CS 4513 – Fall 2013
Instructions for PL/SQL

- Anonymous Blocks
- Declaration Statements, Data types and Variables
- Executable Statements (Some operators, Conditional Logic, Looping)
- More operators
- Cursors
- Procedures
- Functions
- Packages
- Exceptions
- How to edit and execute PL/SQL programs in Oracle SQL Developer
- A Complete Example
- How to call a PL/SQL procedure in a Java program
- Reference

1. **Anonymous Blocks**

You have to put your PL/SQL code in blocks. The simplest type of blocks is Anonymous Blocks.

```
/*General Structure of an Anonymous Block*/
[DECLARE
    declaration statements]
BEGIN
    executable statement(s)
[EXCEPTION
    exception handlers]
 END;
```

Legend:

The square brackets [] mark off sections that are optional, i.e. declare and exception sections are optional.

UPPERCASE words are keywords.

Let's start with a simple example – an anonymous block that prints out 'hello, world!' on the screen. To enable printing on screen you have to type the following command in the worksheet of Oracle SQL Developer 3.2:

```
SET SERVEROUTPUT ON;
/*only once per session, that is until you disconnect from the connection*/
```

And here is the PL/SQL code:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Hello, world!');
END;
```

2. Declaration Statements

In PL/SQL before you can work with any kind of variable, you must first declare it; that is you must give it a name and specify its data type.

The most common data types in PL/SQL are String, Number, Data and Boolean. Here are some examples of declaration statements for every kind of data type.

Note that each declaration ends with semicolon.

2.1   String

Declaration: variable_name VARCHAR2(n);

   where n is the maximum length of the string content (up to 32767)

Example:

```
small_string VARCHAR2(4);
```

2.2   Number

Declaration: variable_name NUMBER[(precision [,scale])];

   where precision and scale are integers

Example:

```
bean_counter (10, 3);
/*can hold values with up to 10 digits of precision, 3 of which are to the right of the decimal point*/
```

2.3 Date

Declaration: variable_name DATE;

Example:

birth_date DATE;

2.4 Boolean

Declaration: variable_name BOOLEAN;

Example:

too_young_to_vote BOOLEAN;

Note that in PL/SQL a BOOLEAN variable can be TRUE, FALSE or NULL.

3. Executable Statements

An executable statement can be any SQL statement (except data definition statements such as CREATE TABLE), a procedure/function call (see sections 6 and 7), or any of the statements below, ending with semicolon.

3.1 Assignment statement – to assign a value to an already declared variable.

The assignment operator to do this is := (a colon followed by an equal sign).

For DATE the format is 'DD-MON-YYYY'. For example: 22-APR-2012.

Example: let's expand the example from section 1 – declare string and date variables, assign values to them and print their content on the screen.

```
DECLARE
   myMessage VARCHAR2(20); /*declaration*/
   myDate DATE;   /*declaration*/
BEGIN
   myMessage := 'hello, world!'; /*assignment*/
   myDate := '20-SEP-2012'; /*assignment*/
   DBMS_OUTPUT.PUT_LINE(myMessage);
   DBMS_OUTPUT.PUT_LINE(myDate);
END;
```

3.2 Output statement – to print on the screen.

PL/SQL has limited ability to print out. To do this you have to put what you want to print in the DBMS_OUTPUT buffer. You have already seen the command for this:

```
DBMS_OUTPUT.PUT_LINE(X);
```

where x can be the name of a variable (as in the example in section 3.1) or a literal value (as in the example in section 1). Note that all the lines put in the buffer are printed on the screen after the program completes its execution.

3.3 Input statement – to read from keyboard

The operator to do this is & (ampersand) followed by the name of the variable you want to read a value for. To automatically cast the input from keyboard to string or date you enclose the whole thing in ' ' (single quotes). In this way the user of your program does not need to enter them enclosed in single quotes. Numbers and Booleans do not require casting.

Note that the & operator is not part of PL/SQL, it is Oracle SQL Developer operator. In the Oracle SQL Developer environment, Oracle SQL Developer substitution variables allow portions of command syntax to be stored and edited into the command before it is run. Substitution variables are variables that you can use to pass runtime values, number or character, into a PL/SQL block. You can reference them within a PL/SQL with a preceding ampersand in the same manner as you reference Oracle SQL Developer substitution variables in a SQL statement. The text values are substituted into the PL/SQL block before PL/SQL block is executed. Therefore you cannot substitute different values for the substitution variables by using a loop. Only one value will replace the substitution variable.

Example: Read a string, a number, a date, and a boolean from keyboard and print them on the screen.

```
/*Example of input statements*/
DECLARE
   myMessage VARCHAR2(20); /*declaration*/
   myNumber NUMBER;        /*declaration*/
   myDate DATE;            /*declaration*/
   printOut BOOLEAN;       /*declaration*/
BEGIN
   myMessage := '&myMessage'; /*reading from keyboard and assignment*/
   myNumber := &myNumber;     /*reading from keyboard and assignment*/
   myDate := '&myDate';       /*reading from keyboard and assignment*/
   printOut := &youSure;      /*reading from keyboard and assignment*/
```

```
   DBMS_OUTPUT.PUT_LINE(myMessage);
   DBMS_OUTPUT.PUT_LINE(myNumber);
   DBMS_OUTPUT.PUT_LINE(myDate);
 END;
```

3.4   IF statement (Conditional Logic)

The general format of the IF statement is as follows.

```
/*General format of IF statement*/
IF condition 1
THEN
    executable statements
[ELSIF condition 2
THEN
    executable statements]
[ELSIF condition n
THEN
    executable statements]
[ELSE
THEN
    executable last statements]
END IF;
```

Note that the keyword is ELSIF, not ELSEIF.

Example: let's modify the example in section 3.3 so that it prints the contents of myMessage, myNumber and myDate only if the Boolean printOut is TRUE, else print a message 'No print on screen'.

```
/*Example of IF statement*/
DECLARE
   myMessage VARCHAR2(20); /*declaration*/
   myNumber NUMBER;        /*declaration*/
   myDate DATE;            /*declaration*/
   printOut BOOLEAN;       /*declaration*/
BEGIN
   myMessage := '&myMessage'; /*reading from keyboard and assignment*/
   myNumber := &myNumber;    /*reading from keyboard and assignment*/
   myDate := '&myDate';      /*reading from keyboard and assignment*/
   printOut := &youSure;    /*reading from keyboard and assignment*/

   IF printOut
```

```
    THEN
      DBMS_OUTPUT.PUT_LINE(myMessage);
      DBMS_OUTPUT.PUT_LINE(myNumber);
      DBMS_OUTPUT.PUT_LINE(myDate);
    ELSE
      DBMS_OUTPUT.PUT_LINE('No print on screen');
    END IF;
  END;
```

Also, there are CASE statements for conditional logic (check the references).

3.5   Looping

3.5.1 FOR Loop – you know how many times you want to execute the statements inside the loop. The general format of the FOR loop is as follows.

```
/*General format of FOR loop*/
FOR loop_counter IN [REVERSE] lower_bound .. upper_bound
LOOP
    statements
END LOOP;
```

Example: Print out the value stored in the counter variable.

```
/*Example of FOR loop*/
DECLARE
    counter NUMBER;
BEGIN
    FOR counter IN 1 .. 6
    LOOP
        DBMS_OUTPUT.PUT_LINE(counter);
    END LOOP;
END;
```

The counter will get values 1, 2, 3, 4, 5, 6 and they will be printed on the screen.

If you want to assign values to the counter in reverse order, use the REVERSE option.

Example: the same loop, but the counter will get values in reverse order.

```
/*Example of FOR loop in reverse order*/
DECLARE
    counter NUMBER;
BEGIN
    FOR counter IN REVERSE 1 .. 6
```

```
   LOOP
      DBMS_OUTPUT.PUT_LINE(counter);
   END LOOP;
END;
```

The counter will get values 6, 5, 4, 3, 2, 1.

3.5.2 Simple (Infinite) Loop – you don't know the exact number of repetitions, but it will execute at least once. To get out of the loop, you specify a condition, when it becomes true, the program gets out of the loop. The general format of the simple loop is as follows.

```
/*General format of simple loop*/
LOOP
   statements
   EXIT WHEN condition
END LOOP;
```

Example: same as the example in section 3.5.1, but will stop when counter becomes equal to 4.

```
/*Example of simple loop*/
DECLARE
   counter NUMBER;
BEGIN
   counter := 1;
   LOOP
      DBMS_OUTPUT.PUT_LINE(counter);
      counter := counter + 1;
      EXIT WHEN counter >= 4;
   END LOOP;
END;
```

There is also a WHILE loop (check the references).

4. More operators

Here are some more PL/SQL operators.

Notation	Meaning
+,-,/,*,**	Addition, subtraction, division, multiplication, exponentiation
AND, OR, NOT	Boolean operators
=	Equality (of non-nulls)
!=	Inequality
<,>,<=,>=	Less than, greater than, less than or equal, greater than or equal
IN	Equality disjunction
BETWEEN	Range test
IS NULL	Nullity test
IS NOT NULL	Non-nullity test
LIKE	Wildcard matching (for strings)
||	Concatenation (for strings)
--	A line starting with – is a comment line
/*block*/	A block insider /* and */ is treated as comment – multiline comment

5. Cursors

A cursor is a named space in memory where the program can get selected data out of the database.

For every cursor you should associate a corresponding SELECT statement.

To retrieve data you OPEN the cursor, FETCH from it and CLOSE it.

Example: Suppose we have a table "students" in which we have stored student_id and student_gpa. Now we want to count how many students who have gpa greater than or equal to 3.0.

Here is the PL/SQL code:

```
/*Example of using the cursor*/
DECLARE
    number_of_students NUMBER;
    /*the following four rows is the declaration of the cursor*/
    CURSOR ns_cur IS
        SELECT COUNT(*)
        FROM students
        WHERE student_gpa >= 3.0;
BEGIN
    number_of_students := 0;

    OPEN ns_cur;                    --open the cursor
    FETCH ns_cur INTO number_of_students; --fetch from it
```

```
    CLOSE ns_cur;                    --close the cursor
    --print out the number of students with gpa >=3.0
    DBMS_OUTPUT.PUT_LINE(number_of_students);
END;
```

6. Procedures

A procedure is a named program that executes some predefined statements and then returns control to whatever calls it. After defining a procedure you can invoke it by calling its name and providing any needed parameters.

There are 2 different types of procedures.

The first type is a "run time" procedure, that is, they exist only when you execute a program that contains them – much like methods in other languages as C++, Java, etc. The general structure of a "run time" procedure is as follows.

```
/*General Structure of a Run Time Procedure*/
PROCEDURE procedure_name
    (parameter1 MODE DATATYPE [DEFAULT expression],
     parameter2 MODE DATATYPE [DEFAULT expression],
     …)
AS
[variable1 DATATYPE;
 Variable2 DATATYPE;
 …]
BEGIN
    statements;
[EXCEPTION
    WHEN exception_name
    THEN
        exception handlers]
END;
```

MODE: can be IN, OUT or IN OUT.

-   IN means calling program supplies the values of the parameters and you cannot change it inside the procedure.
-   OUT means the procedure sets the value of the parameter and the calling program can read it.
-   IN OUT – the calling program supplies a value, the procedure can read it and change it, and the calling program can read the updated value.

[DEFAULT expression] – you can initialize the parameters to some default value. This is optional.

You should define a run time procedure in the declaration section.

Example: A procedure that inserts a tuple <student_id, student_gpa> into the table "students" which has attributes student_id and student_gpa.

```
PROCEDURE insertStudent
    (id IN VARCHAR2, gpa IN NUMBER)
AS
BEGIN
    --the next two lines are pure SQL DML statements
    INSERT INTO students(student_id, student_gpa)
    VALUES(id, gpa);
END;
```

The call to this procedure can be done in the anonymous block that contains this procedure, or from another procedure or function in the same anonymous block.

For example, consider the following anonymous block.

```
/*Example of calling a run time procedure*/
DECLARE
  …
  id VARCHAR2(20);
  gpa NUMBER;
  …
  PROCEDURE insertStudent
  (id IN VARCHAR2, pga IN NUMBER)
   AS
   BEGIN
     --the next two lines are pure SQL DML statements
     INSERT INTO student(student_id, student_gpa)
     VALUES(id, gpa);
   END;
BEGIN
  …
  id := '111-111-1111';
  gpa := 3.5;
  …
  /*call the run time procedure insertStudent*/
  insertStudent(id,gpa);
  …
END;
```

The second type is a "stored" procedure. When you define a stored procedure, it is stored in the database and you can invoke it from different anonymous blocks, procedures and functions, and from various environments that can access the database. This feature in PL/SQL is very attractive as it allows for sharing of PL/SQL code by different applications running at different places. The general structure of a stored procedure is as follows.

```
/*General Structure of a Stored Procedure*/
CREATE [OR REPLACE] PROCEDURE procedure_name
    (parameter1 MODE DATATYPE [DEFAULT expression],
     parameter2 MODE DATATYPE [DEFAULT expression],
     …)
AS
[variable1 DATATYPE;
 Variable2 DATATYPE;
 …]
BEGIN
    statements;
[EXCEPTION
    WHEN exception_name
    THEN
        Exception handlers]
END;
```

The code is almost the same as the run time procedure, except the CREATE [OR REPLACE] keywords.

MODE: same as in run time procedures.

Example: A stored procedure that inserts a tuple <student_id, student_gpa> into the table "students" which has attributes student_id and student_gpa.

```
CREATE OR REPLACE PROCEDURE insertStudent
    (id IN VARCHAR2, gpa IN NUMBER)
AS
BEGIN
    --the next two lines are pure SQL DML statements
    INSERT INTO students(student_id, student_gpa)
    VALUES(id, gpa);
END;
```

The call to a stored procedure can be done from any anonymous block, another procedure or a function, or directly from the Oracle SQL Developer Worksheet.

In case of an anonymous block, here is an example.

```
/*Example of calling a stored procedure*/
DECLARE
   …
   id VARCHAR2(20);
   gpa NUMBER;
   …
  /*note that here we did not declare the procedure
    It already exists*/
BEGIN
   …
   id := '111-111-1111';
   gpa := 3.5;
   …
   /*call the run time procedure insertStudent*/
   insertStudent(id,gpa);
   …
END;
```

7. Functions

A function is similar to a procedure, except that it returns one single value to the program that calls it. The data returned by a function is always of a specific, predefined data type.

There are also "run time" and "stored" functions.

Run time functions are defined as follows.

```
/*General Structure of a Run Time Function*/
FUNCTION function_name
    (parameter1 MODE DATATYPE [DEFAULT expression],
     parameter2 MODE DATATYPE [DEFAULT expression],
     …)
RETURN DATATYPE
AS
[variable1 DATATYPE;
 Variable2 DATATYPE;
 …]
BEGIN
   statements;
```

```
    RETURN expression
[EXCEPTION
    WHEN exception_name
    THEN
        exception handlers]
END;
```

Example: A run time function that returns the number of students with GPA greater than some value from the table "students".

```
/*Example of a run time function*/
FUNCTION getNumStudents (gpa IN NUMBER)
/*note that we declare the type of data to be returned*/
RETURN NUMBER
AS
    number_of_students NUMBER;
    /*the following four rows is the declaration of the cursor*/
    CURSOR ns_cur IS
        SELECT COUNT(*)
        FROM students
        WHERE student_gpa >= gpa;
BEGIN
    number_of_students := 0;

    OPEN ns_cur;                    --open the cursor
    FETCH ns_cur INTO number_of_students; --fetch from it
    CLOSE ns_cur;                   --close the cursor

    RETURN number_of_students;
END;
```

Same as the run time procedures, a run time function should be defined in the declaration section.

The call to this function should be done in the anonymous block that contains the function, or from another function or procedure in the same anonymous block.

For example, consider the following anonymous block:

```
/*Example of calling a run time function*/
DECLARE
    …
    countStudents NUMBER;

    …
    FUNCTION getNumStudents (gpa IN NUMBER)
   /*note that we declare the type of data to be returned*/
   RETURN NUMBER
    AS
        number_of_students NUMBER;
   /*the following four rows is the declaration of the cursor*/
        CURSOR ns_cur IS
        SELECT COUNT(*)
        FROM students
        WHERE student_gpa >= gpa;
BEGIN
    number_of_students := 0;

    OPEN ns_cur;                    --open the cursor
    FETCH ns_cur INTO number_of_students; --fetch from it
    CLOSE ns_cur;                   --close the cursor

    RETURN number_of_students;
END;

BEGIN
    …
    countStudents := getNumStudents(3.5);

    …
END;
```

On the other hand, stored functions are defined as follows.

```
/*General Structure of a Stored Function*/
CREATE [OR REPLACE] FUNCTION function_name
    (parameter1 MODE DATATYPE [DEFAULT expression],
    parameter2 MODE DATATYPE [DEFAULT expression],
    …)
RETURN DATATYPE
AS
[variable1 DATATYPE;
 Variable2 DATATYPE;
 …]
```

```
BEGIN
    statements;
    RETURN expression
[EXCEPTION
    WHEN exception_name
    THEN
        exception handlers]
END;
```

Example: A stored function that returns the number of students with GPA greater than some value from table "students".

```
/*Example of a stored function*/
CREATE [OR REPLACE] FUNCTION getNumStudents (gpa IN NUMBER)
/*note that we declare the type of data to be returned*/
RETURN NUMBER
AS
    number_of_students NUMBER;
    /*the following four rows is the declaration of the cursor*/
    CURSOR ns_cur IS
        SELECT COUNT(*)
        FROM students
        WHERE student_gpa >= gpa;
BEGIN
    number_of_students := 0;

    OPEN ns_cur;                    --open the cursor
    FETCH ns_cur INTO number_of_students; --fetch from it
    CLOSE ns_cur;                   --close the cursor

    RETURN number_of_students;
END;
```

The call to a stored function can be done from any anonymous block, another function or a procedure, or directly from the Oracle SQL Developer Worksheet.

For example, consider the following anonymous block:

```
/*Example of calling a stored function*/
DECLARE
    …
```

```
    countStudents NUMBER;
    …
/*note that here we did not declare the function, it already exists*/

BEGIN
    …
    /*call the stored function*/
    countStudents := getNumStudents(3.5);
    …
END;
```

8. Packages

You can use packages to organize your code. When you have several related stored procedures and/or functions you should combine them in a package. The package is stored in the database, as well as the procedures and functions it contains, and can be shared. Every package consists of:

8.1 Package specification – describes what can be done using this package (lists the headers of all procedures/functions).

A template looks like:

```
CREATE OR REPLACE PACKAGE package_name
AS
    Program1 header;
    Program2 header;
    …
END package_name;
```

8.2 Package body – describes how to do it (implementing procedures/functions).

```
CREATE OR REPLACE PACKAGE BODY package_name
AS
    Private_programs
  /*optional, you cannot invoke these programs from outside*/

    Program1 body;
    Program2 body;
    …
END package_name;
```

An example of package using the examples in sections 6 and 7:

```
/*An example of package*/
--package specification--
CREATE OR REPLACE PACKAGE student_pack
AS
   PROCEDURE insertStudent(id IN VARCHAR2, gpa IN NUMBER);
   FUNCTION getNumStudents(gpa IN NUMBER);
END student_pack;


--package body—
CREATE OR REPLACE PACKAGE BODY student_pack
AS
         /*the procedure*/
         CREATE OR REPLACE PROCEDURE insertStudent
   (id IN VARCHAR2, gpa IN NUMBER)
   AS
   BEGIN
     --the next two lines are pure SQL DML statements
     INSERT INTO students(student_id, student_gpa)
     VALUES(id, gpa);
   END insertStudent;


         /*the function*/
  CREATE OR REPLACE FUNCTION getNumStudents (gpa IN NUMBER)
   /*note that we declare the type of data to be returned*/
   RETURN NUMBER
   AS
     number_of_students NUMBER;
     CURSOR ns_cur IS
       SELECT COUNT(*)
       FROM students
       WHERE student_gpa >= gpa;
   BEGIN
     number_of_students := 0;

     OPEN ns_cur;                --open the cursor
     FETCH ns_cur INTO number_of_students; --fetch from it
     CLOSE ns_cur;                --close the cursor
     RETURN number_of_students;
     END getNumStudents;
END  student_pack;
```

Now, if you want to call the procedure or the function, you use the syntax:

```
package_name.function_name/procedure_name(parameters)
```

For example, to call them from an anonymous block (also you can do this from another function or procedure or from the Oracle SQL Developer Worksheet):

```
/*Example of calling a packet*/
DECLARE
   …
   numStudents NUMBER;
   …
BEGIN
   …
   student_pack.insertStudent('111-234',3.5);
   numStudents := student_pack.getNumStudents(3.6);
   …
END;
```

You will see another example of this (as well as calling a procedure/function directly from Oracle SQL Worksheet) in section 11.

9. Exceptions

PL/SQL implements run-time error handling via exceptions and exception handlers. When an error occurs during the execution of a PL/SQL program, an exception is raised. Immediately, program control is transferred to the exception section of the block in which the exception was raised. For more information about exceptions and exception handling refer to the materials listed in Reference.

10. How to edit and execute PL/SQL programs in Oracle SQL Developer?

Please refer to the following example and the materials listed in the Reference.

An example:

In the following example, we first create a table "student" with two attributes "student_id" and "name" via SQL, then we insert two rows into the table and print out the rows.

Step 1. Create the table "student" using SQL statements.

/*SQL code to create table student*/

create table student (student_ID NUMBER,
                                Name VARCHAR(30),
PRIMARY KEY (student_ID));

Step 2. Create a PL/SQL procedure to insert two rows into table student and print all the student records.
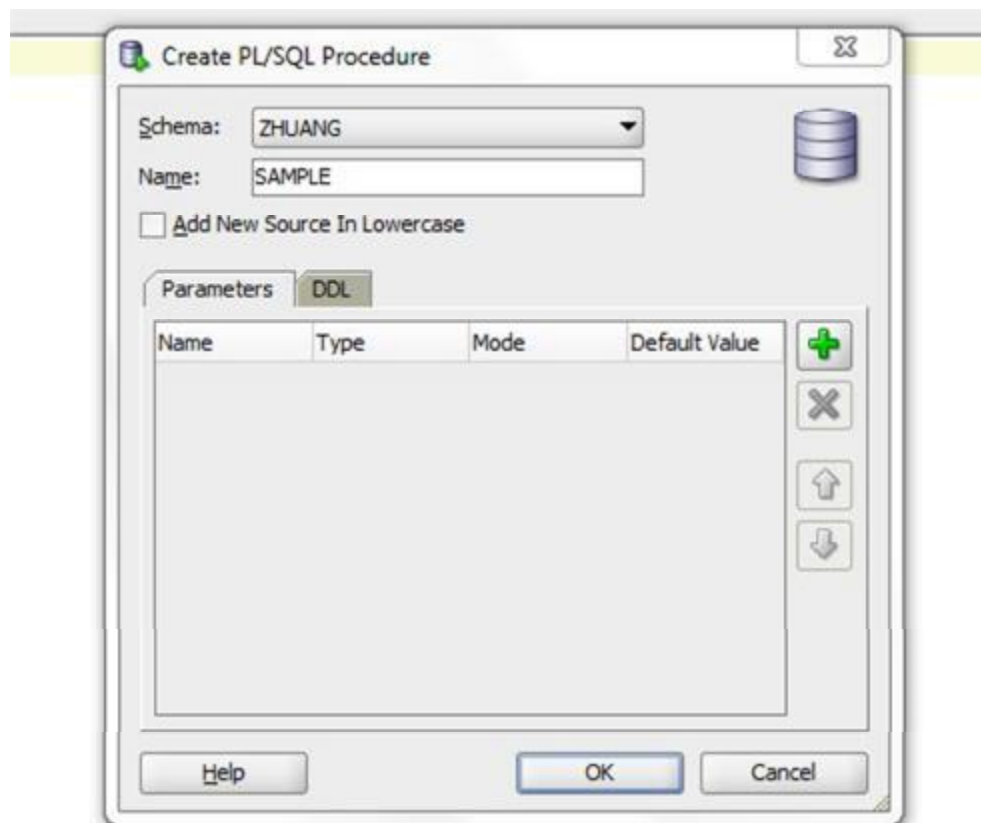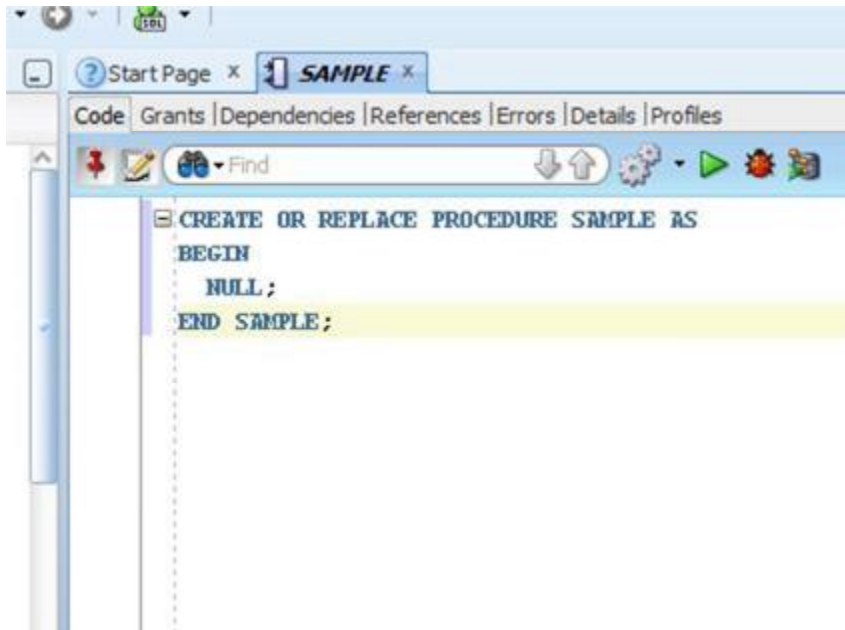
2.1 Click File -> New -> Procedure -> OK.
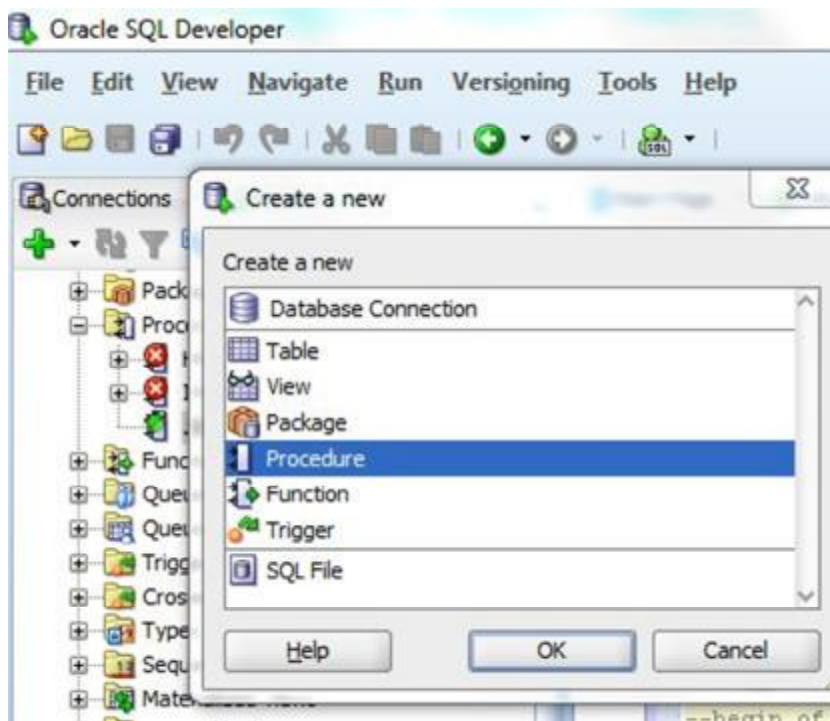
2.2 Select a connection and enter you login name and password.
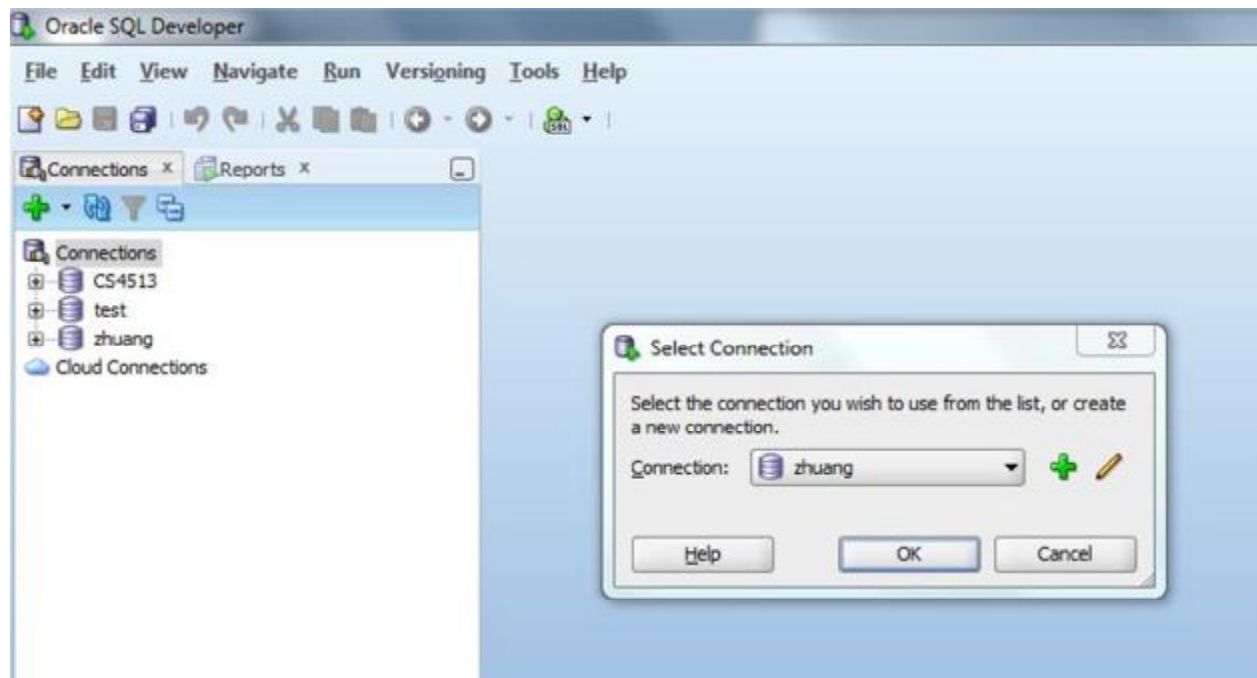
2.3 Give a name for the procedure.

2.4 Once you clicked the "OK" button, a procedure under that name with a null body will be displayed.

Step 3. Fill the procedure body with the procedure code.

Here is an example to insert two rows into the table "student" and print out all the student records.

```
/*Sample procedure
  Insert two rows in the table student
  Print out the table
 */
create or replace
PROCEDURE SAMPLE AS

    /* the next are DECLARE section */
    student_id_1 NUMBER;
    student_id_2 NUMBER;
    student_name_1 VARCHAR2(30);
    student_name_2 VARCHAR2(30);
    /* nested procedure to insert into the table */
    PROCEDURE insert_Student (student_id in NUMBER,
student_name in VARCHAR2) AS
    BEGIN
        /*the next two lines are pure SQL statements*/
        INSERT INTO STUDENT(student_id, name)
        VALUES (student_id, student_name);
    END;
```

```
/* begin of procedure SAMPLE */
BEGIN
    /*assign values to the variables declared above*/
    student_id_1 := 100;
    student_name_1 := 'Alice';
    student_id_2 := 200;
    student_name_2 := 'Bob';
    /*call the predefined procedure to insert*/
    insert_Student(student_id_1, student_name_1);
    insert_Student(student_id_2, student_name_2);
    /*Print out the table*/
    DBMS_OUTPUT.PUT_LINE('Student_id' || '  ' ||
'Student.name');
    FOR someone IN (select * from student)
       LOOP
        DBMS_OUTPUT.PUT_LINE(someone.student_id || '         '
|| someone.name);
    END LOOP;


END SAMPLE;
```

Step 4. Compile and debug the procedure by clicking the star button pointed to by the arrow in the figure below.

```
            student_id_1 NUMBER;
            student_id_2 NUMBER;
            student_name_1 VARCHAR2(30);
            student_name_2 VARCHAR2(30);
            -- nested procedure
            PROCEDURE insert_Student (student_id in NUMBER, student_name in VARCHAR2) AS
            BEGIN
                -- the next two lines are pure SQL statements
                INSERT INTO STUDENT(student_id, name)
                VALUES (student_id, student_name);
            END;
--begin of procedure
BEGIN
            student_id_1 := 100;
            student_name_1 := 'Alice';
            student_id_2 := 200;
            student_name_2 := 'Bob';
            insert_Student(student_id_1, student_name_1);
            insert_Student(student_id_2, student_name_2);
            /*Print the table*/
            DBMS_OUTPUT.PUT_LINE('Student_id' || '   ' || 'Student.name');
            FOR someone IN (select * from student)
            LOOP
                DBMS_OUTPUT.PUT_LINE(someone.student_id || '          ' || someone.name);
```

Messages - Log ×

Compiled

Step 5. Run the procedure by clicking the triangle button pointed to by the arrow below.

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

```
        student_id_1 NUMBER;
        student_id_2 NUMBER;
        student_name_1 VARCHAR2(30);
        student_name_2 VARCHAR2(30);
        -- nested procedure
        PROCEDURE insert_Student (student_id in NUMBER, student_name in VARCHAR2) AS
        BEGIN
            -- the next two lines are pure SQL statements
            INSERT INTO STUDENT(student_id, name)
            VALUES (student_id, student_name);
        END;
--begin of procedure
BEGIN
        student_id_1 := 100;
        student_name_1 := 'Alice';
        student_id_2 := 200;
        student_name_2 := 'Bob';
        insert_Student(student_id_1, student_name_1);
        insert_Student(student_id_2, student_name_2);
        /*Print the table*/
        DBMS_OUTPUT.PUT_LINE('Student_id' || ' ' || 'Student.name');
        FOR someone IN (select * from student)
        LOOP
            DBMS_OUTPUT.PUT_LINE(someone.student_id || '       ' || someone.name);
```

Running: IdeConnections%23zhuang.jpr - Log ×

```
Connecting to the database zhuang.
Student_id  Student.name
100         Alice
200         Bob
Process exited.
Disconnecting from the database zhuang.
```

11. A Complete Example

Write a set of SQL and PL/SQL programs that will help a library to store the information about books and book copies.

A book has an isbn code (primary key) and title, and a book copy has a barcode id (primary key).

E-R diagram
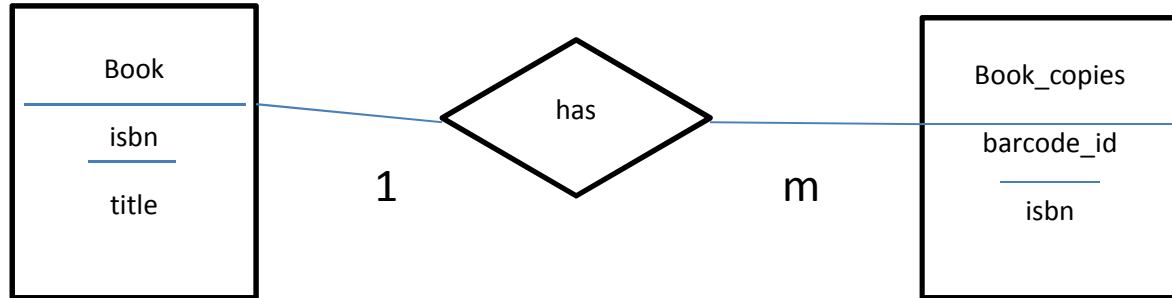
Step 1. Create tables.

Create a file "createTables.sql" with the following code:

```
/*SQL code to create table books and book_copies*/
CREATE TABLE books(
     isbn VARCHAR2(13) NOT NULL PRIMARY KEY,
     title VARCHAR2(200)
);
CREATE TABLE book_copies(
     Barcode_id VARCHAR2(100) NOT NULL PRIMARY KEY,
     isbn VARCHAR2(13),
     FOREIGN KEY (isbn) REFERENCES books (isbn)
);
```

Now assume that we want to write a set of PL/SQL programs to do the following tasks:

- Add book and book copy if barcode_id is provided
- Add a book copy given an isbn of a book and the barcode_id of the copy
- List all books
- Remove a book and all of its copies given the isbn of the book
- Display the title of a book given its isbn

The best way to organize the code is to use packages.

Step 2 Create a package "book".

Step 2.1 Click File -> New -> Package -> OK.
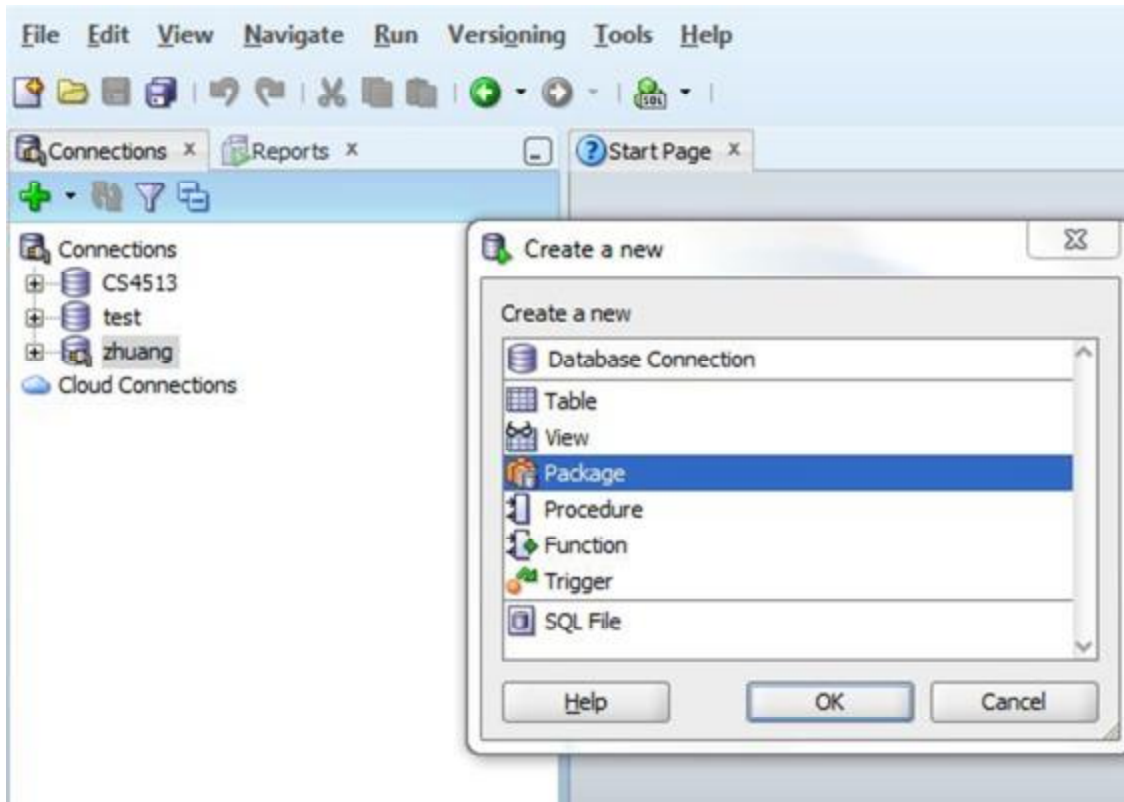
Step 2.2 Select a connection and enter your login name and password if necessary.
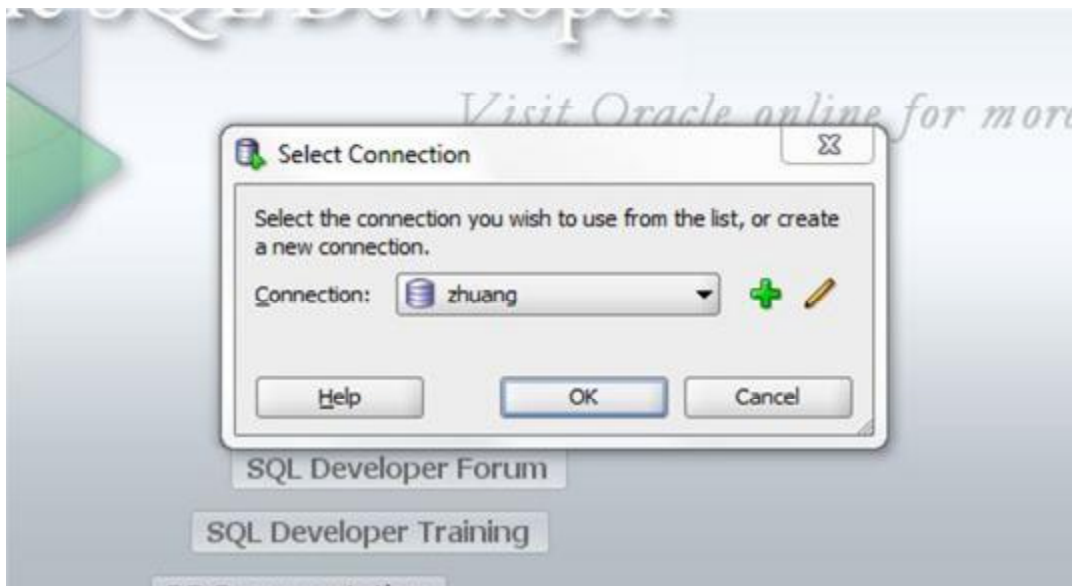
Step 2.3 Give a name for the package and click "OK".
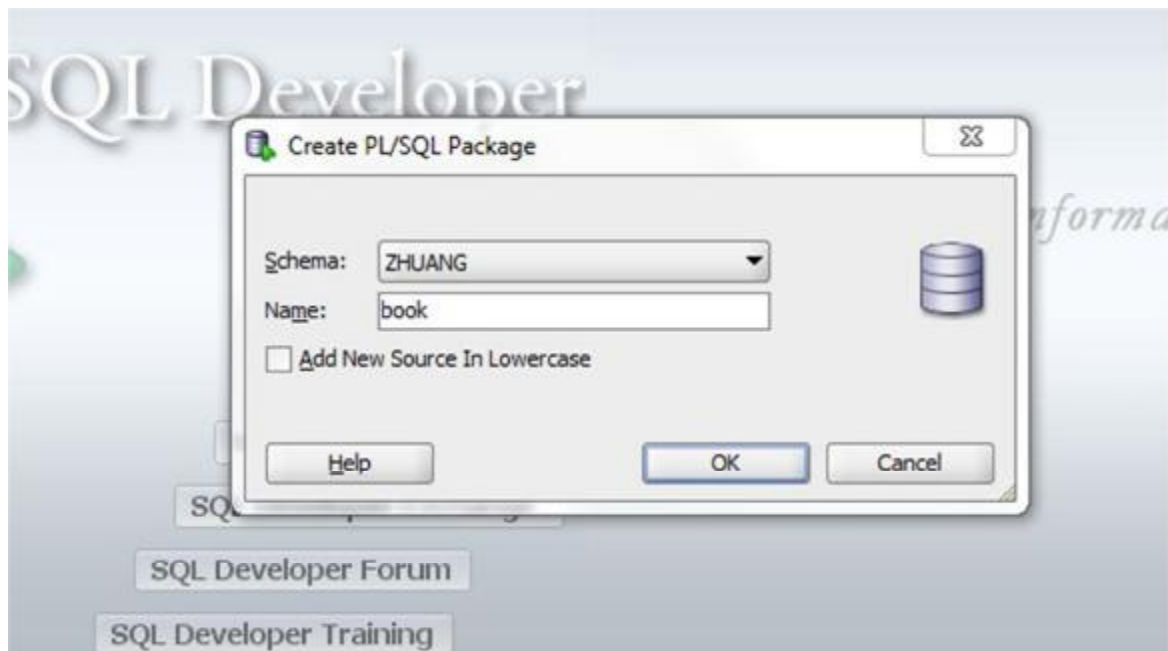
Step 2.4 Once you clicked the "OK" button, a package specification under that name with a null body will be displayed.
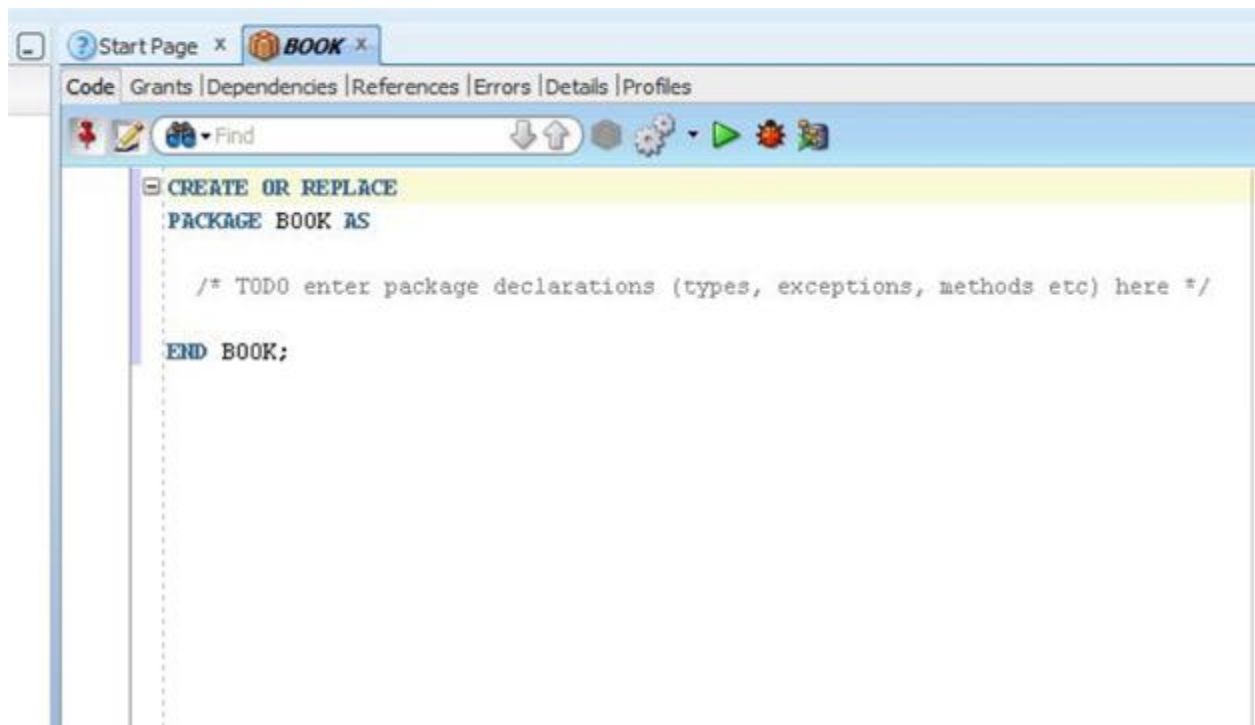
Step 3. Fill the package specification with the code.

```
/*Package specification*/
CREATE OR REPLACE PACKAGE book
AS

    /*procedure to add book and copy if barcode_id_in is
provided*/
    PROCEDURE add(isbn_in IN VARCHAR2, title_in IN VARCHAR2,
    barcode_id_in IN VARCHAR2 DEFAULT NULL);

    /*procedure to add a copy*/
    PROCEDURE add_copy(isbn_in IN VARCHAR2, barcode_id_in IN
VARCHAR2);

    /*procedure to list all the books*/
    PROCEDURE list_books;

    /*procedure to remove a book and all of its copies*/
    PROCEDURE weed(isbn_in IN VARCHAR2);

    /*procedure to display the title of a book given its isbn*/
    PROCEDURE getTitle(isbn_in IN VARCHAR2);

END book;
```

Step 4. Click the "save" button pointed to by the arrow in the figure below to save the specification.
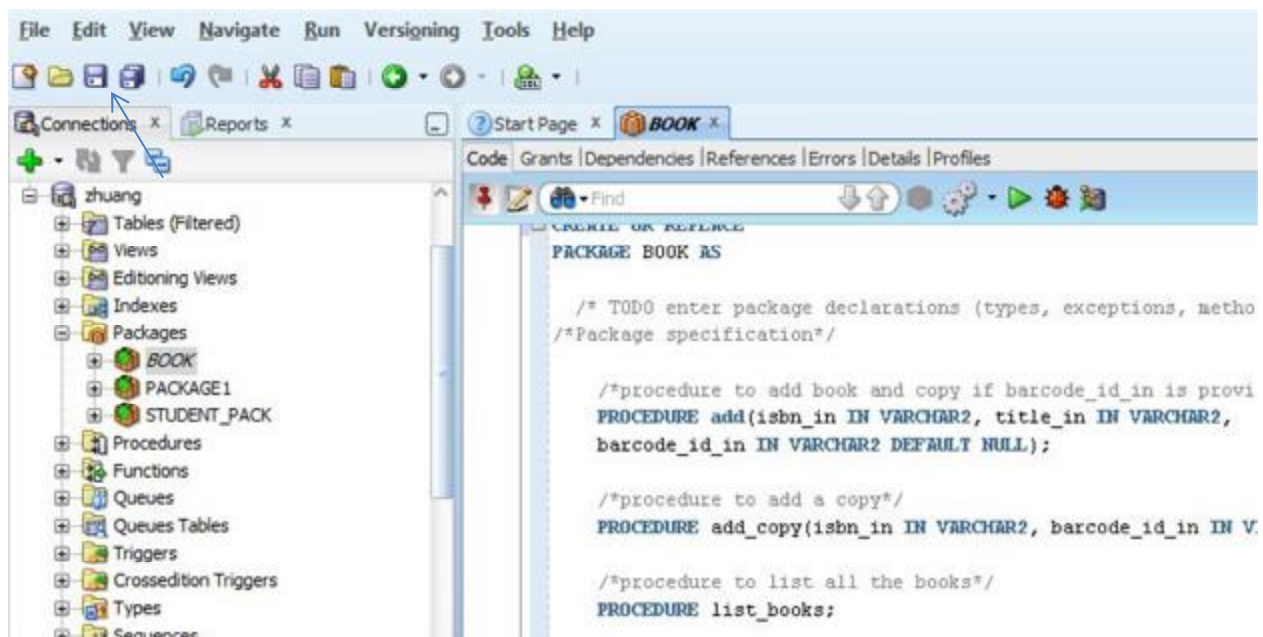
Step 5. Create the body of the package. In the navigator, find the package "book", right click it and select "Create Body".
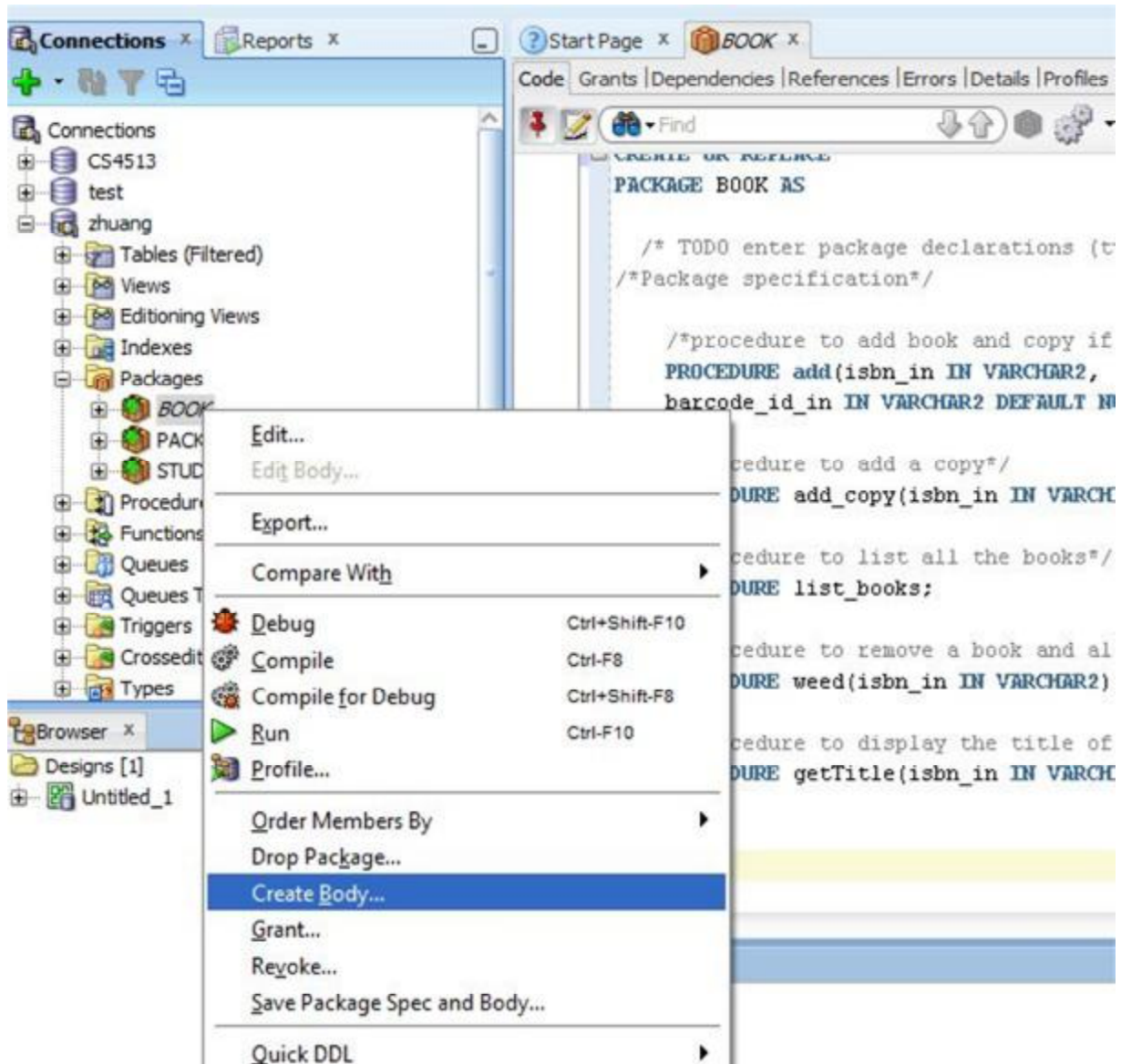
After that, a null body for the package will be displayed.
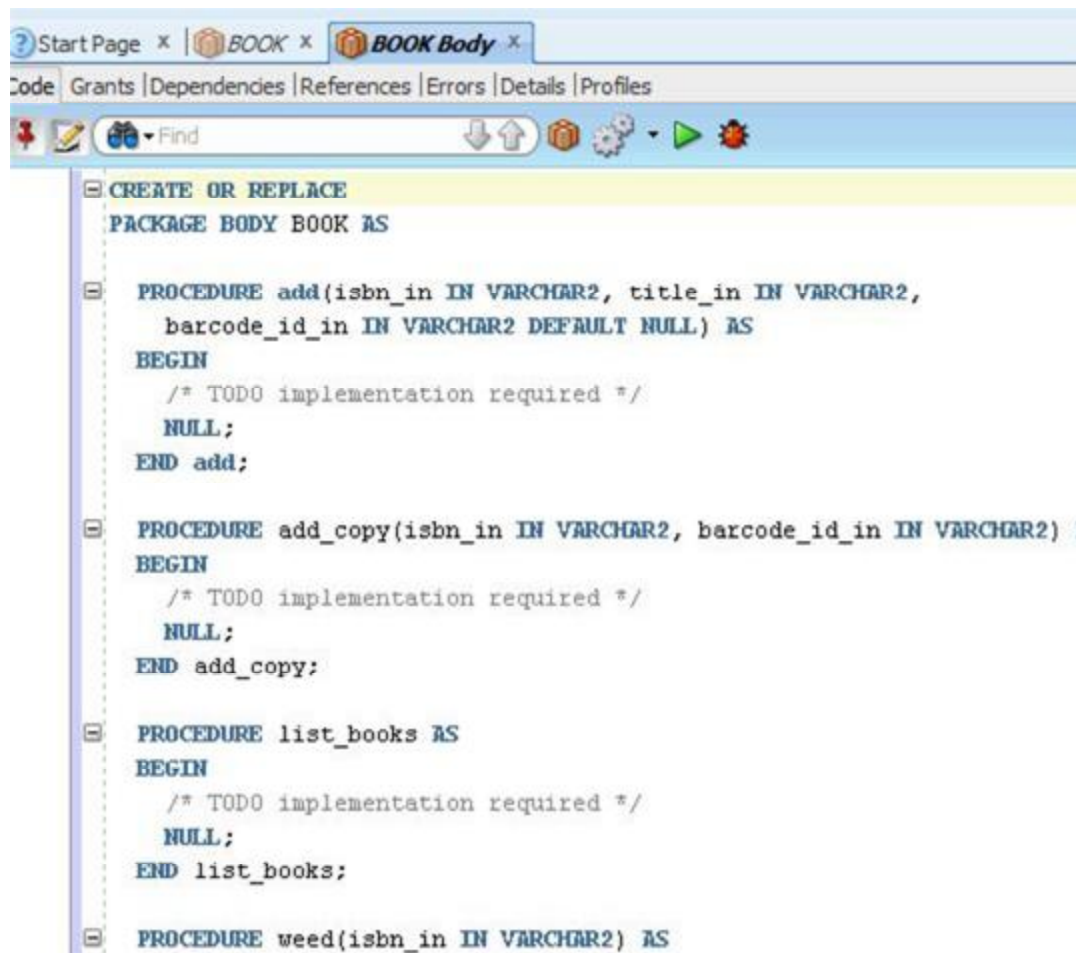
Step 6. Fill the package body with the code.

```
/*Package body*/
CREATE OR REPLACE
PACKAGE BODY BOOK
AS

  /*private procedure to be used in this package body only*/
  PROCEDURE assert_notnull(tested_variable IN VARCHAR2)
  IS
  BEGIN
     IF tested_variable IS NULL
     THEN
         RAISE VALUE_ERROR;
     END IF;
  END assert_notnull;
```

```
  /*procedure to add book and copy if barcode_id_in is
provided*/
  PROCEDURE add(isbn_in IN VARCHAR2, title_in IN VARCHAR2,
  barcode_id_in IN VARCHAR2 DEFAULT NULL)
  AS
  BEGIN
    /* TODO implementation required */
    assert_notnull(isbn_in);

    INSERT INTO books(isbn, title)
    VALUES(isbn_in,title_in);

    IF barcode_id_in IS NOT NULL
    THEN
        add_copy(isbn_in,barcode_id_in);
    END IF;
    END add;

   /*procedure to add a copy*/
  PROCEDURE add_copy(isbn_in IN VARCHAR2, barcode_id_in IN
VARCHAR2)
  AS
  BEGIN
    assert_notnull(isbn_in);
    assert_notnull(barcode_id_in);

    INSERT INTO book_copies(isbn,barcode_id)
    VALUES(isbn_in, barcode_id_in);
  END add_copy;

  /*procedure to list all books*/
  PROCEDURE list_books
  AS
     isbn_out VARCHAR2(50);
     title_out VARCHAR2(50);

     CURSOR lb_cur is
        SELECT isbn, titl
        FROM books;
  BEGIN
    /*open the cursor*/
    OPEN lb_cur;

    /*start the loop*/
    LOOP
        FETCH lb_cur INTO isbn_out, title_out;
    /*exit the loop when there is no more rows*/
```

```
        EXIT WHEN lb_cur%NOTFOUND;

        /*print one row at a time*/
        DBMS_OUTPUT.PUT_LINE(isbn_out||' '||title_out);
        END LOOP;

        /*close the cursor*/
        CLOSE lb_cur;
    END list_books;

    /*procedure to remove a book and all of its copies*/
    PROCEDURE weed(isbn_in IN VARCHAR2) AS
    BEGIN
      assert_notnull(isbn_in);

        /*note the order of delete stataments*/
        DELETE book_copies
        WHERE isbn=isbn_in;
        DELETE books
        WHERE isbn=isbn_in;
    END weed;

    /*procedure to display the title of a book given its isbn*/
    PROCEDURE getTitle(isbn_in IN VARCHAR2) AS
      title_out VARCHAR2(50);

      CURSOR gt_cur IS
         SELECT title
         FROM books
         WHERE isbn=isbn_in;

    BEGIN
      /*open the cursor*/
      OPEN gt_cur;
      /*start the loop*/
      LOOP
        FETCH gt_cur INTO title_out;
        /*exit when there is no more rows*/
        EXIT WHEN gt_cur%NOTFOUND;
        /*print one row at a time*/
        DBMS_OUTPUT.PUT_LINE('The title of a book with isbn=
'||isbn_in||' is '||title_out);
        END LOOP;
        /*close the cursor*/
        CLOSE gt_cur;
    END getTitle;
END BOOK;
```

Step 7. Compile the package body by clicking the button pointed to by the arrow in the figure below.
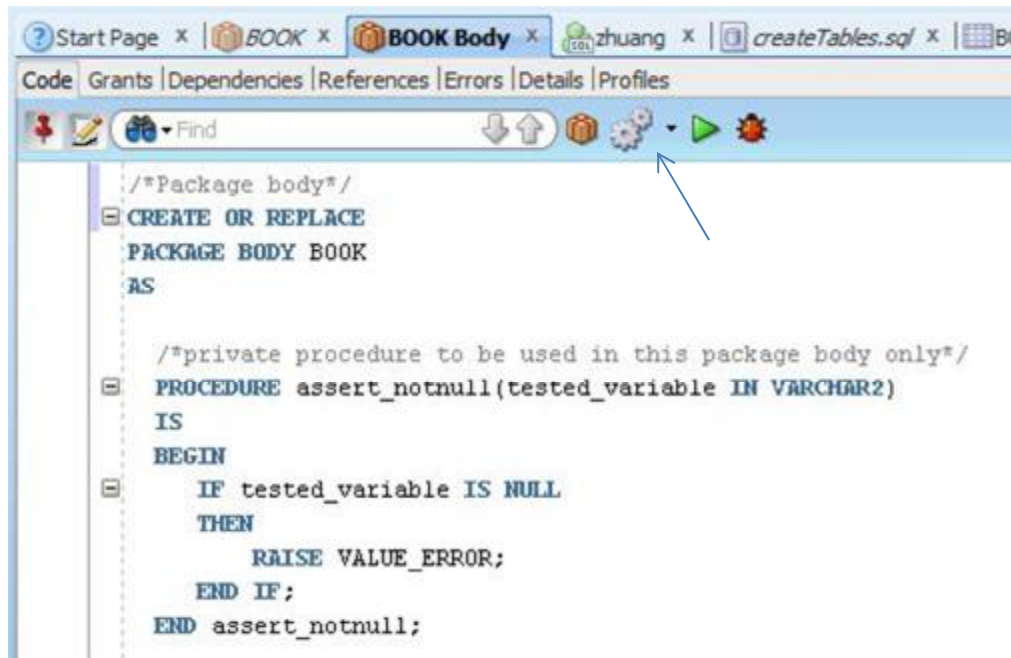
Step 8. The last thing that remains to be done is to write different anonymous blocks that will call the procedures stored in the "book" package. We will invoke the procedure getTitle(isbn_in) directly from the Oracle SQL Developer Worksheet, so there is no need to write an anonymous block that will make the call.

Step 8.1 To add a book and a copy.

Right click the connection name in the navigator and select "Open SQL Worksheet".
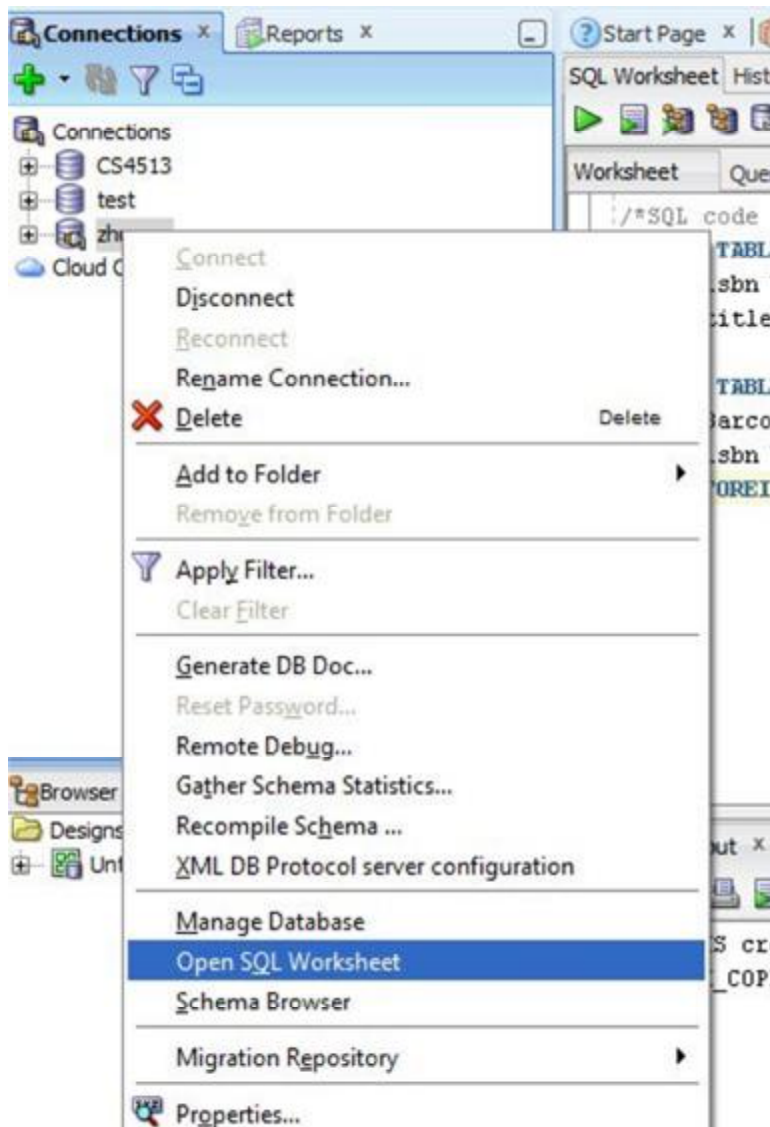
Write the following code in the worksheet:

```
BEGIN
    book.add('&isbn_in','&title_in','&barcode_id_in');
END;
```

Save the code by clicking the SAVE button pointed to by the arrow in the figure below, give it a name "call_add" and select the file type ".pls".
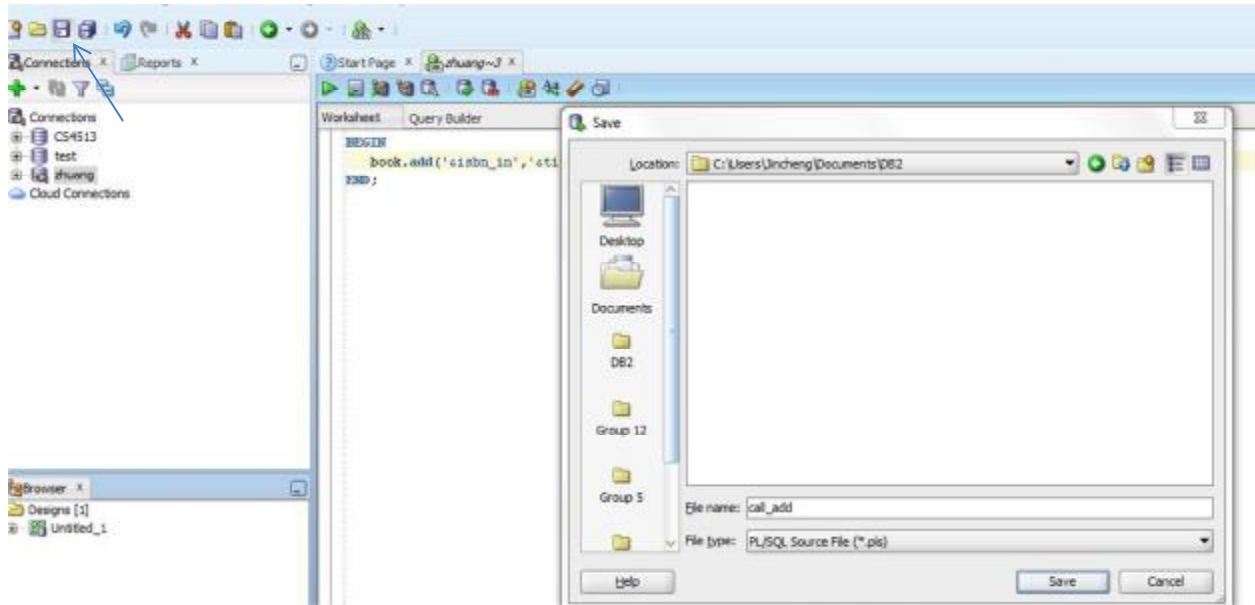
Step 8.2 To add a copy.

Follow the same way as Step 8.1, open another worksheet and write the following code in the worksheet:

```
BEGIN
    book.add_copy('&isbn_in','&barcode_id_in');
END;
```
Save it in the file with the name "call_add_copy" and type ".pls".

Step 8.3 To list all books.

Follow the same way as Step 8.1, open another worksheet and write the following code in the worksheet:

```
BEGIN
    Book.list_books;
END;
```
Save it in the file with the name "call_list_books" and type ".pls".

Step 8.4 To remove a book and all its copies.

Follow the same way as Step 8.1, open another worksheet and write the following code in the worksheet:

```
BEGIN
    book.weed('&isbn_in');
END;
```

Save it in the file with the name "call_weed" and type ".pls".

Step 9. Run the programs.

Step 9.1 Create tables by running the file "createTables.sql".

 Step 9.2 Add a book and a copy. Click File -> Open -> select "call_add.pls". Run the file "call_add.pls" by clicking the "RUN Script" button pointed to by the arrow in the figure below and enter input information.
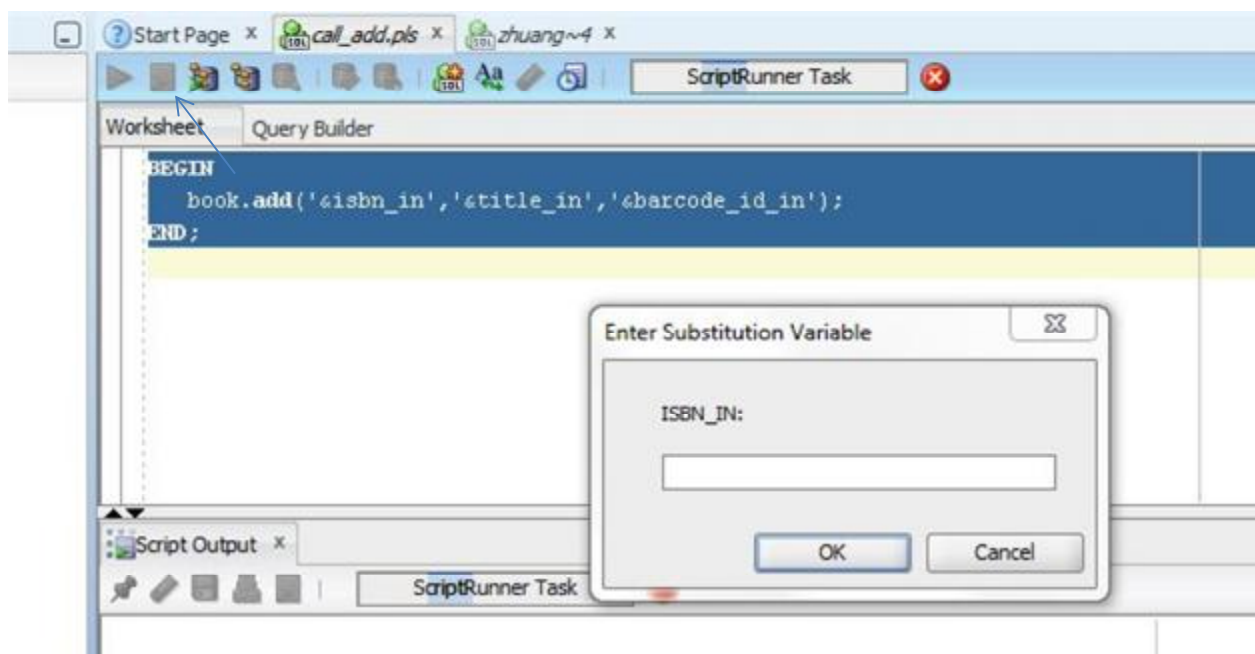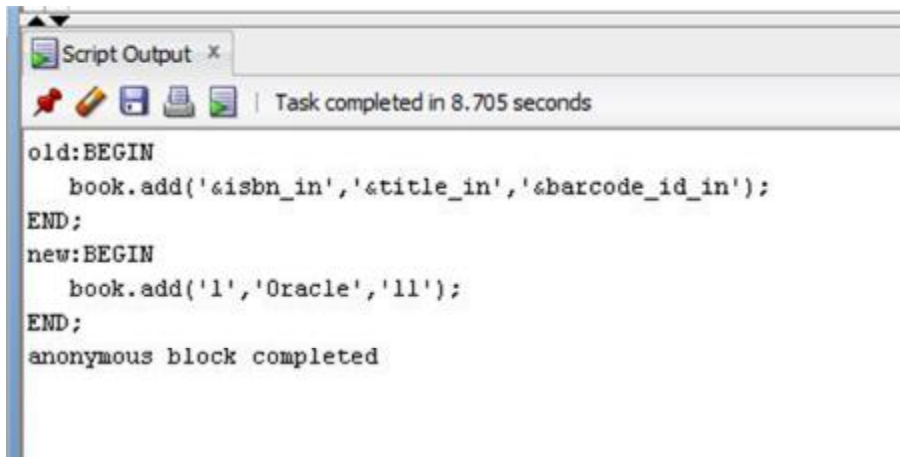
Enter value for ISBN_IN: 1

Enter value for TITLE_IN: Oracle

Enter value for BARCODE_ID_IN: 11

The result is shown in the "Script Output".

```
Script Output ×
📌 ✏ 💾 🖨 📄  |  Task completed in 8.705 seconds
old:BEGIN
    book.add('&isbn_in','&title_in','&barcode_id_in');
END;
new:BEGIN
    book.add('1','Oracle','11');
END;
anonymous block completed
```
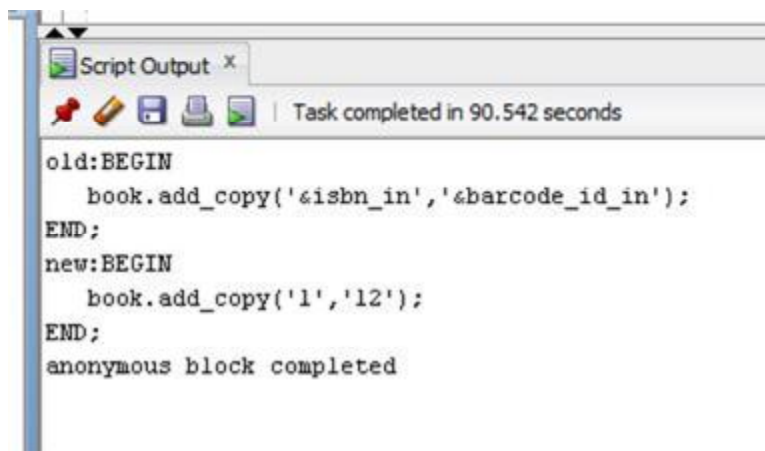
Step 9.3 Add another book and a copy.
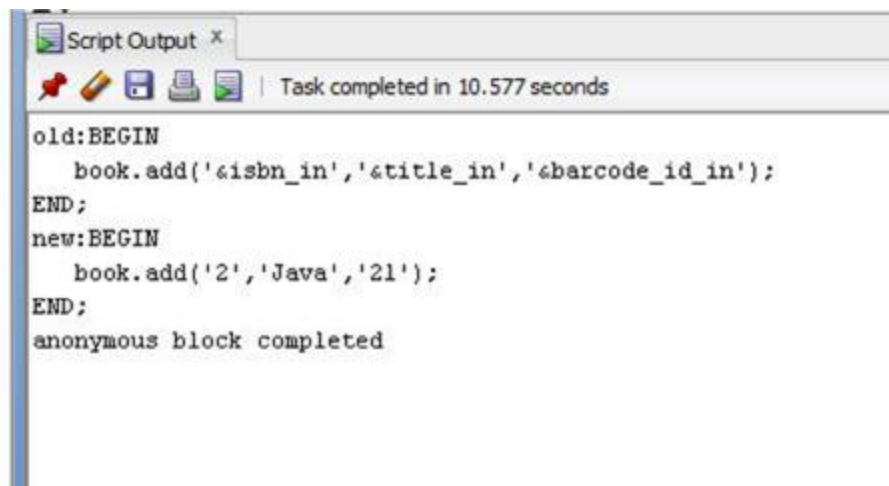
Run the file "call_add.pls" again and enter the following information:

Enter value for ISBN_IN: 2

Enter value for TITLE_IN: Java

Enter value for BARCODE_ID_IN: 21

The result is shown in the "Script Output".


```
Script Output ×
📌 ✏ 💾 🖨 📄  |  Task completed in 10.577 seconds
old:BEGIN
    book.add('&isbn_in','&title_in','&barcode_id_in');
END;
new:BEGIN
    book.add('2','Java','21');
END;
anonymous block completed
```

Step 9.4 Add a copy.

Click File -> Open -> select "call_add_copy.pls". Run the file "call_add_copy.pls" by clicking the "Run Script" button and enter the information.
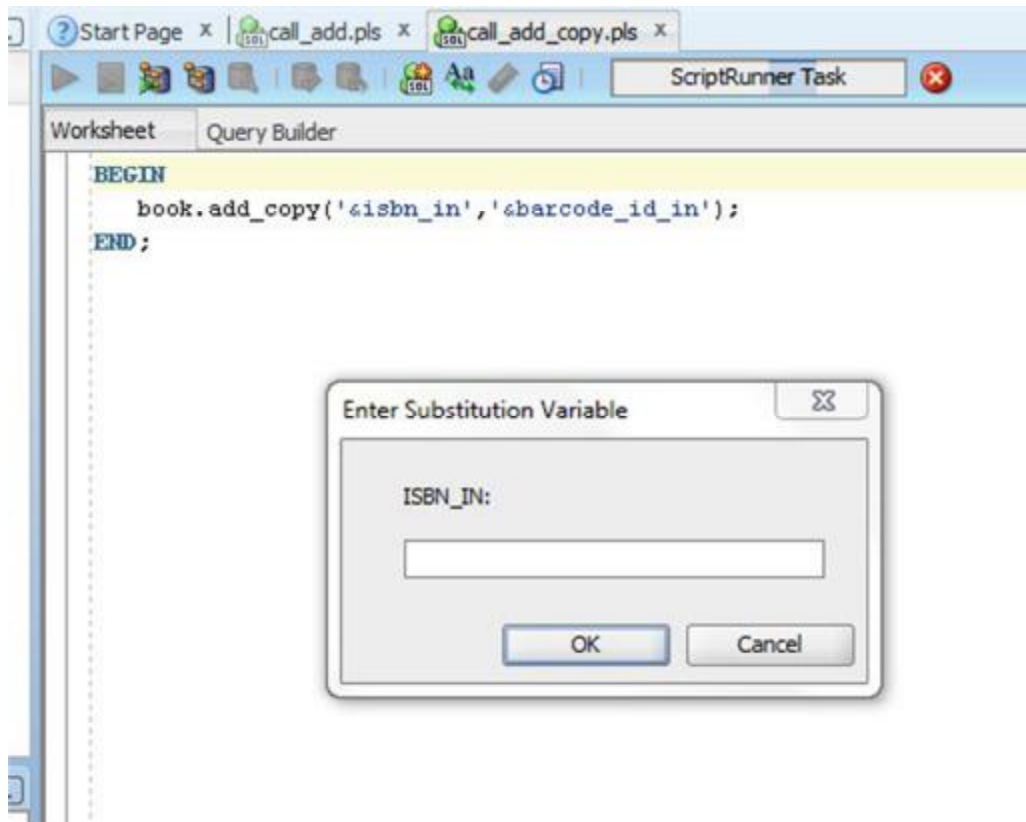
Enter value for isbn_in: 1

Enter value for barcode_id_in: 12
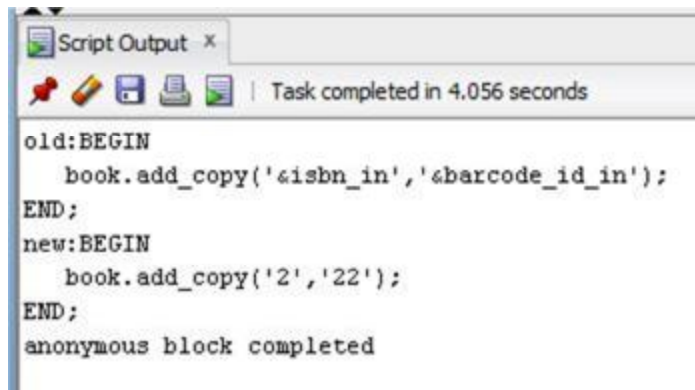
The result is shown in the "Script Output".

Step 9.5 Add another copy.

Run the file "call_add_copy.pls" again and enter the following information:

Enter value for isbn_in: 2

Enter value for barcode_id_in: 22

The result is shown in the "Script Output".

```
Script Output  ×
   📌  ✏  💾  🖨  📋  | Task completed in 4.056 seconds
old:BEGIN
    book.add_copy('&isbn_in','&barcode_id_in');
END;
new:BEGIN
    book.add_copy('2','22');
END;
anonymous block completed
```
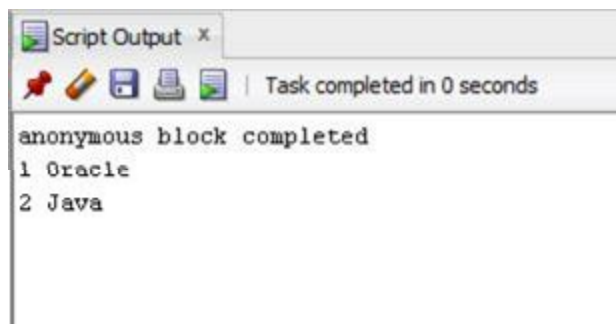
Step 9.6 List the books.

Note that you need to run the following command in the worksheet once per session to display the result on the screen.

```
set serveroutput on;
```

Click File -> Open -> select "call_list_books.pls". Run the file "call_list_books.pls" by clicking the "Run Script" button.

The result is shown in the "Script Output".

```
Script Output  ×
   📌  ✏  💾  🖨  📋  | Task completed in 0 seconds
anonymous block completed
1 Oracle
2 Java
```
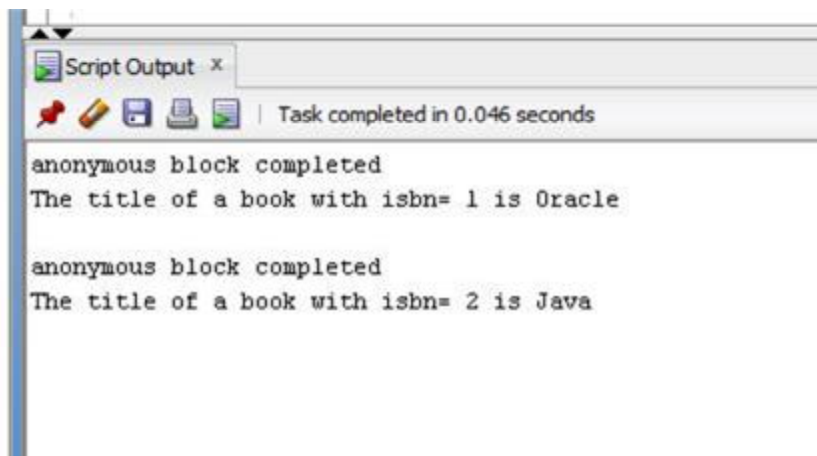
Step 9.7 Call the procedure "book.getTitle" in the worksheet directly.

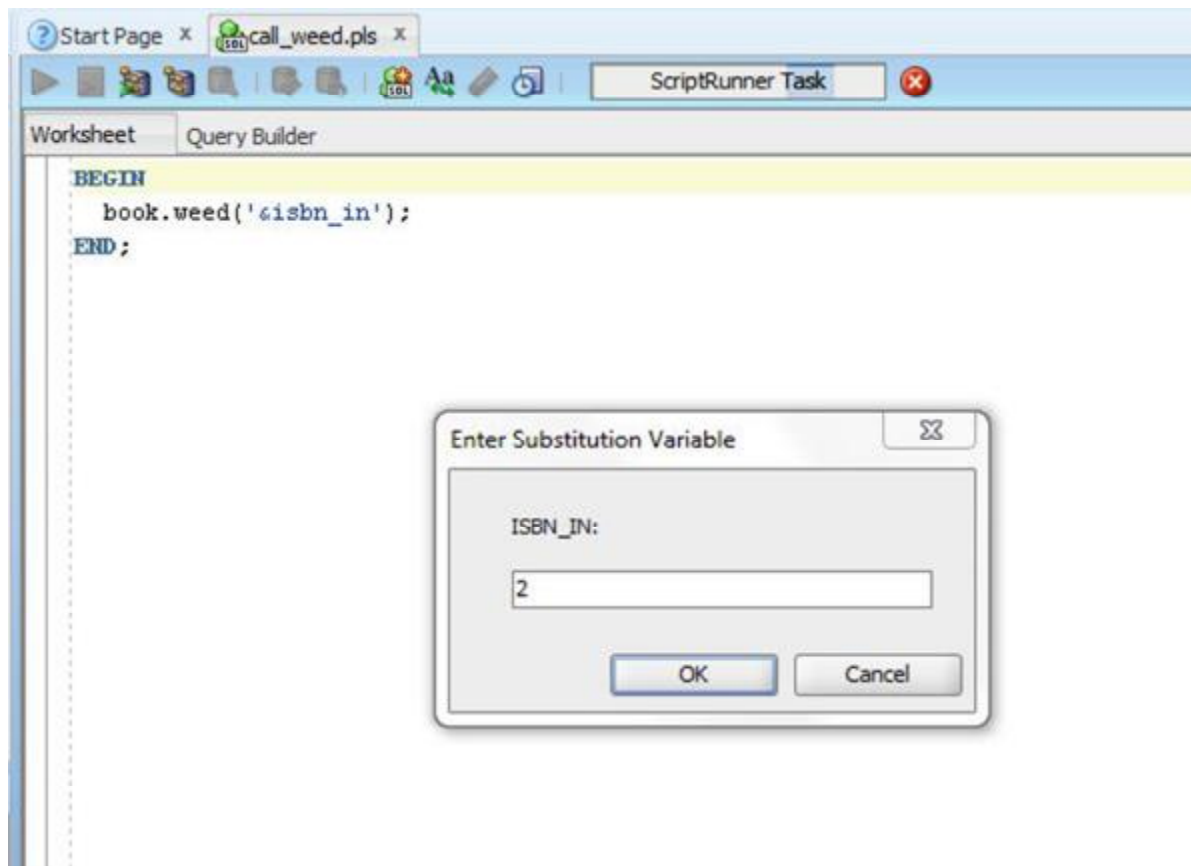Enter the code as follows in the worksheet and click the "Run Script" button.
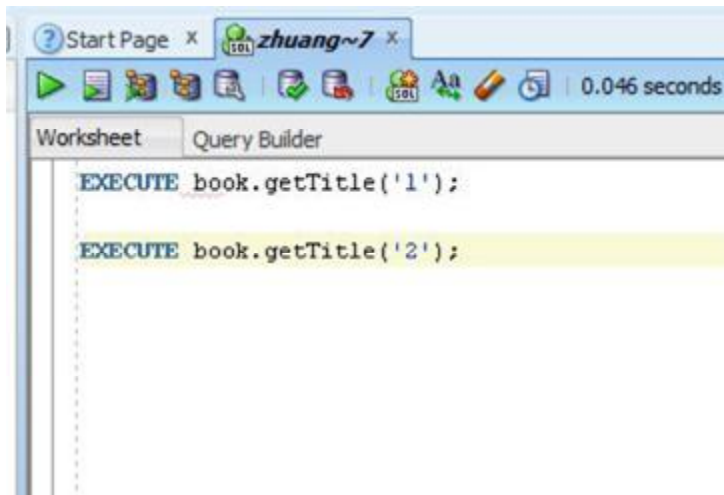
The result is shown in the "Script Output".

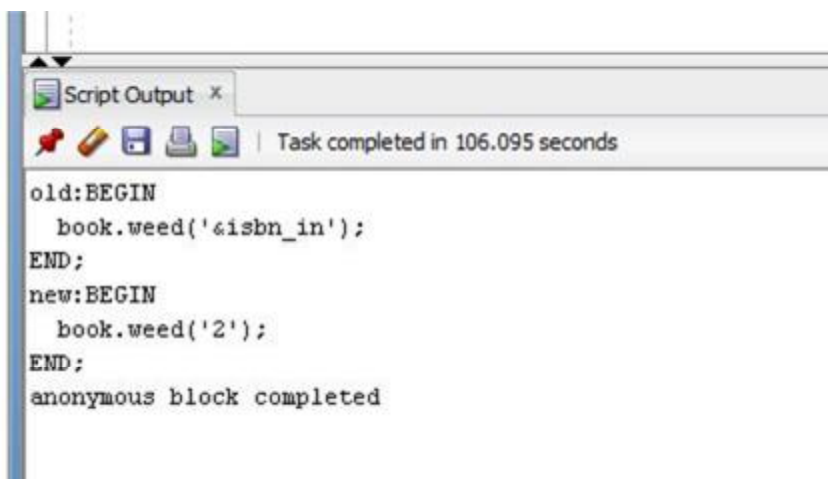Step 9.8 Remove a book and all its copies.

Click File -> Open -> select "call_weed.pls". Run the file "call_weed.pls" by clicking the "Run Script" button.

Enter value for isbn_in: 2
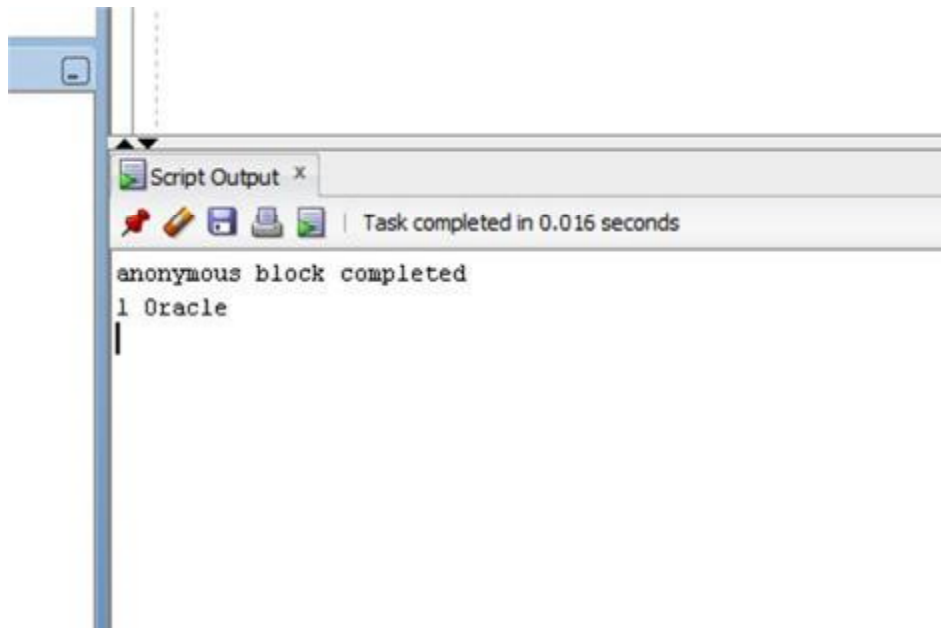
The result is shown in the "Script Output".

Step 9.9 List all the books.

Click File -> Open -> select "call_list_books.pls". Run the file "call_list_books.pls" by clicking the "Run Script" button.

The result is shown in the "Script Output".


```
Script Output ×
📌 🖊 💾 🖨 ▶  | Task completed in 0.016 seconds
anonymous block completed
1 Oracle
```

Note that there is only one book now.

12. How to call a PL/SQL procedure in a Java program

Please refer to the following example to call a PL/SQL procedure in a Java program.

Step 1. Create the table "REGION" with two attributes and insert some data.

```
// create table "REGION"
CREATE  TABLE  REGION
(   R_REGIONKEY NUMBER(2,0) NOT NULL PRIMARY KEY,
    R_NAME VARCHAR2 (25) NOT NULL
) ;
INSERT  INTO REGION VALUES (1, 'Alice');
INSERT  INTO REGION VALUES (2, 'Bob');
```

Step 2. Create a PL/SQL procedure named "find_region" in Oracle SQL Developer that returns the region's name ("regionname") corresponding to the input of the region's key ("regionkey").

The procedure has two parameters: the first one is input parameter and the second one is output parameter. Note that when you create the procedure "find_region" you need to specify the parameters (name, type and mode) by clicking the "+" symbol pointed to by the arrow in the

figure below. In this example, the mode of the parameter "regionkey" is "IN" and the mode of the parameter "regionname" is "OUT".
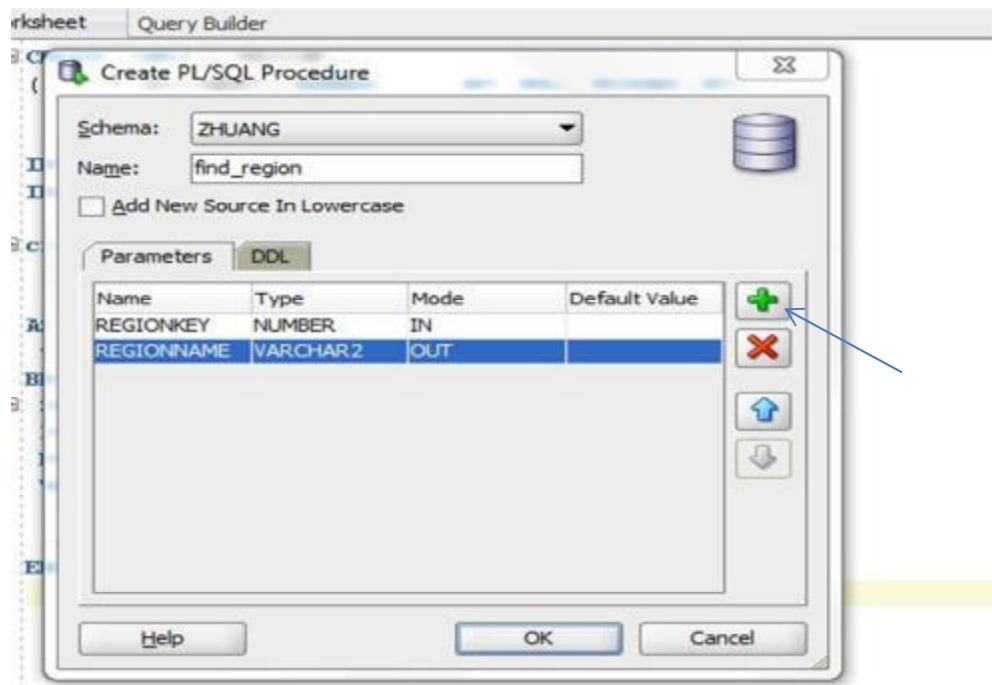
The content of the procedure is as follows:

```
// create the procedure "find_region"
CREATE OR REPLACE PROCEDURE FIND_REGION
(
  REGIONKEY IN NUMBER
, REGIONNAME OUT VARCHAR2
)
AS
  v_name VARCHAR2(25);
BEGIN
 SELECT r_name
 INTO v_name
 FROM REGION
 WHERE r_regionkey = regionkey;

 regionname := v_name;

END FIND_REGION;
```

Step 3. Write the Java program to call the procedure "find_region".

An example of a Java program using JDBC connection to call a procedure in Oracle 11g

```java
// a sample example of using java to call PL/SQL procedure find_region
// return the value of "r_regionname" corresponding to r_regionkey=1
package callProc;
import java.io.*;// java package for i/o
import java.sql.*;// java package for sql statements
class callProc {
        public static void main(String[] args)
        throws SQLException, IOException
        {

          // Step 1. Load a database driver
           try {
                          Class.forName("oracle.jdbc.OracleDriver");
                    }
            catch(Exception x){
                          System.out.println( "Unable to load the driver class!" );
                    }


     // Step 2. Creating an Oracle JDBC Connection.  The following example assumes
      // that the login name is smith1234 and the password is johnsmith

   Connection conn = DriverManager.getConnection
   ("jdbc:oracle:thin:@stu-oracle.cs.ou.edu:1521:cs4513","smith1234", "johnsmith");

       // Step 3. Call the procedure "find_region"
      try {

              int i_rkey = 1;//input parameter of the procedure
              //call procedure FIND_REGION(pram1,pram2)
              CallableStatement pstmt = conn.prepareCall("{call
find_region(?,?)}");//call the procedure

              pstmt.setInt(1, i_rkey);/*the first parameter (input parameter) to
procedure find_region*/
              pstmt.registerOutParameter(2, Types.CHAR);/*the second parameter (output
parameter) from procedure find_region*/

              pstmt.executeUpdate();//execute procedure
              String o_rname = pstmt.getString(2);
              System.out.println("region " + i_rkey + " is: " + o_rname);/*print out
the output*/

              pstmt.close();//close the pstmt


              }
      catch (SQLException e) {
              System.err.println(e.getMessage());

              }
```
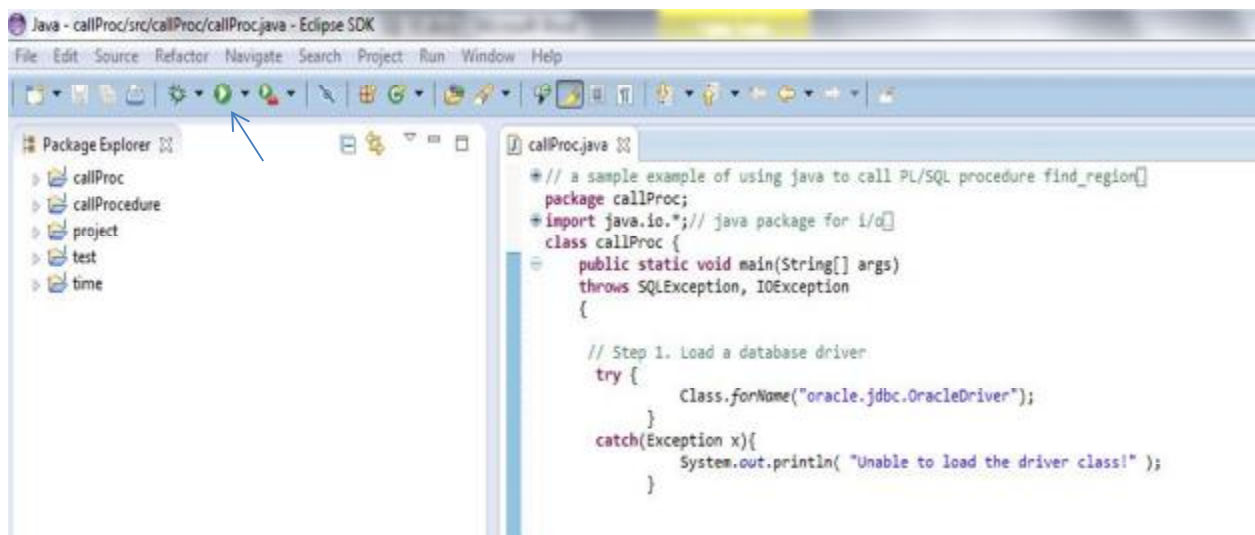
```
    conn.close();
        }
}
```

Step 3. To compile and run the program in *eclipse*, press the RUN button pointed to by the arrow in the figure below.

References

1. Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)
   http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/toc.htm
2. "Oracle PL/SQL by Example" (fourth edition), Benjamin Rosenzweig and Elena
   Silvestrova Rakhimov, 2009 published by Addison-Wesley.
3. Developing and Debugging PL/SQL using Oracle SQL Developer
   http://stcurriculum.oracle.com/obe/db/11g/r2/prod/appdev/sqldev/plsql_debug/plsql_debu
   g_otn.htm
4. Working with PL/SQL by Sue Harper
   http://www.oracle.com/technetwork/issue-archive/2007/07-nov/o67sql-101793.html