'ORA61.DELETE_EMP_TRG'
```

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the `JOBS` table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

1) Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salaries. Implement this requirement through a package procedure called by a trigger on the `JOBS` table. When you attempt to update the minimum salary in the `JOBS` table and try to update the employees' salaries, the `CHECK_SALARY` trigger attempts to read the `JOBS` table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.

    a. Update your `EMP_PKG` package (that you last updated in Practice 8) as follows:

        i. Add a procedure called `SET_SALARY` that updates the employees' salaries.

        ii. The `SET_SALARY` procedure accepts the following two parameters: The job ID for those salaries that may have to be updated, and the new minimum salary for the job ID

    Open the `sol_10_01_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown as follows. To compile the trigger, right-click the package's name in the Object Navigation tree, and then click Compile. The newly added code is highlighted in bold letters in the following code box.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS


  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
```

```
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type);

  PROCEDURE init_departments;

  PROCEDURE print_employee(p_rec_emp employees%rowtype);

  PROCEDURE show_employees;

  /* New set_salary procedure */

  PROCEDURE set_salary(p_jobid VARCHAR2, p_min_salary NUMBER);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tab_type IS TABLE OF BOOLEAN
     INDEX BY BINARY_INTEGER;

  valid_departments boolean_tab_type;
  emp_table         emp_tab_type;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
    RETURN BOOLEAN;
```

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

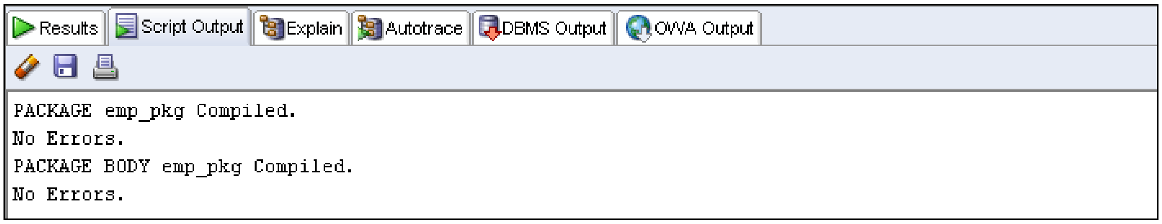
```
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS

    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time,
name)
      VALUES (log_newemp_seq.NEXTVAL, user_id,
sysdate,p_first_name||' '||p_last_name);
      COMMIT;
    END audit_newemp;

  BEGIN -- add_employee
    IF valid_deptid(p_deptid) THEN
      audit_newemp;
      INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
  BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
  END;

  PROCEDURE get_employee(
```

```
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
  END get_employee;

  FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p_emp_id;
    RETURN rec_emp;
  END;

  FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
  BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
  END;

  PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
  BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
  END;

  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

  PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id ||' '||
                         p_rec_emp.employee_id||' '||
```

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
                         p_rec_emp.first_name||' '||
                         p_rec_emp.last_name||' '||
                         p_rec_emp.job_id||' '||
                         p_rec_emp.salary);
  END;

  PROCEDURE show_employees IS
  BEGIN
    IF emp_table IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employees in Package table');
      FOR i IN 1 .. emp_table.COUNT
      LOOP
        print_employee(emp_table(i));
      END LOOP;
    END IF;
  END show_employees;

  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
   RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;

/* New set_salary procedure */

PROCEDURE set_salary(p_jobid VARCHAR2, p_min_salary NUMBER) IS
    CURSOR cur_emp IS
      SELECT employee_id
      FROM employees
      WHERE job_id = p_jobid AND salary < p_min_salary;
  BEGIN
    FOR rec_emp IN cur_emp
    LOOP
      UPDATE employees
        SET salary = p_min_salary
      WHERE employee_id = rec_emp.employee_id;
    END LOOP;
  END set_salary;

BEGIN
  init_departments;
END emp_pkg;


/
SHOW ERRORS
```

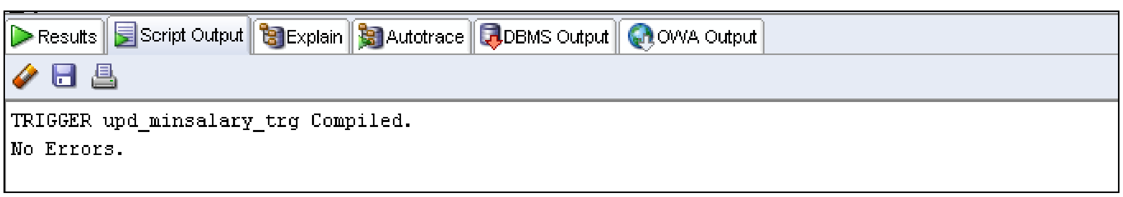
Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)



b. Create a row trigger named `UPD_MINSALARY_TRG` on the `JOBS` table that invokes the `EMP_PKG.SET_SALARY` procedure, when the minimum salary in the `JOBS` table is updated for a specified job ID.

Open the `sol_10_01_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the trigger, right-click the trigger's name in the Object Navigation tree, and then click Compile. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER upd_minsalary_trg
AFTER UPDATE OF min_salary ON JOBS
FOR EACH ROW
BEGIN
  emp_pkg.set_salary(:new.job_id, :new.min_salary);
END;
/
SHOW ERRORS
```



c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their `JOB_ID` is `'IT_PROG'`. Then, update the minimum salary in the `JOBS` table to increase it by $1,000. What happens?

Open the `sol_10_01_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
  SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG';
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
EMPLOYEE_ID            LAST_NAME                SALARY
---------------------- ------------------------ ----------------------
103                    Hunold                   9000
104                    Ernst                    6000
105                    Austin                   4800
106                    Pataballa                4800
107                    Lorentz                  4200
214                    Beh                      9000

6 rows selected


Error starting at line 5 in command:
UPDATE jobs
  SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG'
Error report:
SQL Error: ORA-04091: table ORA61.JOBS is mutating, trigger/function may not see it
ORA-06512: at "ORA61.CHECK_SALARY", line 5
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
ORA-06512: at "ORA61.EMP_PKG", line 143
ORA-06512: at "ORA61.UPD_MINSALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.UPD_MINSALARY_TRG'
04091. 00000 -  "table %s.%s is mutating, trigger/function may not see it"
*Cause:    A trigger (or a user defined plsql function that is referenced in
           this statement) attempted to look at (or modify) a table that was
           in the middle of being modified by the statement which fired it.
*Action:   Rewrite the trigger (or function) so it does not read that table.
```

The update of the `min_salary` column for job `'IT_PROG'` fails because the `UPD_MINSALARY_TRG` trigger on the `JOBS` table attempts to update the employees' salaries by calling the `EMP_PKG.SET_SALARY` procedure. The `SET_SALARY` procedure causes the `CHECK_SALARY_TRG` trigger to fire (a cascading effect). The `CHECK_SALARY_TRG` calls the `CHECK_SALARY` procedure, which attempts to read the `JOBS` table data, this encountering the mutating table exception on the `JOBS` table, which is the table that is subject to the original update operation.

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

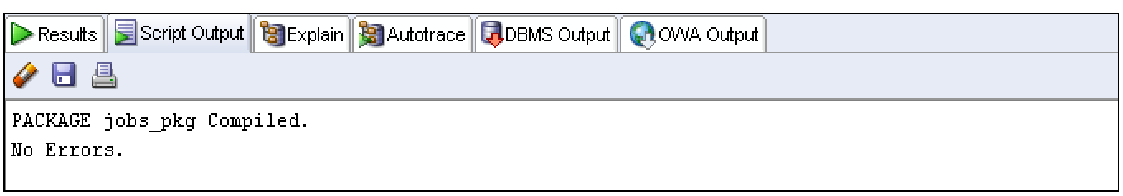
2) To resolve the mutating table issue, create a `JOBS_PKG` package to maintain in memory a copy of the rows in the `JOBS` table. Next, modify the `CHECK_SALARY` procedure to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, you must create a `BEFORE INSERT OR UPDATE` statement trigger on the `EMPLOYEES` table to initialize the `JOBS_PKG` package state before the `CHECK_SALARY` row trigger is fired.

   a. Create a new package called `JOBS_PKG` with the following specification:

```
PROCEDURE initialize;
FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(jobid VARCHAR2,min_salary
                        NUMBER);
PROCEDURE set_maxsalary(jobid VARCHAR2,max_salary
                       NUMBER);
```

   Open the `sol_10_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package's specification, right-click the package's name or body in the Object Navigator tree, and then Select Compile.

```
CREATE OR REPLACE PACKAGE jobs_pkg IS
  PROCEDURE initialize;
  FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER;
  FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER;
  PROCEDURE set_minsalary(p_jobid VARCHAR2, p_min_salary
NUMBER);
  PROCEDURE set_maxsalary(p_jobid VARCHAR2, p_max_salary
NUMBER);
END jobs_pkg;
/
SHOW ERRORS
```



   b. Implement the body of `JOBS_PKG` as follows:

      i. Declare a private PL/SQL index-by table called `jobs_tab_type` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.

ii. Declare a private variable called `jobstab` based on the `jobs_tab_type`.

iii. The `INITIALIZE` procedure reads the rows in the `JOBS` table by using a cursor loop, and uses the `JOB_ID` value for the `jobstab` index that is assigned its corresponding row.

iv. The `GET_MINSALARY` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `min_salary` for that element.

v. The `GET_MAXSALARY` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `max_salary` for that element.

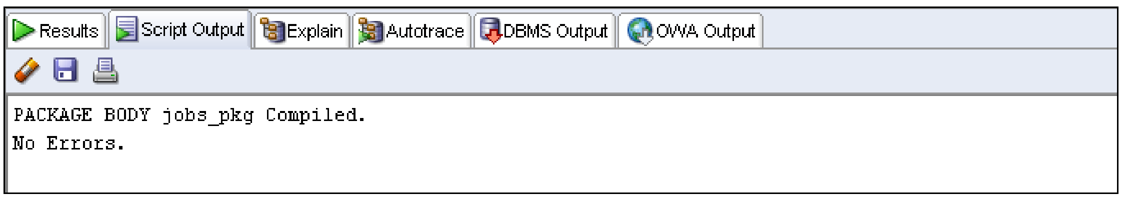
vi. The `SET_MINSALARY` procedure uses its `p_jobid` as an index to the `jobstab` to set the `min_salary` field of its element to the value in the `min_salary` parameter.

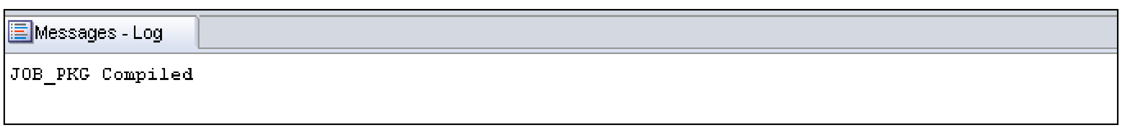
vii. The `SET_MAXSALARY` procedure uses its `p_jobid` as an index to the `jobstab` to set the `max_salary` field of its element to the value in the `max_salary` parameter.

Open the `sol_10_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package's body, right-click the package's name or body in the Object Navigator tree, and then Select Compile.

```
CREATE OR REPLACE PACKAGE BODY jobs_pkg IS
  TYPE jobs_tab_type IS TABLE OF jobs%rowtype
    INDEX BY jobs.job_id%type;
  jobstab jobs_tab_type;

  PROCEDURE initialize IS
  BEGIN
    FOR rec_job IN (SELECT * FROM jobs)
    LOOP
      jobstab(rec_job.job_id) := rec_job;
    END LOOP;
  END initialize;

  FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(p_jobid).min_salary;
  END get_minsalary;

  FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(p_jobid).max_salary;
  END get_maxsalary;
```

```
  PROCEDURE set_minsalary(p_jobid VARCHAR2, p_min_salary
NUMBER) IS
  BEGIN
    jobstab(p_jobid).max_salary := p_min_salary;
  END set_minsalary;

  PROCEDURE set_maxsalary(p_jobid VARCHAR2, p_max_salary
NUMBER) IS
  BEGIN
    jobstab(p_jobid).max_salary := p_max_salary;
  END set_maxsalary;

END jobs_pkg;
/
SHOW ERRORS
```





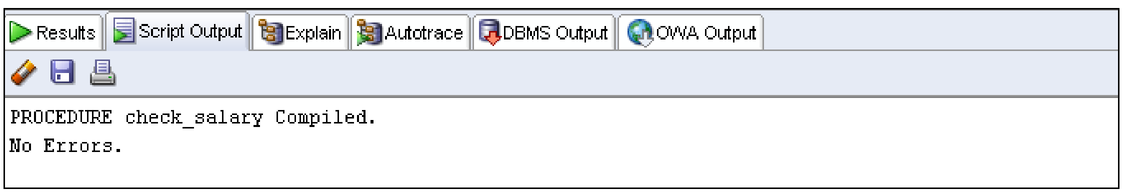
c. Copy the `CHECK_SALARY` procedure from Practice 10, Exercise 1a, and modify the code by replacing the query on the `JOBS` table with statements to set the local `minsal` and `maxsal` variables with values from the `JOBS_PKG` data by calling the appropriate `GET_*SALARY` functions. This step should eliminate the mutating trigger exception.

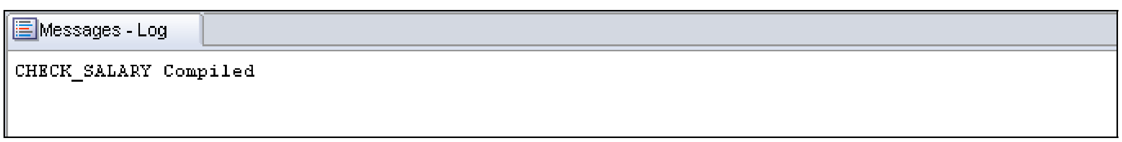
Open the `sol_10_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the procedure, right-click the procedure's name in the Object Navigator, and then select Compile.

```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN
  /*
```

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
  ** Commented out to avoid mutating trigger exception on the
JOBS table
  SELECT min_salary, max_salary INTO v_minsal, v_maxsal
  FROM jobs
  WHERE job_id = UPPER(p_the_job);
  */
  v_minsal := jobs_pkg.get_minsalary(UPPER(p_the_job));
  v_maxsal := jobs_pkg.get_maxsalary(UPPER(p_the_job));
  IF p_the_salary NOT BETWEEN v_minsal AND v_maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $'||p_the_salary||'. '||
      'Salaries for job '|| p_the_job ||
      ' must be between $'|| v_minsal ||' and $' || v_maxsal);
  END IF;
END;
/
SHOW ERRORS
```

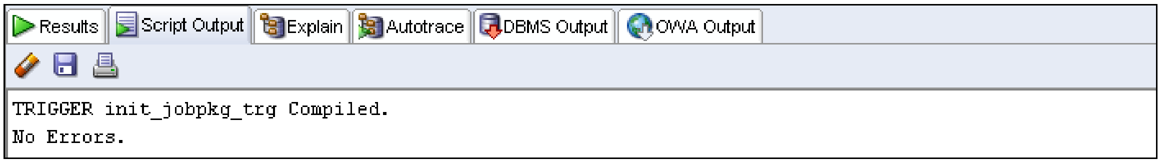


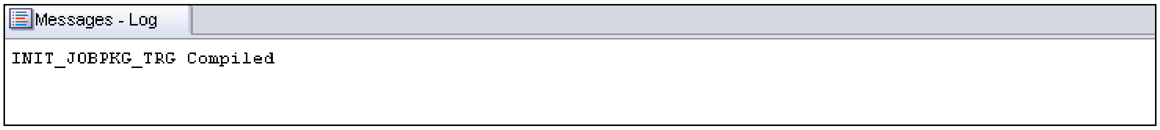


d. Implement a `BEFORE INSERT OR UPDATE` statement trigger called `INIT_JOBPKG_TRG` that uses the `CALL` syntax to invoke the `JOBS_PKG.INITIALIZE` procedure to ensure that the package state is current before the DML operations are performed.

Open the `sol_10_02_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the trigger, right-click the trigger's name in the Object Navigator, and then select Compile.

```
CREATE OR REPLACE TRIGGER init_jobpkg_trg
BEFORE INSERT OR UPDATE ON jobs
CALL jobs_pkg.initialize
/
SHOW ERRORS
```





e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the `IT_PROG` job type by 1,000 in the `JOBS` table. Follow this up with a query on the employees with the `IT_PROG` job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?

Open the `sol_10_02_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
  SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG';

SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';
```

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

EMPLOYEE_ID            LAST_NAME                 SALARY
---------------------- ------------------------- ----------------------
103                    Hunold                    9000
104                    Ernst                     6000
105                    Austin                    4800
106                    Pataballa                 4800
107                    Lorentz                   4200
214                    Beh                       9000

6 rows selected

1 rows updated
EMPLOYEE_ID            LAST_NAME                 SALARY
---------------------- ------------------------- ----------------------
103                    Hunold                    9000
104                    Ernst                     6000
105                    Austin                    5000
106                    Pataballa                 5000
107                    Lorentz                   5000
214                    Beh                       9000

6 rows selected
```

The employees with last names `Austin`, `Pataballa`, and `Lorentz` have all had their salaries updated. No exception occurred during this process, and you implemented a solution for the mutating table trigger exception.

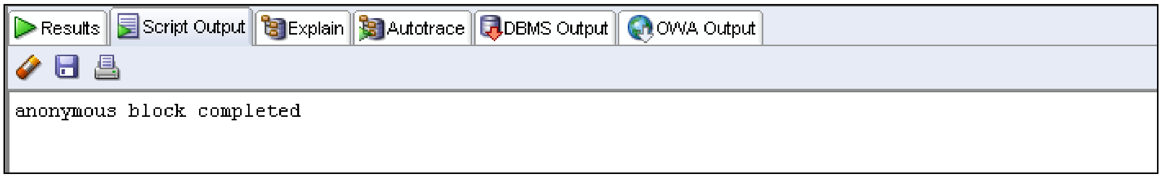
3) Because the `CHECK_SALARY` procedure is fired by `CHECK_SALARY_TRG` before inserting or updating an employee, you must check whether this still works as expected.

   a. Test this by adding a new employee using `EMP_PKG.ADD_EMPLOYEE` with the following parameters: `('Steve', 'Morse', 'SMORSE', and sal => 6500)`. What happens?

   Open the `sol_10_03_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal
=> 6500)
```

Practice 10: Managing Data Integrity Rules and Mutating Table Exceptions (continued)



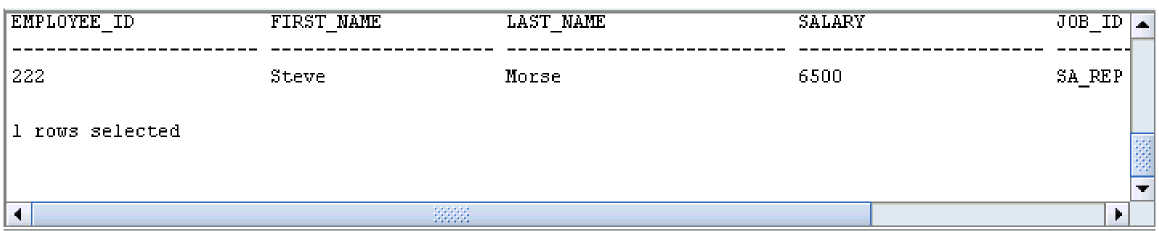
b. To correct the problem encountered when adding or updating an employee:

   i. Create a `BEFORE INSERT OR UPDATE` statement trigger called `EMPLOYEE_INITJOBS_TRG` on the `EMPLOYEES` table that calls the `JOBS_PKG.INITIALIZE` procedure.

   ii. Use the `CALL` syntax in the trigger body.

c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the `EMPLOYEES` table by displaying the employee ID, first and last names, salary, job ID, and department ID.

Open the `sol_10_03_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.



Practice 11: Using the PL/SQL Compiler Parameters and Warnings

In this practice, you display the compiler initialization parameters. You then enable native compilation for your session and compile a procedure. You then suppress all compiler-warning categories and then restore the original session-warning settings. Finally, you identify the categories for some compiler-warning message numbers.

1) Create and run a `lab_11_01` script to display the following information about compiler-initialization parameters by using the `USER_PLSQL_OBJECT_SETTINGS` data dictionary view. Note the settings for the `ADD_JOB_HISTORY` object.
   Note: Use the Execute Statement (F9) icon to display the results in the Results tab.

   a) Object name

   b) Object type

   c) The object's compilation mode

   d) The compilation optimization level

   Open the `sol_11_01.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to run the query. The code and a sample of the result are shown below.

```
SELECT name, type,plsql_code_type as code_type,
plsql_optimize_level as opt_lvl
FROM   user_plsql_object_settings;
```

Practice 11: Using the PL/SQL Compiler Parameters and Warnings (continued)

Results | Script Output | Explain | Autotrace | DBMS Output | OWA

Results:

	NAME	TYPE	CODE_TYPE	OPT_LVL
8	GET_EMPLOYEE	PROCEDURE	INTERPRETED	2
9	GET_ANNUAL_COMP	FUNCTION	INTERPRETED	2
10	EMP_PKG	PACKAGE BODY	INTERPRETED	2
11	EMP_PKG	PACKAGE	INTERPRETED	2
12	EMP_LIST	PROCEDURE	INTERPRETED	2
13	EMP_ACTIONS	PACKAGE BODY	INTERPRETED	2
14	EMP_ACTIONS	PACKAGE	INTERPRETED	2
15	EMPLOYEE_REPORT	PROCEDURE	INTERPRETED	2
16	DEPTREE_FILL	PROCEDURE	INTERPRETED	2
17	DEL_JOB	PROCEDURE	INTERPRETED	2
18	DELETE_EMP_TRG	TRIGGER	INTERPRETED	2
19	COMPILE_PKG	PACKAGE BODY	INTERPRETED	2
20	COMPILE_PKG	PACKAGE	INTERPRETED	2
21	CHECK_SALARY_TRG	TRIGGER	INTERPRETED	2
22	CHECK_SALARY	PROCEDURE	INTERPRETED	2
23	ADD_JOB_HISTORY	PROCEDURE	INTERPRETED	2
24	ADD_EMPLOYEE	PROCEDURE	INTERPRETED	2

. . .

2) Alter the `PLSQL_CODE_TYPE` parameter to enable native compilation for your session, and compile `ADD_JOB_HISTORY`.

   a) Execute the `ALTER SESSION` command to enable native compilation for the session.

   Open the `sol_11_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE';
```

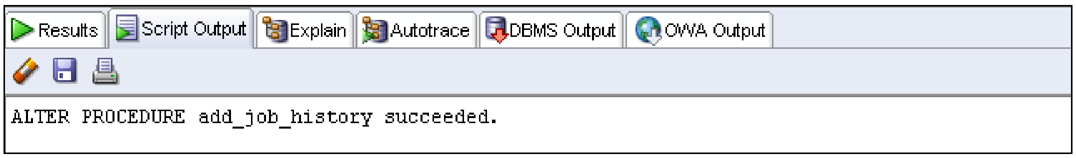
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
ALTER SESSION SET succeeded.
```

b) Compile the `ADD_JOB_HISTORY` procedure.

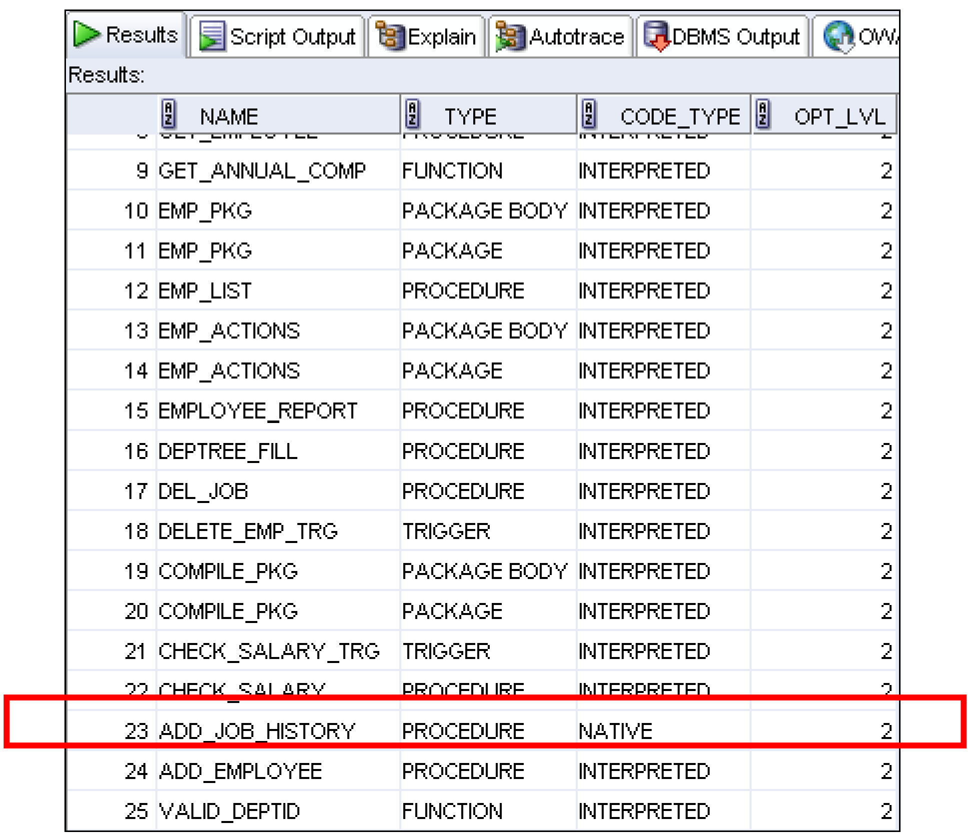
Open the `sol_11_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
ALTER PROCEDURE add_job_history COMPILE;
```



c) Rerun the `sol_11_01` script. Note the `PLSQL_CODE_TYPE` parameter.

```
SELECT name, type, plsql_code_type as code_type,
plsql_optimize_level as opt_lvl
FROM   user_plsql_object_settings;
```

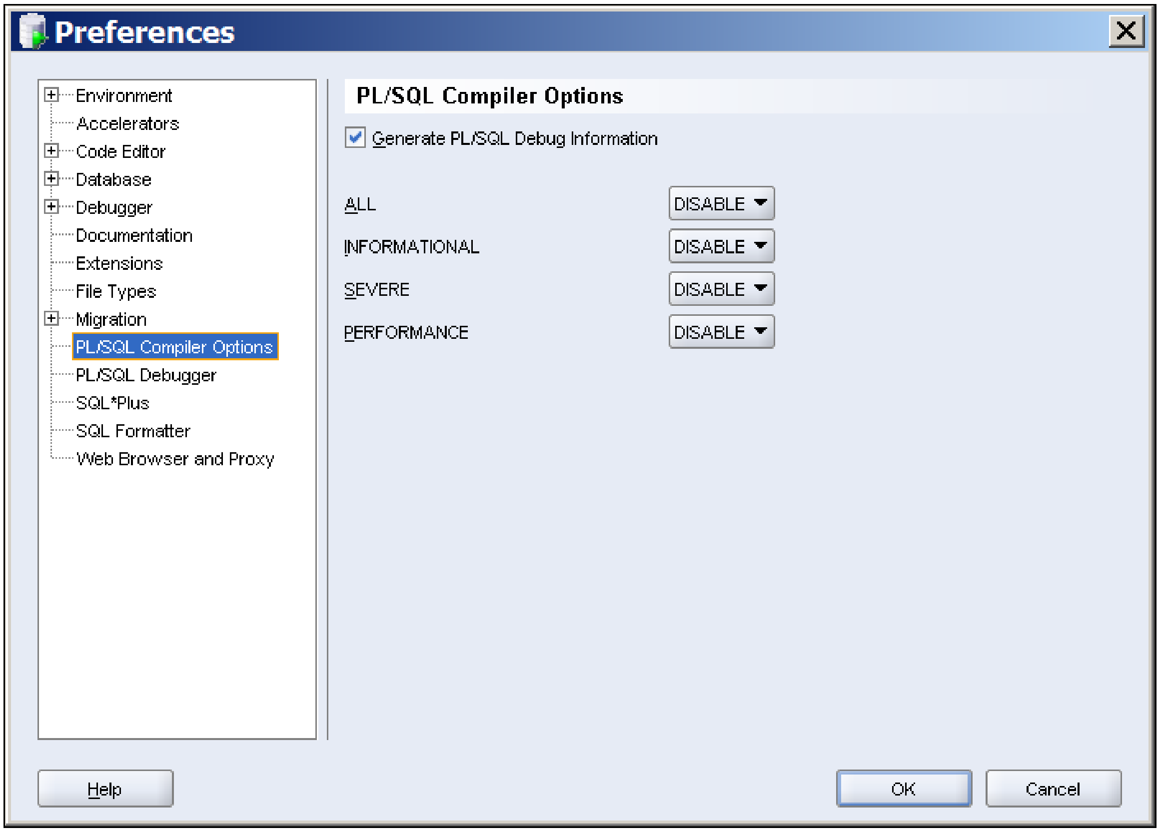


	NAME	TYPE	CODE_TYPE	OPT_LVL
9	GET_ANNUAL_COMP	FUNCTION	INTERPRETED	2
10	EMP_PKG	PACKAGE BODY	INTERPRETED	2
11	EMP_PKG	PACKAGE	INTERPRETED	2
12	EMP_LIST	PROCEDURE	INTERPRETED	2
13	EMP_ACTIONS	PACKAGE BODY	INTERPRETED	2
14	EMP_ACTIONS	PACKAGE	INTERPRETED	2
15	EMPLOYEE_REPORT	PROCEDURE	INTERPRETED	2
16	DEPTREE_FILL	PROCEDURE	INTERPRETED	2
17	DEL_JOB	PROCEDURE	INTERPRETED	2
18	DELETE_EMP_TRG	TRIGGER	INTERPRETED	2
19	COMPILE_PKG	PACKAGE BODY	INTERPRETED	2
20	COMPILE_PKG	PACKAGE	INTERPRETED	2
21	CHECK_SALARY_TRG	TRIGGER	INTERPRETED	2
22	CHECK_SALARY	PROCEDURE	INTERPRETED	2
23	ADD_JOB_HISTORY	PROCEDURE	NATIVE	2
24	ADD_EMPLOYEE	PROCEDURE	INTERPRETED	2
25	VALID_DEPTID	FUNCTION	INTERPRETED	2

d) Switch compilation to use interpreted compilation mode as follows:

```
ALTER SESSION SET PLSQL_CODE_TYPE = 'INTERPRETED';
```

3) Use the Tools > Preferences > PL/SQL Compiler Options region to disable all compiler warnings categories.



Select DISABLE for all four PL/SQL compiler warnings categories, and then click OK.

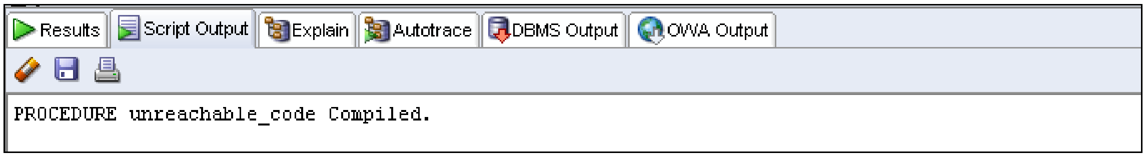
4) Edit, examine, and execute the `lab_11_04.sql` script to create the `UNREACHABLE_CODE` procedure. Click the Run Script icon (F5) to create the procedure. Use the procedure name in the Navigation tree to compile the procedure.

Open the `sol_11_04.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
CREATE OR REPLACE PROCEDURE unreachable_code AS
  c_x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF c_x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
```

Practice 11: Using the PL/SQL Compiler Parameters and Warnings (continued)

```
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/
```

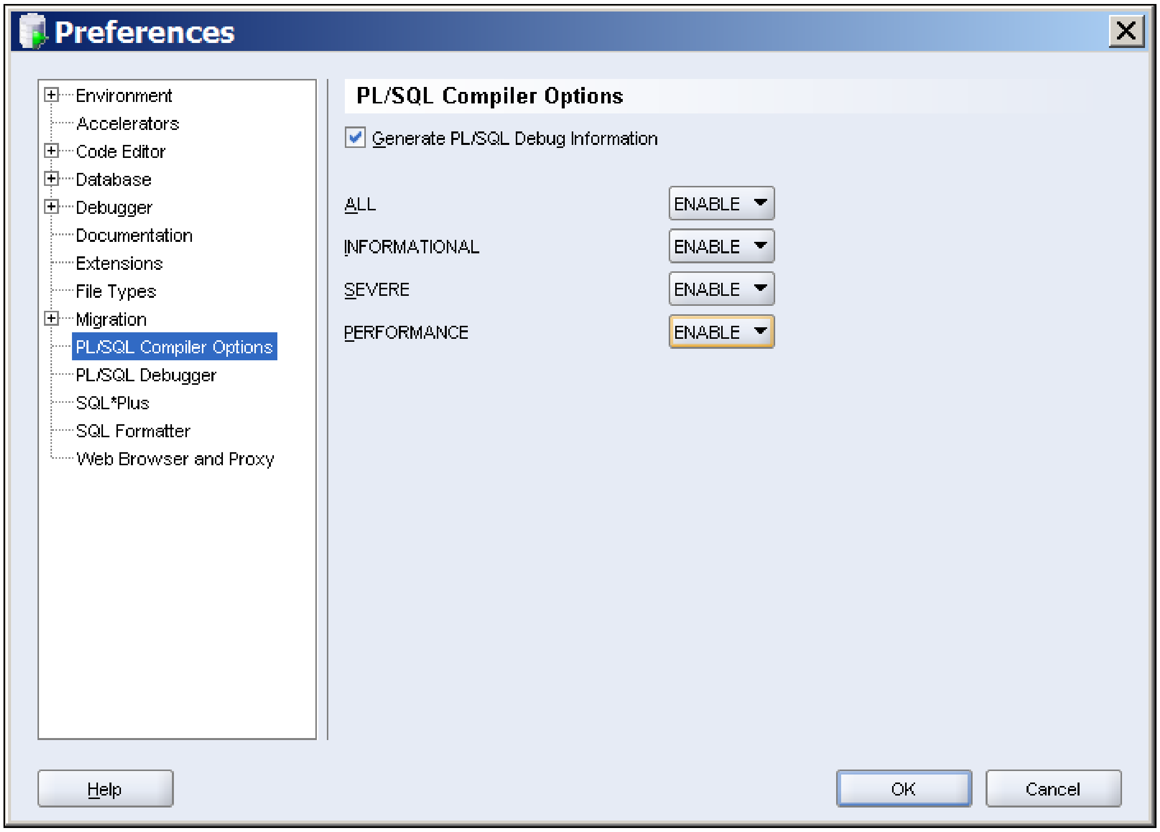


5) What are the compiler warnings that are displayed in the Compiler – Log tab, if any?

   None, because you disabled the compiler warnings in step 3.

6) Enable all compiler-warning messages for this session using the Preferences window.

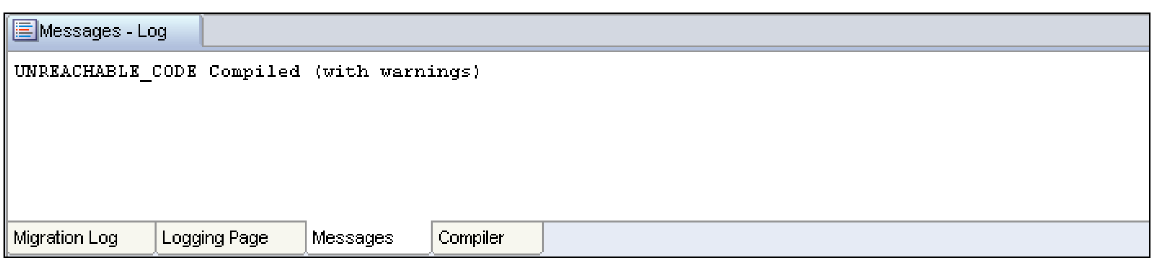
Practice 11: Using the PL/SQL Compiler Parameters and Warnings (continued)

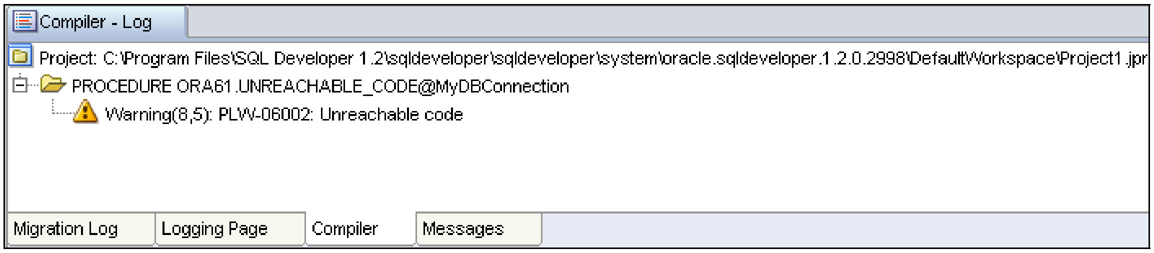


Select ENABLE for all four PL/SQL compiler warnings, and then click OK.

7) Recompile the `UNREACHABLE_CODE` procedure using the Object Navigation tree. What compiler warnings are displayed, if any?

Right-click the procedure's name in the Object Navigation tree, and then select Compile. Note the messages displayed in the Messages and Compiler subtabs in the Compiler – Log tab.



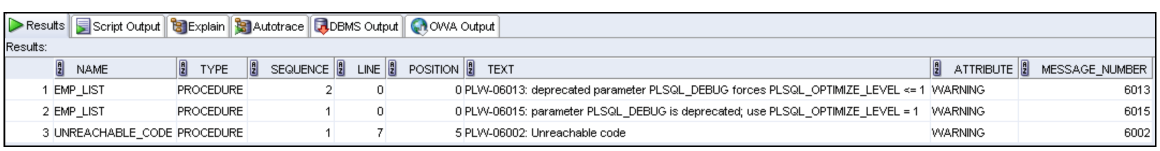


8) Use the `USER_ERRORS` data dictionary view to display the compiler-warning messages details as follows.

```
DESCRIBE user_errors
```

```
DESCRIBE user_errors
Name                           Null     Type
------------------------------ -------- -------------------------------------------------------
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(12)
SEQUENCE                       NOT NULL NUMBER
LINE                           NOT NULL NUMBER
POSITION                       NOT NULL NUMBER
TEXT                           NOT NULL VARCHAR2(4000)
ATTRIBUTE                               VARCHAR2(9)
MESSAGE_NUMBER                          NUMBER

8 rows selected
```

```
SELECT *
FROM user_errors;
```



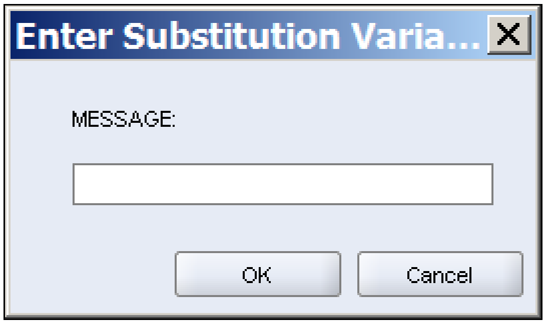
	NAME	TYPE	SEQUENCE	LINE	POSITION	TEXT	ATTRIBUTE	MESSAGE_NUMBER
1	EMP_LIST	PROCEDURE	2	0	0	PLW-06013: deprecated parameter PLSQL_DEBUG forces PLSQL_OPTIMIZE_LEVEL <= 1	WARNING	6013
2	EMP_LIST	PROCEDURE	1	0	0	PLW-06015: parameter PLSQL_DEBUG is deprecated; use PLSQL_OPTIMIZE_LEVEL = 1	WARNING	6015
3	UNREACHABLE_CODE	PROCEDURE	1	7	5	PLW-06002: Unreachable code	WARNING	6002

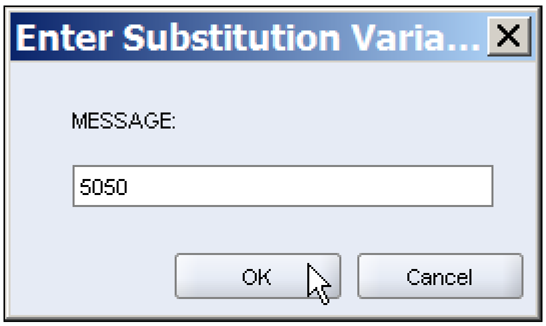
9) Create a script named `warning_msgs` that uses the `EXECUTE DBMS_OUTPUT` and the `DBMS_WARNING` packages to identify the categories for the following compiler-warning message numbers: 5050, 6075, and 7100.

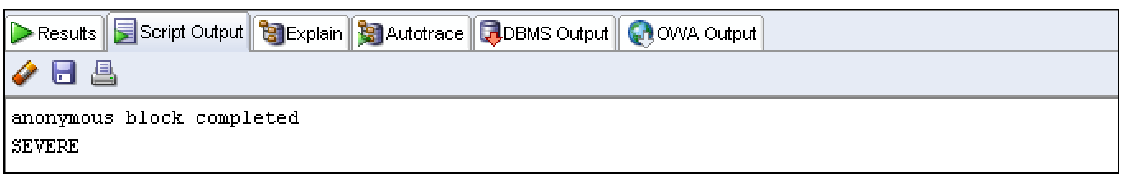
Open the `sol_11_09.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

Practice 11: Using the PL/SQL Compiler Parameters and Warnings (continued)

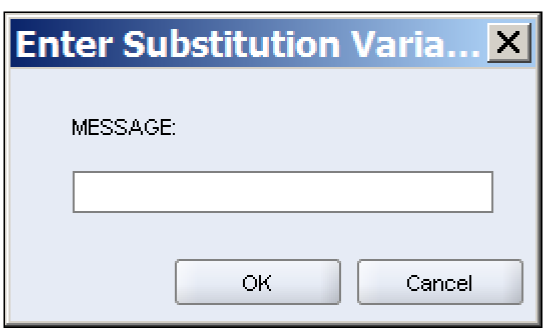
```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```



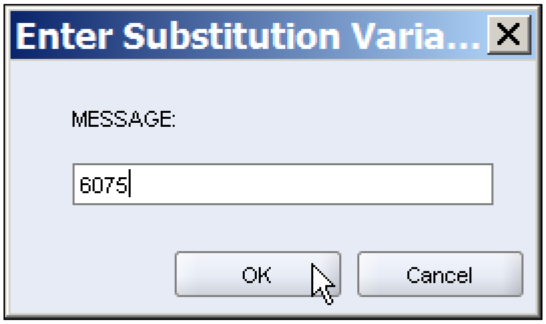


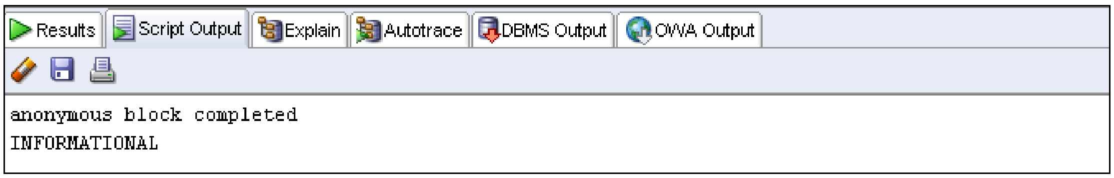



```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```

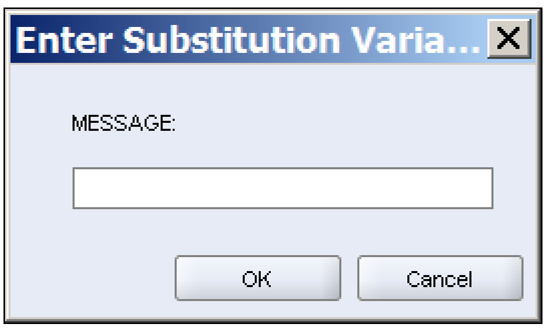


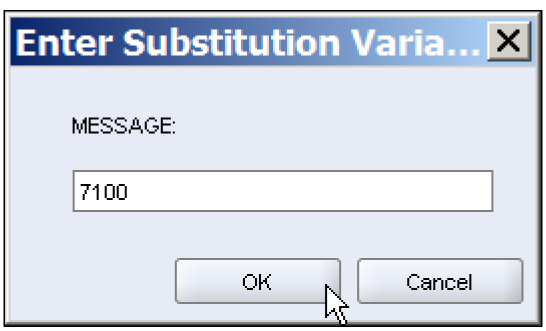
Practice 11: Using the PL/SQL Compiler Parameters and Warnings (continued)

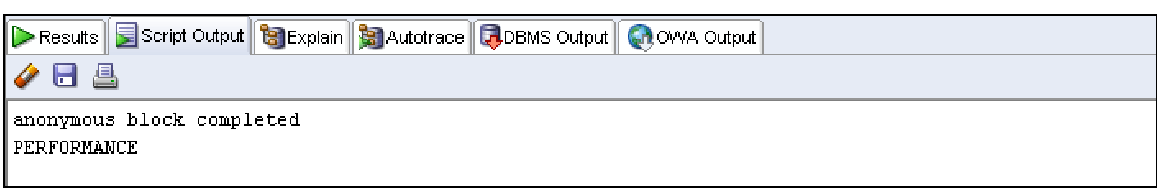





```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```







Practice 12: Using Conditional Compilation

In this practice, you create a package and a procedure that use conditional compilation. In addition, you use the appropriate package to retrieve the postprocessed source text of the PL/SQL unit. You also obfuscate some PL/SQL code.

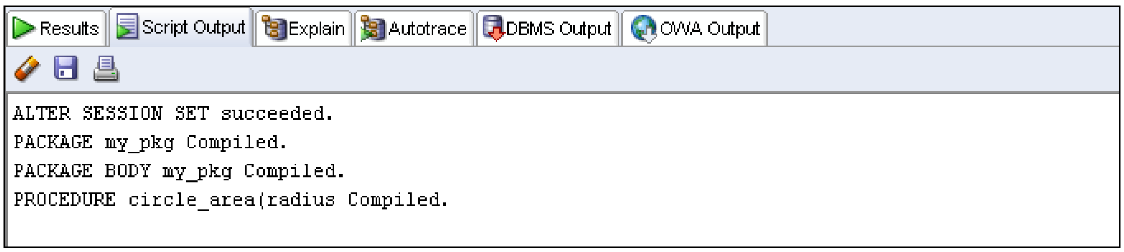
1) Examine and then execute the `lab_12_01.sql` script. This script sets flags for displaying debugging code and tracing information. The script also creates the `my_pkg` package and the `circle_area` procedure.

   Open the `sol_12_01.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package, right-click the package's name in the Object Navigator tree, and then select Compile. To compile the procedure, right-click the procedure's name in the Object Navigator tree, and then select Compile.

```
ALTER SESSION SET PLSQL_CCFLAGS = 'my_debug:FALSE,
my_tracing:FALSE';

CREATE OR REPLACE PACKAGE my_pkg AS
  SUBTYPE my_real IS
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN NUMBER; -- check
database version
      $ELSE                               BINARY_DOUBLE;
    $END
  my_pi my_real; my_e my_real;
END my_pkg;
/
CREATE OR REPLACE PACKAGE BODY my_pkg AS
BEGIN
  $IF DBMS_DB_VERSION.VERSION < 10 $THEN
       my_pi := 3.14016408289008292431940027343666863227;
       my_e  := 2.71828182845904523536028747135266249775;
  $ELSE
       my_pi := 3.14016408289008292431940027343666863227d;
       my_e  := 2.71828182845904523536028747135266249775d;
  $END
END my_pkg;
/

CREATE OR REPLACE PROCEDURE circle_area(radius my_pkg.my_real)
IS
  my_area my_pkg.my_real;
  my_datatype VARCHAR2(30);
BEGIN
  my_area := my_pkg.my_pi * radius;
  DBMS_OUTPUT.PUT_LINE('Radius: ' || TO_CHAR(radius)
```

Practice 12: Using Conditional Compilation (continued)

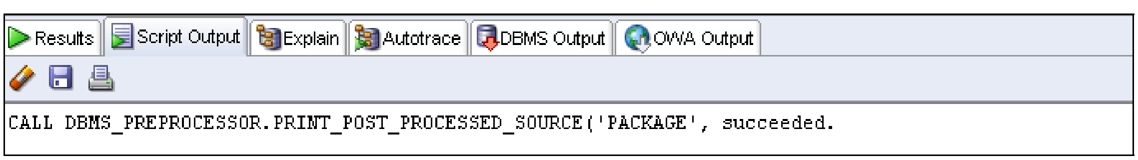
```
                        || ' Area: ' || TO_CHAR(my_area) );
  $IF $$my_debug $THEN
-- if my_debug is TRUE, run some debugging code


   SELECT DATA_TYPE INTO my_datatype FROM USER_ARGUMENTS
       WHERE OBJECT_NAME = 'CIRCLE_AREA' AND ARGUMENT_NAME =
'RADIUS';
     DBMS_OUTPUT.PUT_LINE('Datatype of the RADIUS argument is:
' || my_datatype);
  $END
END;
/
```



2) Use the DBMS_PREPROCESSOR subprogram to retrieve the postprocessed source text of the PL/SQL unit after processing the conditional compilation directives from lab_12_01.

Open the sol_12_02.sql file in the D:\labs\PLPU\solns folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
-- The code example assumes you are the student with the
-- account ora70. Substitute ora70 with your account
-- information.

CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE('PACKAGE',
'ORA70', 'MY_PKG');
```



Practice 12: Using Conditional Compilation (continued)

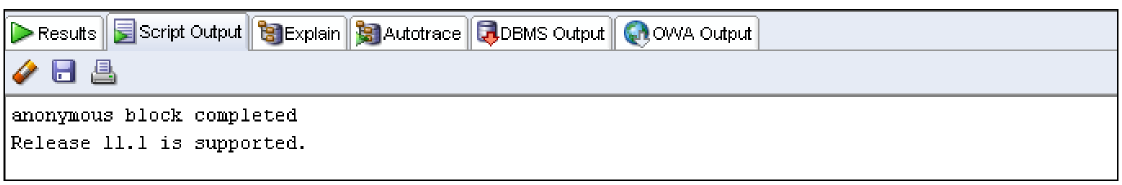
3) Create a PL/SQL script that uses the `DBMS_DB_VERSION` constant with conditional compilation. The code should test for the Oracle database version:

   a) If the database version is less than or equal to 10.1, it should display the following error message:
   `Unsupported database release.`

   b) If the database version is 11.1 or higher, it should display the following message:
   `Release 11.1 is supported.`

   Open the `sol_12_03.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
BEGIN
$IF DBMS_DB_VERSION.VER_LE_10_1 $THEN
$ERROR 'unsupported database release.' $END

$ELSE
  DBMS_OUTPUT.PUT_LINE ('Release ' || DBMS_DB_VERSION.VERSION
|| '.' ||
                        DBMS_DB_VERSION.RELEASE || ' is
supported.');
  -- Note that this COMMIT syntax is newly supported in 10.2
  COMMIT WRITE IMMEDIATE NOWAIT;
$END
END;
/
```



4) Consider the following code in the `lab_12_04.sql` script that uses `CREATE_WRAPPED` to dynamically create and wrap a package specification and a package body in a database. Edit the `lab_12_04.sql` script to add the needed code to obfuscate the PL/SQL code. Save and then execute the script.

```
DECLARE
-- the package_text variable contains the text to create
-- the package spec and body
  package_text VARCHAR2(32767);
  FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2
AS
  BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AS
     PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
```

```
        PROCEDURE fire_employee (emp_id NUMBER);
        END ' || pkgname || ';';
  END generate_spec;
  FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2
    AS
  BEGIN
     RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
       PROCEDURE raise_salary (emp_id NUMBER, amount
NUMBER) IS
       BEGIN
         UPDATE employees SET salary = salary + amount
WHERE employee_id = emp_id;
       END raise_salary;

      PROCEDURE fire_employee (emp_id NUMBER) IS
       BEGIN
         DELETE FROM employees WHERE employee_id = emp_id;
       END fire_employee;
       END ' || pkgname || ';';
  END generate_body;
```

a) Generate the package specification while passing the `emp_actions` parameter.

b) Create and wrap the package specification.

c) Generate the package body.

d) Create and wrap the package body.

e) Call a procedure from the wrapped package as follows:

```
CALL emp_actions.raise_salary(120, 100);
```

f) Use the `USER_SOURCE` data dictionary view to verify that the code is hidden as follows:

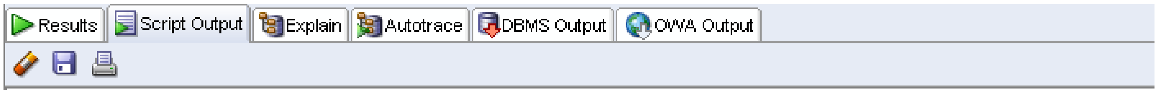
```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

Open the `soln_12_04.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
DECLARE
-- the package_text variable contains the text to create the
package spec and body
  package_text VARCHAR2(32767);
  FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
     RETURN 'CREATE PACKAGE ' || pkgname || ' AS
       PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
       PROCEDURE fire_employee (emp_id NUMBER);
       END ' || pkgname || ';';
```

```
  END generate_spec;
  FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
     RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
       PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER)
IS
       BEGIN
UPDATE employees SET salary = salary + amount WHERE
employee_id = emp_id;
       END raise_salary;
       PROCEDURE fire_employee (emp_id NUMBER) IS
       BEGIN
         DELETE FROM employees WHERE employee_id = emp_id;
       END fire_employee;
       END ' || pkgname || ';';
  END generate_body;

BEGIN

-- generate package spec
  package_text := generate_spec('emp_actions');

-- create and wrap the package spec
  SYS.DBMS_DDL.CREATE_WRAPPED(package_text);

-- generate package body
  package_text := generate_body('emp_actions');

-- create and wrap the package body
  SYS.DBMS_DDL.CREATE_WRAPPED(package_text);
END;
/

-- call a procedure from the wrapped package

CALL emp_actions.raise_salary(120, 100);

-- Use the USER_SOURCE data dictionary view to verify that --
the code is hidden as follows:

SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

Practice 12: Using Conditional Compilation (continued)




```
anonymous block completed
CALL emp_actions.raise_salary(120, succeeded.
TEXT
-----------------------------------------------------------------------------------------------
PACKAGE emp_actions wrapped
a000000
369
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
9
9d b6
J/5HL9fHral0qRCb2WqJ2palyfEwg5m49T0f9b9cWtdi465Z0fpHctUruHSLCampqcqqF+qc
UMrqAsqAcnCxznCALMY0WkSy6pqpMK4fRJlpD0mxyqREgA+laZ0vFurrql7EExfI9nppt9Cq
GgrV7J9GI+okH0TIiYzH47CdD0sqHab0lBYj
```

```
PACKAGE BODY emp_actions wrapped
a000000
369
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
b
178 10f
Qr07JITUletMtbw7mSy0dcAUvfUwg/BK7cusfC/GkEIY/SqS0UbuDlvf5gVxBQTolARbWyTT
ELXnN+WFBi6/vrXGcnAtahjKsVEC60Mcy8bB0p8QpoM2QH/YmdNBvQRjY3AtYEg4ofC9Camb
Ltw0+36g6kh66sqa+W7FJtMgNnus/8Mep4WsbFsLw2+cMlkxIPQkBrwblmuW8C0a2zYWkE6n
8T7byCyPrajb88NeB6PCSLwwtaLs9dwi7QSZSQ3CKQZEoNxM4A==



2 rows selected
```

Practice 13: Managing Dependencies in Your Schema

In this practice, you use the DEPTREE_FILL procedure and the IDEPTREE view to investigate dependencies in your schema. In addition, you recompile invalid procedures, functions, packages, and views.

1) Create a tree structure showing all dependencies involving your add_employee procedure and your valid_deptid function.

   Note: add_employee and valid_deptid were created in the lesson titled "Creating Functions." You can run the solution scripts for Practice 3 if you need to create the procedure and function.

   a) Load and execute the utldtree.sql script, which is located in the D:\lab\labs folder.

      Open the utldtree.sql file in the D:\labs\PLPU\solns folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
Rem
Rem $Header: utldtree.sql,v 1.2 1992/10/26 16:24:44 RKOOI Stab
$
Rem
Rem  Copyright (c) 1991 by Oracle Corporation
Rem   NAME
Rem   deptree.sql - Show objects recursively dependent on
Rem   given object
Rem   DESCRIPTION
Rem   This procedure, view and temp table will allow you to
see Rem   all objects that are (recursively) dependent on the
given Rem   object.
Rem   Note: you will only see objects for which you have
Rem   permission.
Rem   Examples:
Rem   execute deptree_fill('procedure', 'scott', 'billing');
Rem   select * from deptree order by seq#;
Rem
Rem   execute deptree_fill('table', 'scott', 'emp');
Rem   select * from deptree order by seq#;
Rem


Rem   execute deptree_fill('package body', 'scott',
Rem   'accts_payable');
Rem   select * from deptree order by seq#;
Rem
```

```
Rem   A prettier way to display this information than
Rem   select * from deptree order by seq#;
Rem     is
Rem   select * from ideptree;
Rem   This shows the dependency relationship via indenting.
Rem   Notice that no order by clause is needed with ideptree.
Rem   RETURNS
Rem
Rem   NOTES
Rem   Run this script once for each schema that needs this
Rem   utility.
Rem   MODIFIED   (MM/DD/YY)
Rem   rkooi      10/26/92 -  owner -> schema for SQL2
Rem   glumpkin   10/20/92 -  Renamed from DEPTREE.SQL
Rem   rkooi      09/02/92 -  change ORU errors
Rem   rkooi      06/10/92 -  add rae errors
Rem   rkooi      01/13/92 -  update for sys vs. regular user
Rem   rkooi      01/10/92 -  fix ideptree
Rem   rkooi      01/10/92 -  Better formatting, add ideptree
view
Rem   rkooi      12/02/91 -  deal with cursors
Rem   rkooi      10/19/91 -  Creation

DROP SEQUENCE deptree_seq
/
CREATE SEQUENCE deptree_seq cache 200
/* cache 200 to make sequence faster */

/
DROP TABLE deptree_temptab
/
CREATE TABLE deptree_temptab
(
  object_id            number,
  referenced_object_id number,
  nest_level           number,
  seq#                 number
)
/
CREATE OR REPLACE PROCEDURE deptree_fill (type char, schema
char, name char) IS
  obj_id number;
BEGIN
  DELETE FROM deptree_temptab;
  COMMITT;
  SELECT object_id INTO obj_id FROM all_objects
    WHERE owner = upper(deptree_fill.schema)

AND   object_name  = upper(deptree_fill.name)
    AND   object_type  = upper(deptree_fill.type);
  INSERT INTO deptree_temptab
```

Practice 13: Managing Dependencies in Your Schema (continued)

```
    VALUES(obj_id, 0, 0, 0);
  INSERT INTO deptree_temptab
    SELECT object_id, referenced_object_id,
        level, deptree_seq.nextval
      FROM public_dependency
      CONNECT BY PRIOR object_id = referenced_object_id
      START WITH referenced_object_id = deptree_fill.obj_id;
EXCEPTION
  WHEN no_data_found then
    raise_application_error(-20000, 'ORU-10013: ' ||
      type || ' ' || schema || '.' || name || ' was not
found.');
END;
/

DROP VIEW deptree
/

SET ECHO ON

REM This view will succeed if current user is sys.  This view
REM shows which shared cursors depend on the given object.  If
REM the current user is not sys, then this view get an error
REM either about lack of privileges or about the non-existence
of REM table x$kglxs.

SET ECHO OFF
CREATE VIEW sys.deptree
  (nested_level, type, schema, name, seq#)
AS
  SELECT d.nest_level, o.object_type, o.owner, o.object_name,
d.seq#
  FROM deptree_temptab d, dba_objects o
  WHERE d.object_id = o.object_id (+)
UNION ALL
  SELECT d.nest_level+1, 'CURSOR', '<shared>',
'"'||c.kglnaobj||'"', d.seq#+.5
  FROM deptree_temptab d, x$kgldp k, x$kglob g, obj$ o, user$
u, x$kglob c,
      x$kglxs a
    WHERE d.object_id = o.obj#
    AND   o.name = g.kglnaobj
    AND   o.owner# = u.user#
    AND   u.name = g.kglnaown
    AND   g.kglhdadr = k.kglrfhdl
    AND   k.kglhdadr = a.kglhdadr   /* make sure it is not a
transitive */
    AND   k.kgldepno = a.kglxsdep   /* reference, but a direct
one */
    AND   k.kglhdadr = c.kglhdadr
    AND   c.kglhdnsp = 0 /* a cursor */
```

Practice 13: Managing Dependencies in Your Schema (continued)

```
/

SET ECHO ON

REM This view will succeed if current user is not sys.  This
view
REM does *not* show which shared cursors depend on the given
REM object.
REM If the current user is sys then this view will get an
error
REM indicating that the view already exists (since prior view
REM create will have succeeded).

SET ECHO OFF
CREATE VIEW deptree
  (nested_level, type, schema, name, seq#)
AS
  select d.nest_level, o.object_type, o.owner, o.object_name,
d.seq#
  FROM deptree_temptab d, all_objects o
  WHERE d.object_id = o.object_id (+)
/
DROP VIEW ideptree
/
CREATE VIEW ideptree (dependencies)
AS
  SELECT lpad(' ',3*(max(nested_level))) || max(nvl(type, '<no
permission>')
    || ' ' || schema || decode(type, NULL, '', '.') || name)
  FROM deptree
  GROUP BY seq# /* So user can omit sort-by when selecting
from ideptree */
/
```

Practice 13: Managing Dependencies in Your Schema (continued)

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
Error starting at line 47 in command:
DROP SEQUENCE deptree_seq
Error report:
SQL Error: ORA-02289: sequence does not exist
02289. 00000 -  "sequence does not exist"
*Cause:    The specified sequence does not exist, or the user does
           not have the required privilege to perform this operation.
*Action:   Make sure the sequence name is correct, and that you have
           the right to perform the desired operation on this sequence.
CREATE SEQUENCE succeeded.

Error starting at line 53 in command:
DROP TABLE deptree_temptab
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
CREATE TABLE succeeded.
Warning: execution completed with warning
PROCEDURE deptree_fill Compiled.

Error starting at line 88 in command:
DROP VIEW deptree
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
REM This view will succeed if current user is sys.  This view

REM shows which shared cursors depend on the given object.  If REM the current user is not sys, then this view get an error

REM either about lack of privileges or about the non-existence of REM table x$kglxs.

SET ECHO OFF
```

Practice 13: Managing Dependencies in Your Schema (continued)

```
Error starting at line 98 in command:
CREATE VIEW sys.deptree
  (nested_level, type, schema, name, seq#)
AS
  SELECT d.nest_level, o.object_type, o.owner, o.object_name, d.seq#
  FROM deptree_temptab d, dba_objects o
  WHERE d.object_id = o.object_id (+)
UNION ALL
  SELECT d.nest_level+1, 'CURSOR', '<shared>', '"'||c.kglnaobj||'"', d.seq#+.5
  FROM deptree_temptab d, x$kgldp k, x$kglob g, obj$ o, user$ u, x$kglob c,
      x$kglxs a
    WHERE d.object_id = o.obj#
    AND   o.name = g.kglnaobj
    AND   o.owner# = u.user#
    AND   u.name = g.kglnaown
    AND   g.kglhdadr = k.kglrfhdl
    AND   k.kglhdadr = a.kglhdadr   /* make sure it is not a transitive */
    AND   k.kgldepno = a.kglxsdep   /* reference, but a direct one */
    AND   k.kglhdadr = c.kglhdadr
    AND   c.kglhdnsp = 0 /* a cursor */
Error at Command Line:102 Column:7
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
REM This view will succeed if current user is not sys.  This view

REM does *not* show which shared cursors depend on the given

REM object.

REM If the current user is sys then this view will get an error

REM indicating that the view already exists (since prior view
```

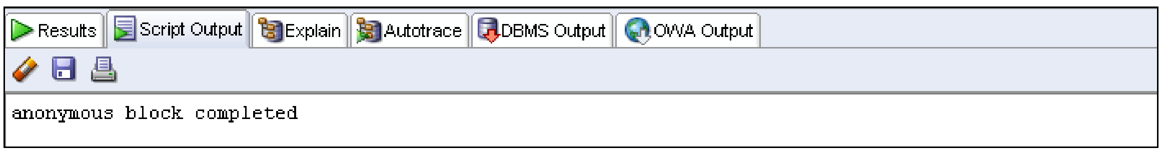
```
REM create will have succeeded).

SET ECHO OFF

CREATE VIEW succeeded.

Error starting at line 136 in command:
DROP VIEW ideptree
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
CREATE VIEW succeeded.
```

Practice 13: Managing Dependencies in Your Schema (continued)

b) Execute the `deptree_fill` procedure for the `add_employee` procedure.

Open the `sol_13_01_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

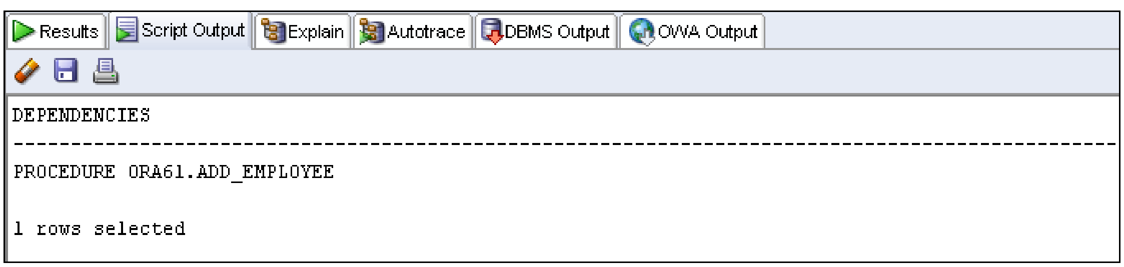
```
EXECUTE deptree_fill('PROCEDURE', USER, 'add_employee')
```



c) Query the `IDEPTREE` view to see your results.

Open the `sol_13_01_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

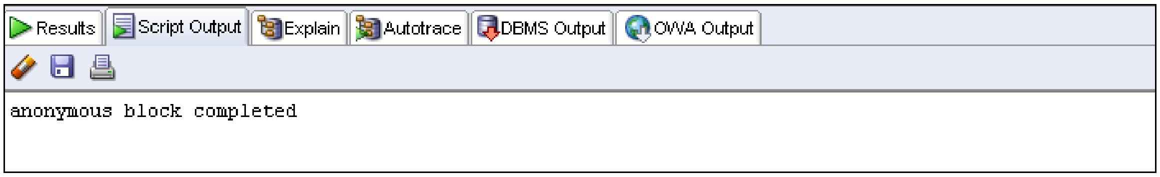
```
SELECT * FROM IDEPTREE;
```



d) Execute the `deptree_fill` procedure for the `valid_deptid` function.

Open the `sol_13_01_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

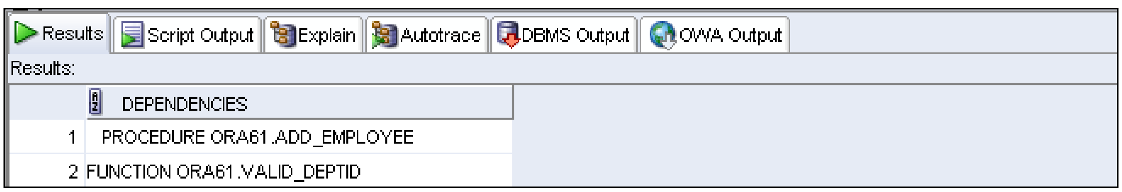
```
EXECUTE deptree_fill('FUNCTION', USER, 'valid_deptid')
```



e) Query the `IDEPTREE` view to see your results.

Open the `sol_13_01_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT * FROM IDEPTREE;
```



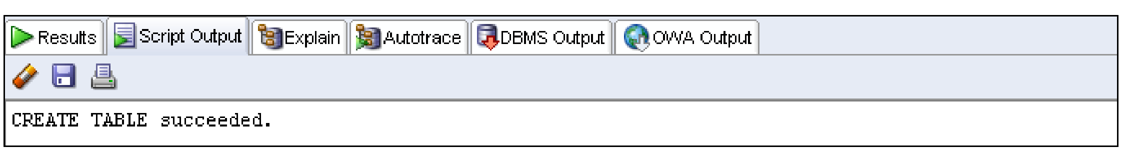
If you have time, complete the following exercise:

2) Dynamically validate invalid objects.

a) Make a copy of your `EMPLOYEES` table, called `EMPS`.

Open the `sol_13_02_a.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE TABLE emps AS
  SELECT * FROM employees;
```



b) Alter your `EMPLOYEES` table and add the column `TOTSAL` with data type `NUMBER(9,2)`.

Practice 13: Managing Dependencies in Your Schema (continued)

Open the `sol_13_02_b.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
ALTER TABLE employees
  ADD (totsal NUMBER(9,2));
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
ALTER TABLE employees succeeded.
```

c) Create and save a query to display the name, type, and status of all invalid objects.

Open the `sol_13_02_c.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

Results | Script Output | Explain | Autotrace | DBMS

Results:

	OBJECT_NAME	OBJECT_TYPE	STATUS
1	EMP_PKG	PACKAGE	INVALID
2	EMP_PKG	PACKAGE BODY	INVALID
3	GET_EMPLOYEE	PROCEDURE	INVALID
4	SECURE_EMPLOYEES	TRIGGER	INVALID
5	UPDATE_JOB_HISTORY	TRIGGER	INVALID
6	CHECK_SALARY_TRG	TRIGGER	INVALID
7	DELETE_EMP_TRG	TRIGGER	INVALID
8	MY_PKG	PACKAGE BODY	INVALID
9	EMP_ACTIONS	PACKAGE BODY	INVALID

. . .

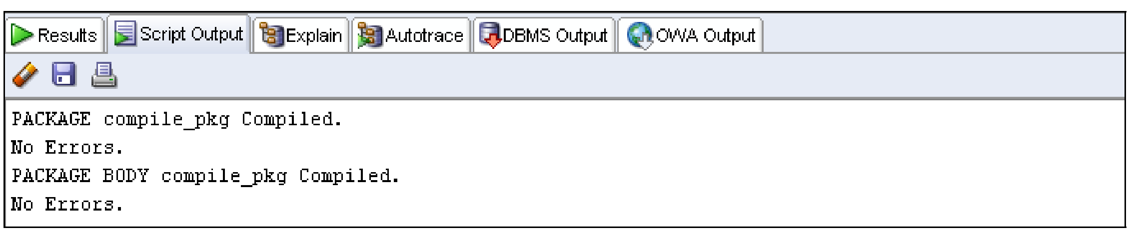
d) In the `compile_pkg` (created in Practice 7 in the lesson titled “Using Dynamic SQL”), add a procedure called `recompile` that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to alter the invalid object type and compile it.

Practice 13: Managing Dependencies in Your Schema (continued)

Open the `sol_13_02_d.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. The newly added code is highlighted in bold letters in the following code box.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  PROCEDURE make(name VARCHAR2);
  PROCEDURE recompile;
END compile_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY compile_pkg IS

  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;

  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
     */
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(name)
    AND ROWNUM = 1;
    RETURN proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

  PROCEDURE make(name VARCHAR2) IS
    stmt       VARCHAR2(100);
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      stmt := 'ALTER '|| proc_type ||' '|| name ||' COMPILE';
```

```
  execute(stmt);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Subprogram '''|| name ||''' does not exist');
    END IF;
  END make;

  PROCEDURE recompile IS
    stmt VARCHAR2(200);
    obj_name user_objects.object_name%type;
    obj_type user_objects.object_type%type;
  BEGIN
    FOR objrec IN (SELECT object_name, object_type
                   FROM user_objects
                   WHERE status = 'INVALID'
                   AND object_type <> 'PACKAGE BODY')
    LOOP
      stmt := 'ALTER '|| objrec.object_type ||' '||
                   objrec.object_name ||' COMPILE';
      execute(stmt);
    END LOOP;
  END recompile;

END compile_pkg;
/
SHOW ERRORS
```



e) Execute the `compile_pkg.recompile` procedure.

Open the `sol_13_02_e.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE compile_pkg.recompile
```

Practice 13: Managing Dependencies in Your Schema (continued)

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
anonymous block completed
ALTER PACKAGE EMP_PKG COMPILE
ALTER PROCEDURE GET_EMPLOYEE COMPILE
ALTER TRIGGER SECURE_EMPLOYEES COMPILE
ALTER TRIGGER UPDATE_JOB_HISTORY COMPILE
ALTER TRIGGER CHECK_SALARY_TRG COMPILE
ALTER TRIGGER DELETE_EMP_TRG COMPILE
```

f) Run the script file that you created in step 3 c. to check the status column value. Do you still have objects with an `INVALID` status?

Open the `sol_13_02_f.sql` file in the `D:\labs\PLPU\solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	OBJECT_NAME	OBJECT_TYPE	STATUS
1	EMP_ACTIONS	PACKAGE BODY	INVALID

B

Table Descriptions

Schema Description

Overall Description

The Oracle database sample schemas portray a sample company that operates worldwide to fill orders for several different products. The company has three divisions:

- **Human Resources:** Tracks information about the employees and facilities
- **Order Entry:** Tracks product inventories and sales through various channels
- **Sales History:** Tracks business statistics to facilitate business decisions

Each of these divisions is represented by a schema. In this course, you have access to the objects in all the schemas. However, the emphasis of the examples, demonstrations, and practices is on the Human Resources (`HR`) schema.

All scripts necessary to create the sample schemas reside in the `$ORACLE_HOME/demo/schema/` folder.

Human Resources (HR)

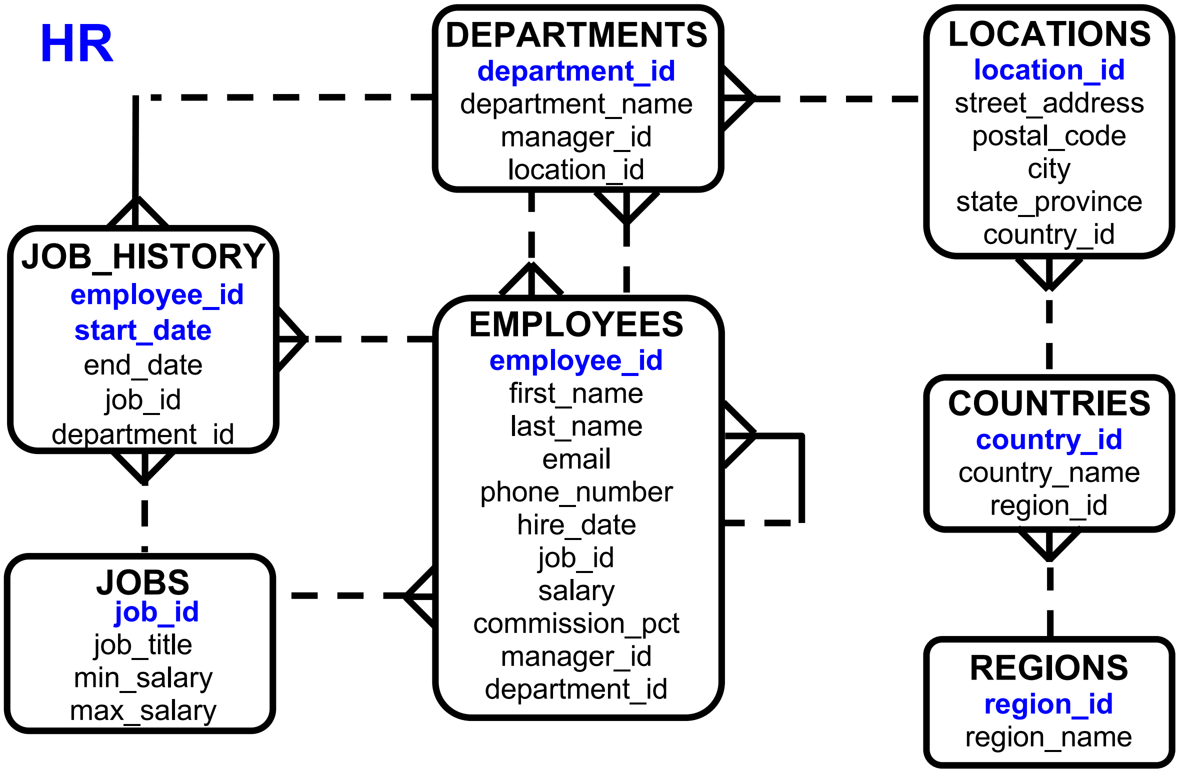
This is the schema that is used in this course. In the Human Resource (`HR`) records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee resigns, the duration the employee was working, the job identification number, and the department are recorded.

The sample company is regionally diverse, so it tracks the locations of its warehouses and departments. Each employee is assigned to a department, and each department is identified either by a unique department number or a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

In places where the departments and warehouses are located, the company records details such as the country name, currency symbol, currency name, and the region where the country is located geographically.

The HR Entity Relationship Diagram



The Human Resources (HR) Table Descriptions

```
DESCRIBE countries

 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------
 COUNTRY_ID                                NOT NULL CHAR(2)
 COUNTRY_NAME                                       VARCHAR2(40)
 REGION_ID                                          NUMBER

SELECT * FROM countries
```

	COUNTRY_ID	COUNTRY_NAME	REGION_ID
1	AR	Argentina	2
2	AU	Australia	3
3	BE	Belgium	1
4	BR	Brazil	2
5	CA	Canada	2
6	CH	Switzerland	1
7	CN	China	3
8	DE	Germany	1
9	DK	Denmark	1
10	EG	Egypt	4
11	FR	France	1
12	HK	HongKong	3
13	IL	Israel	4
14	IN	India	3
15	IT	Italy	1
16	JP	Japan	3
17	KW	Kuwait	4
18	MX	Mexico	2
19	NG	Nigeria	4
20	NL	Netherlands	1
21	SG	Singapore	3
22	UK	United Kingdom	1
23	US	United States of ...	2
24	ZM	Zambia	4
25	ZW	Zimbabwe	4

```
DESCRIBE departments

Name                                      Null?    Type
----------------------------------------- -------- -------------------
DEPARTMENT_ID                             NOT NULL NUMBER(4)
DEPARTMENT_NAME                           NOT NULL VARCHAR2(30)
MANAGER_ID                                         NUMBER(6)
LOCATION_ID                                        NUMBER(4)
```

```
SELECT * FROM departments
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500
6	60	IT	103	1400
7	70	Public Relations	204	2700
8	80	Sales	145	2500
9	90	Executive	100	1700
10	100	Finance	108	1700
11	110	Accounting	205	1700
12	120	Treasury	(null)	1700
13	130	Corporate Tax	(null)	1700
14	140	Control And Credit	(null)	1700
15	150	Shareholder Services	(null)	1700
16	160	Benefits	(null)	1700
17	170	Manufacturing	(null)	1700
18	180	Construction	(null)	1700
19	190	Contracting	(null)	1700
20	200	Operations	(null)	1700
21	210	IT Support	(null)	1700
22	220	NOC	(null)	1700
23	230	IT Helpdesk	(null)	1700
24	240	Government Sales	(null)	1700
25	250	Retail Sales	(null)	1700
26	260	Recruiting	(null)	1700
27	270	Payroll	(null)	1700
28	980	Education	(null)	2500
29	280	Training	(null)	2400

The Human Resources (HR) Table Descriptions (continued)

```
DESCRIBE employees
```

```
Name                                      Null?    Type
----------------------------------------- -------- ----------------
EMPLOYEE_ID                               NOT NULL NUMBER(6)
FIRST_NAME                                         VARCHAR2(20)
LAST_NAME                                 NOT NULL VARCHAR2(25)
EMAIL                                     NOT NULL VARCHAR2(25)
PHONE_NUMBER                                       VARCHAR2(20)
HIRE_DATE                                 NOT NULL DATE
JOB_ID                                    NOT NULL VARCHAR2(10)
SALARY                                             NUMBER(8,2)
COMMISSION_PCT                                     NUMBER(2,2)
MANAGER_ID                                         NUMBER(6)
DEPARTMENT_ID                                      NUMBER(4)
```

```
SELECT * FROM employees
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	(null)	(null)	90
1	100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	(null)	(null)	90
3	102	Lex	De Haan	LDE...	515.123.4569	13-JAN-93	AD_VP	17000	(null)	100	90
4	103	Alexander	Hunold	AHU...	590.423.4567	03-JAN-90	IT_PROG	9000	(null)	102	60
5	104	Bruce	Ernst	BER...	590.423.4568	21-MAY-91	IT_PROG	6000	(null)	103	60
6	105	David	Austin	DAU...	590.423.4569	25-JUN-97	IT_PROG	4800	(null)	103	60
7	106	Valli	Pataballa	VPA...	590.423.4560	05-FEB-98	IT_PROG	4800	(null)	103	60
8	107	Diana	Lorentz	DLO...	590.423.5567	07-FEB-99	IT_PROG	4200	(null)	103	60
9	108	Nancy	Greenberg	NGR...	515.124.4569	17-AUG-94	FI_MGR	12000	(null)	101	100
10	109	Daniel	Faviet	DFA...	515.124.4169	16-AUG-94	FI_ACCOUNT	9000	(null)	108	100
11	110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200	(null)	108	100
12	111	Ismael	Sciarra	ISCI...	515.124.4369	30-SEP-97	FI_ACCOUNT	7700	(null)	108	100
13	112	Jose Manuel	Urman	JMU...	515.124.4469	07-MAR-98	FI_ACCOUNT	7800	(null)	108	100
14	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900	(null)	108	100
15	114	Den	Raphaely	DRA...	515.127.4561	07-DEC-94	PU_MAN	11000	(null)	100	30
16	115	Alexander	Khoo	AKH...	515.127.4562	18-MAY-95	PU_CLERK	3100	(null)	114	30
17	116	Shelli	Baida	SBAI...	515.127.4563	24-DEC-97	PU_CLERK	2900	(null)	114	30
18	117	Sigal	Tobias	STO...	515.127.4564	24-JUL-97	PU_CLERK	2800	(null)	114	30
19	118	Guy	Himuro	GHIM...	515.127.4565	15-NOV-98	PU_CLERK	2600	(null)	114	30
20	119	Karen	Colmenares	KCO...	515.127.4566	10-AUG-99	PU_CLERK	2500	(null)	114	30
21	120	Matthew	Weiss	MWE...	650.123.1234	18-JUL-96	ST_MAN	8000	(null)	100	50
22	121	Adam	Fripp	AFRI...	650.123.2234	10-APR-97	ST_MAN	8200	(null)	100	50
23	122	Payam	Kaufling	PKA...	650.123.3234	01-MAY-95	ST_MAN	7900	(null)	100	50
24	123	Shanta	Vollman	SVO...	650.123.4234	10-OCT-97	ST_MAN	6500	(null)	100	50
25	124	Kevin	Mourgos	KMO...	650.123.5234	16-NOV-99	ST_MAN	5800	(null)	100	50
26	125	Julia	Nayer	JNA...	650.124.1214	16-JUL-97	ST_CLERK	3200	(null)	120	50
27	126	Irene	Mikkilineni	IMIK...	650.124.1224	28-SEP-98	ST_CLERK	2700	(null)	120	50
28	127	James	Landry	JLA...	650.124.1334	14-JAN-99	ST_CLERK	2400	(null)	120	50
29	128	Steven	Markle	SMA...	650.124.1434	08-MAR-00	ST_CLERK	2200	(null)	120	50
30	129	Laura	Bissot	LBIS...	650.124.5234	20-AUG-97	ST_CLERK	3300	(null)	121	50
31	130	Mozhe	Atkinson	MAT...	650.124.6234	30-OCT-97	ST_CLERK	2800	(null)	121	50
32	131	James	Marlow	JAM...	650.124.7234	16-FEB-97	ST_CLERK	2500	(null)	121	50
33	132	TJ	Olson	TJOL...	650.124.8234	10-APR-99	ST_CLERK	2100	(null)	121	50
34	133	Jason	Mallin	JMA...	650.127.1934	14-JUN-96	ST_CLERK	3300	(null)	122	50
35	134	Michael	Rogers	MRO...	650.127.1834	26-AUG-98	ST_CLERK	2900	(null)	122	50
36	135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400	(null)	122	50
37	136	Hazel	Philtanker	HPHI...	650.127.1634	06-FEB-00	ST_CLERK	2200	(null)	122	50
38	137	Renske	Ladwig	RLA...	650.121.1234	14-JUL-95	ST_CLERK	3600	(null)	123	50

. . .

The Human Resources (HR) Table Descriptions (continued)

`Employees` (continued)

. . .

39	138	Stephen	Stiles	SSTI...	650.121.2034	26-OCT-97	ST_CLERK	3200	(null)	123	50
40	139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700	(null)	123	50
41	140	Joshua	Patel	JPAT...	650.121.1834	06-APR-98	ST_CLERK	2500	(null)	123	50
42	141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500	(null)	124	50
43	142	Curtis	Davies	CDA...	650.121.2994	29-JAN-97	ST_CLERK	3100	(null)	124	50
44	143	Randall	Matos	RMA...	650.121.2874	15-MAR-98	ST_CLERK	2600	(null)	124	50
45	144	Peter	Vargas	PVA...	650.121.2004	09-JUL-98	ST_CLERK	2500	(null)	124	50
46	145	John	Russell	JRU...	011.44.1344.42...	01-OCT-96	SA_MAN	14000	0.4	100	80
47	146	Karen	Partners	KPA...	011.44.1344.46...	05-JAN-97	SA_MAN	13500	0.3	100	80
48	147	Alberto	Errazuriz	AER...	011.44.1344.42...	10-MAR-97	SA_MAN	12000	0.3	100	80
49	148	Gerald	Cambrault	GCA...	011.44.1344.61...	15-OCT-99	SA_MAN	11000	0.3	100	80
50	149	Eleni	Zlotkey	EZL...	011.44.1344.42...	29-JAN-00	SA_MAN	10500	0.2	100	80
51	150	Peter	Tucker	PTU...	011.44.1344.12...	30-JAN-97	SA_REP	10000	0.3	145	80
52	151	David	Bernstein	DBE...	011.44.1344.34...	24-MAR-97	SA_REP	9500	0.25	145	80
53	152	Peter	Hall	PHALL	011.44.1344.47...	20-AUG-97	SA_REP	9000	0.25	145	80
54	153	Christopher	Olsen	COL...	011.44.1344.49...	30-MAR-98	SA_REP	8000	0.2	145	80
55	154	Nanette	Cambrault	NCA...	011.44.1344.98...	09-DEC-98	SA_REP	7500	0.2	145	80
56	155	Oliver	Tuvault	OTU...	011.44.1344.48...	23-NOV-99	SA_REP	7000	0.15	145	80
57	156	Janette	King	JKING	011.44.1345.42...	30-JAN-96	SA_REP	10000	0.35	146	80
58	157	Patrick	Sully	PSU...	011.44.1345.92...	04-MAR-96	SA_REP	9500	0.35	146	80
59	158	Allan	McEwen	AMC...	011.44.1345.82...	01-AUG-96	SA_REP	9000	0.35	146	80
60	159	Lindsey	Smith	LSMI...	011.44.1345.72...	10-MAR-97	SA_REP	8000	0.3	146	80
61	160	Louise	Doran	LDO...	011.44.1345.62...	15-DEC-97	SA_REP	7500	0.3	146	80
62	161	Sarath	Sewall	SSE...	011.44.1345.52...	03-NOV-98	SA_REP	7000	0.25	146	80
63	162	Clara	Vishney	CVIS...	011.44.1346.12...	11-NOV-97	SA_REP	10500	0.25	147	80
64	163	Danielle	Greene	DGR...	011.44.1346.22...	19-MAR-99	SA_REP	9500	0.15	147	80
65	164	Mattea	Marvins	MMA...	011.44.1346.32...	24-JAN-00	SA_REP	7200	0.1	147	80
66	165	David	Lee	DLEE	011.44.1346.52...	23-FEB-00	SA_REP	6800	0.1	147	80
67	166	Sundar	Ande	SAN...	011.44.1346.62...	24-MAR-00	SA_REP	6400	0.1	147	80
68	167	Amit	Banda	ABA...	011.44.1346.72...	21-APR-00	SA_REP	6200	0.1	147	80
69	168	Lisa	Ozer	LOZER	011.44.1343.92...	11-MAR-97	SA_REP	11500	0.25	148	80
70	169	Harrison	Bloom	HBL...	011.44.1343.82...	23-MAR-98	SA_REP	10000	0.2	148	80
71	170	Tayler	Fox	TFOX	011.44.1343.72...	24-JAN-98	SA_REP	9600	0.2	148	80
72	171	William	Smith	WSM...	011.44.1343.62...	23-FEB-99	SA_REP	7400	0.15	148	80
73	172	Elizabeth	Bates	EBA...	011.44.1343.52...	24-MAR-99	SA_REP	7300	0.15	148	80
74	173	Sundita	Kumar	SKU...	011.44.1343.32...	21-APR-00	SA_REP	6100	0.1	148	80

. . .

The Human Resources (HR) Table Descriptions (continued)

Employees (continued

. . .

75	174	Ellen	Abel	EABEL	011.44.1644.42...	11-MAY-96	SA_REP	11000	0.3	149	80
76	175	Alyssa	Hutton	AHU...	011.44.1644.42...	19-MAR-97	SA_REP	8800	0.25	149	80
77	176	Jonathon	Taylor	JTA...	011.44.1644.42...	24-MAR-98	SA_REP	8600	0.2	149	80
78	177	Jack	Livingston	JLIVI...	011.44.1644.42...	23-APR-98	SA_REP	8400	0.2	149	80
79	178	Kimberely	Grant	KGR...	011.44.1644.42...	24-MAY-99	SA_REP	7000	0.15	149	(null)
80	179	Charles	Johnson	CJO...	011.44.1644.42...	04-JAN-00	SA_REP	6200	0.1	149	80
81	180	Winston	Taylor	WTA...	650.507.9876	24-JAN-98	SH_CLERK	3200	(null)	120	50
82	181	Jean	Fleaur	JFLE...	650.507.9877	23-FEB-98	SH_CLERK	3100	(null)	120	50
83	182	Martha	Sullivan	MSU...	650.507.9878	21-JUN-99	SH_CLERK	2500	(null)	120	50
84	183	Girard	Geoni	GGE...	650.507.9879	03-FEB-00	SH_CLERK	2800	(null)	120	50
85	184	Nandita	Sarchand	NSA...	650.509.1876	27-JAN-96	SH_CLERK	4200	(null)	121	50
86	185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100	(null)	121	50
87	186	Julia	Dellinger	JDEL...	650.509.3876	24-JUN-98	SH_CLERK	3400	(null)	121	50
88	187	Anthony	Cabrio	ACA...	650.509.4876	07-FEB-99	SH_CLERK	3000	(null)	121	50
89	188	Kelly	Chung	KCH...	650.505.1876	14-JUN-97	SH_CLERK	3800	(null)	122	50
90	189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600	(null)	122	50
91	190	Timothy	Gates	TGA...	650.505.3876	11-JUL-98	SH_CLERK	2900	(null)	122	50
92	191	Randall	Perkins	RPE...	650.505.4876	19-DEC-99	SH_CLERK	2500	(null)	122	50
93	192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000	(null)	123	50
94	193	Britney	Everett	BEV...	650.501.2876	03-MAR-97	SH_CLERK	3900	(null)	123	50
95	194	Samuel	McCain	SMC...	650.501.3876	01-JUL-98	SH_CLERK	3200	(null)	123	50
96	195	Vance	Jones	VJO...	650.501.4876	17-MAR-99	SH_CLERK	2800	(null)	123	50
97	196	Alana	Walsh	AW...	650.507.9811	24-APR-98	SH_CLERK	3100	(null)	124	50
98	197	Kevin	Feeney	KFEE...	650.507.9822	23-MAY-98	SH_CLERK	3000	(null)	124	50
99	198	Donald	OConnell	DOC...	650.507.9833	21-JUN-99	SH_CLERK	2600	(null)	124	50
100	199	Douglas	Grant	DGR...	650.507.9844	13-JAN-00	SH_CLERK	2600	(null)	124	50
101	200	Jennifer	Whalen	JWH...	515.123.4444	17-SEP-87	AD_ASST	4400	(null)	101	10
102	201	Michael	Hartstein	MHA...	515.123.5555	17-FEB-96	MK_MAN	13000	(null)	100	20
103	202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000	(null)	201	20
104	203	Susan	Mavris	SMA...	515.123.7777	07-JUN-94	HR_REP	6500	(null)	101	40
105	204	Hermann	Baer	HBA...	515.123.8888	07-JUN-94	PR_REP	10000	(null)	101	70
106	205	Shelley	Higgins	SHIG...	515.123.8080	07-JUN-94	AC_MGR	12000	(null)	101	110
107	206	William	Gietz	WGI...	515.123.8181	07-JUN-94	AC_ACCOUNT	8300	(null)	205	110

. . .

The Human Resources (HR) Table Descriptions (continued)

```
DESCRIBE job_history
```

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------------
EMPLOYEE_ID                               NOT NULL NUMBER(6)
START_DATE                                NOT NULL DATE
END_DATE                                  NOT NULL DATE
JOB_ID                                    NOT NULL VARCHAR2(10)
DEPARTMENT_ID                                      NUMBER(4)
```

```
SELECT * FROM job_history
```

	EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
1	102	13-JAN-93	24-JUL-98	IT_PROG	60
2	101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
3	101	28-OCT-93	15-MAR-97	AC_MGR	110
4	201	17-FEB-96	19-DEC-99	MK_REP	20
5	114	24-MAR-98	31-DEC-99	ST_CLERK	50
6	122	01-JAN-99	31-DEC-99	ST_CLERK	50
7	200	17-SEP-87	17-JUN-93	AD_ASST	90
8	176	24-MAR-98	31-DEC-98	SA_REP	80
9	176	01-JAN-99	31-DEC-99	SA_MAN	80
10	200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

The Human Resources (HR) Table Descriptions (continued)

```
DESCRIBE jobs
```

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------
JOB_ID                                    NOT NULL VARCHAR2(10)
JOB_TITLE                                 NOT NULL VARCHAR2(35)
MIN_SALARY                                         NUMBER(6)
MAX_SALARY                                         NUMBER(6)
```

```
SELECT * FROM jobs
```

	JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	AD_PRES	President	20000	40000
2	AD_VP	Administration Vice President	15000	30000
3	AD_ASST	Administration Assistant	3000	6000
4	FI_MGR	Finance Manager	8200	16000
5	FI_ACCOUNT	Accountant	4200	9000
6	AC_MGR	Accounting Manager	8200	16000
7	AC_ACCOUNT	Public Accountant	4200	9000
8	SA_MAN	Sales Manager	10000	20000
9	SA_REP	Sales Representative	6000	12000
10	PU_MAN	Purchasing Manager	8000	15000
11	PU_CLERK	Purchasing Clerk	2500	5500
12	ST_MAN	Stock Manager	5500	8500
13	ST_CLERK	Stock Clerk	2000	5000
14	SH_CLERK	Shipping Clerk	2500	5500
15	IT_PROG	Programmer	4000	10000
16	MK_MAN	Marketing Manager	9000	15000
17	MK_REP	Marketing Representative	4000	9000
18	HR_REP	Human Resources Representative	4000	9000
19	PR_REP	Public Relations Representative	4500	10500

```
DESCRIBE locations
```

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------
JOB_ID                                    NOT NULL VARCHAR2(10)
JOB_TITLE                                 NOT NULL VARCHAR2(35)
MIN_SALARY                                         NUMBER(6)
MAX_SALARY                                         NUMBER(6)
```

```
SELECT * FROM locations
```

	LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	COUNTRY_ID
1	1000	1297 Via Cola di Rie	00989	Roma	(null)	IT
2	1100	93091 Calle della Testa	10934	Venice	(null)	IT
3	1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
4	1300	9450 Kamiya-cho	6823	Hiroshima	(null)	JP
5	1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
6	1500	2011 Interiors Blvd	99236	South San Francisco	California	US
7	1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
8	1700	2004 Charade Rd	98199	Seattle	Washington	US
9	1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
10	1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
11	2000	40-5-12 Laogianggen	190518	Beijing	(null)	CN
12	2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
13	2200	12-98 Victoria Street	2901	Sydney	New South Wales	AU
14	2300	198 Clementi North	540198	Singapore	(null)	SG
15	2400	8204 Arthur St	(null)	London	(null)	UK
16	2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
17	2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
18	2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
19	2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
20	2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
21	3000	Murtenstrasse 921	3095	Bern	BE	CH
22	3100	Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
23	3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

The Human Resources (HR) Table Descriptions (continued)

```
DESCRIBE regions
```

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------
REGION_ID                                 NOT NULL NUMBER
REGION_NAME                                        VARCHAR2(25)
```

```
SELECT * FROM locations
```

	REGION_ID	REGION_NAME
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

Using SQL Developer

Objectives

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

- List the key features of Oracle SQL Developer
- Install Oracle SQL Developer 1.2.1
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and Run SQL scripts
- Create and save reports
- Install and use Oracle SQL Developer 1.5.3

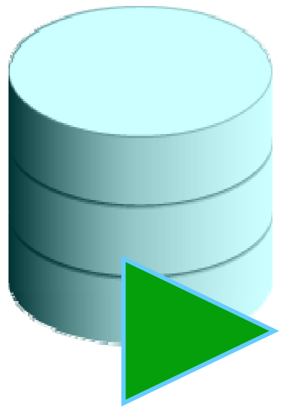
ORACLE

Objectives

In this appendix, you are introduced to the graphical tool called SQL Developer. You learn how to use SQL Developer for your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts.

What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.



What Is Oracle SQL Developer?

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When connected, you can perform operations on objects in the database.

Note: The SQL Developer 1.2 release is called the *Migration release* because it tightly integrates with *Developer Migration Workbench* that provides users with a single point to browse database objects and data in third-party databases, and to migrate from these databases to Oracle. You can also connect to schemas for selected third-party (non-Oracle) databases such as MySQL, Microsoft SQL Server, and Microsoft Access, and you can view metadata and data in these databases.

Additionally, SQL Developer includes support for Oracle Application Express 3.0.1 (Oracle APEX).

Specifications of SQL Developer

- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
  – Unzip the downloaded SQL Developer kit and double-click `sqldeveloper.exe` to start SQL Developer.
- Connects to Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
  – http://www.oracle.com/technology/products/database/sql_developer/index.html
- Needs JDK 1.5 installed on your system that can be downloaded from the following link:
  – http://java.sun.com/javase/downloads/index_jdk5.jsp

ORACLE

Specifications of SQL Developer

Oracle SQL Developer is developed in Java leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms. You can install SQL Developer on the Database Server and connect remotely from your desktop, thus avoiding client/server network traffic.

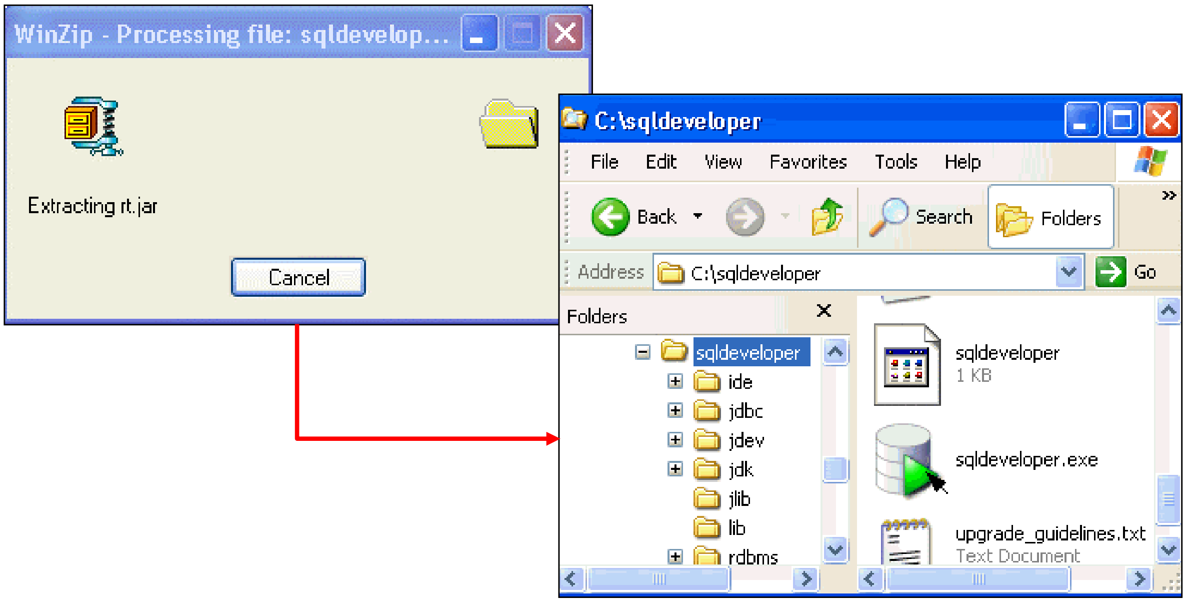
Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, and therefore, no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions including Express Edition.

SQL Developer can be downloaded with the following packaging options:

- Oracle SQL Developer for Windows (option to download with or without JDK 1.5)
- Oracle SQL Developer for Multiple Platforms (you should have JDK 1.5 already installed)
- Oracle SQL Developer for Mac OS X platforms (you should have JDK 1.5 already installed)
- Oracle SQL Developer RPM for Linux (you should have JDK 1.5 already installed)

Installing SQL Developer

Download the Oracle SQL Developer kit and unzip into any directory on your machine.



Installing SQL Developer

Oracle SQL Developer does not require an installer. To install SQL Developer, you need an unzip tool.

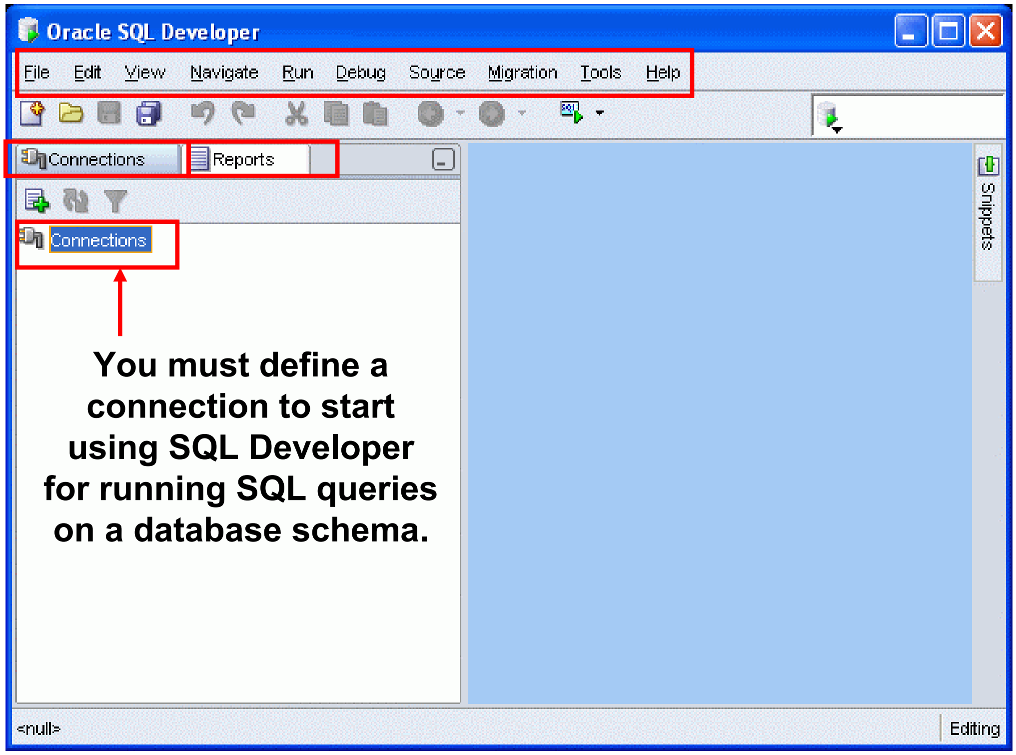
To install SQL Developer, perform the following steps:

1. Create a folder as `<local drive>:\SQL Developer`.
2. Download the SQL Developer kit from http://www.oracle.com/technology/products/database/sql_developer/index.html.
3. Unzip the downloaded SQL Developer kit into the folder created in step 1.

To start SQL Developer, go to `<local drive>:\SQL Developer`, and double-click `sqldeveloper.exe`.

Notes: SQL Developer 1.2 is already installed on the classroom machine. The installation kit for SQL Developer 1.5.3 is also on the classroom machine. You may use either version of SQL Developer in this course. Instructions for installing SQL Developer version 1.5.3 are available at the end of this appendix.

SQL Developer 1.2 Interface



ORACLE

SQL Developer 1.2 Interface

SQL Developer has two main navigation tabs:

- **Connections Navigator:** By using this, you can browse database objects and users to which you have access.
- **Reports tab:** By using this tab, you can run predefined reports or create and add your own reports.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences. The following menus contain standard entries, plus entries for features specific to SQL Developer:

- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to panes and in the execution of subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected
- **Debug:** Contains options that are relevant when a function or procedure is selected for debugging
- **Source:** Contains options for use when you edit functions and procedures
- **Migration:** Contains options related to migrating third-party databases to Oracle
- **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures/functions.

Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for:
  - Multiple databases
  - Multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

ORACLE

Creating a Database Connection

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

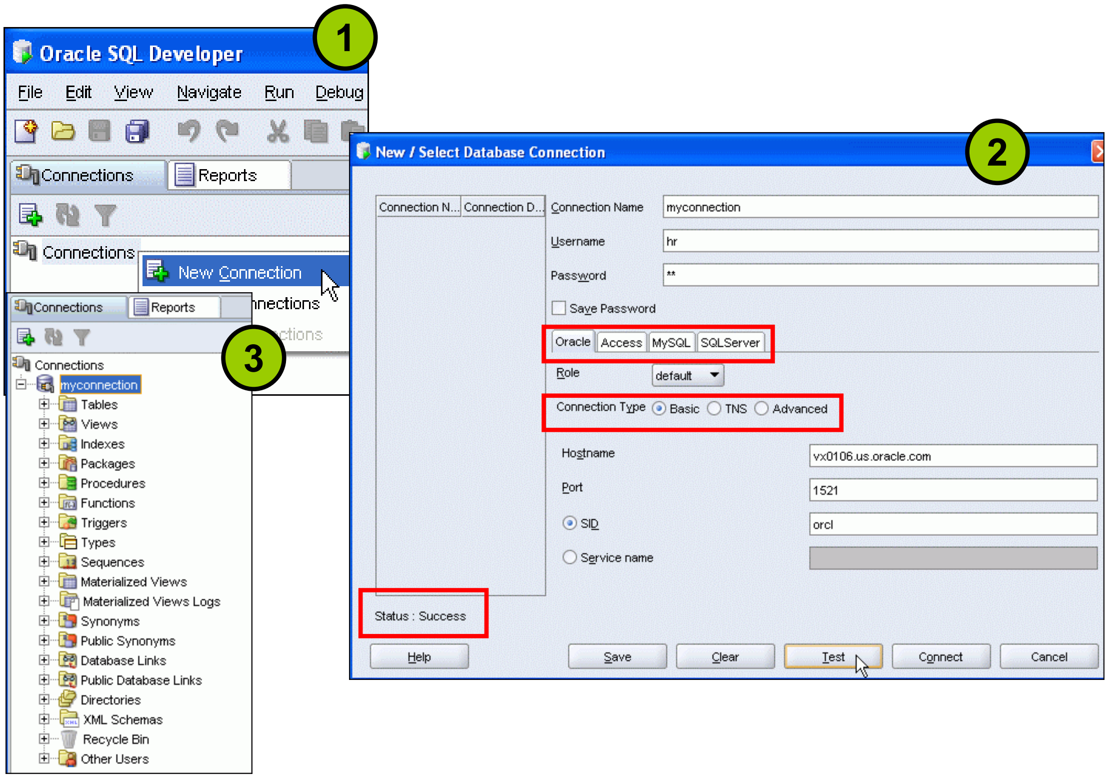
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows, if the `tnsnames.ora` file exists but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it later.

You can create additional connections as different users to the same database or to connect to the different databases.

Creating a Database Connection



ORACLE

Creating a Database Connection (continued)

To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click **Connections** and select **New Connection**.
2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.
    1. From the Role drop-down box, you can select either *default* or SYSDBA (you choose SYSDBA for the `sys` user or any user with database administrator privileges).
    2. You can select the connection type as:
        - **Basic:** In this type, enter hostname and SID for the database you want to connect to. Port is already set to 1521. Or you can also choose to enter the Service name directly if you use a remote database connection.
        - **TNS:** You can select any one of the database aliases imported from the `tnsnames.ora` file.
        - **Advanced:** You can define a custom Java Database Connectivity (JDBC) URL to connect to the database.
    3. Click Test to ensure that the connection has been set correctly.
    4. Click Connect.

Creating a Database Connection (continued)

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you are not prompted for the password.

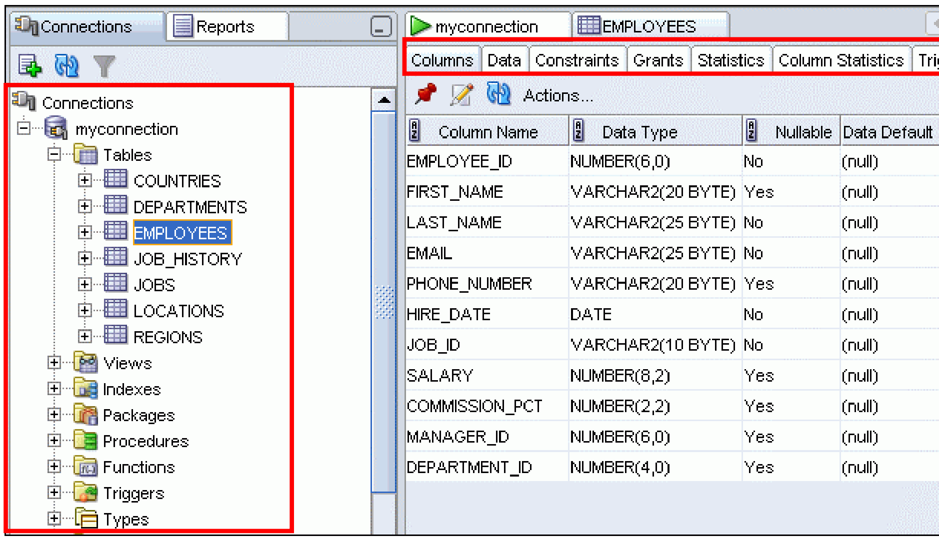
3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions, for example, dependencies, details, statistics, and so on.

Note: From the same New/Select Database Connection window, you can define connections to non-Oracle data sources using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.

Browsing Database Objects

Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



Browsing Database Objects

After you create a database connection, you can use the Connections Navigator to browse through many objects in a database schema including Tables, Views, Indexes, Packages, Procedures, Triggers, and Types.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

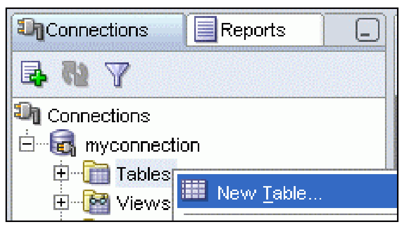
You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the `EMPLOYEES` table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click EMPLOYEES. By default, the Columns tab is selected. It shows the column description of the table. Using the Data tab, you can view the table data and also enter new rows, update data, and commit these changes to the database.

Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
  - Executing a SQL statement in SQL Worksheet
  - Using the context menu
- Edit the objects by using an edit dialog or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.



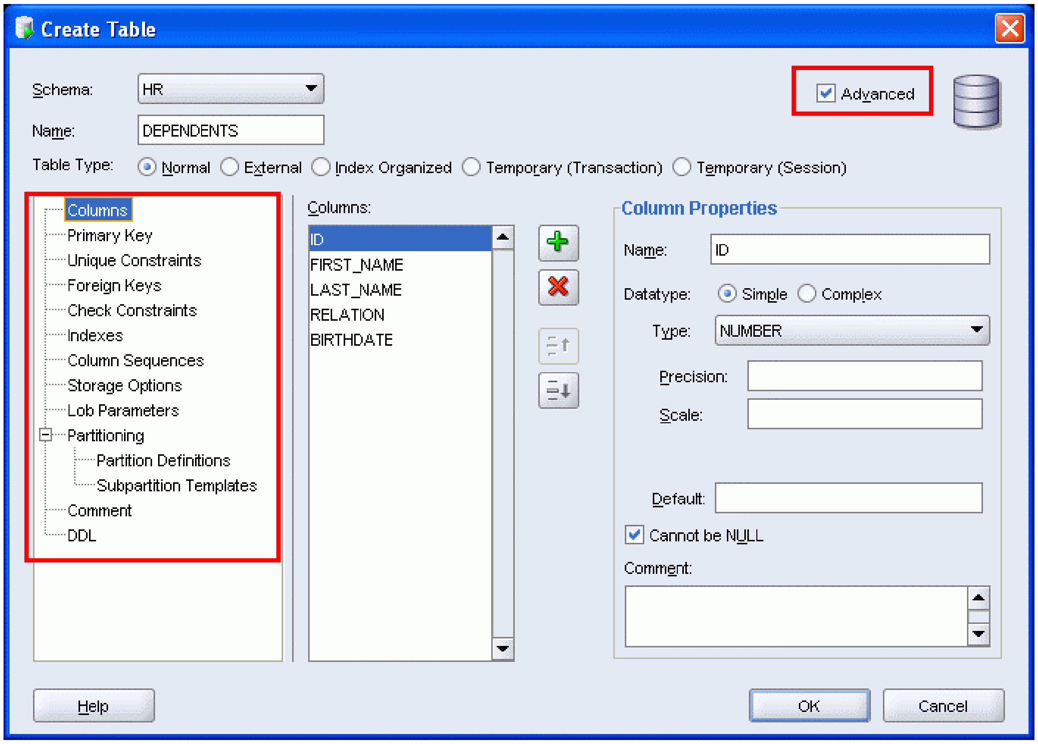
Creating a Schema Object

SQL Developer supports the creation of any schema object by executing a SQL statement in SQL Worksheet. Alternatively, you can create objects using the context menus. When created, you can edit the objects using an edit dialog or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the DDL for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows how to create a table using the context menu. To open a dialog box for creating a new table, right-click Tables and select New Table. The dialog boxes to create and edit database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



Creating a New Table: Example

In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the Advanced check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while you create the table.

The example in the slide shows how to create the `DEPENDENTS` table by selecting the Advanced check box.

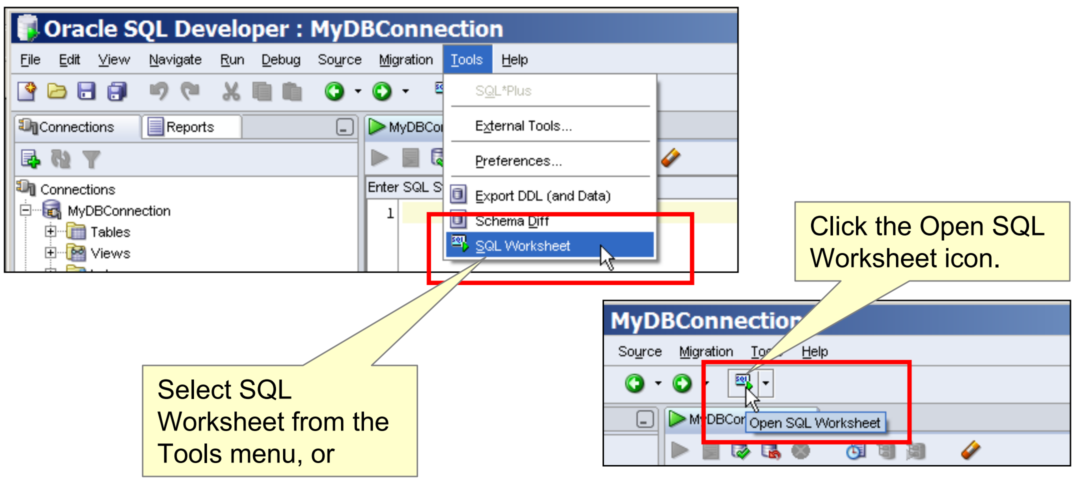
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables.
2. Select Create TABLE.
3. In the Create Table dialog box, select Advanced.
4. Specify column information.
5. Click OK.

Although it is not required, you should also specify a primary key by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created; to do so, right-click the table in the Connections Navigator and select Edit.

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL *Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



Using the SQL Worksheet

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

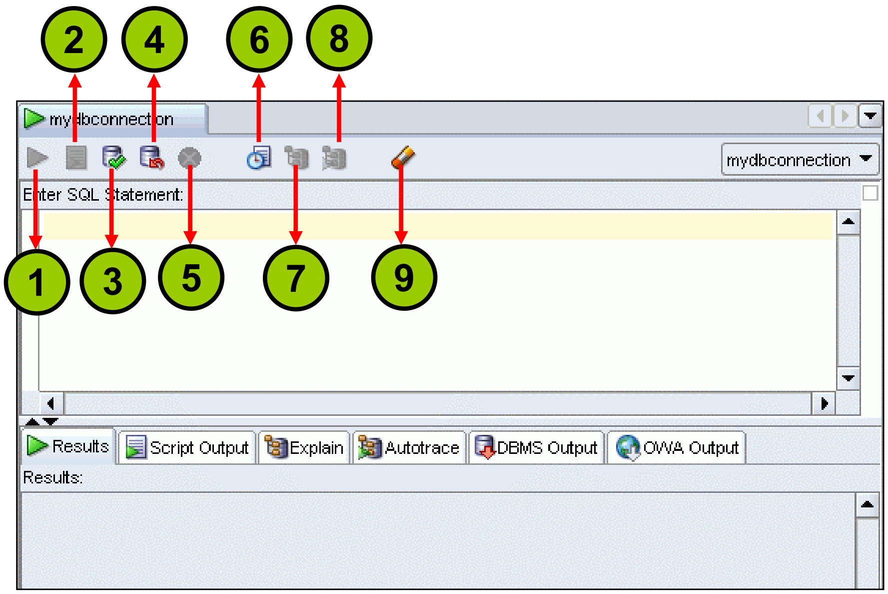
You can specify actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL Worksheet by using one of the following:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Using the SQL Worksheet



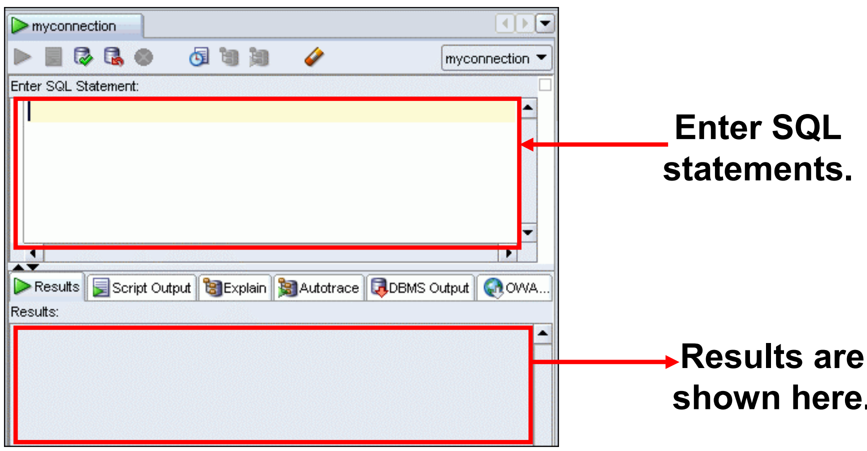
Using the SQL Worksheet (continued)

You may want to use the shortcut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement:** Executes the statement where the cursor is located in the Enter SQL Statement box. You can use bind variables in the SQL statements, but not substitution variables.
2. **Run Script:** Executes all statements in the Enter SQL Statement box by using the Script Runner. You can use substitution variables in the SQL statements, but not bind variables.
3. **Commit:** Writes any changes to the database and ends the transaction
4. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction
5. **Cancel:** Stops the execution of any statements currently being executed
6. **SQL History:** Displays a dialog box with information about SQL statements that you have executed
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
8. **Autotrace:** Generates trace information for the statement
9. **Clear:** Erases the statement or statements in the Enter SQL Statement box

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



Using the SQL Worksheet (continued)

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL Worksheet to the Oracle database. SQL*Plus commands used in the SQL Developer have to be interpreted by the SQL Worksheet before being passed to the database.

The SQL Worksheet currently supports a number of SQL*Plus commands. Commands not supported by the SQL Worksheet are ignored and are not sent to the Oracle database. Through the SQL Worksheet, you can execute SQL statements and some of the SQL*Plus commands.

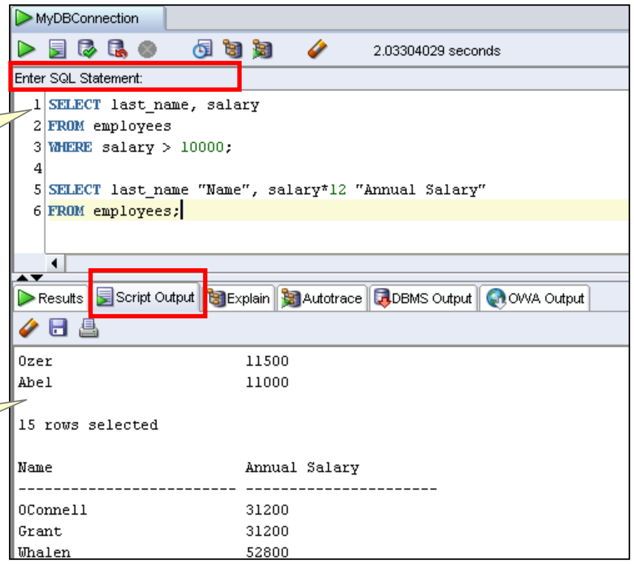
You can display a SQL Worksheet by using any of the following two options:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

Use the Enter SQL Statement box to enter single or multiple SQL statements.



View the results on the Script Output tabbed page.

Executing SQL Statements

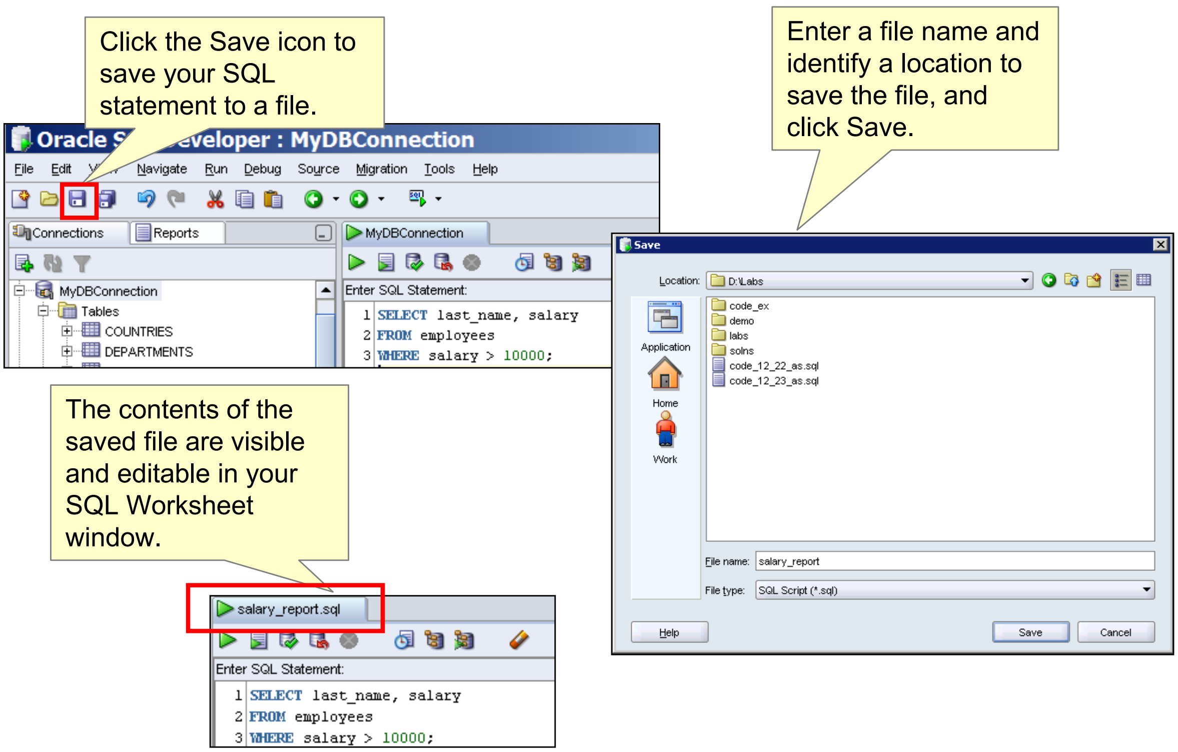
In the SQL Worksheet, you can use the Enter SQL Statement box to enter single or multiple SQL statements. For a single statement, the semicolon at the end is optional.

When you enter the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the Execute Statement icon. Alternatively, you can press the F9 key.

To execute multiple SQL statements and see the results, click the Run Script icon. Alternatively, you can press the F5 key.

In the example in the slide, because there are multiple SQL statements, the first statement is terminated with a semicolon. The cursor is in the first statement, and therefore, when the statement is executed, results corresponding to the first statement are displayed in the Results box.

Saving SQL Scripts



Saving SQL Scripts

You can save your SQL statements from the SQL Worksheet into a text file. To save the contents of the Enter SQL Statement box, follow these steps:

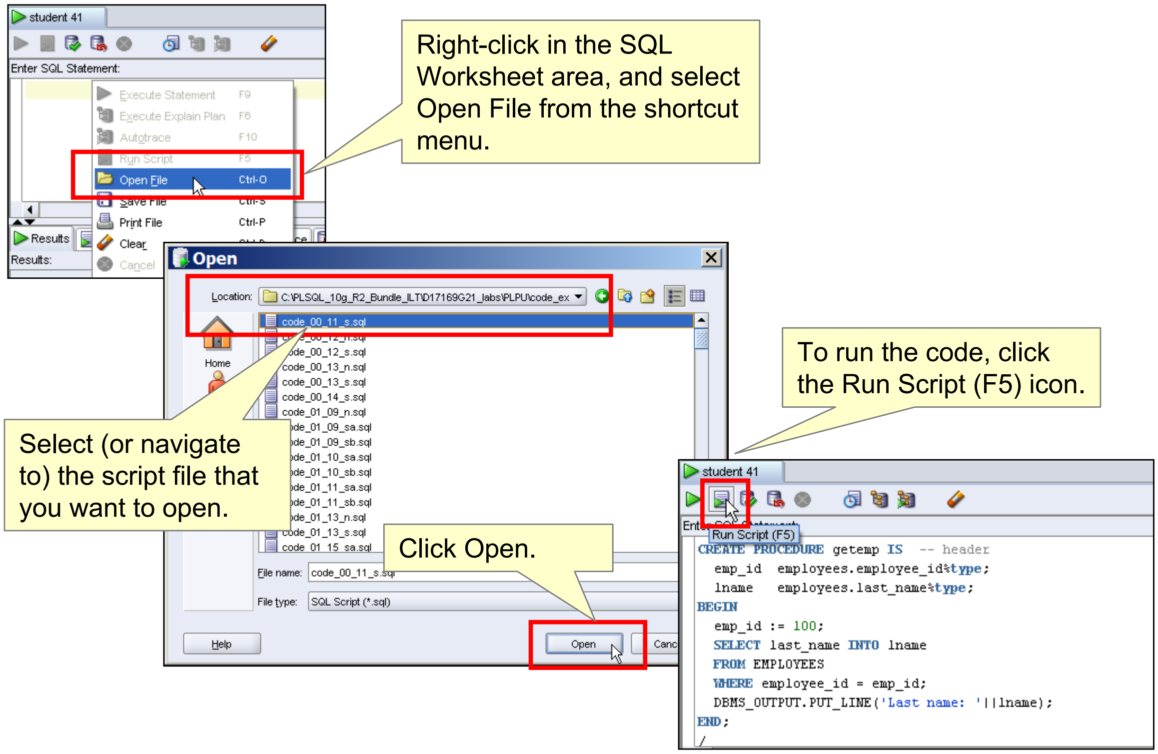
1. Click the Save icon or use the File > Save menu item.
2. In the Windows Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

After you save the contents to a file, the Enter SQL Statement window displays a tabbed page of your file contents. You can have multiple files open at the same time. Each file displays as a tabbed page.

Script Pathing

You can select a default path to look for scripts and to save scripts. Under Tools > Preferences > Database > Worksheet Parameters, enter a value in the “Select default path to look for scripts” field.

Executing Saved Script Files: Method 1

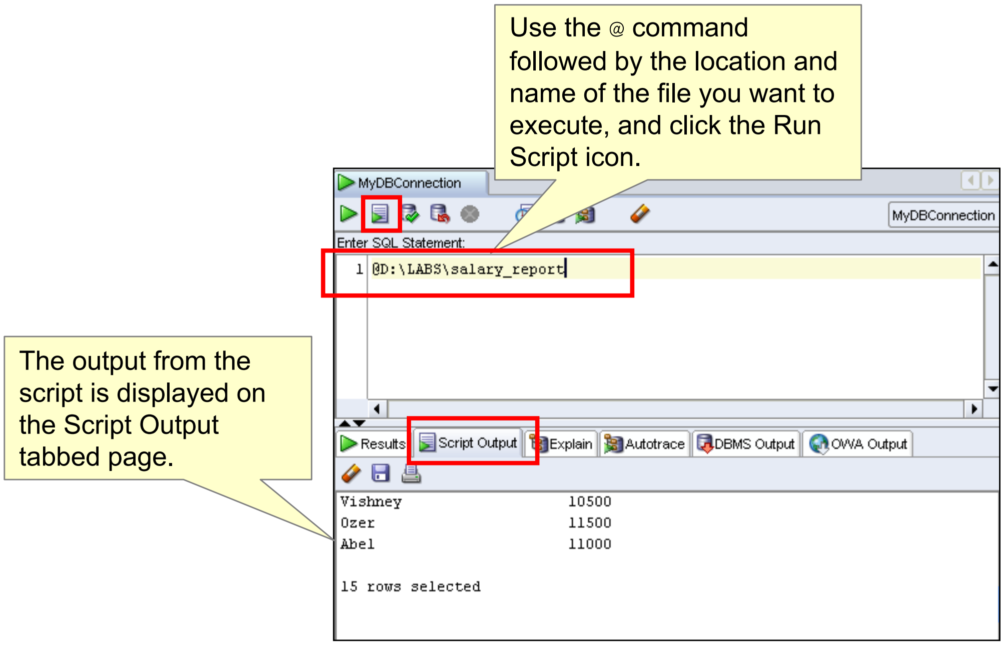


Executing Saved Script Files: Method 1

To open a script file and display the code in the SQL Worksheet area, perform the following:

1. Right-click in the SQL Worksheet area, and select Open File from the menu. The Open dialog box is displayed.
2. In the Open dialog box, select (or navigate to) the script file that you want to open.
3. Click Open. The code of the script file is displayed in the SQL Worksheet area.
4. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar.

Executing Saved Script Files: Method 2



Executing Saved Script Files: Method 2

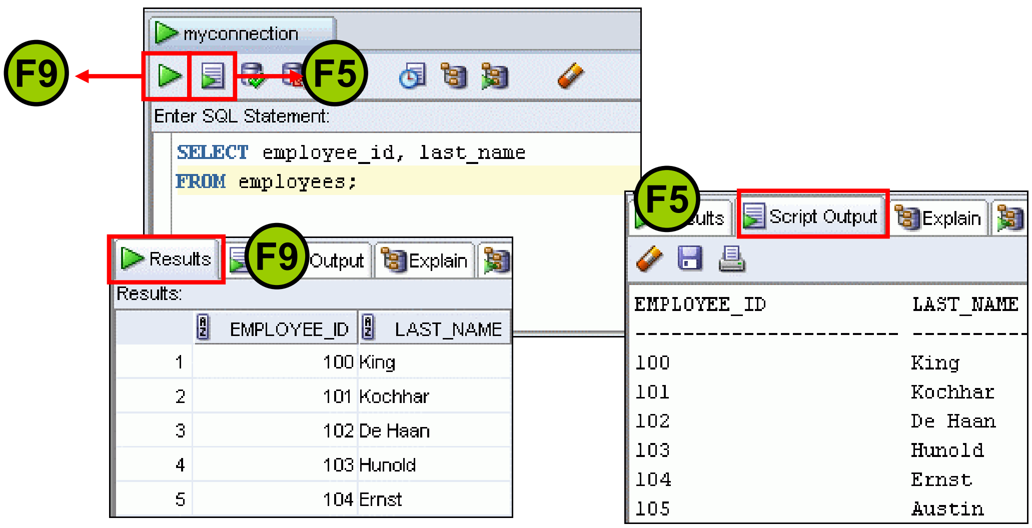
To run a saved SQL script, perform the following:

1. Use the **@** command, followed by the location, and name of the file you want to run, in the Enter SQL Statement window.
2. Click the Run Script icon.

The results from running the file are displayed on the Script Output tabbed page. You can also save the script output by clicking the Save icon on the Script Output tabbed page. The Windows File Save dialog box appears and you can identify a name and location for your file.

Executing SQL Statements

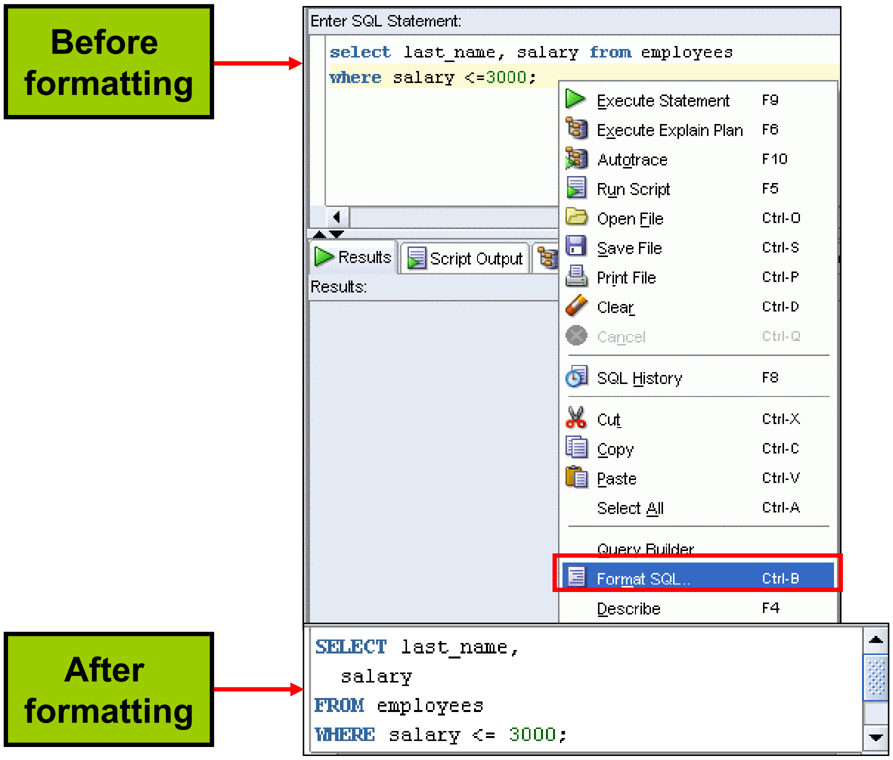
Use the Enter SQL Statement box to enter single or multiple SQL statements.



Executing SQL Statements

The example in the slide shows the difference in output for the same query when the [F9] key or Execute Statement is used versus the output when [F5] or Run Script is used.

Formatting the SQL Code



Formatting the SQL Code

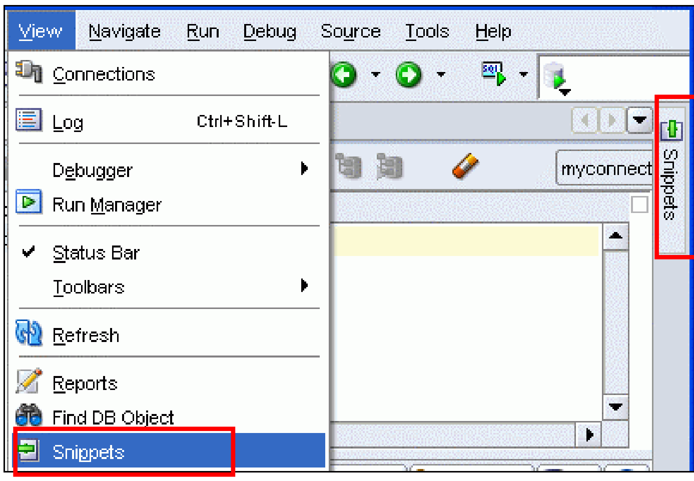
You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

To format the SQL code, right-click in the statement area, and select Format SQL.

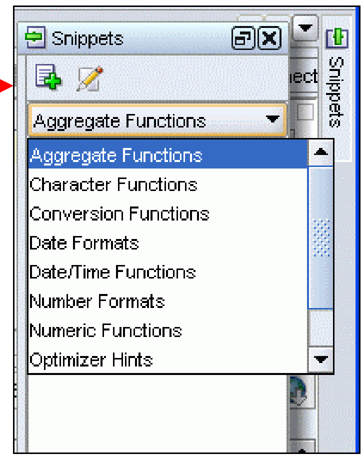
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

Using Snippets

Snippets are code fragments that may be just syntax or examples.



When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category you want.



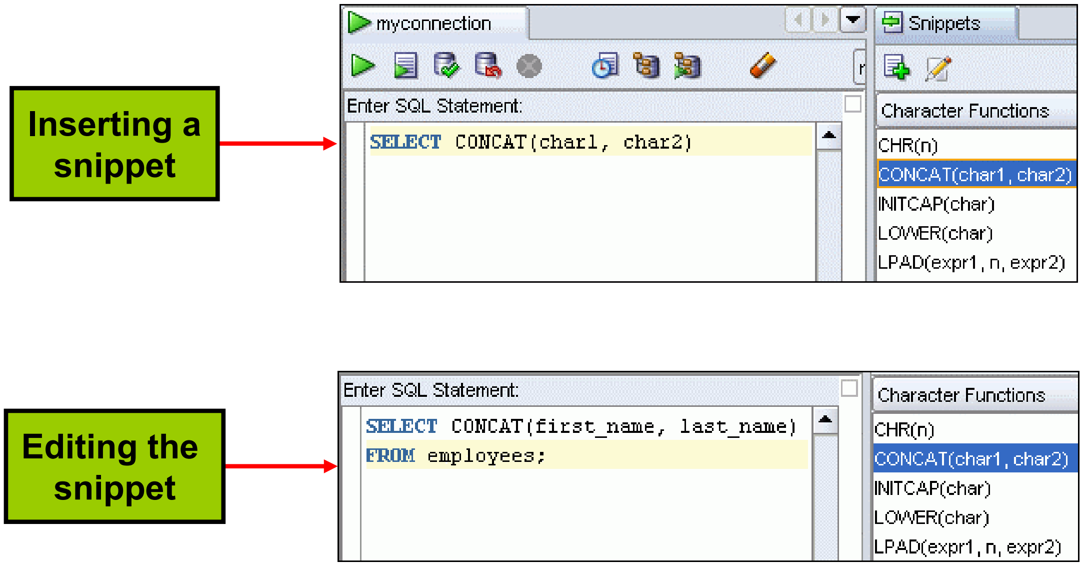
Using Snippets

You may want to use certain code fragments when you use the SQL Worksheet or create or edit a PL/SQL function or procedure. SQL Developer has the feature called Snippets. Snippets are code fragments such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets into the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed at the right side. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Using Snippets: Example



ORACLE

Using Snippets: Example

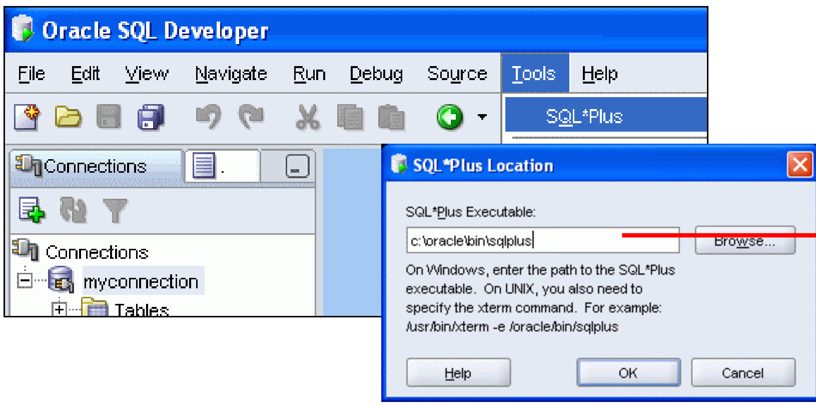
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added as in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Using SQL*Plus

- You can invoke the SQL*Plus command-line interface from SQL Developer.
- Close all the SQL Worksheets to enable the SQL*Plus menu option.



Provide the location of the `sqlplus.exe` file only the first time you invoke SQL*Plus.

Using SQL*Plus

The SQL Worksheet supports most of the SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL Worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database. To display the SQL*Plus command window, from the Tools menu, select **SQL*Plus**. To use this feature, the system on which you use SQL Developer must have an Oracle home directory or folder, with a SQL*Plus executable under that location. If the location of the SQL*Plus executable is not already stored in your SQL Developer preferences, you are asked to specify its location.

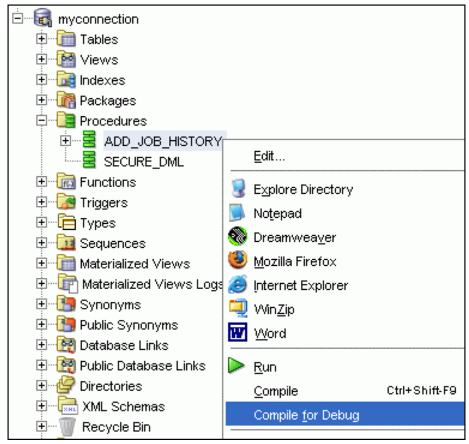
For example, some of the SQL*Plus statements that are not supported by SQL Worksheet are:

- `append`
- `archive`
- `attribute`
- `break`

For the complete list of SQL*Plus statements that are either supported or not supported by SQL Worksheet, refer to the *SQL*Plus Statements Supported and Not Supported in SQL Worksheet* topic in the SQL Developer online Help.

Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the Compile for Debug option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use Debug menu options to set breakpoints, and to perform step into, step over tasks.



Debugging Procedures and Functions

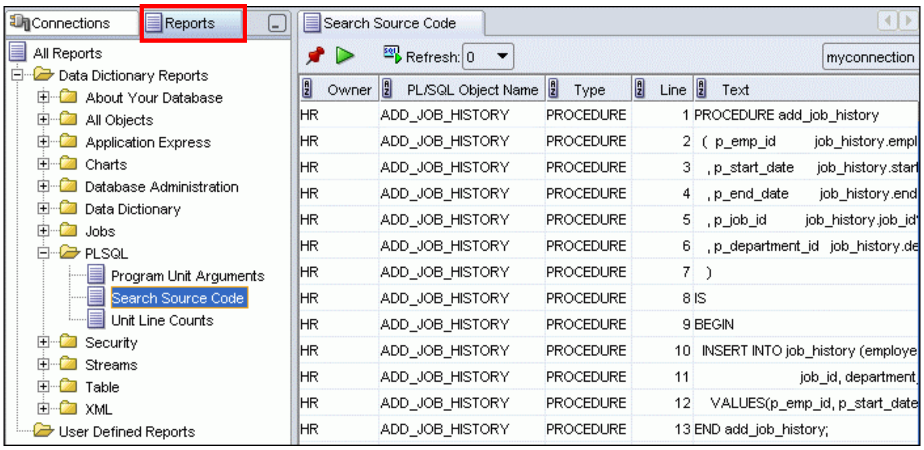
In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon in the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of more frequently accessed and more valid objects.

These options are also available as icons in the debugging toolbar.

Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.



ORACLE

Database Reporting

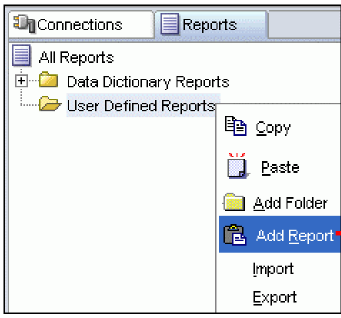
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

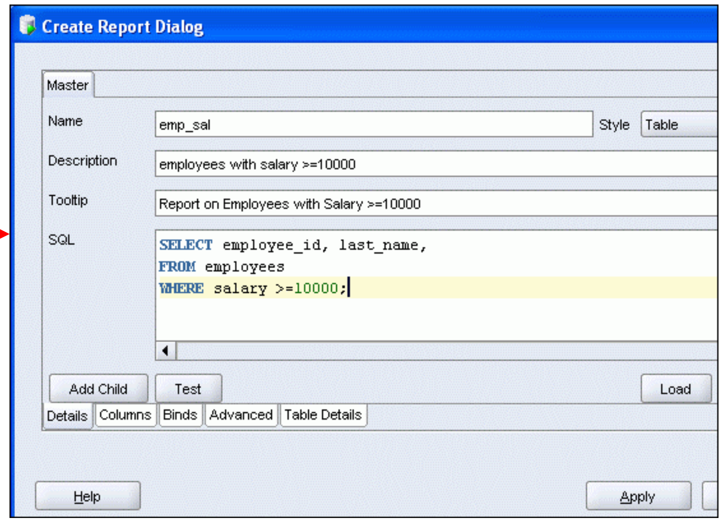
- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-Defined reports

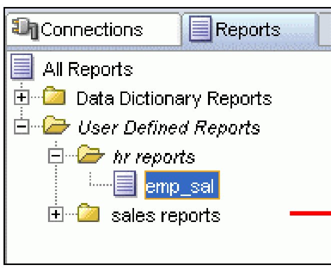
To display reports, click the Reports tab at the left side of the window. Individual reports are displayed in tabbed panes at the right side of the window; and for each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.







Organize reports in folders.

Creating a User-Defined Report

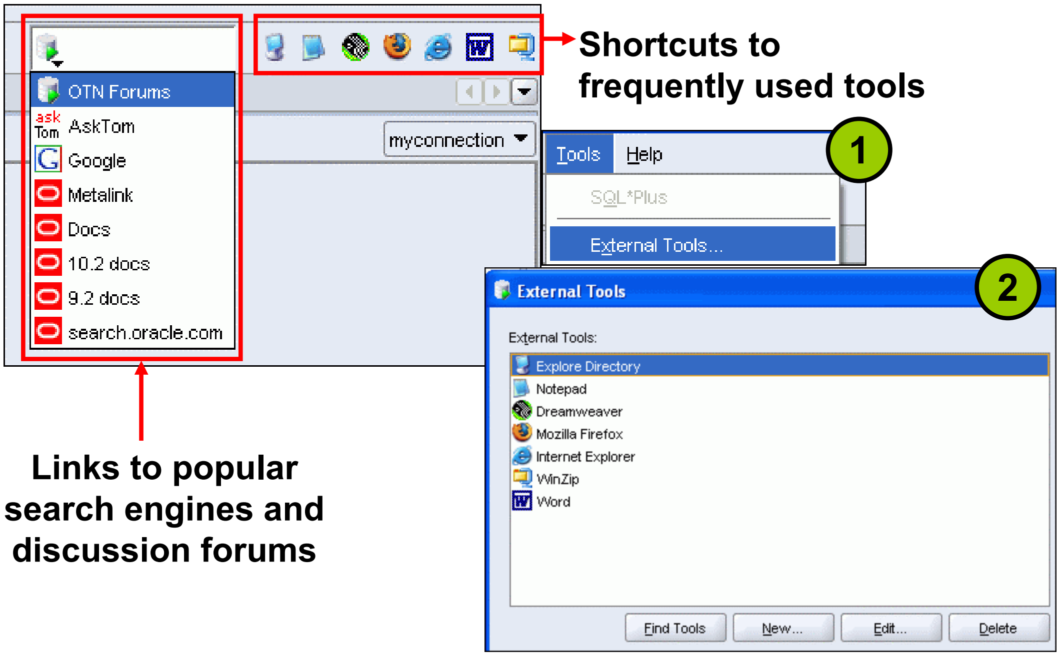
User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

1. Right-click the User Defined Reports node under Reports, and select Add Report.
2. In the Create Report Dialog box, specify the report name and the SQL query to retrieve information for the report. Then, click Apply.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details of employees with `salary >= 10000`. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the cursor stays briefly over the report name in the Reports navigator display.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined Reports node or any folder name under that node and select Add Folder. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Search Engines and External Tools



Search Engines and External Tools

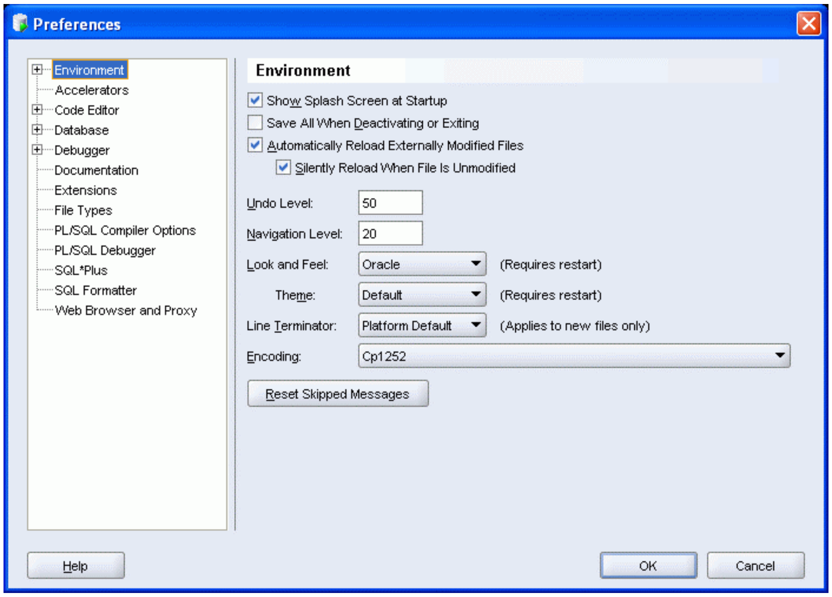
To enhance productivity of the SQL developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to tools that you do not use frequently. To do so, perform the following:

1. From the Tools menu, select External Tools.
2. In the External Tools dialog box, select New to add new tools. Select Delete to remove any tool from the list.

Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.



Setting Preferences

You can customize many aspects of the SQL Developer interface and environment by modifying SQL Developer preferences according to your preferences and needs. To modify SQL Developer preferences, select Tools, then Preferences.

Following are some of the categories that the preferences are grouped into:

- Environment
- Accelerators (Keyboard shortcuts)
- Code Editors
- Database
- Debugger
- Documentation
- Extensions
- File Types
- Migration
- PL/SQL Compilers
- PL/SQL Debugger

Specifications of SQL Developer 1.5.3

- SQL Developer 1.5.3 is the first translation release, and is a patch to Oracle SQL Developer 1.5.
- New feature list is available at:
  - http://www.oracle.com/technology/products/database/sql_developer/files/newFeatures_v15.html
- Supports Windows, Linux, and Mac OS X platforms
- To install, unzip the downloaded SQL Developer kit, which includes the required minimum JDK (JDK1.5.0_06).
- To start, double-click `sqldeveloper.exe`
- Connects to Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
  - http://www.oracle.com/technology/products/database/sql_developer/index.html

Specifications of SQL Developer 1.5.3

SQL Developer 1.5.3 is also available, as it is the latest version of the product that was available at the time of the release of this of course

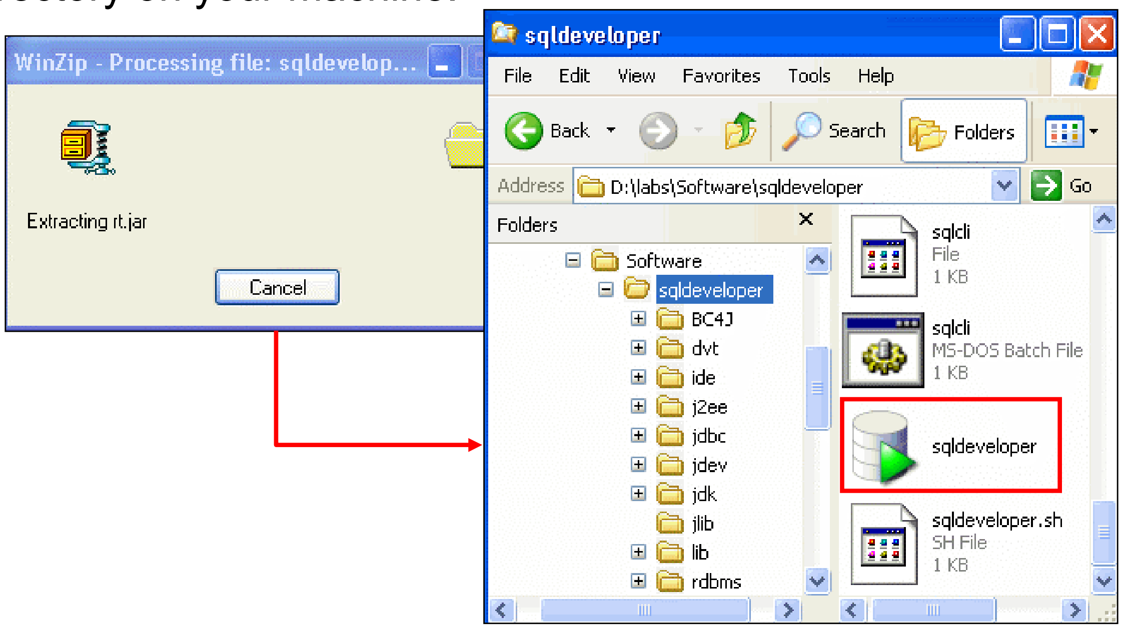
Like version 1.2, SQL Developer 1.5.3 is developed in Java leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms. You can install SQL Developer on the Database Server and connect remotely from your desktop, thus avoiding client/server network traffic.

Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, and therefore, no Oracle Home is required. The JDBC drivers that are shipped with version 1.5.3 support 11g R1. Therefore, users will no longer be able to connect to an Oracle 8.1.7 database.

SQL Developer does not require an installer and you need to simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions including Express Edition.

Installing SQL Developer 1.5.3

Download the Oracle SQL Developer kit and unzip into any directory on your machine.



Installing SQL Developer 1.5.3

Oracle SQL Developer does not require an installer. To install SQL Developer, you need an unzip tool.

To install SQL Developer, perform the following steps:

1. Create a folder. For example: `<local drive>:\software`
2. Download the SQL Developer kit from http://www.oracle.com/technology/products/database/sql_developer/index.html.
3. Unzip the downloaded SQL Developer kit into the folder created in step 1.

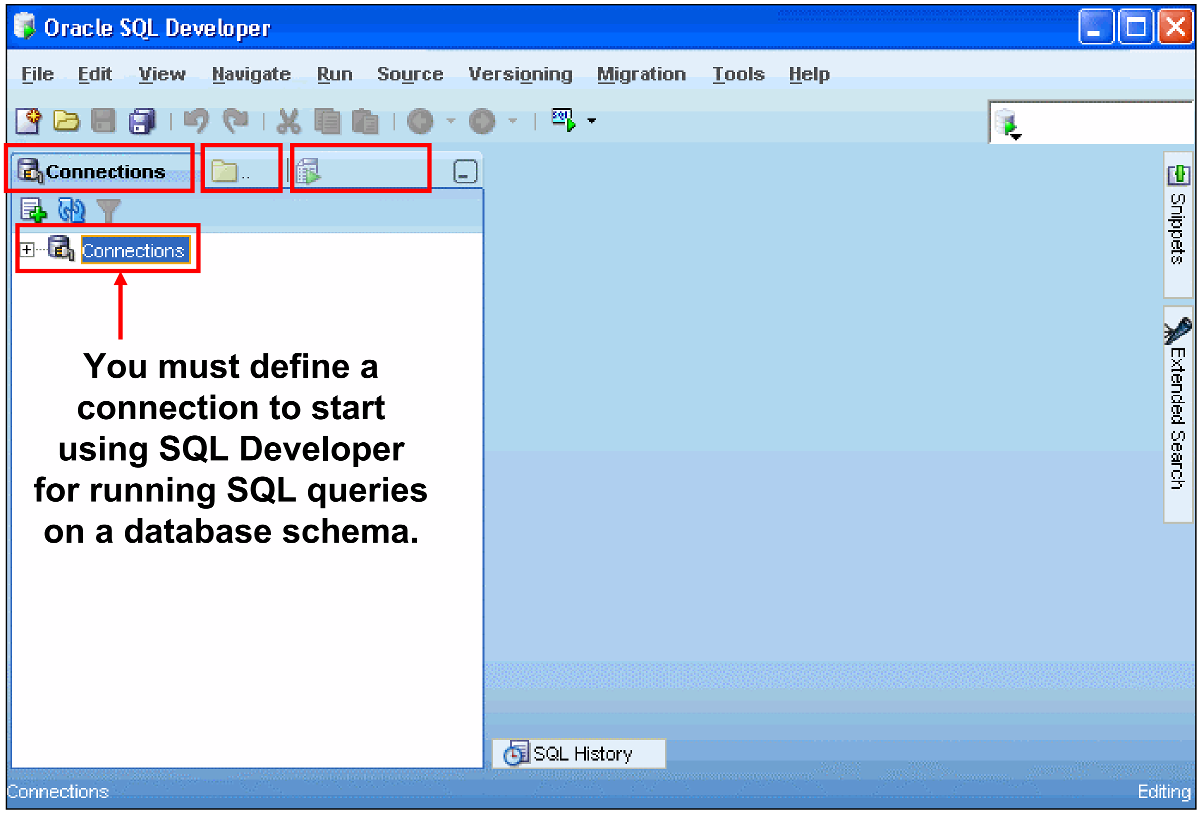
Starting SQL Developer

To start SQL Developer, go to `<local drive>:\software\sqldeveloper`, and double-click `sqldeveloper.exe`.

Notes:

- The SQL Developer 1.5.3 kit, named sqldeveloper-5783.zip, is located in is `d:\labs\software` on your classroom machine.
- When you open SQL Developer 1.5.3 for the first time, select **No** when prompted to migrate settings from a previous release.

SQL Developer 1.5.3 Interface



SQL Developer 1.5.3 Interface

The SQL Developer 1.5.3 interface contains all of the features found in version 1.2, and also some additional features.

Version 1.5.3 contains three main navigation tabs, from left to right:

- **Connections tab:** By using this tab, you can browse database objects and users to which you have access.
- **Files tab**: Identified by the Files folder icon, this tab enables you to access files from your local machine without having to use the File > Open menu.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.

General Navigation and Use

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

The features and functions that have been covered previously in this lesson for version 1.2, such as Creating a Connection, Browsing Database Objects, Creating Schema Objects, Using the SQL Worksheet, Using Snippets, Creating Reports, and Setting Preferences, are equivalent in the 1.5.3 interface.

Note: As with version 1.2, you need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures/functions.

SQL Developer 1.5.3 Interface (Continued)

Menus

The following menus contain standard entries, plus entries for features specific to SQL Developer:

- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to panes and in the execution of subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options.
- **Source:** Contains options for use when you edit functions and procedures
- **Versioning:** Provides integrated support for the following versioning and source control systems: CVS (Concurrent Versions System) and Subversion.
- **Migration:** Contains options related to migrating third-party databases to Oracle
- **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet

Note: The Run menu also contains options that are relevant when a function or procedure is selected for debugging. These are the same options that are found in the Debug menu in version 1.2.

Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports

Summary

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use SQL Worksheet to run SQL statements and scripts. SQL Developer enables you to create and save your own special set of reports for repeated use.

Version 1.2 is the default version set up for this class. Version 1.5.3 is also available on the classroom machine for use with all code examples, demos, and practices.

D

Review of PL/SQL

Block Structure for Anonymous PL/SQL Blocks

- `DECLARE` (optional)
  - Declare PL/SQL objects to be used within this block.
- `BEGIN` (mandatory)
  - Define the executable statements.
- `EXCEPTION` (optional)
  - Define the actions that take place if an error or exception arises.
- `END;` (mandatory)

ORACLE

Anonymous Blocks

Anonymous blocks do not have names. You declare them at the point in an application where they are to be run, and they are passed to the PL/SQL engine for execution at run time.

- The section between the keywords `DECLARE` and `BEGIN` is referred to as the declaration section. In the declaration section, you define the PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block. The `DECLARE` keyword is optional if you do not declare any PL/SQL objects.
- The `BEGIN` and `END` keywords are mandatory and enclose the body of actions to be performed. This section is referred to as the executable section of the block.
- The section between `EXCEPTION` and `END` is referred to as the exception section. The exception section traps error conditions. In it, you define actions to take if a specified condition arises. The exception section is optional.

The keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not followed by semicolons, but `END` and all other PL/SQL statements do require semicolons.

Declaring PL/SQL Variables

- Syntax:

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

- Examples:

```
Declare
  v_hiredate       DATE;
  v_deptno         NUMBER(2) NOT NULL := 10;
  v_location       VARCHAR2(13) := 'Atlanta';
  c_ comm          CONSTANT NUMBER := 1400;
  v_count          BINARY_INTEGER := 0;
  v_valid          BOOLEAN NOT NULL := TRUE;
```

ORACLE

Declaring PL/SQL Variables

You need to declare all PL/SQL identifiers within the declaration section before referencing them within the PL/SQL block. You have the option to assign an initial value. You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must be sure to declare them separately in a previous statement.

In the syntax:

`Identifier` is the name of the variable

`CONSTANT` constrains the variable so that its value cannot change; constants must be initialized.

`Datatype` is a scalar, composite, reference, or `LOB` data type (This course covers only scalar and composite data types.)

`NOT NULL` constrains the variable so that it must contain a value; `NOT NULL` variables must be initialized.

`expr` is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions

Declaring Variables with the %TYPE Attribute: Examples

```
...
  v_ename                employees.last_name%TYPE;
  v_balance              NUMBER(7,2);
  v_min_balance          v_balance%TYPE := 10;
...
```


Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

```
...
v_ename          employees.last_name%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10.

```
...
v_balance        NUMBER(7,2);
v_min_balance    v_balance%TYPE := 10;
...
```

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

Creating a PL/SQL Record

- Declare variables to store the name, job, and salary of a new employee.

```
...
  TYPE emp_record_type IS RECORD
    (ename     VARCHAR2(25),
     job          VARCHAR2(10),
     sal       NUMBER(8,2));
  emp_record      emp_record_type;
...
```


Creating a PL/SQL Record

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

The following example shows that you can use the %TYPE attribute to specify a field data type:

```
DECLARE
  TYPE emp_record_type IS RECORD
    (empid  NUMBER(6) NOT NULL := 100,
     ename  employees.last_name%TYPE,
     job    employees.job_id%TYPE);
  emp_record     emp_record_type;
...
```

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

%ROWTYPE Attribute: Examples

- Declare a variable to store the same information about a department as is stored in the DEPARTMENTS table.

```
dept_record    departments%ROWTYPE;
```

- Declare a variable to store the same information about an employee as is stored in the EMPLOYEES table.

```
emp_record     employees%ROWTYPE;
```


Examples

The first declaration in the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID.

The second declaration in the slide creates a record with the same field names and field data types as a row in the EMPLOYEES table. The fields are EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID.

In the following example, you select column values into a record named job_record.

```
DECLARE
   job_record  jobs%ROWTYPE;
   ...
BEGIN
   SELECT * INTO job_record
   FROM   jobs
   WHERE  ...
```

Creating a PL/SQL Table

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


Creating a PL/SQL Table

There are no predefined data types for PL/SQL tables, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

Referencing a PL/SQL Table

Syntax

```
pl/sql_table_name(primary_key_value)
```

In this syntax, `primary_key_value` belongs to the `BINARY_INTEGER` type.

Reference the third row in a PL/SQL table `ENAME_TABLE`.

```
ename_table(3) ...
```

The magnitude range of a `BINARY_INTEGER` is –2,147,483,647 through 2,147,483,647. The primary key value can therefore be negative. Indexing need not start with 1.

Note: The `table.EXISTS(i)` statement returns `TRUE` if at least one row with index `i` is returned. Use the `EXISTS` statement to prevent an error that is raised in reference to a nonexistent table element.

`SELECT` Statements in PL/SQL: Example

The `INTO` clause is mandatory.

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


`INTO` Clause

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

You use the `INTO` clause to populate either PL/SQL variables or host variables.

Queries Must Return One and Only One Row

`SELECT` statements within a PL/SQL block fall into the ANSI classification of Embedded SQL, for which the following rule applies:

Queries must return one and only one row. More than one row or no row generates an error.

PL/SQL deals with these errors by raising standard exceptions, which you can trap in the exception section of the block with the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions. You should code `SELECT` statements to return a single row.

Inserting Data: Example

Add new employee information to the `EMPLOYEES` table.

```
DECLARE
  v_empid  employees.employee_id%TYPE;
BEGIN
  SELECT  employees_seq.NEXTVAL
  INTO    v_empno
  FROM    dual;
  INSERT INTO  employees(employee_id, last_name,
                         job_id, department_id)
  VALUES(v_empid, 'HARDING', 'PU_CLERK', 30);
END;
```


Inserting Data

- Use SQL functions, such as `USER` and `SYSDATE`.
- Generate primary key values by using database sequences.
- Derive values in the PL/SQL block.
- Add column default values.

Note: There is no possibility for ambiguity with identifiers and column names in the `INSERT` statement. Any identifier in the `INSERT` clause must be a database column name.

Updating Data: Example

Increase the salary of all employees in the EMPLOYEES table who are purchasing clerks.

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 2000;
BEGIN
  UPDATE  employees
  SET     salary = salary + v_sal_increase
  WHERE   job_id = 'PU_CLERK';
END;
```


Updating Data

There may be ambiguity in the SET clause of the UPDATE statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable.

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

Note: PL/SQL variable assignments always use := and SQL column assignments always use =.. Remember that if column names and identifier names are identical in the WHERE clause, the Oracle server looks to the database first for the name.

Deleting Data: Example

Delete rows that belong to department 190 from the `EMPLOYEES` table.

```
DECLARE
  v_deptid   employees.department_id%TYPE := 190;
BEGIN
  DELETE FROM employees
  WHERE department_id = v_deptid;
END;
```


Deleting Data

Delete a specific job:

```
DECLARE
  v_jobid    jobs.job_id%TYPE := 'PR_REP';
BEGIN
  DELETE FROM jobs
  WHERE job_id = v_jobid;
END;
```

COMMIT and ROLLBACK Statements

- Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK statement.
- Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.

Controlling Transactions

You control the logic of transactions with COMMIT and ROLLBACK SQL statements, rendering some groups of database changes permanent while discarding others. As with the Oracle server, data manipulation language (DML) transactions start at the first command to follow a COMMIT or ROLLBACK and end on the next successful COMMIT or ROLLBACK. These actions may occur within a PL/SQL block or as a result of events in the host environment. A COMMIT ends the current transaction by making all pending changes to the database permanent.

Syntax

```
COMMIT [WORK];
ROLLBACK [WORK];
```

In this syntax, WORK is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as LOCK TABLE and SELECT ... FOR UPDATE) in a block. They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.

SQL Cursor Attributes

You can use SQL cursor attributes to test the outcome of your SQL statements.

SQL Cursor Attributes	Description
`SQL%ROWCOUNT`	Number of rows affected by the most recent SQL statement (an integer value)
`SQL%FOUND`	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement affects one or more rows
`SQL%NOTFOUND`	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement does not affect any rows
`SQL%ISOPEN`	Boolean attribute that always evaluates to `FALSE` because PL/SQL closes implicit cursors immediately after they are executed

SQL Cursor Attributes

SQL cursor attributes enable you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN` attributes in the exception section of a block to gather information about the execution of a DML statement. In PL/SQL, a DML statement that does not change any rows is not seen as an error condition, whereas the `SELECT` statement will return an exception if it cannot locate any rows.

IF, THEN, and ELSIF Statements: Example

For a given value entered, return a calculated value.

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := 0.5 * v_start;
ELSE
  v_start := 0.1 * v_start;
END IF;
. . .
```


IF, THEN, and ELSIF Statements

When possible, use the ELSIF clause instead of nesting IF statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the ELSE clause consists purely of another IF statement, it is more convenient to use the ELSIF clause. This makes the code clearer by removing the need for nested END IFs at the end of each further set of conditions and actions.

Example

```
IF condition1 THEN
  statement1;
ELSIF condition2 THEN
  statement2;
ELSIF condition3 THEN
  statement3;
END IF;
```

The statement in the slide is further defined as follows:

For a given value entered, return a calculated value. If the entered value is over 100, then the calculated value is two times the entered value. If the entered value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.

Basic Loop: Example

```
DECLARE
  v_ordid      order_items.order_id%TYPE := 101;
  v_counter    NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```


Basic Loop

The basic loop example shown in the slide is defined as follows:

Insert the first 10 new line items for order number 101.

Note: A basic loop enables execution of its statements at least once, even if the condition has been met upon entering the loop.

FOR Loop: Example

Insert the first 10 new line items for order number 101.

```
DECLARE
  v_ordid      order_items.order_id%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, i);
  END LOOP;
END;
```


FOR Loop

The slide shows a FOR loop that inserts 10 rows into the order_items table.

WHILE Loop: Example

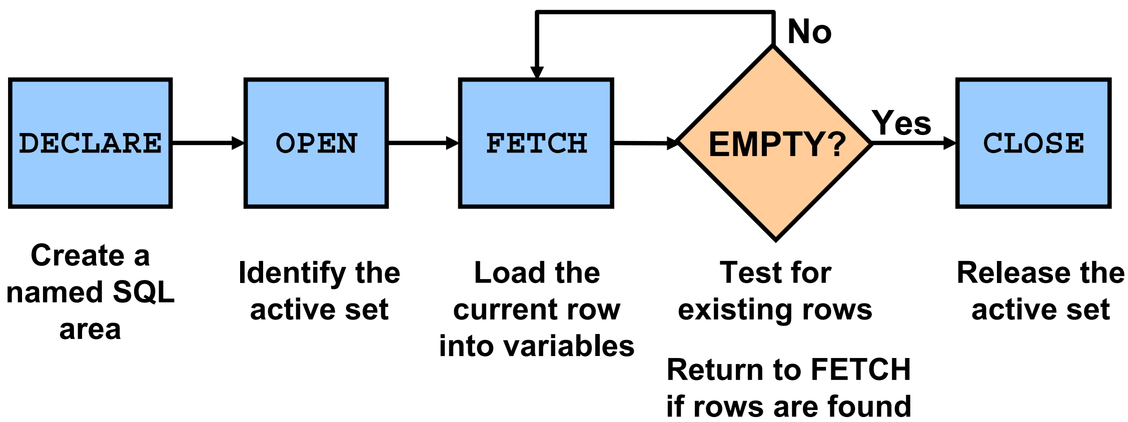
```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot -
 PROMPT 'Enter the maximum total for purchase of item: '
DECLARE
...
v_qty                NUMBER(8)  := 1;
v_running_total      NUMBER(7,2)  := 0;

BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
  v_qty := v_qty + 1;
  v_running_total := v_qty * &p_price;
  END LOOP;
...
```

ORACLE

WHILE Loop

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

Controlling Explicit Cursors



Explicit Cursors

Controlling Explicit Cursors Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The `OPEN` statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor. The `FETCH` statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore, each fetch accesses a different row returned by the query. In the flow diagram in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise, it closes the cursor.
4. Close the cursor. The `CLOSE` statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Declaring the Cursor: Example

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR c2 IS
    SELECT *
    FROM   departments
    WHERE  department_id = 10;
BEGIN
  ...
```


Explicit Cursor Declaration

Retrieve the employees one by one.

```
DECLARE
  v_empid   employees.employee_id%TYPE;
  v_ename   employees.last_name%TYPE;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
...
```

Note: You can reference variables in the query, but you must declare them before the CURSOR statement.

Opening the Cursor

```
OPEN cursor_name;
```

- Open the cursor to execute the query and identify the active set.
- If the query returns no rows, no exception is raised.
- Use cursor attributes to test the outcome after a fetch.

`OPEN` Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax, `cursor_name` is the name of the previously declared cursor.

`OPEN` is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information
2. Parses the `SELECT` statement
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the `OPEN` statement is executed. Rather, the `FETCH` statement retrieves the rows.
5. Positions the pointer just before the first row in the active set

Note: If the query returns no rows when the cursor is opened, then PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared by using the `FOR UPDATE` clause, the `OPEN` statement also locks those rows.

Fetching Data from the Cursor: Examples

```
FETCH c1 INTO v_empid, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```


`FETCH` Statement

You use the `FETCH` statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the `INTO` list. Also, their data types must be compatible. Retrieve the first 10 employees one by one:

```
DECLARE
  v_empid   employees.employee_id%TYPE;
  v_ename   employees.last_name%TYPE;
  i            NUMBER := 1;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename;
    ...
  END LOOP;
END;
```

Closing the Cursor

```
CLOSE   cursor_name;
```

- Close the cursor after completing the processing of the rows.
- Reopen the cursor, if required.
- Do not attempt to fetch data from a cursor after it has been closed.

ORACLE

`CLOSE` Statement

The `CLOSE` statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the `SELECT` statement. This step allows the cursor to be reopened, if required. Therefore, you can establish an active set several times.

In the syntax, `cursor_name` is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor after it has been closed, or the `INVALID_CURSOR` exception will be raised.

Note: The `CLOSE` statement releases the context area. Although it is possible to terminate the PL/SQL block without closing cursors, you should always close any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the `OPEN_CURSORS` parameter in the database parameter field. By default, the maximum number of `OPEN_CURSORS` is 50.

```
...
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename; ...
  END LOOP;
  CLOSE c1;
END;
```

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
`ISOPEN`	`BOOLEAN`	Evaluates to `TRUE` if the cursor is open
`%NOTFOUND`	`BOOLEAN`	Evaluates to `TRUE` if the most recent fetch does not return a row
`%FOUND`	`BOOLEAN`	Evaluates to `TRUE` if the most recent fetch returns a row; complement of `%NOTFOUND`
`%ROWCOUNT`	`NUMBER`	Evaluates to the total number of rows returned so far

Explicit Cursor Attributes

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

Note: Do not reference cursor attributes directly in a SQL statement.

Cursor FOR Loops: Example

Retrieve employees one by one until there are no more left.

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```


Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. The cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched. In the slide example, `emp_record` in the cursor for loop is an implicitly declared record that is used in the `FOR LOOP` construct.

`FOR UPDATE` Clause: Example

Retrieve the orders for amounts over $1,000 that were processed today.

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
      AND  order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```


`FOR UPDATE` Clause

If the database server cannot acquire the locks on the rows it needs in a `SELECT FOR UPDATE`, then it waits indefinitely. You can use the `NOWAIT` clause in the `SELECT FOR UPDATE` statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore, you can retry opening the cursor *n* times before terminating the PL/SQL block.

If you intend to update or delete rows by using the `WHERE CURRENT OF` clause, you must specify a column name in the `FOR UPDATE OF` clause.

If you have a large table, you can achieve better performance by using the `LOCK TABLE` statement to lock all rows in the table. However, when using `LOCK TABLE`, you cannot use the `WHERE CURRENT OF` clause and must use the notation `WHERE` *`column = identifier`*.

WHERE CURRENT OF Clause: Example

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
    ...
  END LOOP;
  COMMIT;
END;
```


WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your `DELETE` or `UPDATE` statement to contain the `WHERE CURRENT OF cursor_name` clause to refer to the latest row processed by the `FETCH` statement. When you use this clause, the cursor you reference must exist and must contain the `FOR UPDATE` clause in the cursor query; otherwise, you get an error. This clause enables you to apply updates and deletes to the currently addressed row without the need to explicitly reference the `ROWID` pseudocolumn.

Trapping Predefined Oracle Server Errors

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

ORACLE

Trapping Predefined Oracle Server Errors

Trap a predefined Oracle server error by referencing its standard name within the corresponding exception-handling routine.

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

It is a good idea to always consider the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions, which are the most common.

Trapping Predefined Oracle Server Errors: Example

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```


Trapping Predefined Oracle Server Exceptions: Example

In the example in the slide, a message is printed out to the user for each exception. Only one exception is raised and handled at any time.

Non-Predefined Error

Trap for Oracle server error number –2292, which is an integrity constraint violation.

```
DECLARE
  e_products_invalid  EXCEPTION;
  PRAGMA EXCEPTION_INIT (
   e_products_invalid, -2292);
  v_message VARCHAR2(50);
BEGIN
. . .
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product ID
   specified is not valid.';
. . .
END;
```

1 2 3

Trapping a Non-Predefined Oracle Server Exception

1. Declare the name for the exception within the declarative section.
   Syntax

   ```
   exception   EXCEPTION;
   ```

   In this syntax, *exception* is the name of the exception.
2. Associate the declared exception with the standard Oracle server error number, using the `PRAGMA EXCEPTION_INIT` statement.
   Syntax

   ```
   PRAGMA EXCEPTION_INIT(exception, error_number);
   ```

   In this syntax:

exception	Is the previously declared exception
error_number	Is a standard Oracle server error number
3. Reference the declared exception within the corresponding exception-handling routine.
   In the slide example: If there is product in stock, halt processing and print a message to the user.

User-Defined Exceptions: Example

```
[DECLARE]
  e_amount_remaining EXCEPTION;
. . .
BEGIN
. . .
  RAISE e_amount_remaining;
. . .

EXCEPTION
  WHEN e_amount_remaining THEN
    :g_message := 'There is still an amount
                   in stock.';
. . .
END;
```


Trapping User-Defined Exceptions

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

1. Declare the name for the user-defined exception within the declarative section.
   Syntax: `exception EXCEPTION;`
   where: *`exception`* Is the name of the exception
2. Use the `RAISE` statement to raise the exception explicitly within the executable section.
   Syntax: `RAISE exception;`
   where: *`exception`* Is the previously declared exception
3. Reference the declared exception within the corresponding exception-handling routine.

In the slide example: This customer has a business rule that states that a product cannot be removed from its database if there is any inventory left in stock for this product. Because there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a `DELETE` on the `PRODUCT_INFORMATION` table, the block queries the `INVENTORIES` table to see whether there is any stock for the product in question. If there is stock, raise an exception.

Note: Use the `RAISE` statement by itself within an exception handler to raise the same exception back to the calling environment.

RAISE_APPLICATION_ERROR Procedure

```
raise_application_error (error_number,
   message[, {TRUE | FALSE}]);
```

- Enables you to issue user-defined error messages from stored subprograms
- Is called from an executing stored subprogram only

RAISE_APPLICATION_ERROR Procedure

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax, *error_number* is a user-specified number for the exception between –20,000 and –20,999. The *message* is the user-specified message for the exception. It is a character string that is up to 2,048 bytes long.

TRUE | FALSE is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example:

```
 ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20201,
     'Manager is not a valid employee.');
END;
```

RAISE_APPLICATION_ERROR Procedure

- Is used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

ORACLE

RAISE_APPLICATION_ERROR Procedure: Example

```
...
DELETE FROM employees
WHERE  manager_id = v_mgr;
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20202,
    'This is not a valid manager');
END IF;
...
```

Using SQL*Plus

Objectives

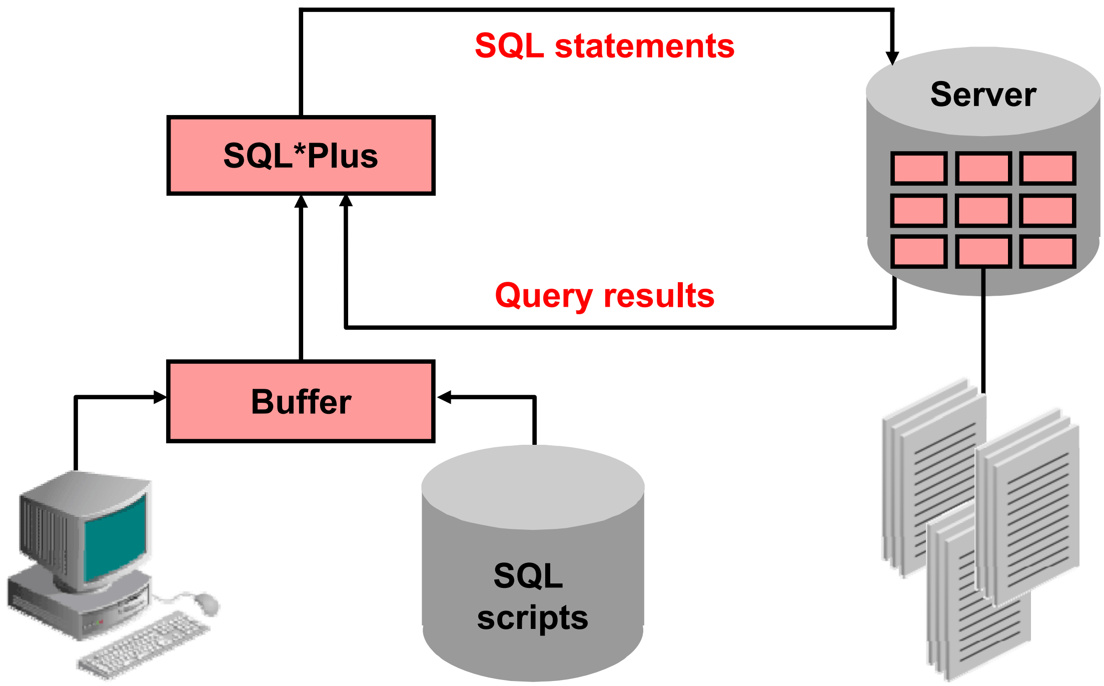
After completing this appendix, you should be able to do the following:

- Log in to SQL*Plus
- Edit SQL commands
- Format output using SQL*Plus commands
- Interact with script files

Objectives

You might want to create `SELECT` statements that can be used again and again. This appendix also covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



SQL and SQL*Plus

SQL is a command language for communication with the Oracle Server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the *SQL buffer* and remains there until you enter a new SQL statement. SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle Server for execution. It contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases

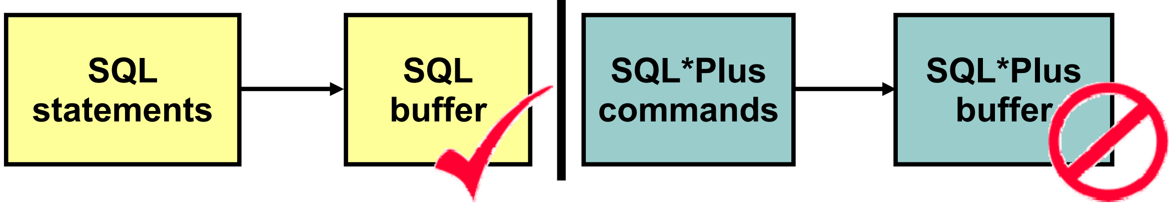
SQL Statements Versus SQL*Plus Commands

SQL

- A language
- ANSI-standard
- Keywords cannot be abbreviated
- Statements manipulate data and table definitions in the database

SQL*Plus

- An environment
- Oracle-proprietary
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database



SQL and SQL*Plus (continued)

The following table compares SQL and SQL*Plus:

SQL	SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI)–standard SQL	Is the Oracle-proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time, not stored in the SQL buffer
Does not have a continuation character	Uses a dash (–) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute commands immediately	Does not require termination characters; executes commands immediately
Uses functions to perform some formatting	Uses commands to format data

Overview of SQL*Plus

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from file to buffer to edit.

SQL*Plus

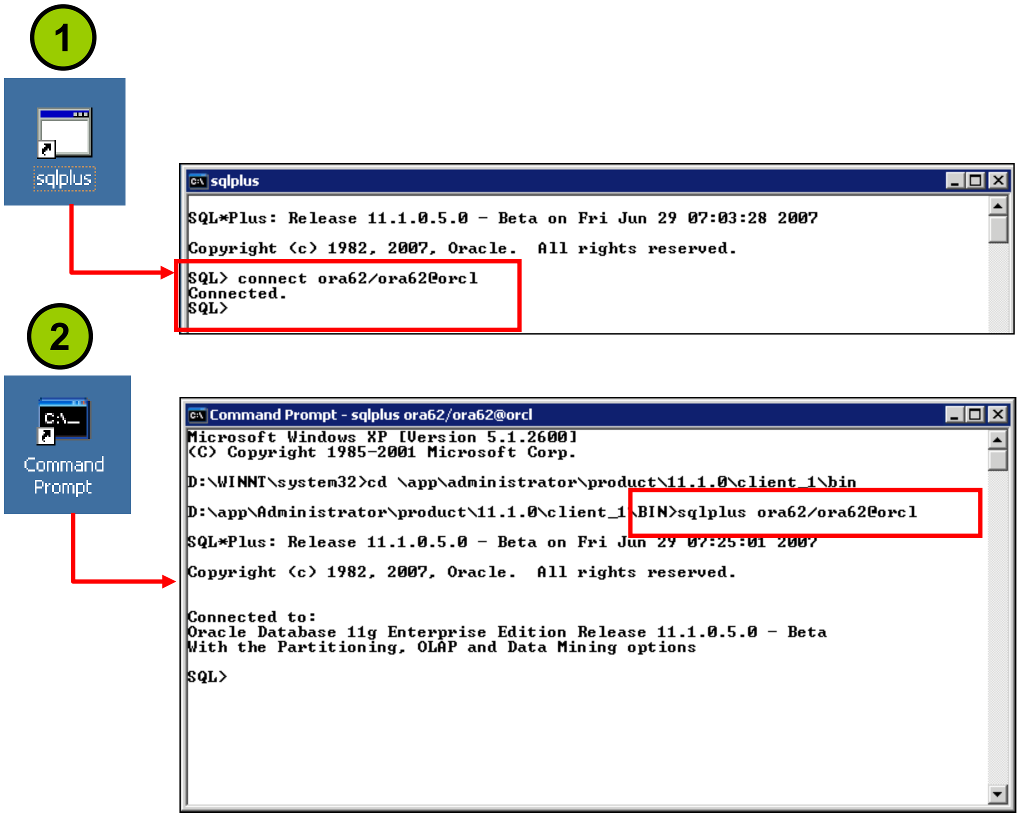
SQL*Plus is an environment in which you can do the following:

- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repeated use in the future

SQL*Plus commands can be divided into the following main categories:

Category	Purpose
Environment	Affect the general behavior of SQL statements for the session
Format	Format query results
File manipulation	Save, load, and run script files
Execution	Send SQL statements from the SQL buffer to the Oracle server
Edit	Modify SQL statements in the buffer
Interaction	Create and pass variables to SQL statements, print variable values, and print messages to the screen
Miscellaneous	Connect to the database, manipulate the SQL*Plus environment, and display column definitions

Logging In to SQL*Plus: Available Methods



Logging In to SQL*Plus

How you invoke SQL*Plus depends on which type of operating system or Windows environment you are running.

To log in from a Windows environment:

1. Select Start > Programs > Oracle > Application Development > SQL*Plus.
2. Enter the username, password, and database name.

To log in from a command-line environment:

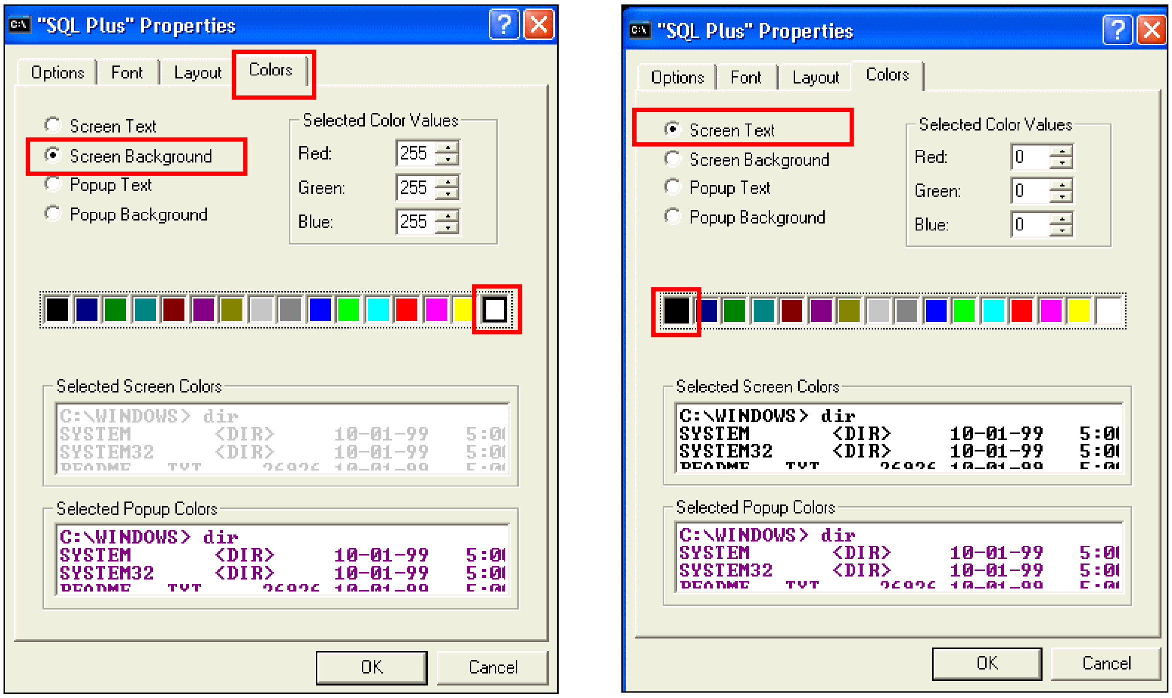
1. Log on to your machine.
2. Enter the `sqlplus` command shown in the slide.

In the syntax:

username	Your database username
password	Your database password (Your password is visible if you enter it here.)
@database	The database connect string

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the password prompt.

Customizing the SQL*Plus Environment



Changing Settings of the SQL*Plus Environment

You can optionally change the look of the SQL*Plus environment by using the SQL*Plus Properties dialog box.

In the SQL*Plus window, right-click the title bar and in the shortcut menu that appears, select Properties. You can then use the colors tab of the SQL*Plus Properties dialog box to set Screen Background and Screen Text.

Displaying Table Structure

Use the SQL*Plus `DESCRIBE` command to display the structure of a table:

```
DESC[RIBE]  tablename
```


Displaying Table Structure

In SQL*Plus, you can display the structure of a table using the `DESCRIBE` command. The result of the command is a display of column names and data types as well as an indication of whether a column must contain data.

In the syntax:

tablename Is the name of any existing table, view, or synonym that is accessible to the user

To describe the `JOB_GRADES` table, use this command:

```
SQL> DESCRIBE job_grades
Name                                  Null?      Type
----------------------------------  --------  -----------
GRADE_LEVEL                                     VARCHAR2(3)
LOWEST_SAL                                      NUMBER
HIGHEST_SAL                                     NUMBER
```

Displaying Table Structure

```
DESCRIBE departments
```

```
Name                        Null?     Type
-----------------------     -------- ------------
DEPARTMENT_ID               NOT NULL NUMBER(4)
DEPARTMENT_NAME             NOT NULL VARCHAR2(30)
MANAGER_ID                  NUMBER(6)
LOCATION_ID                 NUMBER(4)
```

ORACLE

Displaying Table Structure (continued)

The example in the slide displays the information about the structure of the `DEPARTMENTS` table. In the result:

`Null?`: Specifies whether a column must contain data (`NOT NULL` indicates that a column must contain data.)

`Type`: Displays the data type for a column

The following table describes the data types:

Data Type	**Description**
`NUMBER(p,s)`	Number value that has a maximum number of digits *p*, which is the number of digits to the right of the decimal point *s*
`VARCHAR2(s)`	Variable-length character value of maximum size *s*
`DATE`	Date and time value between January 1, 4712 B.C., and A.D. December 31, 9999
`CHAR(s)`	Fixed-length character value of size *s*

SQL*Plus Editing Commands

- `A[PPEND] text`
- `C[HANGE] / old / new`
- `C[HANGE] / text /`
- `CL[EAR] BUFF[ER]`
- `DEL`
- `DEL n`
- `DEL m n`

SQL*Plus Editing Commands

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

Command	Description
`A[PPEND] text`	Adds text to the end of the current line
`C[HANGE] / old / new`	Changes `old` text to `new` in the current line
`C[HANGE] / text /`	Deletes `text` from the current line
`CL[EAR] BUFF[ER]`	Deletes all lines from the SQL buffer
`DEL`	Deletes current line
`DEL n`	Deletes line `n`
`DEL m n`	Deletes lines `m` to `n`

Guidelines

- If you press [Enter] before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer by either entering one of the terminator characters (semicolon or slash) or pressing [Enter] twice. The SQL prompt then appears.

SQL*Plus Editing Commands

- `I[NPUT]`
- `I[NPUT] text`
- `L[IST]`
- `L[IST] n`
- `L[IST] m n`
- `R[UN]`
- `n`
- `n text`
- `0 text`

SQL*Plus Editing Commands (continued)

Command	Description
`I[NPUT]`	Inserts an indefinite number of lines
`I[NPUT] text`	Inserts a line consisting of `text`
`L[IST]`	Lists all lines in the SQL buffer
`L[IST] n`	Lists one line (specified by `n`)
`L[IST] m n`	Lists a range of lines (`m` to `n`)
`R[UN]`	Displays and runs the current SQL statement in the buffer
`n`	Specifies the line to make the current line
`n text`	Replaces line `n` with `text`
`0 text`	Inserts a line before line 1

Note: You can enter only one SQL*Plus command for each SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (-).

Using `LIST`, `n`, and `APPEND`

```
LIST
  1  SELECT last_name
  2* FROM   employees
```

```
1
  1* SELECT last_name
```

```
A , job_id
  1* SELECT last_name, job_id
```

```
LIST
  1  SELECT last_name, job_id
  2* FROM   employees
```


Using `LIST`, `n`, and `APPEND`

- Use the `L[IST]` command to display the contents of the SQL buffer. The asterisk (`*`) beside line 2 in the buffer indicates that line 2 is the current line. Any edits that you made apply to the current line.
- Change the number of the current line by entering the number (*n*) of the line that you want to edit. The new current line is displayed.
- Use the `A[PPEND]` command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the `LIST` command.

Note: Many SQL*Plus commands, including `LIST` and `APPEND`, can be abbreviated to just their first letters. `LIST` can be abbreviated to `L`; `APPEND` can be abbreviated to `A`.

Using the CHANGE Command

```
LIST
  1* SELECT * from employees
```

```
c/employees/departments
  1* SELECT * from departments
```

```
LIST
  1* SELECT * from departments
```


Using the CHANGE Command

- Use L[IST] to display the contents of the buffer.
- Use the C[HANGE] command to alter the contents of the current line in the SQL buffer. In this case, replace the EMPLOYEES table with the DEPARTMENTS table. The new current line is displayed.
- Use the L[IST] command to verify the new contents of the buffer.

SQL*Plus File Commands

- SAVE *filename*
- GET *filename*
- START *filename*
- @ *filename*
- EDIT *filename*
- SPOOL *filename*
- EXIT

SQL*Plus File Commands

SQL statements communicate with the Oracle server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands described in the following table:

Command	Description
`SAV[E] filename [.ext] [REP[LACE]APP[END]]`	Saves current contents of SQL buffer to a file. Use `APPEND` to add to an existing file; use `REPLACE` to overwrite an existing file. The default extension is `.sql`.
`GET filename [.ext]`	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is `.sql`.
`STA[RT] filename [.ext]`	Runs a previously saved command file
`@ filename`	Runs a previously saved command file (same as `START`)
`ED[IT]`	Invokes the editor and saves the buffer contents to a file named `afiedt.buf`
`ED[IT] [filename[.ext]]`	Invokes the editor to edit the contents of a saved file
`SPO[OL] [filename[.ext]| OFF|OUT]`	Stores query results in a file. `OFF` closes the spool file. `OUT` closes the spool file and sends the file results to the printer.
`EXIT`	Quits SQL*Plus

Using the SAVE, START, and EDIT Commands

```
LIST
  1  SELECT last_name, manager_id, department_id
  2* FROM employees
```

```
SAVE my_query
  Created file my_query
```

```
START my_query

LAST_NAME                 MANAGER_ID DEPARTMENT_ID
------------------------- ---------- -------------
King                                            90
Kochhar                          100            90
...
107 rows selected.
```

ORACLE

Using the SAVE, START, and EDIT Commands

SAVE

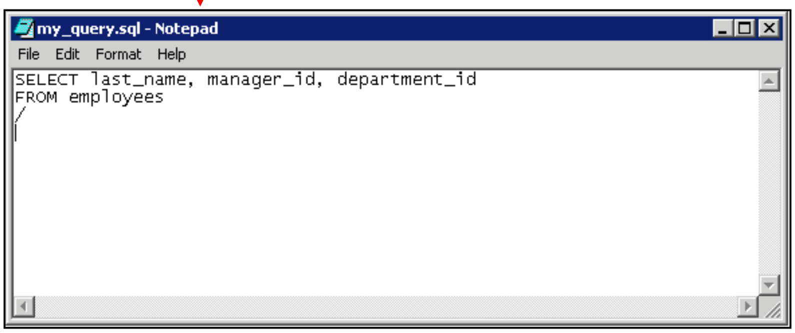
Use the SAVE command to store the current contents of the buffer in a file. In this way, you can store frequently used scripts for use in the future.

START

Use the START command to run a script in SQL*Plus.

Using the `SAVE`, `START`, and `EDIT` Commands

```
EDIT my_query
```



Using the `SAVE`, `START`, and `EDIT` Commands (continued)

`EDIT`

Use the `EDIT` command to edit an existing script. This opens an editor with the script file in it. When you have made the changes, quit the editor to return to the SQL*Plus command line.

SQL*Plus Enhancements Since Oracle Database 10*g*

- Changes to the `SET SERVEROUT[PUT]` command
- White space support in file and path names in Windows
- Three new predefined SQL*Plus variables
- The new `RECYCLEBIN` clause of the `SHOW` command
- The new `APPEND`, `CREATE`, and `REPLACE` extensions to the `SPOOL` command
- New error messages for the `COPY` command
- Change in the `DESCRIBE` command behavior
- New `PAGESIZE` default
- New `SQLPLUS` program compatibility option
- Execution statistics information in the `AUTOTRACE` command report

ORACLE

Changes to the SERVEROUTPUT Command

- Use the SET SERVEROUT[PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32,767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

```
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}]
    [FOR[MAT] {WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}]
```


New SQL*Plus Enhancements Since Oracle Database 10*g*

Most PL/SQL input and output is through SQL statements, to store data in database tables or query those tables. All other PL/SQL I/O is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

Note

- SIZE sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is UNLIMITED. *n* cannot be less than 2,000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see the *Oracle Database PL/SQL User's Guide and Reference 11g Release 1 (11.1)*

White Space Support in File and Path Names in Windows

- In Windows, white space can be included in file names and paths.
- Examples of where white space can be used:
  - `START`, `@`, `@@`, `RUN`, `SPOOL`, `SAVE`, and `EDIT` commands
- To reference files or paths containing spaces, enclose the name or path in double quotation marks.

Examples

```
SAVE "Monthly Report.sql"
START "Monthly Report.sql"
```

Predefined SQL*Plus Variables

Variable Name	Contains
`_CONNECT_IDENTIFIER`	Connection identifier used to make connection, where available
`_DATE`	Current date, or a user-defined fixed string
`_EDITOR`	Specifies the editor used by the `EDIT` command
`_O_VERSION`	Current version of the installed Oracle Database
`O_RELEASE`	Full release number of the installed Oracle Database
`_PRIVILEGE`	Privilege level of the current connection
`_SQLPLUS_RELEASE`	Full release number of installed SQL*Plus component
`_USER`	Username used to make connection

Predefined Variables

There are eight variables defined during SQL*Plus installation. These variables differ from user-defined variables by having only predefined values.

You can view the value of each of these variables with the `DEFINE` command. These variables can be accessed and redefined like any other substitution variable. They can be used in `TTITLE`, in `'&'` substitution variables, or in your SQL*Plus command-line prompt.

You can use the `DEFINE` command to view the definitions of these eight predefined variables in the same way as you view other `DEFINE` definitions. You can also use the `DEFINE` command to redefine their values, or you can use the `UNDEFINE` command to remove their definitions and make them unavailable.

Note: For additional information about the SQL*Plus predefined variables, see the *SQL*Plus User's Guide and Reference Release 11.1.*

Using the New Predefined SQL*Plus Variables: Examples

```
-- Change the SQL*Plus prompt to display the connection
-- identifier

SQL> SET SQLPROMPT '_CONNECT_IDENTIFIER > '
orcl >

-- view the predefined value of the _SQLPLUS_RELEASE
-- substitution variable

orcl > DEFINE _SQLPLUS_RELEASE
DEFINE _SQLPLUS_RELEASE = "1002000100" (CHAR)

-- View the user name connected to the current
-- connection.

orcl > DEFINE _USER
DEFINE _USER            = "HR" (CHAR)
```


Using the Predefined SQL*Plus Variables: Examples

To view all predefined and user-defined variable definitions, enter `DEFINE`. All predefined and all user-defined variable definitions are displayed as shown below:

```
orcl > DEFINE
DEFINE _DATE               = "06-JUL-06" (CHAR)
DEFINE _CONNECT_IDENTIFIER = "orcl" (CHAR)
DEFINE _USER               = "HR" (CHAR)
DEFINE _PRIVILEGE          = "" (CHAR)
DEFINE _SQLPLUS_RELEASE = "1002000100" (CHAR)
DEFINE _EDITOR             = "Notepad" (CHAR)
DEFINE _O_VERSION          = "Oracle Database 10g Enterprise
Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options" (CHAR)
DEFINE _O_RELEASE          = "1002000100" (CHAR)
```

You can use `UNDEFINE` to remove a substitution variable definition and make it unavailable.

The SHOW Command and the New RECYCLEBIN Clause

```
SHOW RECYC[LEBIN] [original_name]
SELECT * FROM USER_RECYCLEBIN
desc user_recyclebin;
Name            Null?    Type
--------------- -------- ------------
OBJECT_NAME     NOT NULL VARCHAR2(30)
ORIGINAL_NAME            VARCHAR2(32)
OPERATION                VARCHAR2(9)
TYPE                     VARCHAR2(25)
TS_NAME                  VARCHAR2(30)
CREATETIME               VARCHAR2(19)
DROPTIME                 VARCHAR2(19)
DROPSCN                  NUMBER
PARTITION_NAME           VARCHAR2(32)
CAN_UNDROP               VARCHAR2(3)
CAN_PURGE                VARCHAR2(3)
RELATED         NOT NULL NUMBER
BASE_OBJECT     NOT NULL NUMBER
PURGE_OBJECT    NOT NULL NUMBER
SPACE                    NUMBER
```

ORACLE

The SHOW Command and the RECYCLEBIN Clause

Using the SHOW command, you can show objects in the recycle bin that can be reverted with the FLASHBACK BEFORE DROP command. You do not need to remember column names, or interpret the less readable output from the query. The following query returns three columns that are displayed in the slide:

```
SELECT * FROM USER_RECYCLEBIN
```

The `SHOW` Command and the `RECYCLEBIN` Clause: Example

```
DROP TABLE test;
Table dropped.

SHOW recyclebin
```

```
ORIGINAL NAME    RECYCLEBIN NAME                OBJECT TYPE  DROP TIME
---------------- ------------------------------ ------------ -------------------
TEST             BIN$SefY+qPKSY6mvV8eDT1r+A==$0 TABLE        2006-07-06:11:12:00
SQL>
```

ORACLE

Using the SQL*Plus SPOOL Command

```
SPO[OL] [file_name[.ext] [CRE[ATE] | REP[LACE] |
    APP[END]] | OFF | OUT]
```

Option	Description
`file_name[.ext]`	Spools output to the specified file name
`CRE[ATE]`	Creates a new file with the name specified
`REP[LACE]`	Replaces the contents of an existing file. If the file does not exist, `REPLACE` creates the file.
`APP[END]`	Adds the contents of the buffer to the end of the file you specify
`OFF`	Stops spooling
`OUT`	Stops spooling and sends the file to your computer's standard (default) printer

Using the SQL*Plus SPOOL Command

The SPOOL command stores query results in a file, or optionally sends the file to a printer. The SPOOL command has been enhanced. You can now append to, or replace an existing file, where previously you could use SPOOL to only create (and replace) a file. REPLACE is the default.

To spool output generated by commands in a script without displaying the output on the screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect output from commands that run interactively.

You must use quotation marks around file names containing white spaces. To create a valid HTML file using SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. Set SQLPLUSCOMPAT[IBILITY] to 9.2 or earlier to disable the CREATE, APPEND, and SAVE parameters.

Using the SQL*Plus SPOOL Command: Examples

```
-- Record the output in the new file DIARY using the
-- default file extension.

SPOOL DIARY CREATE

-- Append the output to the existing file DIARY.

SPOOL DIARY APPEND

-- Record the output to the file DIARY, overwriting the
-- existing content

SPOOL DIARY REPLACE

-- Stop spooling and print the file on your default printer.

SPOOL OUT
```

The `COPY` Command: New Error Messages

```
CPY-0002 Illegal or missing APPEND, CREATE, INSERT, or
REPLACE option

CPY-0003 Internal Error: logical host number out of
Range

CPY-0004 Source and destination table and column names
don't match

CPY-0005 Source and destination column attributes don't
Match

CPY-0006 Select list has more columns than destination
Table

CPY-0007 Select list has fewer columns than destination
table
```

ORACLE

The `COPY` Command: New Error Messages

- **CPY-0002 Illegal or missing `APPEND`, `CREATE`, `INSERT`, or `REPLACE` option:** An internal `COPY` function has invoked `COPY` with a create option (flag) value that is out of range.
- **CPY-0003 Internal Error: Logical host number out of range:** An internal `COPY` function has been invoked with a logical host number value that is out of range.
- **CPY-0004 Source and destination table and column names don't match:** On an `APPEND` operation or an `INSERT` (when the table exists), at least one column name in the destination table does not match the corresponding column name in the optional column name list or in the `SELECT` command. To correct this, respecify the `COPY` command, making sure that the column names and their respective order in the destination table match the column names and column order in the optional column list or in the `SELECT` command.
- **CPY-0005 Source and destination column attributes don't match:** On an `APPEND` operation or an `INSERT` (when the table exists), at least one column in the destination table does not have the same data type as the corresponding column in the `SELECT` command. To correct this, respecify the `COPY` command, making sure that the data types for items being selected agree with the destination. Use `TO_DATE`, `TO_CHAR`, and `TO_NUMBER` to make conversions.

The COPY Command: New Error Messages (continued)

CPY-0006 Select list has more columns than destination table: On an APPEND operation or an INSERT (when the table exists), the number of columns in the SELECT command is greater than the number of columns in the destination table. To correct this, re-specify the COPY command, making sure that the number of columns being selected agrees with the number in the destination table.

CPY-0007 Select list has fewer columns than destination table: On an APPEND operation or INSERT (when the table exists), the number of columns in the SELECT command is less than the number of columns in the destination table. To correct this, re-specify the COPY command, making sure that the number of columns being selected agrees with the number in the destination table.

The `COPY` Command: New Error Messages

```
CPY-0008 More column list names than columns in the
destination table

CPY-0009 Fewer column list names than columns in the
destination table

CPY-0012 Datatype cannot be copied
```


The `COPY` Command: New Error Messages

- **CPY-0008 More column list names than columns in the destination table:** On an `APPEND` operation or an `INSERT` (when the table exists), the number of columns in the column name list is greater than the number of columns in the destination table. To correct this, re-specify the `COPY` command, making sure that the number of columns in the column list agrees with the number in the destination table.
- **CPY-0009 Fewer column list names than columns in the destination table:** On an `APPEND` operation or an `INSERT` (when the table exists), the number of columns in the column name list is less than the number of columns in the destination table. To correct this, re-specify the `COPY` command, making sure that the number of columns in the column list agrees with the number in the destination table.
- **CPY-0012 Datatype cannot be copied:** An attempt was made to copy a data type that is not supported in the `COPY` command. Data types supported by the `COPY` command are `CHAR`, `DATE`, `LONG`, `NUMBER`, and `VARCHAR2`. To correct this, re-specify the `COPY` command, making sure that the unsupported data type column is removed.

Change in the `DESCRIBE` Command Behavior

- Prior to Oracle Database 10*g*, using `DESCRIBE` on an invalidated object failed with the error:
  - `ORA-24372: invalid object for describe`
- The `DESCRIBE` command continued to fail even if the object had since been validated.
- Starting with Oracle Database 10*g*, the `DESCRIBE` command now automatically validates the object and continues if the validation is successful.

The `SET PAGES[IZE]` Command

- It sets the number of rows on each page of the output in SQL*Plus.
- The default `PAGESIZE` has changed from `24` to `14`.
- You can set `PAGESIZE` to zero to suppress all headings, page breaks, titles, the initial blank line, and other formatting information.

```
SET PAGES[IZE] {14 | n}
```


The `SET PAGES[IZE]` Command

The `SET PAGES[IZE]` command sets the number of rows displayed on each page. Error and informational messages are not counted in the page size, so pages may not always be exactly the same length. The default page size for SQL*Plus has changed from 24 to 14.

The SQLPLUS Program and the Compatibility Option

Sets the value of the SQLPLUSCOMPATIBILITY system variable to the SQL*Plus release specified by x.y[.z]

```
SQLPLUS -C[OMPATIBILITY] {x.y[.z]}

-- x is the version number
-- y is the release number
-- z is the update number
```

```
SQLPLUS -C 10.2.0
```


The SQLPLUS Program and the Compatibility Option

The SQL*Plus Compatibility Matrix tabulates behavior affected by each SQL*Plus compatibility setting. SQL*Plus compatibility modes can be set in three ways:

- You can include a `SET SQLPLUSCOMPATIBILITY` command in your site or user profile. On installation, there is no `SET SQLPLUSCOMPATIBILITY` setting in `glogin.sql`. Therefore, the default compatibility is 10.2.
- You can use the `SQLPLUS -C[OMPATIBILITY] {x.y[.z]}` command argument at startup to set the compatibility mode of that session.
- You can use the `SET SQLPLUSCOMPATIBILITY {x.y[.z]}` command during a session to set the SQL*Plus behavior you want for that session.

Note: For a list showing the release of SQL*Plus that introduced the behavior change, see the "SQL*Plus Compatibility Matrix" topic in *SQL*Plus User's Guide and Reference Release 11.1*.

Using the AUTOTRACE Command

- It displays a report after the successful execution of SQL DML statements such as SELECT, INSERT, UPDATE or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]]
    [STAT[ISTICS]]
```

```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics.
```

ORACLE

Using the AUTOTRACE Command

EXPLAIN shows the query execution path by performing an EXPLAIN PLAN. STATISTICS displays SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The additional information and tabular output of AUTOTRACE PLAN is supported when connecting to Oracle Database 10*g* (Release 10.1) or later. When you connect to an earlier database, the older form of AUTOTRACE reporting is used.

The DBMS_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several, predefined formats.

Note

- For additional information about the package and subprograms, see the *Oracle Database PL/SQL Packages and Types Reference 10g Release 2 (10.2)* guide.
- For additional information about the EXPLAIN PLAN, see *Oracle Database SQL Reference 10g Release 2 (10.2)*.
- For additional information about Execution Plans and the statistics, see *Oracle Database Performance Tuning Guide 10g Release 2 (10.2)*.

Displaying a Plan Table Using the DBMS_XPLAN.DISPLAY Package Function

```
-- Execute an explain plan command on a SELECT
-- statement

EXPLAIN PLAN FOR
SELECT * FROM emp e, dept d
   WHERE e.deptno = d.deptno
   AND e.ename='benoit';

-- Display the plan using the DBMS_XPLAN.DISPLAY table
-- function

SET LINESIZE 130
SET PAGESIZE 0
SELECT * FROM table(DBMS_XPLAN.DISPLAY);
```

ORACLE

Displaying a Plan Table Using the DBMS_XPLAN.DISPLAY Package Function

The query in the slide page produces the following output:

```
Plan hash value: 3693697075
---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost  (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    57 |      6  (34)| 00:00:01 |
|*  1 |  HASH JOIN         |      |     1 |    57 |      6  (34)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMP  |     1 |    37 |      3  (34)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| DEPT |     4 |    80 |      3  (34)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
1 - access("E"."DEPTNO"="D"."DEPTNO")
2 - filter("E"."ENAME"='benoit')

15 rows selected.
```

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format output
- Interact with script files

Summary

SQL*Plus is an execution environment that you can use to send SQL commands to the database server and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.

F

Studies for Implementing Triggers

Objectives

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

- Enhance database security with triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities using triggers

Lesson Aim

In this lesson, you learn to develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server. In some cases, it may be sufficient to refrain from using triggers and accept the functionality provided by the Oracle server.

This lesson covers the following business application scenarios:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Controlling Security Within the Server

Using database security with the `GRANT` statement.

```
GRANT SELECT, INSERT, UPDATE, DELETE
 ON    employees
 TO    clerk;                  -- database role
GRANT clerk TO scott;
```


Controlling Security Within the Server

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data-manipulation, and data-definition privileges.

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
dummy PLS_INTEGER;
BEGIN
 IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN')) THEN
   RAISE_APPLICATION_ERROR(-20506,'You may only
     change data during normal business hours.');
 END IF;
 SELECT COUNT(*) INTO dummy FROM holiday
 WHERE holiday_date = TRUNC (SYSDATE);
 IF dummy > 0 THEN
   RAISE_APPLICATION_ERROR(-20507,
     'You may not change data on a holiday.');
 END IF;
END;
/
```


Controlling Security with a Database Trigger

Develop triggers to handle more complex security requirements.

- Base privileges on any database values, such as the time of day, the day of the week, and so on.
- Determine access to tables only.
- Determine data-manipulation privileges only.

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD
  CONSTRAINT ck_salary CHECK (salary >= 500);
```

```
Table altered.
```


Enforcing Data Integrity Within the Server

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key.

Use these rules to:

- Provide constant default values
- Enforce static constraints
- Enable and disable dynamically

Example

The code sample in the slide ensures that the salary is at least $500.

Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
      'Do not decrease salary.');
END;
/
```

ORACLE

Protecting Data Integrity with a Trigger

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

Example

The code sample in the slide ensures that the salary is never decreased.

Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY (department_id)
     REFERENCES departments(department_id)
ON DELETE CASCADE;
```


Enforcing Referential Integrity Within the Server

Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

- Restrict updates and deletes.
- Cascade deletes.
- Enable and disable dynamically.

Example

When a department is removed from the DEPARTMENTS parent table, cascade the deletion to the corresponding rows in the EMPLOYEES child table.

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
 AFTER UPDATE OF department_id ON departments
 FOR EACH ROW
BEGIN
 UPDATE employees
  SET employees.department_id=:NEW.department_id
  WHERE employees.department_id=:OLD.department_id;
 UPDATE job_history
  SET department_id=:NEW.department_id
  WHERE department_id=:OLD.department_id;
END;
/
```

ORACLE

Protecting Referential Integrity with a Trigger

The following referential integrity rules are not supported by declarative constraints:

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.

You can develop triggers to implement these integrity rules.

Example

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table.

For a complete referential integrity solution using triggers, a single trigger is not enough.

Replicating a Table Within the Server

```
CREATE MATERIALIZED VIEW emp_copy
  NEXT sysdate + 7
  AS SELECT * FROM employees@ny;
```


Creating a Materialized View

Materialized views enable you to maintain copies of remote data on your local node for replication purposes. You can select data from a materialized view as you would from a normal database table or view. A materialized view is a database object that contains the results of a query, or a copy of some database on a query. The FROM clause of the query of a materialized view can name tables, views, and other materialized views.

When a materialized view is used, replication is performed implicitly by the Oracle server. This performs better than using user-defined PL/SQL triggers for replication. Materialized views:

- Copy data from local and remote tables asynchronously, at user-defined intervals
- Can be based on multiple master tables
- Are read-only by default, unless using the Oracle Advanced Replication feature
- Improve the performance of data manipulation on the master table

Alternatively, you can replicate tables using triggers.

The example in the slide creates a copy of the remote EMPLOYEES table from New York. The NEXT clause specifies a date-time expression for the interval between automatic refreshes.

Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
 BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN /* Proceed if user initiates data operation,
        NOT through the cascading trigger.*/
  IF INSERTING THEN
   IF :NEW.flag IS NULL THEN
     INSERT INTO employees@sf
     VALUES(:new.employee_id,...,'B');
     :NEW.flag := 'A';
   END IF;
  ELSE   /* Updating. */
   IF :NEW.flag = :OLD.flag THEN
     UPDATE employees@sf
      SET ename=:NEW.last_name,...,flag=:NEW.flag
      WHERE employee_id = :NEW.employee_id;
   END IF;
   IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
                      ELSE :NEW.flag := 'A';
   END IF;
  END IF;
END;
```

ORACLE

Replicating a Table with a Trigger

You can replicate a table with a trigger. By replicating a table, you can:

- Copy tables synchronously, in real time
- Base replicas on a single master table
- Read from replicas as well as write to them

Note: Excessive use of triggers can impair the performance of data manipulation on the master table, particularly if the network fails.

Example

In New York, replicate the local `EMPLOYEES` table to San Francisco.

Computing Derived Data Within the Server

```
UPDATE departments
 SET total_sal=(SELECT SUM(salary)
                FROM employees
                WHERE employees.department_id =
                     departments.department_id);
```


Computing Derived Data Within the Server

By using the server, you can schedule batch jobs or use the database Scheduler for the following scenarios:

- Compute derived column values asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

Example

Keep the salary total for each department within a special `TOTAL_SALARY` column of the `DEPARTMENTS` table.

Computing Derived Values with a Trigger

```
CREATE PROCEDURE increment_salary
  (id NUMBER, new_sal NUMBER) IS
BEGIN
   UPDATE departments
   SET    total_sal = NVL (total_sal, 0)+ new_sal
   WHERE  department_id = id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE
ON employees FOR EACH ROW
BEGIN
 IF DELETING THEN     increment_salary(
     :OLD.department_id,(-1*:OLD.salary));
 ELSIF UPDATING THEN  increment_salary(
     :NEW.department_id,(:NEW.salary-:OLD.salary));
 ELSE   increment_salary(
     :NEW.department_id,:NEW.salary); --INSERT
 END IF;
END;
```


Computing Derived Data Values with a Trigger

By using a trigger, you can perform the following tasks:

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a single pass to the database.

Example

Keep a running total of the salary for each department in the special TOTAL_SALARY column of the DEPARTMENTS table.

Logging Events with a Trigger

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
 ON inventories FOR EACH ROW
DECLARE
 dsc product_descriptions.product_description%TYPE;
 msg_text VARCHAR2(2000);
BEGIN
  IF :NEW.quantity_on_hand <=
     :NEW.reorder_point THEN
    SELECT product_description INTO dsc
    FROM product_descriptions
    WHERE product_id = :NEW.product_id;
  _ msg_text := 'ALERT: INVENTORY LOW ORDER:'||
       'Yours,' ||CHR(10) ||user || '.'|| CHR(10);
  ELSIF :OLD.quantity_on_hand >=
        :NEW.quantity_on_hand THEN
    msg_text := 'Product #'||... CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com','ord@oracle.com',
   message=>msg_text, subject=>'Inventory Notice');
END;
```


Logging Events with a Trigger

In the server, you can log events by querying data and performing operations manually. This sends an email message when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package `UTL_MAIL` to send the email message.

Logging Events Within the Server

1. Query data explicitly to determine whether an operation is necessary.
2. Perform the operation, such as sending a message.

Using Triggers to Log Events

1. Perform operations implicitly, such as firing off an automatic electronic memo.
2. Modify data and perform its dependent operation in a single step.
3. Log events automatically as data is changing.

Logging Events with a Trigger (continued)

Logging Events Transparently

In the trigger code:

- `CHR(10)` is a carriage return
- `Reorder_point` is not `NULL`
- Another transaction can receive and read the message in the pipe

Example

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF amount_in_stock, reorder_point
ON inventory  FOR EACH ROW
DECLARE
  dsc product.descrip%TYPE;
  msg_text VARCHAR2(2000);
BEGIN
  IF  :NEW.amount_in_stock <= :NEW.reorder_point THEN
    SELECT descrip INTO  dsc
    FROM PRODUCT WHERE prodid = :NEW.product_id;
    msg_text := 'ALERT: INVENTORY LOW ORDER:'||CHR(10)||
    'It has come to my personal attention that, due to recent'
    ||CHR(10)||'transactions, our inventory for product # '||
    TO_CHAR(:NEW.product_id)||'-- '|| dsc ||
    ' -- has fallen below acceptable levels.' || CHR(10) ||
    'Yours,' ||CHR(10) ||user || '.'|| CHR(10)|| CHR(10);
  ELSIF :OLD.amount_in_stock >= :NEW.amount_in_stock THEN
    msg_text := 'Product #'|| TO_CHAR(:NEW.product_id)
    ||' ordered. '|| CHR(10)|| CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com', 'ord@oracle.com',
     message => msg_text, subject => 'Inventory Notice');
END;
```

Summary

In this lesson, you should have learned how to:

- Enhance database security with triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities using triggers

Summary

This lesson provides some detailed comparison of using the Oracle database server functionality to implement security, auditing, data integrity, replication, and logging. The lesson also covers how database triggers can be used to implement the same features but go further to enhance the features that the database server provides. In some cases, you must use a trigger to perform some activities (such as computation of derived data) because the Oracle server cannot know how to implement this kind of business rule without some programming effort.

Using the DBMS_SCHEDULER and HTP Packages

Objectives

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

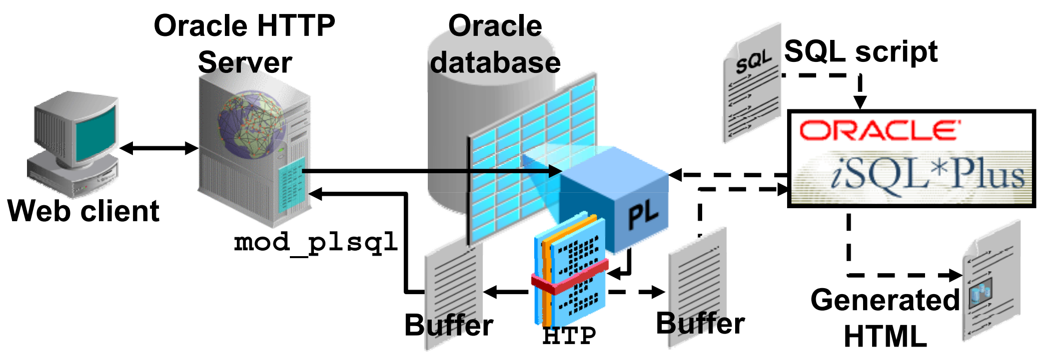
- Use the HTP package to generate a simple Web page
- Call the DBMS_SCHEDULER package to schedule PL/SQL code for execution

Lesson Aim

In this lesson, you learn how to use some of the Oracle-supplied packages and their capabilities. This lesson focuses on the packages that generate Web-based output and the provided scheduling capabilities.

Generating Web Pages with the HTP Package

- The HTP package procedures generate HTML tags.
- The HTP package is used to generate HTML documents dynamically and can be invoked from:
  – A browser using Oracle HTTP Server and PL/SQL Gateway (mod_plsql) services
  – An SQL*Plus script to display HTML output



Generating Web Pages with the HTP Package

The HTP package contains procedures that are used to generate HTML tags. The HTML tags that are generated typically enclose the data provided as parameters to the various procedures. The slide illustrates two ways in which the HTP package can be used:

- Most likely your procedures are invoked by the PL/SQL Gateway services, via the mod_plsql component supplied with Oracle HTTP Server, which is part of the Oracle Application Server product (represented by solid lines in the graphic).
- Alternatively (as represented by dotted lines in the graphic), your procedure can be called from *SQL**Plus that can display the generated HTML output, which can be copied and pasted to a file. This technique is used in this course because Oracle Application Server software is not installed as a part of the course environment.

Note: The HTP procedures output information to a session buffer held in the database server. In the Oracle HTTP Server context, when the procedure completes, the mod_plsql component automatically receives the buffer contents, which are then returned to the browser as the HTTP response. In SQL*Plus, you must manually execute:

- A SET SERVEROUTPUT ON command
- The procedure to generate the HTML into the buffer
- The OWA_UTIL.SHOWPAGE procedure to display the buffer contents

Using the HTP Package Procedures

- Generate one or more HTML tags. For example:

```
htp.bold('Hello');               -- <B>Hello</B>
htp.print('Hi <B>World</B>');    -- Hi <B>World</B>
```

- Are used to create a well-formed HTML document:

```
BEGIN                            -- Generates:
 htp.htmlOpen;    --------->
 htp.headOpen;    --------->     <HTML>
 htp.title('Welcome');   -->     <HEAD>
 htp.headClose;   --------->     <TITLE>Welcome</TITLE>
 htp.bodyOpen;    --------->     </HEAD>
 htp.print('My home page');      <BODY>
 htp.bodyClose;   --------->     My home page
 htp.htmlClose;   --------->     </BODY>
END;                             </HTML>
```


Using the HTP Package Procedures

The HTP package is structured to provide a one-to-one mapping of a procedure to standard HTML tags. For example, to display bold text on a Web page, the text must be enclosed in the HTML tag pair <B> and </B>. The first code box in the slide shows how to generate the word Hello in HTML bold text by using the equivalent HTP package procedure—that is, HTP.BOLD. The HTP.BOLD procedure accepts a text parameter and ensures that it is enclosed in the appropriate HTML tags in the HTML output that is generated.

The HTP.PRINT procedure copies its text parameter to the buffer. The example in the slide shows how the parameter supplied to the HTP.PRINT procedure can contain HTML tags. This technique is recommended only if you need to use HTML tags that cannot be generated by using the set of procedures provided in the HTP package.

The second example in the slide provides a PL/SQL block that generates the basic form of an HTML document. The example serves to illustrate how each of the procedures generates the corresponding HTML line in the enclosed text box on the right.

The benefit of using the HTP package is that you create well-formed HTML documents, eliminating the need to manually type the HTML tags around each piece of data.

Note: For information about all the HTP package procedures, refer to *PL/SQL Packages and Types Reference*.

Creating an HTML File with SQL*Plus

To create an HTML file with SQL*Plus, perform the following steps:

1. Create a SQL script with the following commands:

```
SET SERVEROUTPUT ON
ACCEPT procname PROMPT "Procedure: "
EXECUTE &procname
EXECUTE owa_util.showpage
UNDEFINE proc
```

2. Load and execute the script in SQL*Plus, supplying values for substitution variables.
3. Select, copy, and paste the HTML text that is generated in the browser to an HTML file.
4. Open the HTML file in a browser.

ORACLE

Creating an HTML File with SQL*Plus

The slide example shows the steps for generating HTML by using any procedure and saving the output into an HTML file. You should perform the following steps:

1. Turn on server output with the `SET SERVEROUTPUT ON` command. Without this, you receive exception messages when running procedures that have calls to the `HTP` package.
2. Execute the procedure that contains calls to the `HTP` package.
   Note: This does *not* produce output, unless the procedure has calls to the `DBMS_OUTPUT` package.
3. Execute the `OWA_UTIL.SHOWPAGE` procedure to display the text. This call actually displays the HTML content that is generated from the buffer.

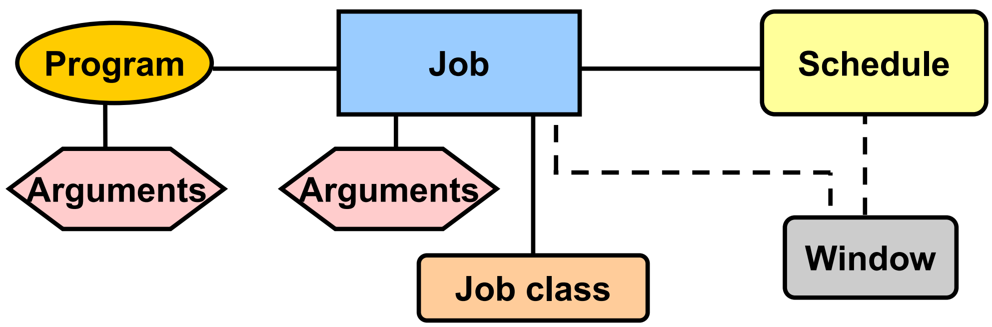
The `ACCEPT` command prompts for the name of the procedure to execute. The call to `OWA_UTIL.SHOWPAGE` displays the HTML tags in the browser window. You can then copy and paste the generated HTML tags from the browser window into an HTML file, typically with an `.htm` or `.html` extension.

Note: If you are using SQL*Plus, then you can use the `SPOOL` command to direct the HTML output directly to an HTML file.

The DBMS_SCHEDULER Package

The database Scheduler comprises several components to enable jobs to be run. Use the DBMS_SCHEDULER package to create each job with:

- A unique job name
- A program ("what" should be executed)
- A schedule ("when" it should run)



DBMS_SCHEDULER Package

Oracle Database provides a collection of subprograms in the DBMS_SCHEDULER package to simplify management and to provide a rich set of functionality for complex scheduling tasks. Collectively, these subprograms are called the Scheduler and can be called from any PL/SQL program. The Scheduler enables database administrators and application developers to control when and where various tasks take place. By ensuring that many routine database tasks occur without manual intervention, you can lower operating costs, implement more reliable routines, and minimize human error.

The diagram shows the following architectural components of the Scheduler:

- A **job** is the combination of a program and a schedule. Arguments required by the program can be provided with the program or the job. All job names have the format `[schema.]name`. When you create a job, you specify the job name, a program, a schedule, and (optionally) job characteristics that can be provided through a **job class**.
- A **program** determines what should be run. Every automated job involves a particular executable, whether it is a PL/SQL block, a stored procedure, a native binary executable, or a shell script. A program provides metadata about a particular executable and may require a list of arguments.
- A **schedule** specifies when and how many times a job is executed.

DBMS_SCHEDULER Package (continued)

- A **job class** defines a category of jobs that share common resource usage requirements and other characteristics. At any given time, each job can belong to only a single job class. A job class has the following attributes:
  - A database **service** name. The jobs in the job class will have an affinity to the particular service specified—that is, the jobs will run on the instances that cater to the specified service.
  - A **resource consumer group**, which classifies a set of user sessions that have common resource-processing requirements. At any given time, a user session or job class can belong to a single resource consumer group. The resource consumer group that the job class associates with determines the resources that are allocated to the job class.
- A **window** is represented by an interval of time with a well-defined beginning and end, and is used to activate different resource plans at different times.

The slide focuses on the job component as the primary entity. However, a program, a schedule, a window, and a job class are components that can be created as individual entities that can be associated with a job to be executed by the Scheduler. When a job is created, it may contain all the information needed inline—that is, in the call that creates the job. Alternatively, creating a job may reference a program or schedule component that was previously defined. Examples of this are discussed on the next few pages.

For more information about the Scheduler, see the Online Course titled *Oracle Database 11g: Configure and Manage Jobs with the Scheduler*.

Creating a Job

- A job can be created in several ways by using a combination of inline parameters, named `Programs`, and named `Schedules`.
- You can create a job with the `CREATE_JOB` procedure by:
  – Using inline information with the “what” and the schedule specified as parameters
  – Using a named (saved) program and specifying the schedule inline
  – Specifying what should be done inline and using a named Schedule
  – Using named Program and Schedule components

ORACLE

Creating a Job

The component that causes something to be executed at a specified time is called a **job**. Use the `DBMS_SCHEDULER.CREATE_JOB` procedure of the `DBMS_SCHEDULER` package to create a job, which is in a disabled state by default. A job becomes active and scheduled when it is explicitly enabled. To create a job, you:

- Provide a name in the format `[schema.]name`
- Need the `CREATE JOB` privilege

Note: A user with the `CREATE ANY JOB` privilege can create a job in any schema except the `SYS` schema. Associating a job with a particular class requires the `EXECUTE` privilege for that class.

In simple terms, a job can be created by specifying all the job details—the program to be executed (what) and its schedule (when)—in the arguments of the `CREATE_JOB` procedure. Alternatively, you can use predefined Program and Schedule components. If you have a named Program and Schedule, then these can be specified or combined with inline arguments for maximum flexibility in the way a job is created.

A simple logical check is performed on the schedule information (that is, checking the date parameters when a job is created). The database checks whether the end date is after the start date. If the start date refers to a time in the past, then the start date is changed to the current date.

Creating a Job with Inline Parameters

Specify the type of code, code, start time, and frequency of the job to be run in the arguments of the CREATE_JOB procedure.

```
-- Schedule a PL/SQL block every hour:

BEGIN
  DBMS_SCHEDULER.CREATE_JOB(
   job_name => 'JOB_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   start_date => SYSTIMESTAMP,
   repeat_interval=>'FREQUENCY=HOURLY;INTERVAL=1',
   enabled => TRUE);
END;
/
```

ORACLE

Creating a Job with Inline Parameters

You can create a job to run a PL/SQL block, stored procedure, or external program by using the DBMS_SCHEDULER.CREATE_JOB procedure. The CREATE_JOB procedure can be used directly without requiring you to create Program or Schedule components.

The example in the slide shows how you can specify all the job details inline. The parameters of the CREATE_JOB procedure define "what" is to be executed, the schedule, and other job attributes. The following parameters define what is to be executed:

- The job_type parameter can be one of the following three values:
  - PLSQL_BLOCK for any PL/SQL block or SQL statement. This type of job cannot accept arguments.
  - STORED_PROCEDURE for any stored stand-alone or packaged procedure. The procedures can accept arguments that are supplied with the job.
  - EXECUTABLE for an executable command-line operating system application
- The schedule is specified by using the following parameters:
  - The start_date accepts a time stamp, and the repeat_interval is string-specified as a calendar or PL/SQL expression. An end_date can be specified.

Note: String expressions that are specified for repeat_interval are discussed later. The example specifies that the job should run every hour.

Creating a Job Using a Program

- Use `CREATE_PROGRAM` to create a program:

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM(
   program_name => 'PROG_NAME',
   program_type => 'PLSQL_BLOCK',
   program_action => 'BEGIN ...; END;');
END;
```

- Use overloaded `CREATE_JOB` procedure with its `program_name` parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   enabled => TRUE);
END;
```

ORACLE

Creating a Job Using a Program

The `DBMS_SCHEDULER.CREATE_PROGRAM` procedure defines a program that must be assigned a unique name. Creating the program separately for a job enables you to:

- Define the action once and then reuse this action within multiple jobs
- Change the schedule for a job without having to re-create the PL/SQL block
- Change the program executed without changing all the jobs

The program action string specifies a procedure, executable name, or PL/SQL block depending on the value of the `program_type` parameter, which can be:

- `PLSQL_BLOCK` to execute an anonymous block or SQL statement
- `STORED_PROCEDURE` to execute a stored procedure, such as PL/SQL, Java, or C
- `EXECUTABLE` to execute operating system command-line programs

The example shown in the slide demonstrates calling an anonymous PL/SQL block. You can also call an external procedure within a program, as in the following example:

```
DBMS_SCHEDULER.CREATE_PROGRAM(program_name => 'GET_DATE',
    program_action => '/usr/local/bin/date',
    program_type => 'EXECUTABLE');
```

To create a job with a program, specify the program name in the `program_name` argument in the call to the `DBMS_SCHEDULER.CREATE_JOB` procedure, as shown in the slide.

Creating a Job for a Program with Arguments

- Create a program:

```
DBMS_SCHEDULER.CREATE_PROGRAM(
  program_name => 'PROG_NAME',
  program_type => 'STORED_PROCEDURE',
  program_action => 'EMP_REPORT');
```

- Define an argument:

```
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'PROG_NAME',
  argument_name => 'DEPT_ID',
  argument_position=> 1, argument_type=> 'NUMBER',
  default_value => '50');
```

- Create a job specifying the number of arguments:

```
DBMS_SCHEDULER.CREATE_JOB('JOB_NAME', program_name
  => 'PROG_NAME', start_date => SYSTIMESTAMP,
  repeat_interval => 'FREQ=DAILY',
  number_of_arguments => 1, enabled => TRUE);
```

ORACLE

Creating a Job for a Program with Arguments

Programs, such as PL/SQL or external procedures, may require input arguments. Using the `DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT` procedure, you can define an argument for an existing program. The `DEFINE_PROGRAM_ARGUMENT` procedure parameters include the following:

- `program_name` specifies an existing program that is to be altered.
- `argument_name` specifies a unique argument name for the program.
- `argument_position` specifies the position in which the argument is passed when the program is called.
- `argument_type` specifies the data type of the argument value that is passed to the called program.
- `default_value` specifies a default value that is supplied to the program if the job that schedules the program does not provide a value.

The slide shows how to create a job executing a program with one argument. The program argument default value is 50. To change the program argument value for a job, use:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE(
  job_name => 'JOB_NAME',
  argument_name => 'DEPT_ID', argument_value => '80');
```

Creating a Job Using a Schedule

- Use `CREATE_SCHEDULE` to create a schedule:

```
BEGIN
  DBMS_SCHEDULER.CREATE_SCHEDULE('SCHED_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   end_date => SYSTIMESTAMP +15);
END;
```

- Use `CREATE_JOB` by referencing the schedule in the `schedule_name` parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   schedule_name => 'SCHED_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   enabled => TRUE);
END;
```

ORACLE

Creating a Job Using a Schedule

You can create a common schedule that can be applied to different jobs without having to specify the schedule details each time. The following are the benefits of creating a schedule:

- It is reusable and can be assigned to different jobs.
- Changing the schedule affects all jobs using the schedule. The job schedules are changed once, not multiple times.

A schedule is precise to only the nearest second. Although the `TIMESTAMP` data type is more accurate, the Scheduler rounds off anything with a higher precision to the nearest second.

The start and end times for a schedule are specified by using the `TIMESTAMP` data type. The `end_date` for a saved schedule is the date after which the schedule is no longer valid. The schedule in the example is valid for 15 days after using it with a specified job.

The `repeat_interval` for a saved schedule must be created by using a calendaring expression. A `NULL` value for `repeat_interval` specifies that the job runs only once.

Note: You cannot use PL/SQL expressions to express the repeat interval for a saved schedule.

Setting the Repeat Interval for a Job

- Using a calendaring expression:

```
repeat_interval=> 'FREQ=HOURLY; INTERVAL=4'
repeat_interval=> 'FREQ=DAILY'
repeat_interval=> 'FREQ=MINUTELY;INTERVAL=15'
repeat_interval=> 'FREQ=YEARLY;
                   BYMONTH=MAR,JUN,SEP,DEC;
                   BYMONTHDAY=15'
```

- Using a PL/SQL expression:

```
repeat_interval=> 'SYSDATE + 36/24'
repeat_interval=> 'SYSDATE + 1'
repeat_interval=> 'SYSDATE + 15/(24*60)'
```

ORACLE

Setting the Repeat Interval for a Job

When scheduling repeat intervals for a job, you can specify either a PL/SQL expression (if it is within a job argument) or a calendaring expression.

The examples in the slide include the following:

- `FREQ=HOURLY;INTERVAL=4` indicates a repeat interval of every four hours.
- `FREQ=DAILY` indicates a repeat interval of every day, at the same time as the start date of the schedule.
- `FREQ=MINUTELY;INTERVAL=15` indicates a repeat interval of every 15 minutes.
- `FREQ=YEARLY;BYMONTH=MAR,JUN,SEP,DEC;BYMONTHDAY=15` indicates a repeat interval of every year on March 15, June 15, September 15, and December 15.

With a calendaring expression, the next start time for a job is calculated using the repeat interval and the start date of the job.

Note: If no repeat interval is specified (that is, if a `NULL` value is provided in the argument), the job runs only once on the specified start date.

Creating a Job Using a Named Program and Schedule

- Create a named program called PROG_NAME by using the CREATE_PROGRAM procedure.
- Create a named schedule called SCHED_NAME by using the CREATE_SCHEDULE procedure.
- Create a job referencing the named program and schedule:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   schedule_name => 'SCHED_NAME',
   enabled => TRUE);
END;
/
```


Creating a Job Using a Named Program and Schedule

The example in the slide shows the final form for using the DBMS_SCHEDULER.CREATE_JOB procedure. In this example, the named program (PROG_NAME) and schedule (SCHED_NAME) are specified in their respective parameters in the call to the DBMS_SCHEDULER.CREATE_JOB procedure.

With this example, you can see how easy it is to create jobs by using a predefined program and schedule.

Some jobs and schedules can be too complex to cover in this course. For example, you can create windows for recurring time plans and associate a resource plan with a window. A resource plan defines attributes about the resources required during the period defined by execution window.

For more information, refer to the online course titled *Oracle Database 11g: Configure and Manage Jobs with the Scheduler*.

Managing Jobs

- Run a job:

```
DBMS_SCHEDULER.RUN_JOB('SCHEMA.JOB_NAME');
```

- Stop a job:

```
DBMS_SCHEDULER.STOP_JOB('SCHEMA.JOB_NAME');
```

- Drop a job even if it is currently running:

```
DBMS_SCHEDULER.DROP_JOB('JOB_NAME', TRUE);
```

ORACLE

Managing Jobs

After a job has been created, you can:

- Run the job by calling the RUN_JOB procedure specifying the name of the job. The job is immediately executed in your current session.
- Stop the job by using the STOP_JOB procedure. If the job is running currently, it is stopped immediately. The STOP_JOB procedure has two arguments:
  - **job_name:** Is the name of the job to be stopped
  - **force:** Attempts to gracefully terminate a job. If this fails and force is set to TRUE, then the job slave is terminated. (Default value is FALSE.) To use force, you must have the MANAGE SCHEDULER system privilege.
- Drop the job with the DROP_JOB procedure. This procedure has two arguments:
  - **job_name:** Is the name of the job to be dropped
  - **force:** Indicates whether the job should be stopped and dropped if it is currently running (Default value is FALSE.)

  If the DROP_JOB procedure is called and the job specified is currently running, then the command fails unless the force option is set to TRUE. If the force option is set to TRUE, then any instance of the job that is running is stopped and the job is dropped.

Note: To run, stop, or drop a job that belongs to another user, you need ALTER privileges on that job or the CREATE ANY JOB system privilege.

Data Dictionary Views

- [DBA | ALL | USER]_SCHEDULER_JOBS
- [DBA | ALL | USER]_SCHEDULER_RUNNING_JOBS
- [DBA | ALL]_SCHEDULER_JOB_CLASSES
- [DBA | ALL | USER]_SCHEDULER_JOB_LOG
- [DBA | ALL | USER]_SCHEDULER_JOB_RUN_DETAILS
- [DBA | ALL | USER]_SCHEDULER_PROGRAMS

Data Dictionary Views

The DBA_SCHEDULER_JOB_LOG view shows all completed job instances, both successful and failed.

To view the state of your jobs, use the following query:

```
SELECT job_name, program_name, job_type, state
FROM USER_SCHEDULER_JOBS;
```

To determine which instance a job is running on, use the following query:

```
SELECT owner, job_name, running_instance,
resource_consumer_group
FROM DBA_SCHEDULER_RUNNING_JOBS;
```

To determine information about how a job ran, use the following query:

```
SELECT job_name, instance_id, req_start_date,
actual_start_date, status
FROM ALL_SCHEDULER_JOB_RUN_DETAILS;
```

To determine the status of your jobs, use the following query:

```
SELECT job_name, status, error#, run_duration, cpu_used
FROM USER_SCHEDULER_JOB_RUN_DETAILS;
```

Summary

In this lesson, you should have learned how to:

- Use the HTP package to generate a simple Web page
- Call the DBMS_SCHEDULER package to schedule PL/SQL code for execution

Summary

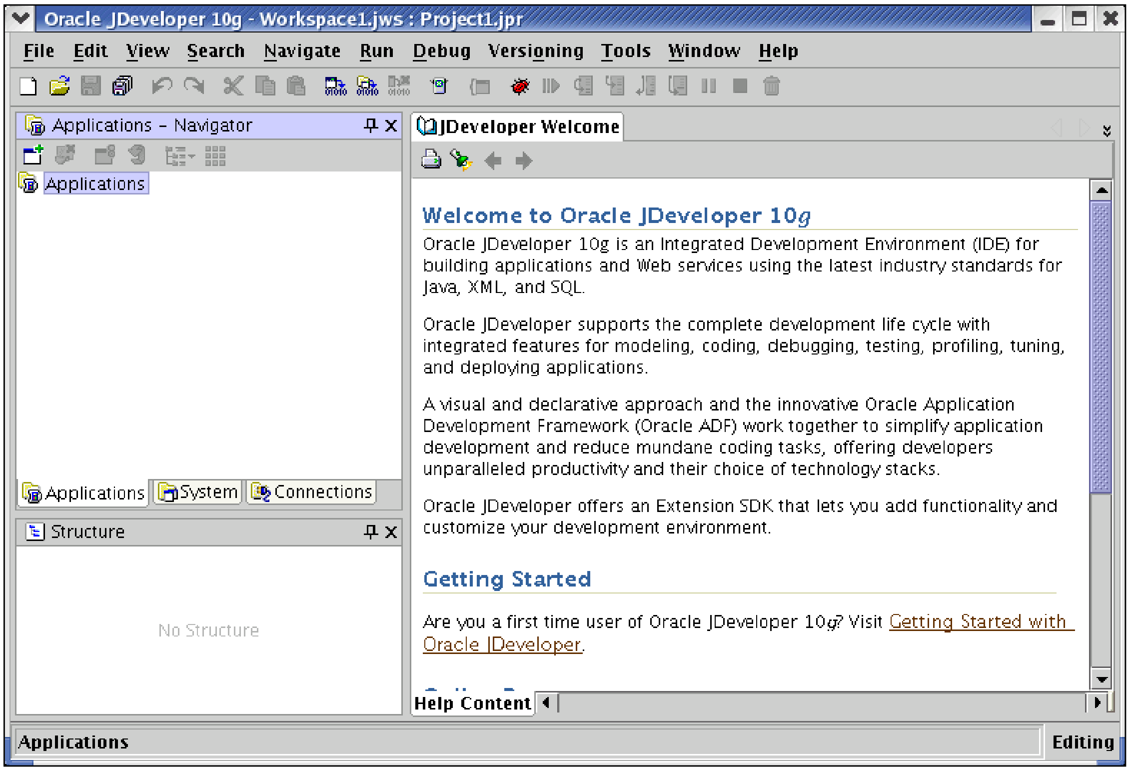
This lesson covers a small subset of packages provided with the Oracle database. You have extensively used DBMS_OUTPUT for debugging purposes and displaying procedurally generated information on the screen in SQL*Plus.

In this lesson, you should have learned how to schedule PL/SQL and external code for execution with the DBMS_SCHEDULER package.

Note: For more information about all PL/SQL packages and types, refer to *PL/SQL Packages and Types Reference*.

Review of JDeveloper

JDeveloper

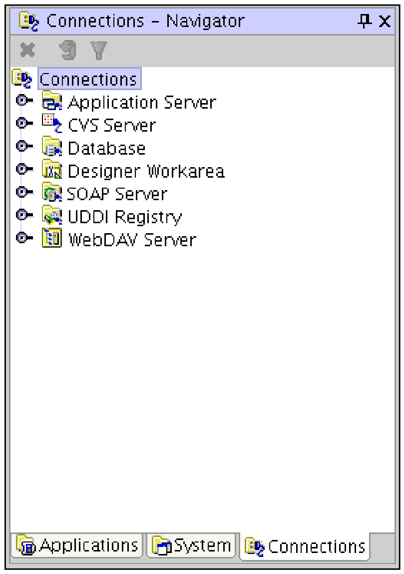


JDeveloper

Oracle JDeveloper 11*g* is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC) from modeling to deploying. It has the features to use the latest industry standards for Java, Extensible Markup Language (XML), and SQL while developing an application.

Oracle JDeveloper 11*g* initiates a new approach to J2EE development with the features that enable visual and declarative development. This innovative approach makes J2EE development simple and efficient.

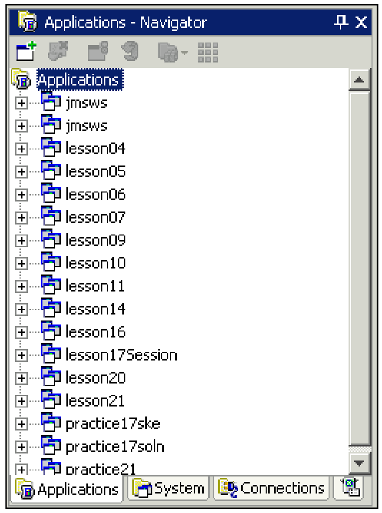
Connection Navigator



Connection Navigator

Using Oracle JDeveloper 11*g*, you can store the information necessary to connect to a database in an object called “connection.” A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.

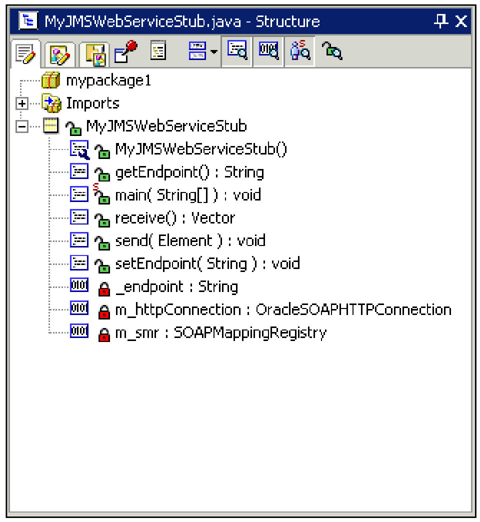
Application Navigator



Application Navigator

The Application Navigator gives you a logical view of your application and the data it contains. The Application Navigator provides an infrastructure that the different extensions can plug into and use to organize their data and menus in a consistent, abstract manner. While the Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, UML (Unified Modeling Language) diagrams, Enterprise JavaBeans (EJB), or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Structure Window



Structure Window

The Structure window offers a structural view of the data in the document currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in a variety of ways. The structures available for display are based upon document type. For a Java file, you can view code structure, user interface (UI) structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, always tracking the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, select a different structure tab.

Editor Window


SHOW_CUST_CALL

```
PROCEDURE show_cust_call (
custid IN NUMBER default 101) AS
 BEGIN NULL;
htp.prn('
');
htp.prn('
');
htp.prn('
<HTML>
<BODY>
<form method="POST" action="show_cust">
<p>Enter the Customer ID:
<input type="text" name="custid">
<input type="submit" value="Submit">
</form>
</BODY>
</HTML>
');
 END;
```

ORACLE

Editor Window

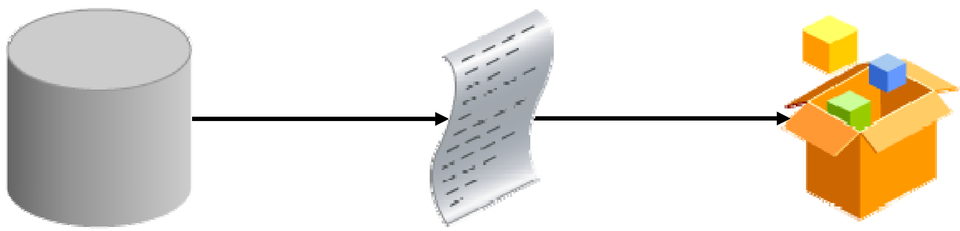
You can view all your project files in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

The tabs at the top of the editor window are the document tabs. Selecting a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:



Deploying Java Stored Procedures

Create a deployment profile for Java stored procedures, then deploy the classes and, optionally, any public static methods in JDeveloper using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call–specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

FormatCreditCardNo.java | CCFORMAT

```java
public class FormatCreditCardNo
{
  public static final void formatCard(String[] cardno)
  {
  int count=0, space=0;
  String oldcc=cardno[0];
 // System.out.println("Printing the card no initially "+oldcc);
  String[] newcc= {""};
  while (count<16)
  {
  newcc[0]+= oldcc.charAt(count);
  space++;
  if (space ==4)
  {  newcc[0]+=" "; space=0;  }
  count++;
  }
  cardno[0]=newcc [0];
  }
}
```

FormatCreditCardNo.java | CCFORMAT

```sql
PROCEDURE ccformat (x IN OUT varchar2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```


Publishing Java to PL/SQL

The slide shows the Java code and how to publish the Java code in a PL/SQL procedure.

Creating Program Units

TEST_JDEV

```
FUNCTION "TEST_JDEV" RETURN VARCHAR2
AS
BEGIN
  RETURN('');
END;
```

Skeleton of the function

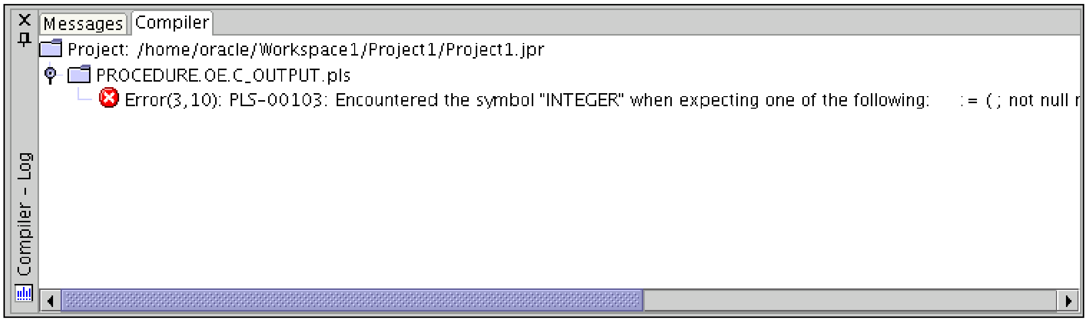
Creating Program Units

To create a PL/SQL program unit:

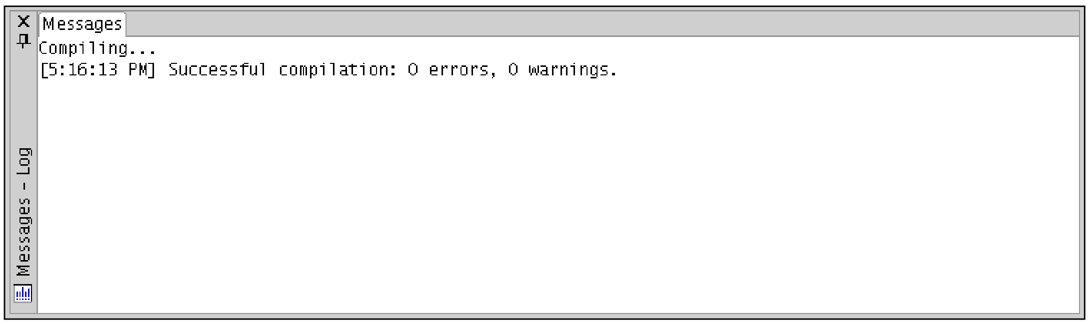
1. Select View > Connection Navigator.
2. Expand Database and select a database connection.
3. In the connection, expand a schema.
4. Right-click a folder corresponding to the object type (Procedures, Packages, and Functions).
5. Choose New PL/SQL object_type. The Create PL/SQL dialog box appears for the function, package, or procedure.
6. Enter a valid name for the function, package, or procedure, and click OK.

A skeleton definition will be created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling



Compilation with errors

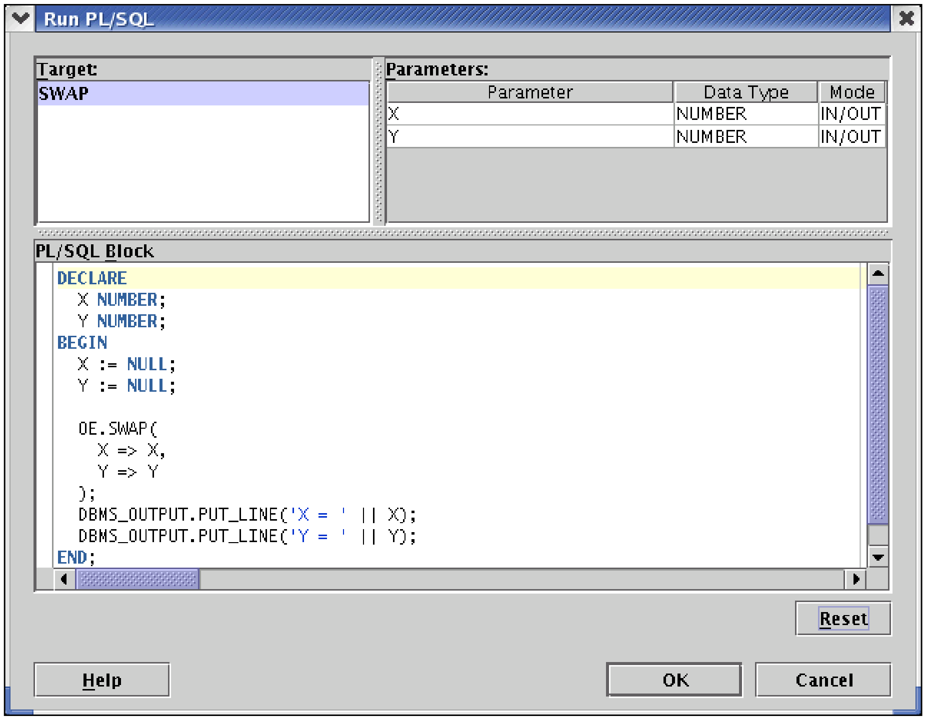


Compilation without errors

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator and then select Compile. Alternatively, you can also press [CTRL] + [SHIFT] + [F9] to compile.

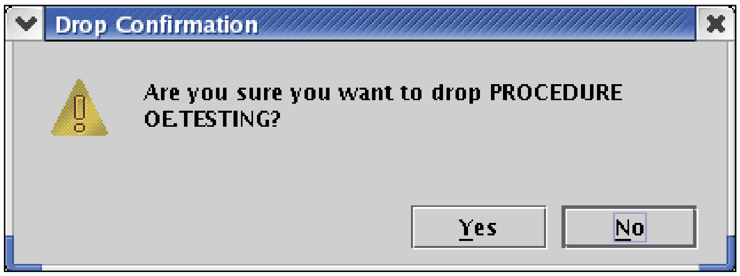
Running a Program Unit



Running a Program Unit

To execute the program unit, right-click the object and select Run. The Run PL/SQL dialog box appears. You may need to change the `NULL` values with reasonable values that are passed into the program unit. After you change the values, click OK. The output will be displayed in the Message-Log window.

Dropping a Program Unit



Dropping a Program Unit

To drop a program unit, right-click the object and select Drop. The Drop Confirmation dialog box appears; click Yes. The object will be dropped from the database.

Debugging PL/SQL Programs

- JDeveloper support two types of debugging:
  - Local
  - Remote
- You need the following privileges to perform PL/SQL debugging:
  - `DEBUG ANY PROCEDURE`
  - `DEBUG CONNECT SESSION`

Debugging PL/SQL Programs

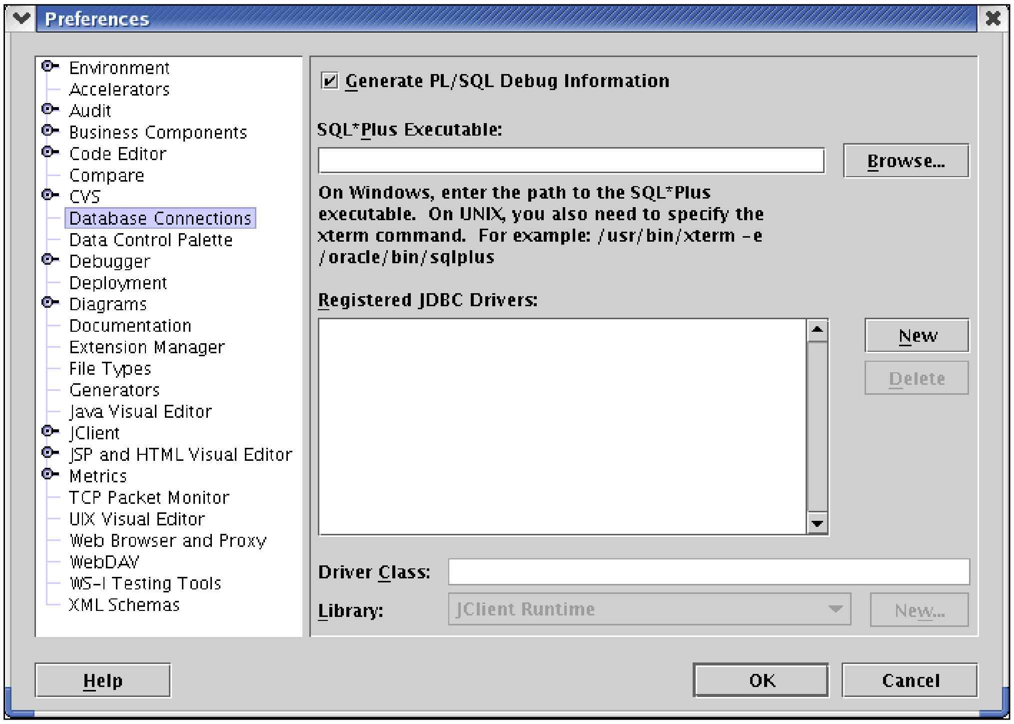
JDeveloper offers both local and remote debugging. A local debugging session is started by setting breakpoints in source files, and then starting the debugger. Remote debugging requires two JDeveloper processes: a `debugger` and a `debuggee`, which may reside on a different platform.

To debug a PL/SQL program, it must be compiled in `INTERPRETED` mode. You cannot debug a PL/SQL program that is compiled in `NATIVE` mode. This mode is set in the database's `init.ora` file.

PL/SQL programs must be compiled with the `DEBUG` option enabled. This option can be enabled using various ways. Using SQL*Plus, execute `ALTER SESSION SET PLSQL_DEBUG = true` to enable the `DEBUG` option. Then you can create or recompile the PL/SQL program you want to debug. Another way of enabling the `DEBUG` option is by using the following command in SQL*Plus:

```
ALTER <procedure, function, package> <name> COMPILE DEBUG;
```

Debugging PL/SQL Programs



Debugging PL/SQL Programs (continued)

Before you start with debugging, make sure that the Generate PL/SQL Debug Information check box is selected. You can access the dialog box by using Tools > Preferences > Database Connections.

Instead of manually testing PL/SQL functions and procedures as you may be accustomed to doing from within SQL*Plus or by running a dummy procedure in the database, JDeveloper enables you to test these objects in an automatic way. With this release of JDeveloper, you can run and debug PL/SQL program units. For example, you can specify parameters being passed or return values from a function giving you more control over what is run and providing you output details about what was tested.

Note: The procedures or functions in the Oracle database can be either stand-alone or within a package.

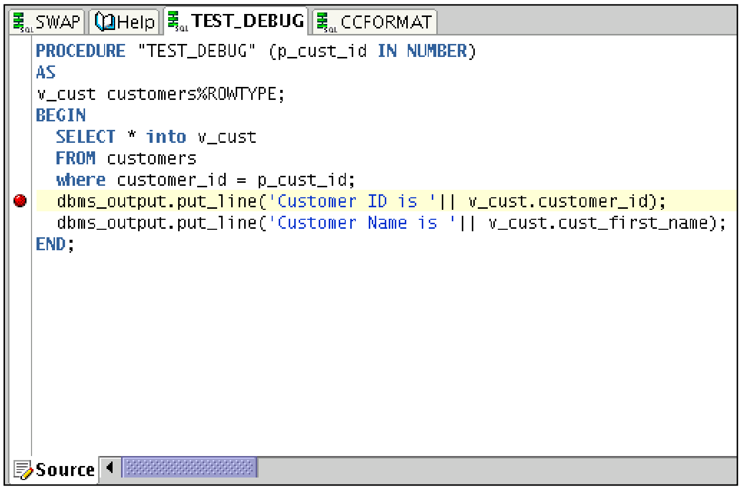
Debugging PL/SQL Programs (continued)

To run or debug functions, procedures, or packages, perform the following steps:

1. Create a database connection by using the Database Wizard.
2. In the Navigator, expand the Database node to display the specific database username and schema name.
3. Expand the Schema node.
4. Expand the appropriate node depending on what you are debugging: Procedure, Function, or Package body.
5. (Optional for debugging only) Select the function, procedure, or package that you want to debug and double-click to open it in the Code Editor.
6. (Optional for debugging only) Set a breakpoint in your PL/SQL code by clicking to the left of the margin.
   Note: The breakpoint must be set on an executable line of code. If the debugger does not stop, the breakpoint may have not been set on an executable line of code (ensure that the breakpoint was verified). Also, verify that the debugging PL/SQL prerequisites were met. In particular, make sure that the PL/SQL program is compiled in `INTERPRETED` mode.
7. Make sure that either the Code Editor or the procedure in the Navigator is currently selected.
8. Click the Debug toolbar button; or, if you want to run without debugging, click the Run toolbar button.
9. The Run PL/SQL dialog box is displayed.
   - Select a target that is the name of the procedure or function that you want to debug. Note that the content in the Parameters and PL/SQL Block boxes change dynamically when the target changes.
     Note: You will have a choice of target only if you choose to run or debug a package that contains more than one program unit.
   - The Parameters box lists the target's arguments (if applicable).
   - The PL/SQL Block box displays code that was custom-generated by JDeveloper for the selected target. Depending on what the function or procedure does, you may need to replace the NULL values with reasonable values so that these are passed into the procedure, function, or package. In some cases, you may need to write additional code to initialize values to be passed as arguments. In this case, you can edit the PL/SQL block text as necessary.
10. Click OK to execute or debug the target.
11. Analyze the output information displayed in the Log window.

In the case of functions, the return value will be displayed. `DBMS_OUTPUT` messages will also be displayed.

Setting Breakpoints

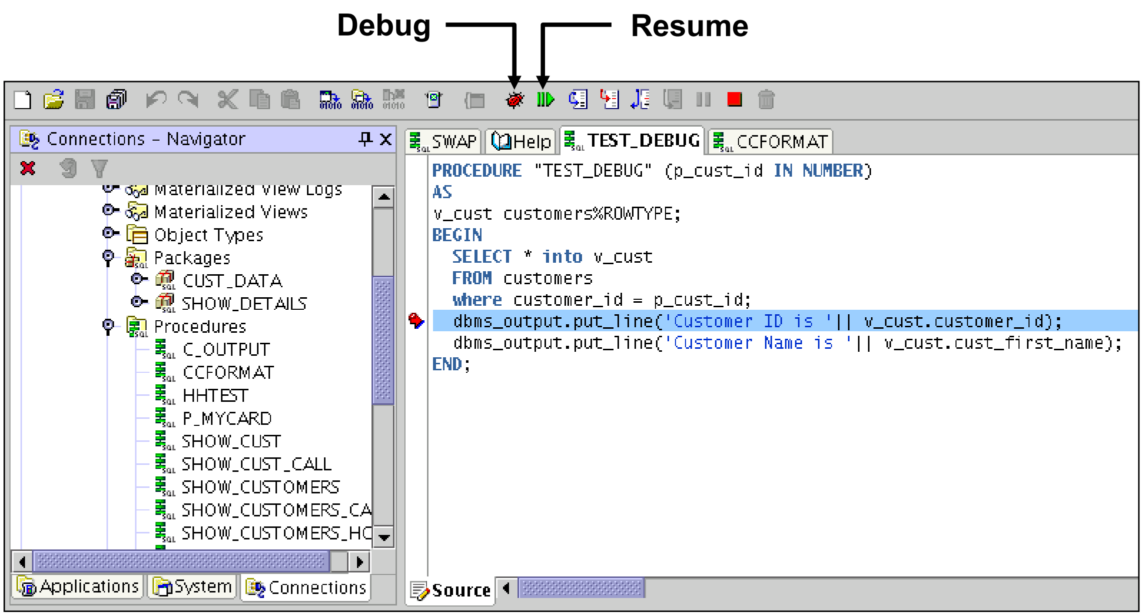


Setting Breakpoints

Breakpoints help you examine the values of the variables in your program. A breakpoint is a trigger in a program that, when reached, pauses program execution allowing you to examine the values of some or all of the program variables. By setting breakpoints in potential problem areas of your source code, you can run your program until its execution reaches a location you want to debug. When your program execution encounters a breakpoint, the program pauses, and the debugger displays the line containing the breakpoint in the Code Editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time while you are debugging.

To set a breakpoint in the Code Editor, click the left margin next to a line of executable code. Breakpoints set on comment lines, blank lines, declaration, and any other nonexecutable lines of code are not verified by the debugger and are treated as invalid.

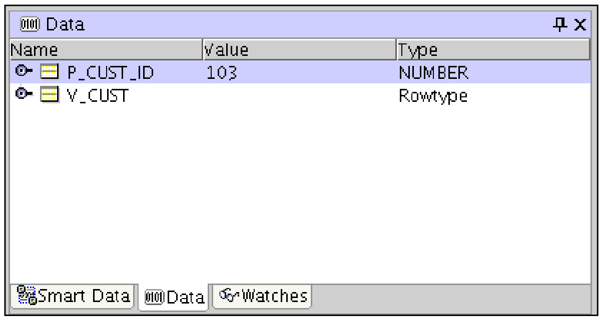
Stepping Through Code



Stepping Through Code

After setting the breakpoint, start the debugger by clicking the Debug icon. The debugger will pause the program execution at the point where the breakpoint is set. At this point, you can check the values of the variables. You can continue with the program execution by clicking the Resume icon. The debugger will then move on to the next breakpoint. After executing all the breakpoints, the debugger will stop the execution of the program and display the results in the Debugging – Log area.

Examining and Modifying Variables



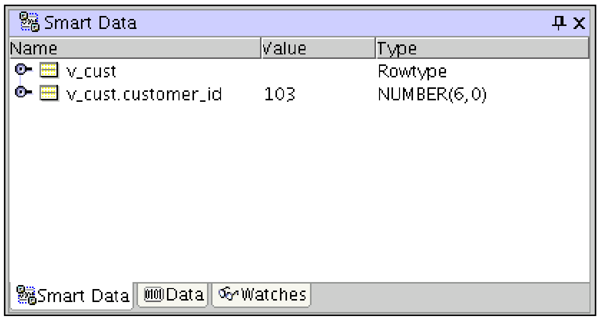
Data window

Examining and Modifying Variables

When the debugger is ON, you can examine and modify the value of the variables using the Data, Smart Data, and Watches windows. You can modify program data values during a debugging session as a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile the program to make the fix permanent.

You use the Data window to display information about variables in your program. The Data window displays the arguments, local variables, and static fields for the current context, which is controlled by the selection in the Stack window. If you move to a new context, the Data window is updated to show the data for the new context. If the current program was compiled without debug information, you will not be able to see the local variables.

Examining and Modifying Variables



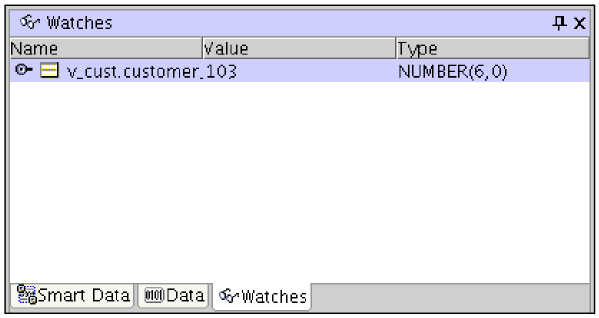
Smart Data window

ORACLE

Examining and Modifying Variables (continued)

Unlike the Data window that displays all the variables in your program, the Smart Data window displays only the data that is relevant to the source code that you are stepping through.

Examining and Modifying Variables

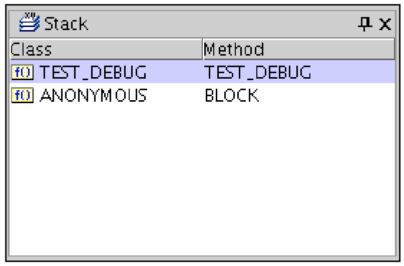


Watches window

Examining and Modifying Variables (continued)

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watch window displays the current value of the expression. As your program runs, the value of the watch changes as your program updates the values of the variables in the watch expression.

Examining and Modifying Variables

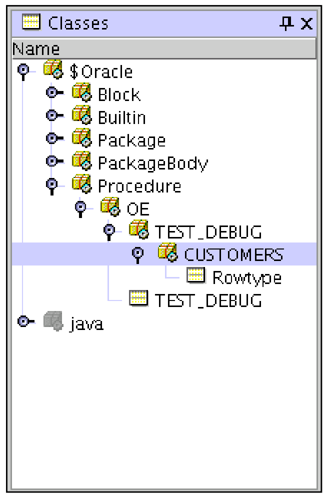


Stack window

Examining and Modifying Variables (continued)

You can activate the Stack window by using View > Debugger > Stack. It displays the call stack for the current thread. When you select a line in the Stack window, the Data window, Watch window, and all other windows are updated to show data for the selected class.

Examining and Modifying Variables



Classes window

Examining and Modifying Variables (continued)

The Classes window displays all the classes that are currently being loaded to execute the program. If used with Oracle Java Virtual Machine (OJVM), it also shows the number of instances of a class and the memory used by those instances.

Index

D

E

E

E

F

F

I

P

P

R

S

S

T

U

V

W