3

Creating Functions and Debugging Subprograms

Objectives

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

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Understand the basic functionality of the SQL Developer debugger

Lesson Agenda

- Working with functions:
  - Differentiating between a procedure and a function
  - Describing the uses of functions
  - Creating, invoking, and removing stored functions
- Introducing the SQL Developer debugger

Overview of Stored Functions

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value for another subprogram
- Can be grouped into PL/SQL packages

Creating Functions

The PL/SQL block must have at least one `RETURN` statement.

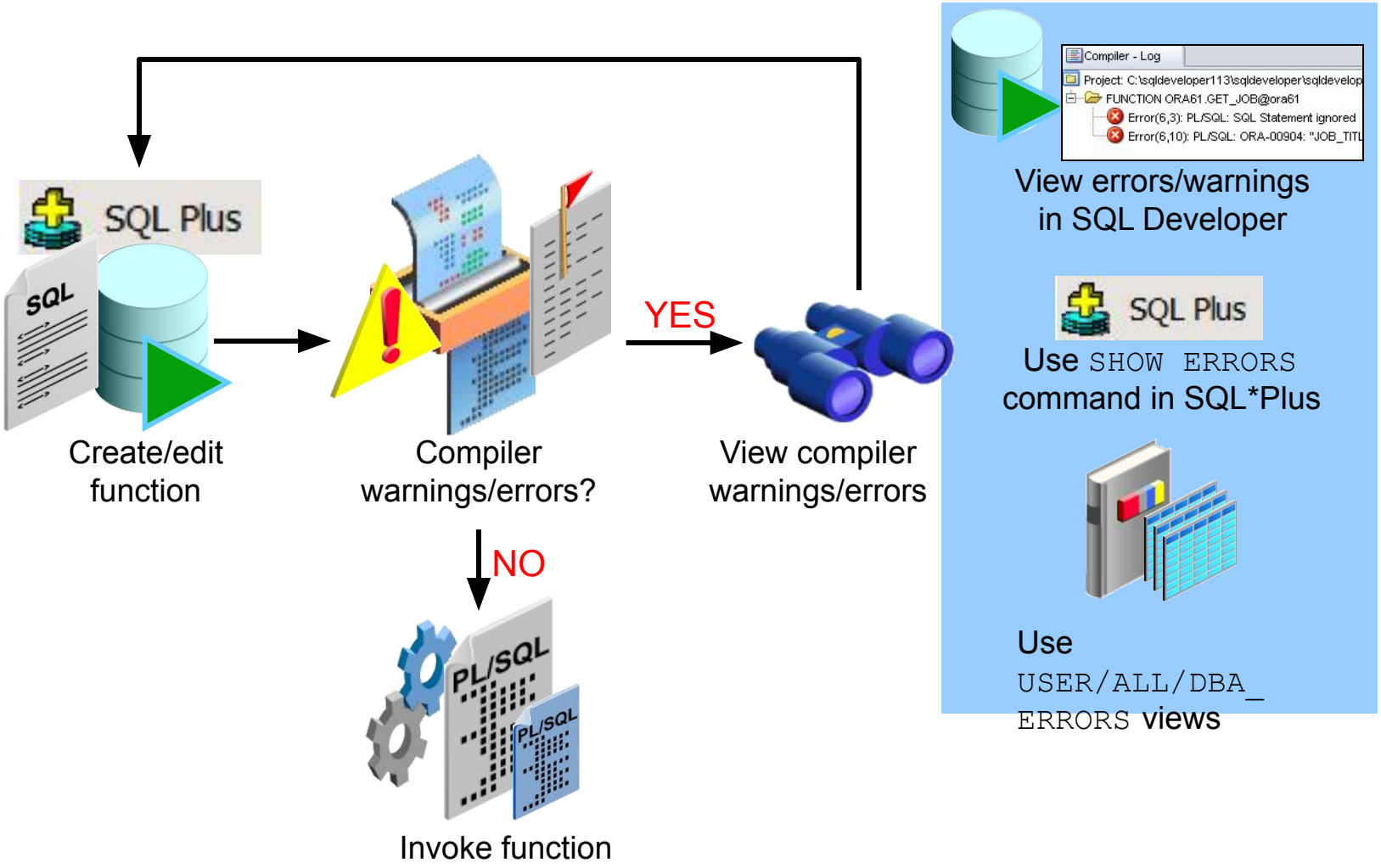
```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, . . .)]
RETURN datatype IS|AS
 [local_variable_declarations;
  . . .]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

The Difference Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain `RETURN` clause in the header	Must contain a `RETURN` clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a `RETURN` statement without a value	Must contain at least one `RETURN` statement

Creating and Running Functions: Overview



Creating and Invoking a Stored Function Using the `CREATE FUNCTION` Statement: Example

```
CREATE OR REPLACE FUNCTION get_sal
 (p_id employees.employee_id%TYPE) RETURN NUMBER IS
  v_sal employees.salary%TYPE := 0;
  BEGIN
    SELECT salary
    INTO   v_sal
    FROM   employees
    WHERE  employee_id = p_id;
    RETURN v_sal;
  END get_sal;
/
```

```
FUNCTION GET_SAL compiled
```

```
-- Invoke the function as an expression or as
-- a parameter value.

EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed
24000
```

Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables

VARIABLE b_salary NUMBER
EXECUTE :b_salary := get_sal(100)
```

```
anonymous block completed
B_SALARY
-----
24000
```

```
-- As a PL/SQL expression, get the results using a local
-- variable
SET SERVEROUTPUT ON
DECLARE
  sal employees.salary%type;
BEGIN
  sal := get_sal(100);
  DBMS_OUTPUT.PUT_LINE('The salary is: '|| sal);
END;
/
```

```
anonymous block completed
The salary is: 24000
```

Using Different Methods for Executing Functions

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

EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed
24000
```

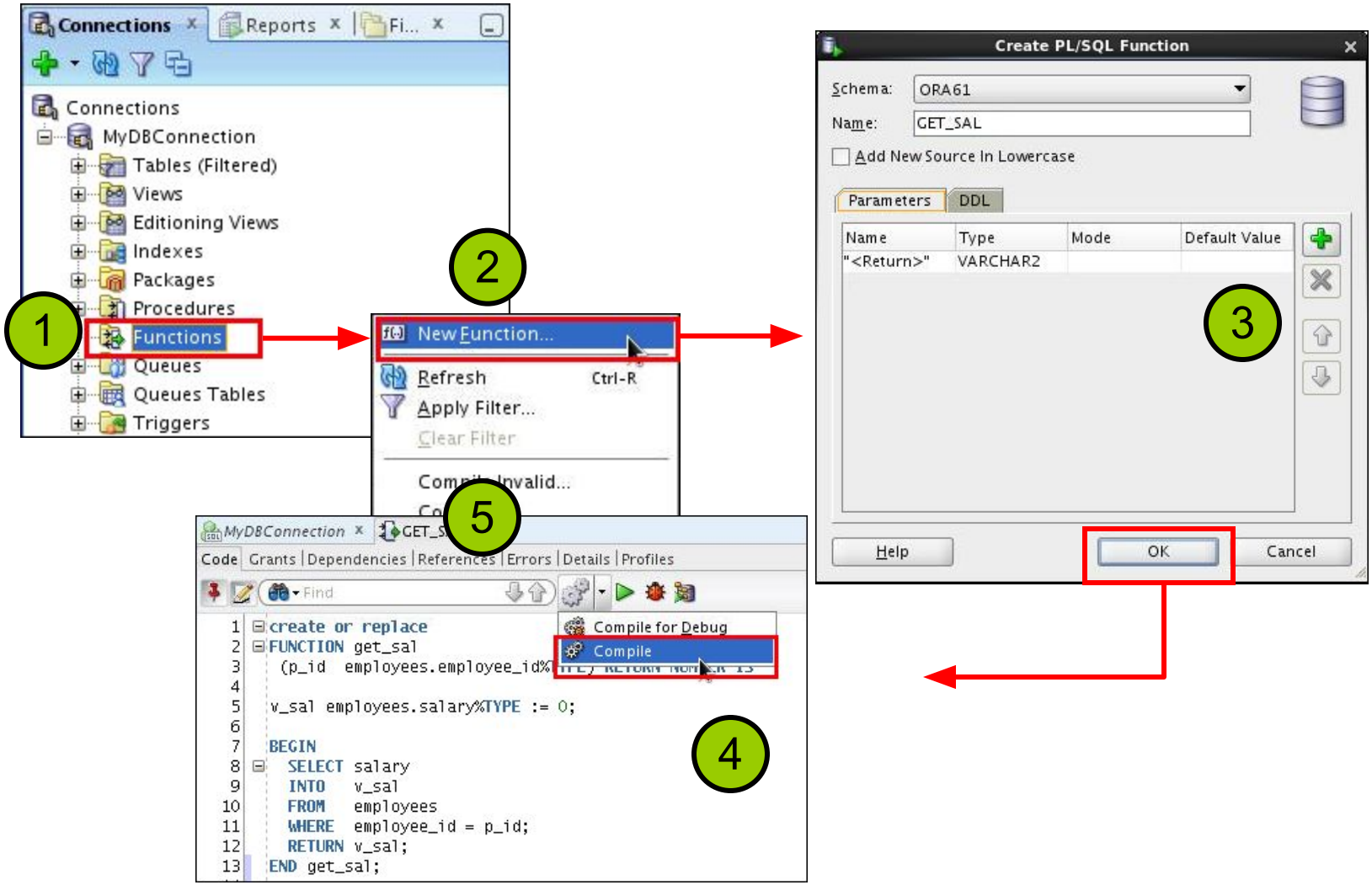
```
-- Use in a SQL statement (subject to restrictions)

SELECT job_id, get_sal(employee_id)
FROM employees;
```

```
JOB_ID     GET_SAL(EMPLOYEE_ID)
---------- --------------------
AC_ACCOUNT                 8300
AC_MGR                    12008
AD_ASST                    4400
AD_PRES                   24000
 . . .
ST_MAN                     6500
ST_MAN                     5800

 107 rows selected
```

Creating and Compiling Functions Using SQL Developer



Executing Functions Using SQL Developer

Connections | Reports | Fi...

Connections
- MyDBConnection
  - Tables (Filtered)
  - Views
  - Editioning Views
  - Indexes
  - Packages
  - Procedures
  - Functions
    - GET_SAL
  - Queues

1

Edit...
Export...
Compare With
Debu... Ctrl+Shift-F10
Comp... Ctrl-F8
Compile for Debug Ctrl+Shift-F8
Run Ctrl-F10
Profile...

2

Run PL/SQL

Target: GET_SAL

Parameters:

Parameter	Data Type	Mode
<Return Value>	NUMBER	OUT
P_ID	NUMBER	IN

PL/SQL Block

```
DECLARE
  P_ID NUMBER;
  v_Return NUMBER;
BEGIN
  P_ID := NULL;

  v_Return := GET_SAL(
    P_ID => 100
  );

DBMS_OUTPUT.PUT_LINE('v_Return = ' || v_Return);
END;
```

Save File... | From File... | Reset

Help | OK | Cancel

4

Running - Log

```
Connecting to the database MyDBConnection.
v_Return = 24000
Process exited.
Disconnecting from the database MyDBConnection.
```

Run PL/SQL

Target: GET_SAL

Parameters:

Parameter
<Return Value>
P_ID

PL/SQL Block

```
DECLARE
  P_ID NUMBER;
  v_Return NUMBER;
BEGIN
  P_ID

  v_Return := GET_SAL(
    P_ID => P_ID
  );
  DBMS_OUTPUT.PUT_LINE('v_Return = ' || v_Return);
END;
```

3

Replace the second P_ID with the actual value 100.

Advantages of User-Defined Functions in SQL Statements

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

Using a Function in a SQL Expression: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

```
FUNCTION TAX compiled
EMPLOYEE_ID LAST_NAME                    SALARY TAX(SALARY)
----------- ------------------------ -------- -----------
        108 Greenberg                     12008      960.64
        109 Faviet                         9000         720
        110 Chen                           8200         656
        111 Sciarra                        7700         616
        112 Urman                          7800         624
        113 Popp                           6900         552

 6 rows selected
```

Calling User-Defined Functions in SQL Statements

User-defined functions act like built-in single-row functions and can be used in:

- The `SELECT` list or clause of a query
- Conditional expressions of the `WHERE` and `HAVING` clauses
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

Restrictions When Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Be stored in the database
  - Accept only `IN` parameters with valid SQL data types and PL/SQL-specific data types
  - Return valid SQL data types and PL/SQL-specific data types
- When calling functions in SQL statements:
  - You must own the function or have the `EXECUTE` privilege
  - You may need to enable the `PARALLEL_ENABLE` keyword to allow a parallel execution of the SQL statement

Controlling Side Effects When Calling Functions from SQL Expressions

Functions called from:

- A `SELECT` statement cannot contain DML statements
- An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`
- SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)

Note: Calls to subprograms that break these restrictions are also not allowed in the function.

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_sal NUMBER)
   RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                  email, hire_date, job_id, salary)
  VALUES(1, 'Frost', 'jfrost@company.com',
         SYSDATE, 'SA_MAN', p_sal);
  RETURN (p_sal + 100);
END;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170;
```

```
FUNCTION DML_CALL_SQL compiled
Error starting at line 127 in command:
UPDATE employees
 SET salary = dml_call_sql(2000)
WHERE employee_id = 170
Error report:
SQL Error: ORA-04091: table ORA61.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA61.DML_CALL_SQL", line 4
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.
```

Named and Mixed Notation from SQL

- PL/SQL allows arguments in a subroutine call to be specified using positional, named, or mixed notation.
- Prior to Oracle Database 11*g*, only the positional notation is supported in calls from SQL.
- Starting in Oracle Database 11*g*, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements.
- For long parameter lists, with most having default values, you can omit values from the optional parameters.
- You can avoid duplicating the default value of the optional parameter at each call site.

Named and Mixed Notation from SQL: Example

```
CREATE OR REPLACE FUNCTION f(
  p_parameter_1 IN NUMBER DEFAULT 1,
  p_parameter_5 IN NUMBER DEFAULT 5)
RETURN NUMBER
IS
 v_var number;
BEGIN
  v_var := p_parameter_1 + (p_parameter_5 * 2);
  RETURN v_var;
END f;
/
```

```
SELECT f(p_parameter_5 => 10) FROM DUAL;
```

```
FUNCTION F compiled
F(P_PARAMETER_5=>10)
--------------------
                  21
```

Viewing Functions Using Data Dictionary Views

```
DESCRIBE USER_SOURCE
```

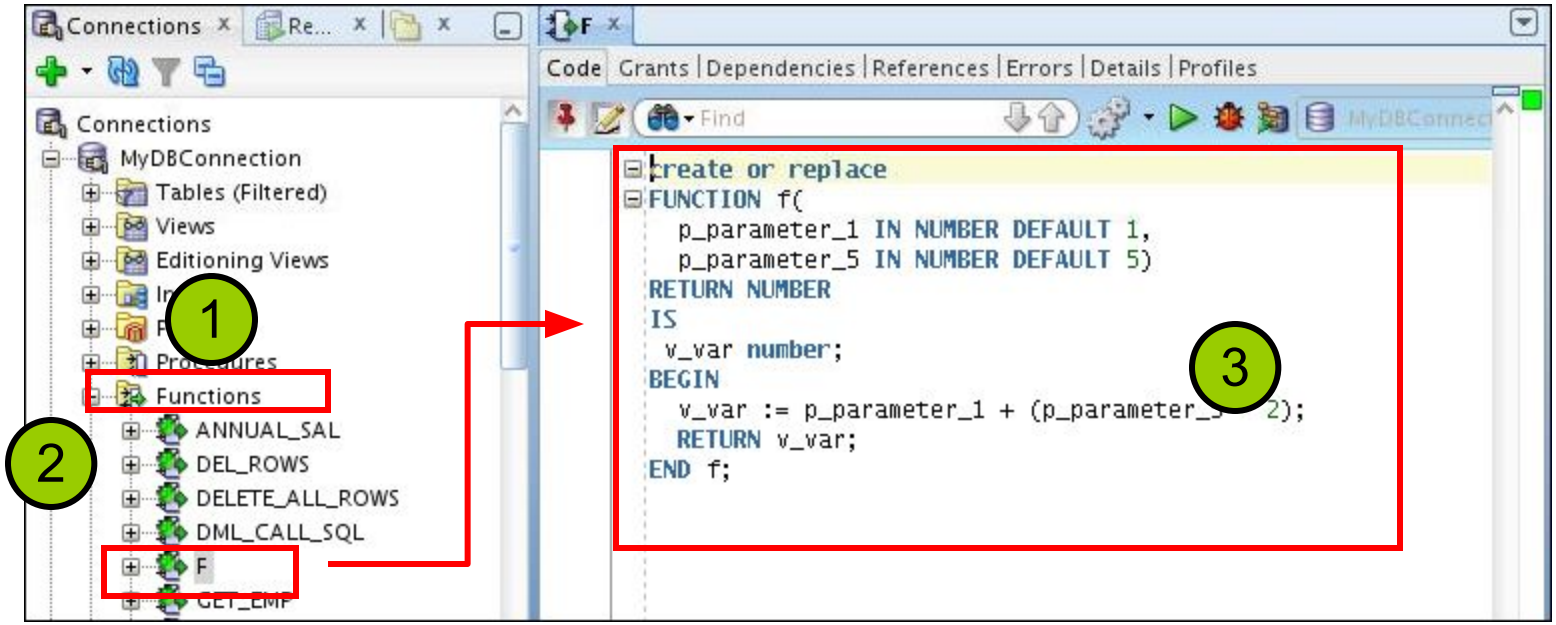
```
DESCRIBE user_source
Name Null Type
---- ---- --------------
NAME      VARCHAR2(128)
TYPE      VARCHAR2(12)
LINE      NUMBER
TEXT      VARCHAR2(4000)
```

```
SELECT   text
FROM     user_source
WHERE    type = 'FUNCTION'
ORDER    BY line;
```

	TEXT
1	FUNCTION dml_call_sql(p_sal NUMBER)
2	FUNCTION tax(p_value IN NUMBER)
3	FUNCTION query_call_sql(p_a NUMBER) RETURN NUMBER IS
4	FUNCTION get_sal
5	RETURN NUMBER IS
6	RETURN NUMBER IS
7	(p_id employees.employee_id%TYPE) RETURN NUMBER IS
8	v_s NUMBER;

. . .

Viewing Functions Information Using SQL Developer

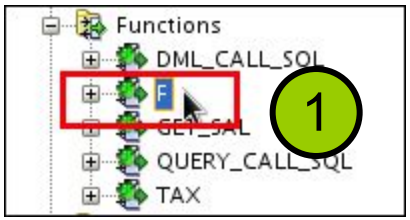


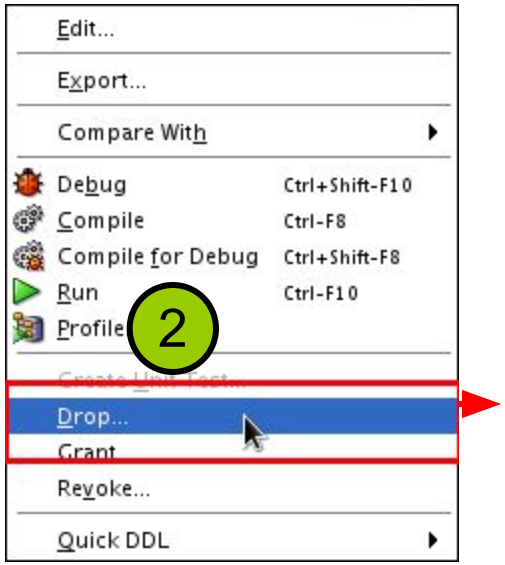
Removing Functions: Using the DROP SQL Statement or SQL Developer

- Using the DROP statement:

```
DROP FUNCTION f;
```

- Using SQL Developer:





Quiz

A PL/SQL stored function:

a. Can be invoked as part of an expression
b. Must contain a `RETURN` clause in the header
c. Must return a single value
d. Must contain at least one `RETURN` statement
e. Does not contain a `RETURN` clause in the header

Practice 3-1: Overview

This practice covers the following topics:

- Creating stored functions:
  - To query a database table and return specific values
  - To be used in a SQL statement
  - To insert a new row, with specified parameter values, into a database table
  - Using default parameter values
- Invoking a stored function from a SQL statement
- Invoking a stored function from a stored procedure

Lesson Agenda

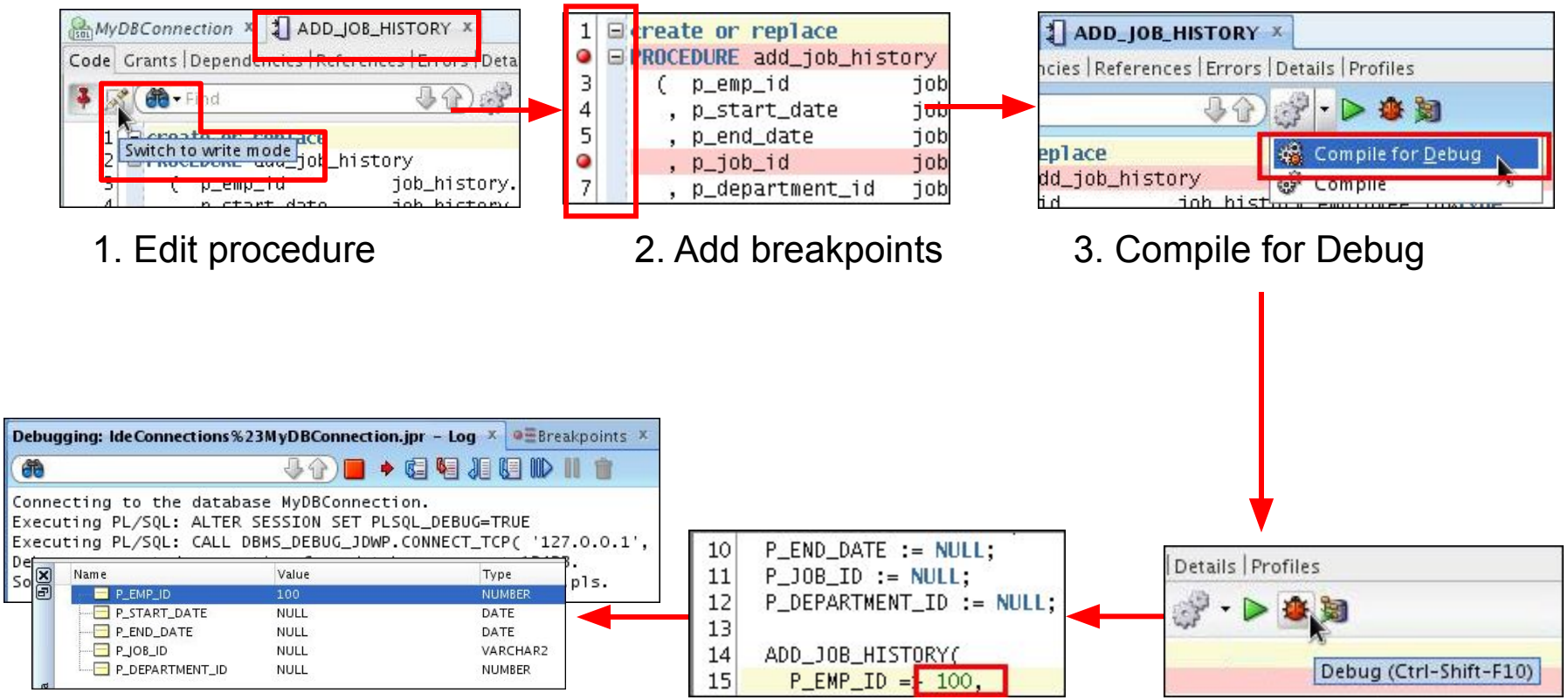
- Working with functions:
  - Differentiating between a procedure and a function
  - Describing the uses of functions
  - Creating, invoking, and removing stored functions
- Introducing the SQL Developer debugger

Debugging PL/SQL Subprograms Using the SQL Developer Debugger

- You can use the debugger to control the execution of your PL/SQL program.
- To debug a PL/SQL subprogram, a *security administrator* needs to grant the following privileges to the application developer:
  - DEBUG ANY PROCEDURE
  - DEBUG CONNECT SESSION

```
GRANT DEBUG ANY PROCEDURE TO ora61;
GRANT DEBUG CONNECT SESSION TO ora61;
```

Debugging a Subprogram: Overview



1. Edit procedure

2. Add breakpoints

3. Compile for Debug

4. Debug

5. Enter parameter value(s)

6. Choose debugging tool, and monitor data

The Procedure or Function Code Editing Tab

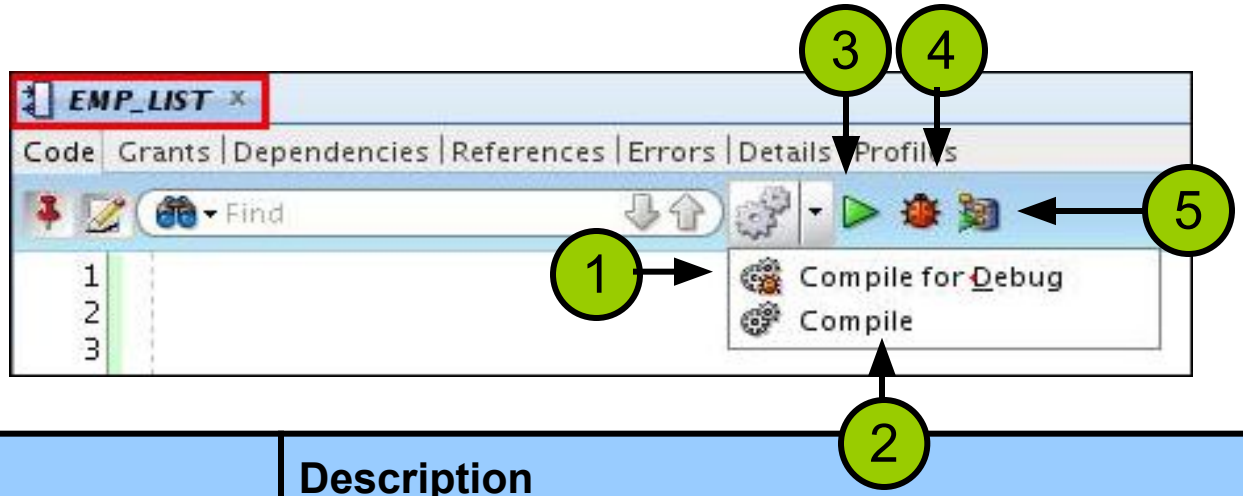
MyDBConnection × | ADD_JOB_HISTORY ×

Code | Grants | Dependencies | References | Errors | Details | Profiles

Find

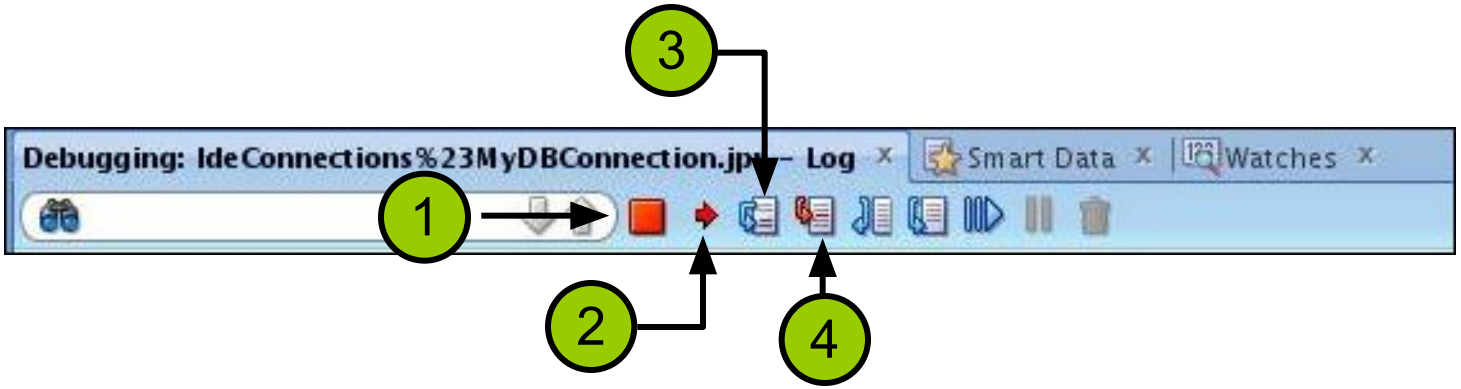
```
1   create or replace
    PROCEDURE add_job_history
3     (  p_emp_id          job_history.employee_id%type
4      , p_start_date      job_history.start_date%type
5      , p_end_date        job_history.end_date%type
       , p_job_id          job_history.job_id%type
7      , p_department_id   job_history.department_id%type
8      )
9   IS
10  BEGIN
11    INSERT INTO job_history (employee_id, start_date, end_date,
12                             job_id, department_id)
13      VALUES(p_emp_id, p_start_date, p_end_date, p_job_id, p_department_id);
14  END add_job_history;
```

The Procedure or Function Tab Toolbar



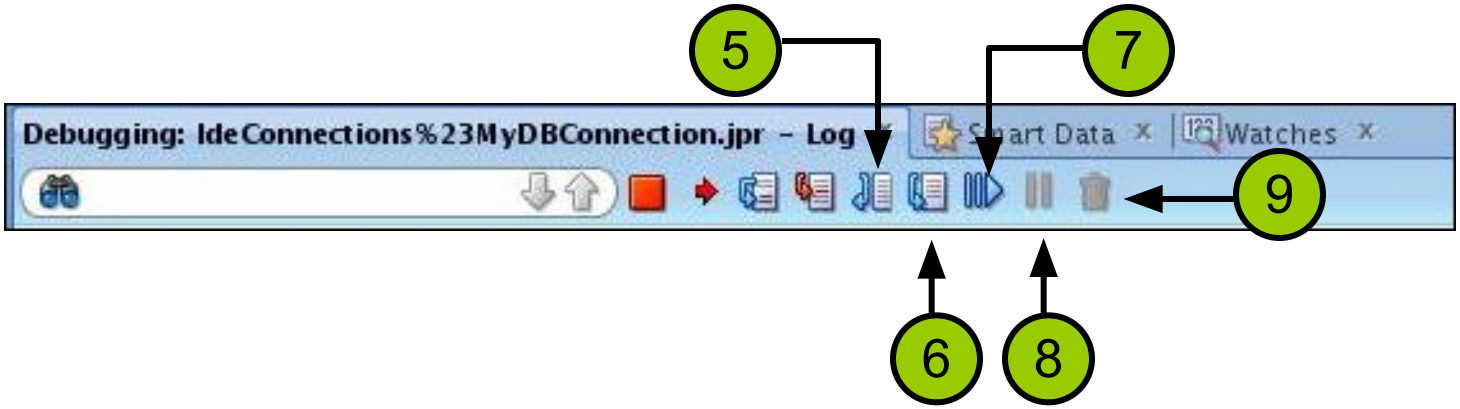
Icon	Description
1. Compile for Debug	Compiles the subprogram so that it can be debugged
2. Compile	Compiles the subprogram
3. Run	Starts normal execution of the function or procedure, and displays the results in the Running - Log tab
4. Debug	Executes the subprogram in debug mode, and displays the Debugging - Log tab, which includes the debugging toolbar for controlling execution
5. Profile	Displays the Profile window that you use to specify parameter values for running, debugging, or profiling a PL/SQL function or procedure

The Debugging – Log Tab Toolbar



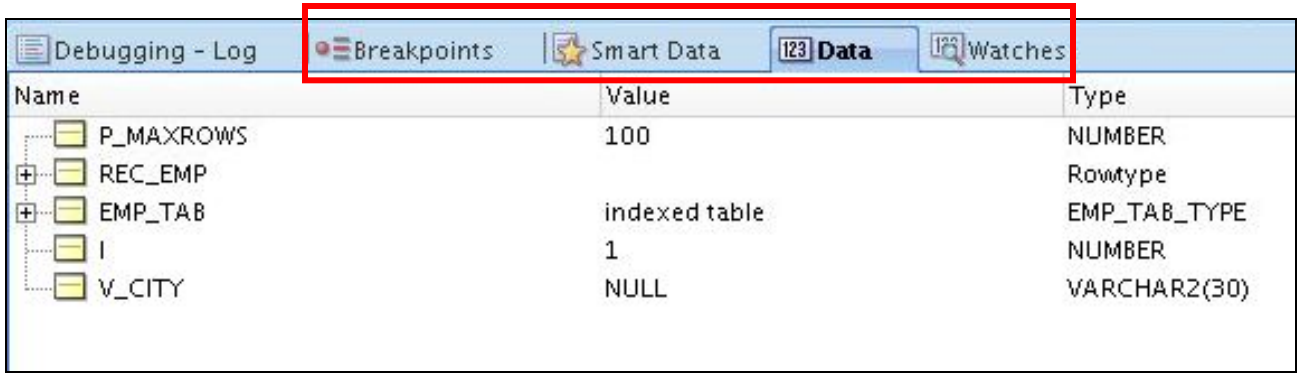
Icon	Description
1. Terminate	Halts and exits the execution
2. Find Execution Point	Goes to the next execution point
3. Step Over	Bypasses the next subprogram and goes to the next statement after the subprogram
4. Step Into	Executes a single program statement at a time. If the execution point is located on a call to a subprogram, it steps into the first statement in that subprogram

The Debugging – Log Tab Toolbar



Icon	Description
5. Step Out	Leaves the current subprogram and goes to the next statement with a breakpoint
6. Step to End of Method	Goes to the last statement of the current subprogram
7. Resume	Continues execution
8. Pause	Halts execution but does not exit
9. Garbage Collect	Removes invalid objects from the cache

Additional Tabs



Tab	Description
Breakpoints	Displays breakpoints, both system-defined and user-defined.
Smart Data	Displays information about variables. You can specify these preferences by right-clicking in the Smart Data window and selecting Preferences.
Data	Located under the code text area; displays information about all variables
Watches	Located under the code text area; displays information about watches

Debugging a Procedure Example: Creating a New `emp_list` Procedure

```
1  CREATE OR REPLACE PROCEDURE emp_list(pmaxrows IN NUMBER) AS
2  CURSOR emp_cursor IS
3  SELECT d.department_name,
4    e.employee_id,
5    e.last_name,
6    e.salary,
7    e.commission_pct
8  FROM departments d,
9    employees e
10 WHERE d.department_id = e.department_id;
11 emp_record emp_cursor % rowtype;
12 type emp_tab_type IS TABLE OF emp_cursor % rowtype INDEX BY binary_integer;
13 emp_tab emp_tab_type;
14 i NUMBER := 1;
15 v_city VARCHAR2(30);
16 BEGIN
17
18   OPEN emp_cursor;
19   FETCH emp_cursor
20   INTO emp_record;
21   emp_tab(i) := emp_record;
22   WHILE(emp_cursor % FOUND)
23    AND(i <= pmaxrows)
24   LOOP
25     i := i + 1;
26     FETCH emp_cursor
27     INTO emp_record;
28     emp_tab(i) := emp_record;
29     v_city := get_location(emp_record.department_name);
30     DBMS_OUTPUT.PUT_LINE('Employee ' || emp_record.last_name || ' works in ' || v_city);
31   END LOOP;
32
33   CLOSE emp_cursor;
34   FOR j IN REVERSE 1 .. i
35   LOOP
36     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
37   END LOOP;
38 END emp_list;
```

Debugging a Procedure Example: Creating a New `get_location` Function

```
1  CREATE OR REPLACE FUNCTION get_location(p_deptname IN VARCHAR2) RETURN VARCHAR2 AS
2  v_loc_id NUMBER;
3  v_city VARCHAR2(30);
4  BEGIN
5    SELECT d.location_id,
6      l.city
7    INTO v_loc_id,
8      v_city
9    FROM departments d,
10     locations l
11   WHERE UPPER(department_name) = UPPER(p_deptname)
12    AND d.location_id = l.location_id;
13   RETURN v_city;
14 END get_location;
```

Setting Breakpoints and Compiling `emp_list` for Debug Mode

MyDBConnection | EMP_LIST

Code | Grants | Dependencies | References | Errors | Details | Profiles

Find

Compile for Debug
Compile

```
1  create or replace
2  PROCEDURE emp_list
3  (p_maxrows IN NUMBER)
4  IS
5  CURSOR cur_emp IS
6    SELECT d.department_name, e.employee_id, e.last_name,
7           e.salary, e.commission_pct
8    FROM  departments d, employees e
9    WHERE d.department_id = e.department_id;
10   rec_emp cur_emp%ROWTYPE;
11   TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
12   emp_tab emp_tab_type;
13 i NUMBER := 1;
14 v_city VARCHAR2(30);
15 BEGIN
16   OPEN cur_emp;
17   FETCH cur_emp INTO rec_emp;
18   emp_tab(i) := rec_emp;
19   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20      i := i + 1;
21      FETCH cur_emp INTO rec_emp;
22      emp_tab(i) := rec_emp;
23      v_city := get_location (rec_emp.department_name);
24      dbms_output.put_line('Employee ' || rec_emp.last_name ||
25        ' works in ' || v_city );
26   END LOOP;
27   CLOSE cur_emp;
28   FOR j IN REVERSE 1..i LOOP
29      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30   END LOOP;
31 END emp_list;
```

Messages - Log | Breakpoints

Compiled

Compiling the `get_location` Function for Debug Mode

MyDBConnection | EMP_LIST | code_ex_03.sql | GET_LOCATION

Code | Grants | Dependencies | References | Errors | Details | Profiles

Find

Compile for Debug

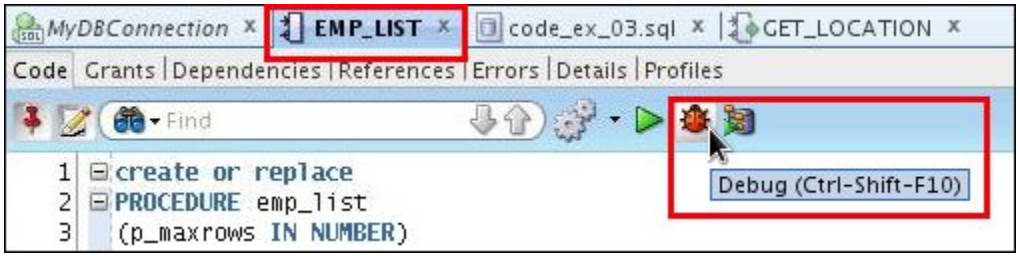
Compile

```
1  create or replace
2  FUNCTION get_location
3  ( p_deptname IN VARCHAR2) RETURN VARCHAR2
4  AS
5    v_loc_id NUMBER;
6    v_city   VARCHAR2(30);
7  BEGIN
8    SELECT d.location_id, l.city INTO v_loc_id, v_city
9    FROM departments d, locations l
10   WHERE upper(department_name) = upper(p_deptname)
11   and d.location_id = l.location_id;
12   RETURN v_city;
13 END GET_LOCATION;
```

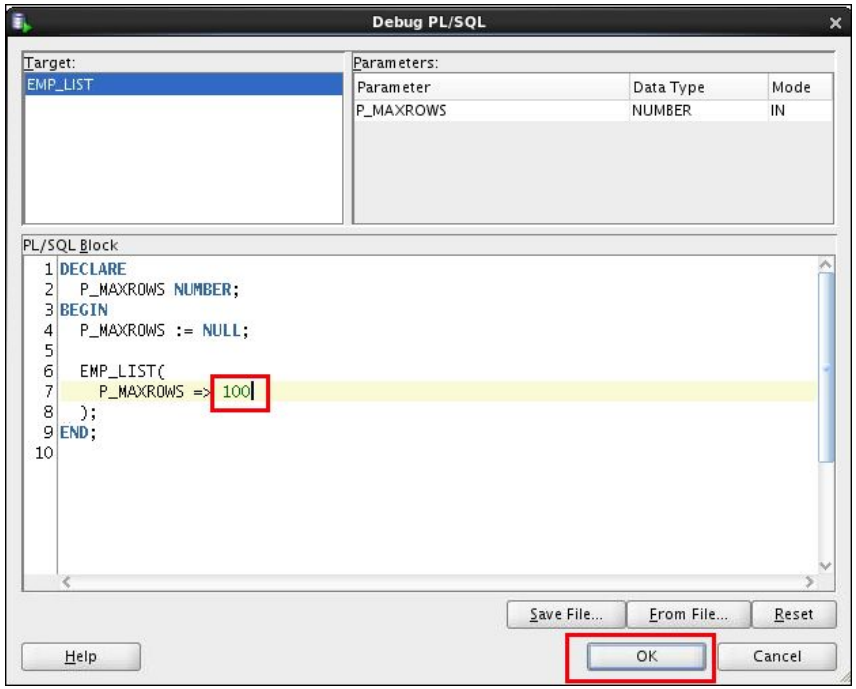
Messages - Log | Breakpoints

Compiled

Debugging `emp_list` and Entering Values for the `PMAXROWS` Parameter



Enter the procedure's parameter value using the anonymous block.



Debugging `emp_list`: Step Into (F7) the Code

Program control stops at first breakpoint.

1

```
1  CREATE OR REPLACE PROCEDURE emp_list
2  (p_maxrows IN NUMBER)
3  IS
4  CURSOR cur_emp IS
5     SELECT d.department_name, e.employee_id, e.last_name,
6            e.salary, e.commission_pct
7     FROM  departments d, employees e
8     WHERE d.department_id = e.department_id;
9     rec_emp cur_emp%ROWTYPE;
10    TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11    emp_tab emp_tab_type;
12  i NUMBER := 1;
13  v_city VARCHAR2(30);
14  BEGIN
16    OPEN cur_emp;
17    FETCH cur_emp INTO rec_emp;
      emp_tab(i) := rec_emp;
      WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19       i := i + 1;
20       FETCH cur_emp INTO rec_emp;
21       emp_tab(i) := rec_emp;
         v_city := get_location (rec_emp.department_name);
23       dbms_output.put_line('Employee ' || rec_emp.last_name ||
24         ' works in ' || v_city );
25    END LOOP;
      CLOSE cur_emp;
27    FOR j IN REVERSE 1..i LOOP
28       DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
29    END LOOP;
30  END emp_list;
```

Debugging: IdeConnections%23MyDBConnection.jpr – Log | Smart Data | Watches

```
Connecting to the database MyDBConnection.
Executing PL/SQL: ALTER SESSION SET PLSQL_DEBUG=TRUE
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.CONNECT_TCP( '127.0.0.1', '15290' )
Debugger accepted connection from database on port 15290.
Source breakpoint occurred at line 16 of EMP_LIST.pls.
```

Debugging `emp_list`: Step Into (F7) the Code

```
create or replace
PROCEDURE emp_list
(p_maxrows IN NUMBER)
IS
CURSOR cur
  SELECT d  ent_name, e.employee_id, e.last_name,
         e.  , e.commission_pct
  FROM  departments d, employees e
  WHERE d.department_id = e.department_id;
  rec_emp cur_emp%ROWTYPE;
  TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
  emp_tab emp_tab_type;
i NUMBER := 1;
v_city VARCHAR2(30);
BEGIN
  OPEN cur_emp;
  FETCH cur_emp INTO rec_emp;
  emp_tab(i) := rec_emp;
  WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
     i := i + 1;
     FETCH cur_emp INTO rec_emp;
     emp_tab(i) := rec_emp;
     v_city := get_location (rec_emp.department_name);
     dbms_output.put_line('Employee ' || rec_emp.last_name ||
       ' works in ' || v_city );
  END LOOP;
  CLOSE cur_emp;
  FOR j IN REVERSE 1..i LOOP
     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
  END LOOP;
END emp_list;
```

1

```
OPEN cur_emp;
FETCH cur_emp INTO rec_emp;
emp_tab(i) := rec_emp;
```

2

Step Into (F7): Steps into and executes the cursor code.

3

```
SELECT d.department_name, e.employee_id,
       e.salary, e.commission_pct
FROM  departments d, employees e
WHERE d.department_id = e.department_id;
rec_emp cur_emp%ROWTYPE;
```

Debugging: IdeConnections%23MyDBConnection.jpr - Log | Breakpoints | Smart Data | Data

```
Connecting to the database MyDBConnection.
Executing PL/SQL: ALTER SESSION SET PLSQL_DEBUG=TRUE
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.CONNECT_TCP( '127.0.0.1', '63138' )
Debugger accepted connection from database on port 63138.
Source breakpoint occurred at line 16 of EMP_LIST.pls.
```

Viewing the Data

```
18
    OPEN emp_cursor;
20  FETCH emp_cursor
```

Debugging - Log | Breakpoints | Smart Data | Data | Watches

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
DEPARTMENT_NAME	NULL	VARCHAR2(30)
EMPLOYEE_ID	NULL	NUMBER(6,0)
LAST_NAME	NULL	VARCHAR2(25)
SALARY	NULL	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB	Indexed table	EMP_TAB_TYPE
_values		EMP_TAB_TYPE elemen...
I	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

```
    OPEN cur_emp;
16  FETCH cur_emp INTO rec_emp;
17  emp_tab(i) := rec_emp;
    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
```

Debugging - Log | Breakpoints | Smart Data | Data | Watches

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
DEPARTMENT_NAME	'Executive'	VARCHAR2(30)
EMPLOYEE_ID	100	NUMBER(6,0)
LAST_NAME	'King'	VARCHAR2(25)
SALARY	26400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB	Indexed table	EMP_TAB_TYPE
_values		EMP_TAB_TYPE elemen...
I	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

Modifying the Variables While Debugging the Code

2

Name | Value | Type

Modify Value...
Watch
Inspect
Watch Class Loader
Inspect Class Loader
Object Preferences...
Export...
Preferences...
Collapse All

Name	Value	Type
P_MA	100	NUMBER
REC_		Rowtype
D	'Executive'	VARCHAR2(30)
E	100	NUMBER(6,0)
L	'King'	VARCHAR2(25)
S	26400	NUMBER(8,2)
C	NULL	NUMBER(2,2)
EMP_	indexed table	EMP_TAB_TYPE
		EMP_TAB_TYPE elem
I	1	NUMBER
V_CITY	NULL	

1

Modify Value

NUMBER I

Current Value:

1

3

Interpret New Value as Object Address

Help | OK

3

4

Debugging: IdeConnections%23MyDBConnecti... | Smart Data | Data | Watches

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
EMP_TAB	indexed table	EMP_TAB_TYPE
I	3	NUMBER
V_CITY	NULL	VARCHAR2(30)

Debugging `emp_list`: Step Over the Code

```
14  BEGIN
    OPEN cur_emp;
16  FETCH cur_emp INTO rec_emp;
17  emp_tab(i) := rec_emp;
    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19      i := i + 1;
20      FETCH cur_emp INTO rec_emp;
21      emp_tab(i) := rec_emp;
        v_city := get_location (rec_emp.department_name);
23      dbms_output.put_line('Employee ' || rec_emp.last_name |
24        ' works in ' || v_city );
```

1

F8

```
14  BEGIN
    OPEN cur_emp;
    FETCH cur_emp INTO rec_emp;
17  emp_tab(i) := rec_emp;
    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19      i := i + 1;
20      FETCH cur_emp INTO rec_emp;
21      emp_tab(i) := rec_emp;
        v_city := get_location (rec_emp.department_name);
23      dbms_output.put_line('Employee ' || rec_emp.last_name ||
24        ' works in ' || v_city );
```

2

F8

```
14  BEGIN
    OPEN cur_emp;
16  FETCH cur_emp INTO rec_emp;
    emp_tab(i) := rec_emp;
    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19      i := i + 1;
20      FETCH cur_emp INTO rec_emp;
21      emp_tab(i) := rec_emp;
        v_city := get_location (rec_emp.department_name);
23      dbms_output.put_line('Employee ' || rec_emp.last_name ||
24        ' works in ' || v_city );
```

3

Step Over (F8):
Executes the Cursor
(same as F7),
but control is not transferred
to Open Cursor code

Debugging `emp_list`: Step Out of the Code (Shift + F7)

1

```
15  BEGIN
16    OPEN cur_emp;
17    FETCH cur_emp INTO rec_emp;
18    emp_tab(i) := rec_emp;
19    WHILE (cur_emp%FOUND) AND (i <
20        i := i + 1;
21        FETCH cur_emp INTO rec_emp;
22        emp_tab(i) := rec_emp;
23        v_city := get_location (rec
24        dbms_output.put_line('Emplo
25          ' works in ' || v_city );
26    END LOOP;
27    CLOSE cur_emp;
```

2

PL/SQL Block

```
1 DECLARE
2   PMAXROWS NUMBER;
3 BEGIN
4   PMAXROWS := NULL;
5
6   EMP_LIST(
7     PMAXROWS => 4
8   );
9 END;
```

3

```
OPEN cur_emp;
FETCH cur_emp INTO rec_emp;
emp_tab(i) := rec_emp;
```

4

```
16  FETCH cur_emp INTO rec_emp;
17  emp_tab(i) := rec_emp;
18  WHILE (cur_emp%FOUND) AND (i
```

5

```
20      FETCH cur_emp INTO rec_em
21      emp_tab(i) := rec_emp;
22      v_city := get_location (r
```

6

```
25  END LOOP;
26  CLOSE cur_emp;
27  FOR j IN REVERSE 1..i LOOP
28      DBMS_OUTPUT.PUT_LINE(emp_
29  END LOOP;
30 END emp_list;
```

7

MyDBConnection | EMP_LIST | EMP_LIST | ANONYMOUS_BLOCK

```
1 DECLARE
2   P_MAXROWS NUMBER;
3 BEGIN
4   P_MAXROWS := NULL;
5
6   EMP_LIST(
7     P_MAXROWS => 4
8   );
9 END;
10
```

8

Debugging - Log

```
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.DISCONNECT()
Employee Kochhar works in Seattle
Employee De Haan works in Seattle
Employee Hunold works in Southlake
Employee Ernst works in Southlake
Ernst
Hunold
De Haan
Kochhar
King
Process exited.
Disconnecting from the database MyDBConnection.
Debugger disconnected from database.
```

Messages | Debugging

Debugging `emp_list`: Run to Cursor (F4)

```
10    emp_record emp_cursor%ROWTYPE;
11    TYPE emp_tab_type IS TABLE OF emp_cursor%ROWTYPE INDEX BY BINARY_INTEGER;
12    emp_tab emp_tab_type;
13  i NUMBER := 1;
14  v_city VARCHAR2(30);
15  BEGIN
16    OPEN emp_cursor;
17    FETCH emp_cursor INTO emp_record;
18    emp_tab(i) := emp_record;
19    WHILE (emp_cursor%FOUND) AND (i <= pMaxRows) LOOP
20       i := i + 1;
21       FETCH emp_cursor INTO emp_record;
22       emp_tab(i) := emp_record;
23       v_city := get_location (emp_record.department_name);
24       dbms_output.put_line('Employee ' || emp_record.last_name ||
25         ' works in ' || v_city );
26    END LOOP;
27    CLOSE emp_cursor;
28    FOR j IN REVERSE 1..i LOOP
29       DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30    END LOOP;
31  END emp_list;
32
```

Debugging - Log | Breakpoints

```
Connecting to the database debugging.
Executing PL/SQL: ALTER SESSION SET PLSQL_DEBUG=TRUE
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.CONNECT_TCP( '12
Debugger accepted connection from database on port 15
Processing 59 classes that have already been prepared.
Finished processing prepared classes.
Source breakpoint occurred at line 16 of EMP_LIST.pls.
```

Name	Value
	5
	indexed
	1
	NULL

Cut	Ctrl-X
Copy	Ctrl-C
Paste	Ctrl-V
Select All	Ctrl-A
Run	
Compile	Ctrl+Shift-F9
Compile for Debug	
Debug	
Run to Cursor	F4
Watch	Ctrl-F5
Inspect...	
Replace With	▸
Preferences...	
Format SQL...	Ctrl-B

Smart Data | Data | Watches

Name	Value	Type
PMAXROWS	5	NUMBER
EMP_RECORD		Rowtype
DEPARTMENT_NAME	'Administration'	VARCHAR2...
EMPLOYEE_ID	200	NUMBER(6,0)
LAST_NAME	'Whalen'	VARCHAR2...
SALARY	4400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB	indexed table	EMP_TAB_...
I	1	NUMBER
V_CITY	NULL	VARCHAR2...

Run to Cursor F4:
Run to your cursor location without having to single step or set a breakpoint.

Debugging `emp_list`: Step to End of Method

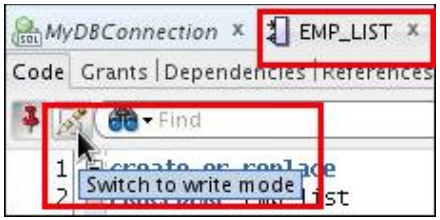
```
15  BEGIN
    OPEN emp_cursor;
17  FETCH emp_cursor INTO emp_record;
18  emp_tab(i) := emp_record;
    WHILE (emp_cursor%FOUND) AND (i <= pMaxRows) LOOP
20      i := i + 1;
21      FETCH emp_cursor INTO emp_record;
22      emp_tab(i) := emp_record;
23      v_city := get_location (emp_record.department_n
24      dbms_output.put_line('Employee ' || emp_record.
25        ' works in ' || v_city );
26  END LOOP;
27  CLOSE emp_cursor;
28  FOR j IN REVERSE 1..i LOOP
29      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30  END LOOP;
31 END emp_list;
```

```
    OPEN emp_cursor;
17  FETCH emp_cursor INTO emp_record;
18  emp_tab(i) := emp_record;
    WHILE (emp_cursor%FOUND) AND (i <= pMaxRows) LOOP
20      i := i + 1;
21      FETCH emp_cursor INTO emp_record;
22      emp_tab(i) := emp_record;
23      v_city := get_location (emp_record.department_name);
24      dbms_output.put_line('Employee ' || emp_record.last_name ||
25        ' works in ' || v_city );
26  END LOOP;
27  CLOSE emp_cursor;
28  FOR j IN REVERSE 1..i LOOP
29      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30  END LOOP;
31 END emp_list;
32
```

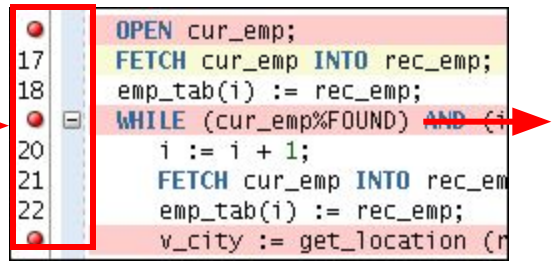
Loops until `i` <> `PMAXROWS`

```
1  DECLARE
2    PMAXROWS NUMBER;
3  BEGIN
4    PMAXROWS := NULL;
5
6    EMP_LIST(
7      PMAXROWS => 2
8    );
   END;
```

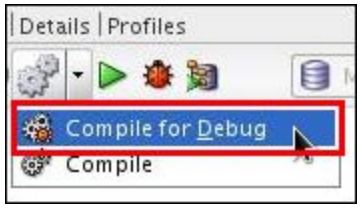
Debugging a Subprogram Remotely: Overview



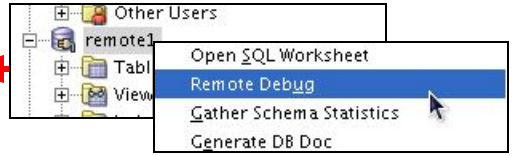
1. Edit procedure



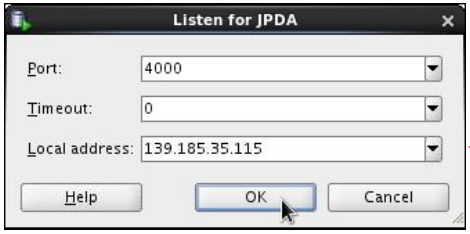
2. Add breakpoints



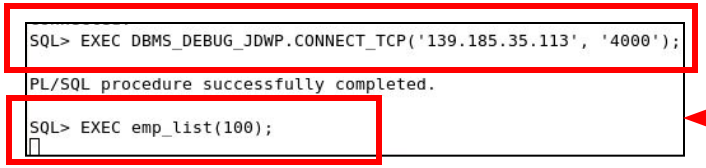
3. Compile for Debug



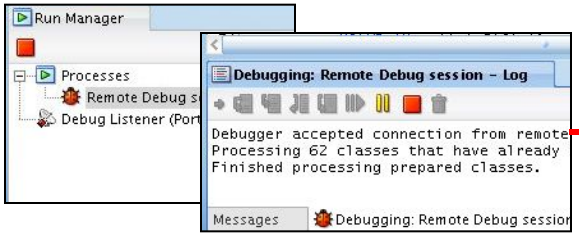
4. Select Remote Debug



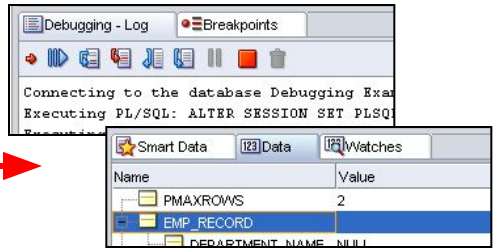
5. Enter local machine IP address and debugging port



6. Issue the debugger connection command and call procedure in another session such as SQL*Plus



7. When the breakpoint is reached, control passes to SQL Developer



8. Debug and monitor data using debugging tools

Summary

In this lesson, you should have learned how to:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Understand the basic functionality of the SQL Developer debugger

Practice 3-2 Overview: Introduction to the SQL Developer Debugger

This practice covers the following topics:

- Creating a procedure and a function
- Inserting breakpoints in the procedure
- Compiling the procedure and function for debug mode
- Debugging the procedure and stepping into the code
- Displaying and modifying the subprograms' variables