APPENDIX D

Answers to the Try It Yourself Sections

Chapter 1, "PL/SQL Concepts"

1) To calculate the area of a circle, you must square the circle's radius and then multiply it by π. Write a program that calculates the area of a circle. The value for the radius should be provided with the help of a substitution variable. Use 3.14 for the value of π. After the area of the circle is calculated, display it on the screen.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   v_radius NUMBER := &sv_radius;
   v_area NUMBER;
BEGIN
   v_area := POWER(v_radius, 2) * 3.14;
   DBMS_OUTPUT.PUT_LINE
      ('The area of the circle is: '||v_area);
END;
```

In this exercise, you declare two variables, `v_radius` and `v_area`, to store the values for the radius of the circle and its area, respectively. Next, you compute the value for the variable `v_area` with the help of the built-in function POWER and the value of the `v_radius`. Finally, you display the value of `v_area` on the screen.

Assume that the number 5 has been entered for the value of the variable `v_radius`. The script produces the following output:

```
Enter value for sv_radius: 5
old   2:    v_radius NUMBER := &sv_radius;
new   2:    v_radius NUMBER := 5;
The area of the circle is: 78.5

PLSQL procedure successfully completed.
```

2) Rewrite the script ch01_2b.sql, version 2.0. In the output produced by the script, extra spaces appear after the day of the week. The new script should remove these extra spaces.

Here's the current output:

```
Today is Sunday   , 20:39
```

The new output should have this format:

```
Today is Sunday, 20:39
```

ANSWER: The new version of the script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_day VARCHAR2(20);
BEGIN
   v_day := TO_CHAR(SYSDATE, 'fmDay, HH24:MI');
   DBMS_OUTPUT.PUT_LINE ('Today is '|| v_day);
END;
```

In this script, you modify the format in which you would like to display the date. Notice that the word `Day` is now prefixed by the letters `fm`. These letters guarantee that extra spaces will be removed from the name of the day. When run, this exercise produces the following output:

```
Today is Tuesday, 18:54

PLSQL procedure successfully completed.
```

Chapter 2, "General Programming Language Fundamentals"

1) Write a PL/SQL block
   - A) That includes declarations for the following variables:
     - I) A VARCHAR2 datatype that can contain the string 'Introduction to Oracle PL/SQL'
     - II) A NUMBER that can be assigned 987654.55, but not 987654.567 or 9876543.55
     - III) A CONSTANT (you choose the correct datatype) that is autoinitialized to the value '603D'
     - IV) A BOOLEAN
     - V) A DATE datatype autoinitialized to one week from today
   - B) In the body of the PL/SQL block, put a DBMS_OUTPUT.PUT_LINE message for each of the variables that received an auto initialization value.
   - C) In a comment at the bottom of the PL/SQL block, state the value of your number datatype.

ANSWER: The answer should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   -- A VARCHAR2 datatype that can contain the string
   -- 'Introduction to Oracle PL/SQL'
   v_descript VARCHAR2(35);

   - A NUMBER that allows for the conditions: can be
   - assigned 987654.55 but not 987654.567 or 9876543.55

   v_number_test NUMBER(8,2);

   -- [a variable] autoinitialized to the value '603D'
   v_location CONSTANT VARCHAR2(4) := '603D';
```

```
   -- A BOOLEAN
   v_boolean_test BOOLEAN;

   -- A DATE datatype auto initialized to one week from today

   v_start_date DATE := TRUNC(SYSDATE) + 7;

BEGIN
   DBMS_OUTPUT.PUT_LINE
      ('The location is: '||v_location||'.');
   DBMS_OUTPUT.PUT_LINE
      ('The starting date is: '||v_start_date||'.');
END;
```

2) Alter the PL/SQL block you just created to conform to the following specifications.
   - A) Remove the DBMS_OUTPUT.PUT_LINE messages.
   - B) In the body of the PL/SQL block, write a selection test (IF) that does the following (use a nested if statement where appropriate):
     - I) Checks whether the VARCHAR2 you created contains the course named "Introduction to Underwater Basketweaving."
     - II) If it does, put a DBMS_OUTPUT.PUT_LINE message on the screen that says so.
     - III) If it does not, test to see if the CONSTANT you created contains the room number 603D.
     - IV) If it does, put a DBMS_OUTPUT.PUT_LINE message on the screen that states the course name and the room number that you've reached in this logic.
     - V) If it does not, put a DBMS_OUTPUT.PUT_LINE message on the screen that states that the course and location could not be determined.
   - C) Add a WHEN OTHERS EXCEPTION that puts a DBMS_OUTPUT.PUT_LINE message on the screen that says that an error occurred.

   ANSWER: The answer should look similar to the following:

```
SET SERVEROUT ON
DECLARE
   -- A VARCHAR2 datatype that can contain the string
   -- 'Introduction to Oracle PL/SQL'
   v_descript VARCHAR2(35);

   -- A NUMBER that allows for the conditions: can be
   - assigned 987654.55 but not 987654.567 or 9876543.55

   v_number_test NUMBER(8,2);

   -- [a variable] auto initialized to the value '603D'
   v_location CONSTANT VARCHAR2(4) := '603D';

   -- A BOOLEAN
   v_boolean_test BOOLEAN;

   -- A DATE datatype autoinitialized to one week from today
   v_start_date DATE := TRUNC(SYSDATE) + 7;
```

```
BEGIN
   IF v_descript = 'Introduction to Underwater Basketweaving'
   THEN
      DBMS_OUTPUT.PUT_LINE ('This course is '||v_descript||'.');

   ELSIF v_location = '603D' THEN

      -- No value has been assigned to v_descript
      IF v_descript IS NOT NULL THEN
         DBMS_OUTPUT.PUT_LINE ('The course is '||v_descript
            ||'.'||' The location is '||v_location||'.');
      ELSE
         DBMS_OUTPUT.PUT_LINE ('The course is unknown.'||
             ' The location is '||v_location||'.');
      END IF;
   ELSE
      DBMS_OUTPUT.PUT_LINE ('The course and location '||
          'could not be determined.');
   END IF;
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ('An error occurred.');
END;
```

Chapter 3, "SQL in PL/SQL"

1) Create a table called CHAP4 with two columns; one is ID (a number) and the other is NAME, which is a VARCHAR2(20).

   ANSWER: The answer should look similar to the following:

```
PROMPT Creating Table 'CHAP4'
CREATE TABLE chap4
   (id   NUMBER,
    name VARCHAR2(20));
```

2) Create a sequence called CHAP4_SEQ that increments by units of 5.

   ANSWER: The answer should look similar to the following:

```
PROMPT Creating Sequence 'CHAP4_SEQ'
CREATE SEQUENCE chap4_seq
   NOMAXVALUE
   NOMINVALUE
   NOCYCLE
   NOCACHE;
```

3) Write a PL/SQL block that does the following, in this order:

   A) Declares two variables: one for the `v_name` and one for `v_id`. The `v_name` variable can be used throughout the block to hold the name that will be inserted. Realize that the value will change in the course of the block.

   B) The block inserts into the table the name of the student who is enrolled in the most classes and uses a sequence for the ID. Afterward there is SAVEPOINT A.

C) The student with the fewest classes is inserted. Afterward there is SAVEPOINT B.

D) The instructor who is teaching the most courses is inserted in the same way. Afterward there is SAVEPOINT C.

E) Using a SELECT INTO statement, hold the value of the instructor in the variable `v_id`.

F) Undo the instructor insertion by using rollback.

G) Insert the instructor teaching the fewest courses, but do not use the sequence to generate the ID. Instead, use the value from the first instructor, whom you have since undone.

H) Insert the instructor teaching the most courses, and use the sequence to populate his or her ID.

Add DBMS_OUTPUT throughout the block to display the values of the variables as they change. (This is a good practice for debugging.)

ANSWER: The script should look similar to the following:

```
DECLARE
    v_name student.last_name%TYPE;
    v_id   student.student_id%TYPE;
BEGIN
   BEGIN
      -- A second block is used to capture the possibility of
      -- multiple students meeting this requirement.
      -- The exception section handles this situation.
      SELECT s.last_name
        INTO v_name
        FROM student s, enrollment e
       WHERE s.student_id = e.student_id
      HAVING COUNT(*) = (SELECT MAX(COUNT(*))
                           FROM student s, enrollment e
                          WHERE s.student_id = e.student_id
                         GROUP BY s.student_id)
      GROUP BY s.last_name;
   EXCEPTION
      WHEN TOO_MANY_ROWS THEN
         v_name := 'Multiple Names';
   END;

   INSERT INTO CHAP4
   VALUES (CHAP4_SEQ.NEXTVAL, v_name);
   SAVEPOINT A;

   BEGIN
      SELECT s.last_name
        INTO v_name
        FROM student s, enrollment e
       WHERE s.student_id = e.student_id
      HAVING COUNT(*) = (SELECT MIN(COUNT(*))
                           FROM student s, enrollment e
                          WHERE s.student_id = e.student_id
                         GROUP BY s.student_id)
      GROUP BY s.last_name;
```

```
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      v_name := 'Multiple Names';
END;

INSERT INTO CHAP4
VALUES (CHAP4_SEQ.NEXTVAL, v_name);
SAVEPOINT B;

BEGIN
   SELECT i.last_name
     INTO v_name
     FROM instructor i, section s
    WHERE s.instructor_id = i.instructor_id
   HAVING COUNT(*) = (SELECT MAX(COUNT(*))
                        FROM instructor i, section s
                       WHERE s.instructor_id = i.instructor_id
                      GROUP BY i.instructor_id)
   GROUP BY i.last_name;
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      v_name := 'Multiple Names';
END;

SAVEPOINT C;

BEGIN
   SELECT instructor_id
     INTO v_id
     FROM instructor
    WHERE last_name = v_name;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      v_id := 999;
END;

INSERT INTO CHAP4
VALUES (v_id, v_name);
ROLLBACK TO SAVEPOINT B;

BEGIN
   SELECT i.last_name
     INTO v_name
     FROM instructor i, section s
    WHERE s.instructor_id = i.instructor_id
   HAVING COUNT(*) = (SELECT MIN(COUNT(*))
                        FROM instructor i, section s
                       WHERE s.instructor_id = i.instructor_id
                      GROUP BY i.instructor_id)
   GROUP BY i.last_name;
```

```
  EXCEPTION
    WHEN TOO_MANY_ROWS THEN
       v_name := 'Multiple Names';
  END;

  INSERT INTO CHAP4
  VALUES (v_id, v_name);

  BEGIN
     SELECT i.last_name
       INTO v_name
       FROM instructor i, section s
      WHERE s.instructor_id = i.instructor_id
     HAVING COUNT(*) = (SELECT MAX(COUNT(*))
                          FROM instructor i, section s
                         WHERE s.instructor_id = i.instructor_id
                        GROUP BY i.instructor_id)
     GROUP BY i.last_name;
  EXCEPTION
     WHEN TOO_MANY_ROWS THEN
        v_name := 'Multiple Names';
  END;

  INSERT INTO CHAP4
  VALUES (CHAP4_SEQ.NEXTVAL, v_name);
END;
```

Chapter 4, "Conditional Control: IF Statements"

1) Rewrite ch04_1a.sql. Instead of getting information from the user for the variable `v_date`, define its value with the help of the function SYSDATE. After it has been determined that a certain day falls on the weekend, check to see if the time is before or after noon. Display the time of day together with the day.

   ANSWER: The script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_day  VARCHAR2(15);
   v_time VARCHAR(8);
BEGIN
   v_day  := TO_CHAR(SYSDATE, 'fmDAY');
   v_time := TO_CHAR(SYSDATE, 'HH24:MI');

   IF v_day IN ('SATURDAY', 'SUNDAY') THEN
      DBMS_OUTPUT.PUT_LINE (v_day||', '||v_time);
      IF v_time BETWEEN '12:01' AND '24:00' THEN
         DBMS_OUTPUT.PUT_LINE ('It''s afternoon');
      ELSE
         DBMS_OUTPUT.PUT_LINE ('It''s morning');
      END IF;
```

```
   END IF;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE('Done...');
END;
```

In this exercise, you remove the variable `v_date` that was used to store the date provided by the user. You add the variable `v_time` to store the time of the day. You also modify the statement

```
v_day := TO_CHAR(SYSDATE, 'fmDAY');
```

so that `DAY` is prefixed by the letters `fm`. This guarantees that extra spaces will be removed from the name of the day. Then you add another statement that determines the current time of day and stores it in the variable `v_time`. Finally, you add an IF-THEN-ELSE statement that checks the time of day and displays the appropriate message.

Notice that two consecutive single quotes are used in the second and third DBMS_OUTPUT.PUT_LINE statements. This allows you to use an apostrophe in your message.

When run, this exercise produces the following output:

```
SUNDAY, 16:19
It's afternoon
Done...

PLSQL procedure successfully completed.
```

2) Create a new script. For a given instructor, determine how many sections he or she is teaching. If the number is greater than or equal to 3, display a message saying that the instructor needs a vacation. Otherwise, display a message saying how many sections this instructor is teaching.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   v_instructor_id NUMBER := &sv_instructor_id;
   v_total NUMBER;
BEGIN
   SELECT COUNT(*)
     INTO v_total
     FROM section
    WHERE instructor_id = v_instructor_id;

   -- check if instructor teaches 3 or more sections
   IF v_total >= 3 THEN
      DBMS_OUTPUT.PUT_LINE ('This instructor needs '||
         'a vacation');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('This instructor teaches '||
         v_total||' sections');
   END IF;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

This script accepts a value for the instructor's ID from a user. Next, it checks the number of sections taught by the given instructor. This is accomplished with the help of the SELECT INTO

statement. Next, it determines what message should be displayed on the screen with the help of the IF-THEN-ELSE statement. If a particular instructor teaches three or more sections, the condition of the IF-THEN-ELSE statement evaluates to TRUE, and the message `This instructor needs a vacation` is displayed to the user. In the opposite case, the message stating how many sections an instructor is teaching is displayed. Assume that value 101 was provided at runtime. Then the script produces the following output:

```
Enter value for sv_instructor_id: 101
old   2:    v_instructor_id NUMBER := &sv_instructor_id;
new   2:    v_instructor_id NUMBER := 101;
This instructor needs a vacation

PLSQL procedure successfully completed.
```

3) Execute the following two PL/SQL blocks, and explain why they produce different output for the same value of the variable `v_num`. Remember to issue the SET SERVEROUTPUT ON command before running this script.

```
-- Block 1
DECLARE
   v_num NUMBER := NULL;
BEGIN
   IF v_num > 0 THEN
      DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('v_num is not greater than 0');
   END IF;
END;

-- Block 2
DECLARE
   v_num NUMBER := NULL;
BEGIN
   IF v_num > 0 THEN
      DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');
   END IF;
   IF NOT (v_num > 0) THEN
      DBMS_OUTPUT.PUT_LINE ('v_num is not greater than 0');
   END IF;
END;
```

ANSWER: Consider the output produced by the preceding scripts:

```
-- Block1
v_num is not greater than 0

PLSQL procedure successfully completed.

-- Block 2
PLSQL procedure successfully completed.
```

The output produced by Block 1 and Block 2 is different, even though in both examples variable `v_num` is defined as NULL.

First, take a closer look at the IF-THEN-ELSE statement used in Block 1:

```
IF v_num > 0 THEN
   DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');
ELSE
   DBMS_OUTPUT.PUT_LINE ('v_num is not greater than 0');
END IF;
```

The condition `v_num > 0` evaluates to FALSE because NULL has been assigned to the variable `v_num`. As a result, control is transferred to the ELSE part of the IF-THEN-ELSE statement. So the message `v_num is not greater than 0` is displayed on the screen.

Second, take a closer look at the IF-THEN statements used in Block 2:

```
IF v_num > 0 THEN
   DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');
END IF;
IF NOT (v_num > 0) THEN
   DBMS_OUTPUT.PUT_LINE ('v_num is not greater than 0');
END IF;
```

The conditions of both IF-THEN statements evaluate to FALSE. As a result, neither message is displayed on the screen.

Chapter 5, "Conditional Control: CASE Statements"

1) Create the following script. Modify the script you created in Chapter 4, project 1 of the "Try It Yourself" section. You can use either the CASE statement or the searched CASE statement. The output should look similar to the output produced by the example you created in Chapter 4.

ANSWER: Consider the script you created in Chapter 4:

```
SET SERVEROUTPUT ON
DECLARE
   v_day  VARCHAR2(15);
   v_time VARCHAR(8);
BEGIN
   v_day  := TO_CHAR(SYSDATE, 'fmDAY');
   v_time := TO_CHAR(SYSDATE, 'HH24:MI');

   IF v_day IN ('SATURDAY', 'SUNDAY') THEN
      DBMS_OUTPUT.PUT_LINE (v_day||', '||v_time);

      IF v_time BETWEEN '12:01' AND '24:00' THEN
         DBMS_OUTPUT.PUT_LINE ('It''s afternoon');
      ELSE
         DBMS_OUTPUT.PUT_LINE ('It''s morning');
      END IF;

   END IF;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

Next, consider the modified version of the script with nested CASE statements. For illustrative purposes, this script uses both CASE and searched CASE statements. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_day  VARCHAR2(15);
   v_time VARCHAR(8);
BEGIN
   v_day  := TO_CHAR(SYSDATE, 'fmDay');
   v_time := TO_CHAR(SYSDATE, 'HH24:MI');

   -- CASE statement
   CASE SUBSTR(v_day, 1, 1)
      WHEN 'S' THEN
         DBMS_OUTPUT.PUT_LINE (v_day||', '||v_time);

         -- searched CASE statement
         CASE
            WHEN v_time BETWEEN '12:01' AND '24:00' THEN
               DBMS_OUTPUT.PUT_LINE ('It''s afternoon');
         ELSE
            DBMS_OUTPUT.PUT_LINE ('It''s morning');
         END CASE;
   END CASE;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE('Done...');
END;
```

In this exercise, you substitute nested CASE statements for nested IF statements. Consider the outer CASE statement. It uses a selector expression

```
SUBSTR(v_day, 1, 1)
```

to check if a current day falls on the weekend. Notice that it derives only the first letter of the day. This is a good solution when using a CASE statement, because only Saturday and Sunday start with S. Furthermore, without using the SUBSTR function, you would need to use a searched CASE statement. Recall that the value of the WHEN expression is compared to the value of the selector. As a result, the WHEN expression must return a similar datatype. In this example, the selector expression returns a string datatype, so the WHEN expression must also return a string datatype.

Next, you use a searched CASE to validate the time of day. Recall that, similar to the IF statement, the WHEN conditions of the searched CASE statement yield Boolean values.

When run, this exercise produces the following output:

```
Saturday, 19:49
It's afternoon
Done...

PLSQL procedure successfully completed.
```

2) Create the following script: Modify the script you created in Chapter 4, project 2 of the "Try It Yourself" section. You can use either the CASE statement or the searched CASE statement. The output should look similar to the output produced by the example you created in Chapter 4.

ANSWER: Consider the script you created in Chapter 4:

```
SET SERVEROUTPUT ON
DECLARE
   v_instructor_id NUMBER := &sv_instructor_id;
   v_total NUMBER;
BEGIN
   SELECT COUNT(*)
     INTO v_total
     FROM section
    WHERE instructor_id = v_instructor_id;

   -- check if instructor teaches 3 or more sections
   IF v_total >= 3 THEN
      DBMS_OUTPUT.PUT_LINE ('This instructor needs '||
         'a vacation');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('This instructor teaches '||
        v_total||' sections');
   END IF;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

Next, consider a modified version of the script, with the searched CASE statement instead of the IF-THEN-ELSE statement. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_instructor_id NUMBER := &sv_instructor_id;
   v_total NUMBER;
BEGIN
   SELECT COUNT(*)
     INTO v_total
     FROM section
    WHERE instructor_id = v_instructor_id;

   -- check if instructor teaches 3 or more sections
   CASE
      WHEN v_total >= 3 THEN
         DBMS_OUTPUT.PUT_LINE ('This instructor needs '||
            'a vacation');
      ELSE
         DBMS_OUTPUT.PUT_LINE ('This instructor teaches '||
           v_total||' sections');
   END CASE;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

Assume that value 109 was provided at runtime. Then the script produces the following output:

```
Enter value for sv_instructor_id: 109
old   2:    v_instructor_id NUMBER := &sv_instructor_id;
new   2:    v_instructor_id NUMBER := 109;
This instructor teaches 1 sections
Done...

PLSQL procedure successfully completed.
```

To use the CASE statement, the searched CASE statement could be modified as follows:

```
CASE SIGN(v_total - 3)
   WHEN -1 THEN
      DBMS_OUTPUT.PUT_LINE ('This instructor teaches '||
         v_total||' sections');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('This instructor needs '||
         'a vacation');
END CASE;
```

Notice that the SIGN function is used to determine if an instructor teaches three or more sections. Recall that the SIGN function returns –1 if `v_total` is less than 3, 0 if `v_total` equals 3, and 1 if `v_total` is greater than 3. In this case, as long as the SIGN function returns –1, the message `This instructor teaches ...` is displayed on the screen. In all other cases, the message `This instructor needs a vacation` is displayed on the screen.

3) Execute the following two SELECT statements, and explain why they produce different output:

```
SELECT e.student_id, e.section_id, e.final_grade, g.numeric_grade,
       COALESCE(g.numeric_grade, e.final_grade) grade
  FROM enrollment e, grade g
 WHERE e.student_id = g.student_id
   AND e.section_id = g.section_id
   AND e.student_id = 102
   AND g.grade_type_code = 'FI';

SELECT e.student_id, e.section_id, e.final_grade, g.numeric_grade,
       NULLIF(g.numeric_grade, e.final_grade) grade
  FROM enrollment e, grade g
 WHERE e.student_id = g.student_id
   AND e.section_id = g.section_id
   AND e.student_id = 102
   AND g.grade_type_code = 'FI';
```

ANSWER: Consider the output produced by the following SELECT statements:

```
STUDENT_ID SECTION_ID FINAL_GRADE NUMERIC_GRADE      GRADE
---------- ---------- ----------- ------------- ----------
       102         86                      85         85
       102         89          92          92         92
```

```
STUDENT_ID SECTION_ID FINAL_GRADE NUMERIC_GRADE      GRADE
---------- ---------- ----------- -------------- ----------
       102         86                        85         85
       102         89          92             92
```

Consider the output returned by the first SELECT statement. This statement uses the COALESCE function to derive the value of GRADE. It equals the value of NUMERIC_GRADE in the first row and the value of FINAL_GRADE in the second row.

The COALESCE function compares the value of FINAL_GRADE to NULL. If it is NULL, the value of NUMERIC_GRADE is compared to NULL. Because the value of NUMERIC_GRADE is not NULL, the COALESCE function returns the value of NUMERIC_GRADE in the first row. In the second row, the COALESCE function returns the value of FINAL_GRADE because it is not NULL.

Next, consider the output returned by the second SELECT statement. This statement uses the NULLIF function to derive the value of GRADE. It equals the value of NUMERIC_GRADE in the first row, and it is NULL in the second row.

The NULLIF function compares the NUMERIC_GRADE value to the FINAL_GRADE value. If these values are equal, the NULLIF function returns NULL. In the opposite case, it returns the value of NUMERIC_GRADE.

Chapter 6, "Iterative Control: Part I"

1) Rewrite script ch06_1a.sql using a WHILE loop instead of a simple loop. Make sure that the output produced by this script does not differ from the output produced by the script ch06_1a.sql.

ANSWER: Consider script ch06_1a.sql:

```
SET SERVEROUTPUT ON
DECLARE
   v_counter BINARY_INTEGER := 0;
BEGIN
   LOOP
      -- increment loop counter by one
      v_counter := v_counter + 1;
      DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

      -- if EXIT condition yields TRUE exit the loop
      IF v_counter = 5 THEN
         EXIT;
      END IF;

   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

Next, consider a new version of the script that uses a WHILE loop. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_counter BINARY_INTEGER := 0;
BEGIN
   WHILE v_counter < 5 LOOP
```

```
      -- increment loop counter by one
      v_counter := v_counter + 1;
      DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
   END LOOP;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE('Done...');
END;
```

In this version of the script, you replace a simple loop with a WHILE loop. It is important to remember that a simple loop executes at least once because the EXIT condition is placed in the body of the loop. On the other hand, a WHILE loop may not execute at all, because a condition is tested outside the body of the loop. So, to achieve the same results using the WHILE loop, the EXIT condition

```
v_counter = 5
```

used in the original version is replaced by the test condition

```
v_counter < 5
```

When run, this example produces the following output:

```
v_counter = 1
v_counter = 2
v_counter = 3
v_counter = 4
v_counter = 5
Done...

PL/SQL procedure successfully completed.
```

2) Rewrite script ch06_3a.sql using a numeric FOR loop instead of a WHILE loop. Make sure that the output produced by this script does not differ from the output produced by the script ch06_3a.sql.

ANSWER: Consider script ch06_3a.sql:

```
SET SERVEROUTPUT ON
DECLARE
   v_counter BINARY_INTEGER := 1;
   v_sum NUMBER := 0;
BEGIN
   WHILE v_counter <= 10 LOOP
      v_sum := v_sum + v_counter;
      DBMS_OUTPUT.PUT_LINE ('Current sum is: '||v_sum);

      -- increment loop counter by one
      v_counter := v_counter + 1;
   END LOOP;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('The sum of integers between 1 '||
                         'and 10 is: '||v_sum);
END;
```

Next, consider a new version of the script that uses a WHILE loop. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_sum NUMBER := 0;
BEGIN
   FOR v_counter IN 1..10 LOOP
      v_sum := v_sum + v_counter;
      DBMS_OUTPUT.PUT_LINE ('Current sum is: '||v_sum);
   END LOOP;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('The sum of integers between 1 '||
                         'and 10 is: '||v_sum);
END;
```

In this version of the script, you replace a WHILE loop with a numeric FOR loop. As a result, there is no need to declare the variable `v_counter` and increment it by 1, because the loop itself handles these steps implicitly.

When run, this version of the script produces output identical to the output produced by the original version:

```
Current sum is: 1
Current sum is: 3
Current sum is: 6
Current sum is: 10
Current sum is: 15
Current sum is: 21
Current sum is: 28
Current sum is: 36
Current sum is: 45
Current sum is: 55
The sum of integers between 1 and 10 is: 55

PL/SQL procedure successfully completed.
```

3) Rewrite script ch06_4a.sql using a simple loop instead of a numeric FOR loop. Make sure that the output produced by this script does not differ from the output produced by the script ch06_4a.sql.

ANSWER: Recall script ch06_4a.sql:

```
SET SERVEROUTPUT ON
DECLARE
   v_factorial NUMBER := 1;
BEGIN
   FOR v_counter IN 1..10 LOOP
      v_factorial := v_factorial * v_counter;
   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Factorial of ten is: '||v_factorial);
END;
```

Next, consider a new version of the script that uses a simple loop. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_counter   NUMBER := 1;
   v_factorial NUMBER := 1;
BEGIN
    LOOP
      v_factorial := v_factorial * v_counter;

      v_counter := v_counter + 1;
      EXIT WHEN v_counter = 10;
   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Factorial of ten is: '||v_factorial);
END;
```

In this version of the script, you replace a numeric FOR loop with a simple loop. As a result, you should make three important changes. First, you need to declare and initialize the loop counter, `v_counter`. This counter is implicitly defined and initialized by the FOR loop. Second, you need to increment the value of the loop counter. This is very important, because if you forget to include the statement

```
v_counter := v_counter + 1;
```

in the body of the simple loop, you end up with an infinite loop. This step is not necessary when you use a numeric FOR loop, because it is done by the loop itself.

Third, you need to specify the EXIT condition for the simple loop. Because you are computing a factorial of 10, the following EXIT condition is specified:

```
EXIT WHEN v_counter = 10;
```

You could specify this EXIT condition using an IF-THEN statement as well:

```
IF v_counter = 10 THEN
   EXIT;
END IF;
```

When run, this example shows the following output:

```
Factorial of ten is: 362880

PL/SQL procedure successfully completed.
```

Chapter 7, "Iterative Control: Part II"

1) Rewrite script ch06_4a.sql to calculate the factorial of even integers only between 1 and 10. The script should use a CONTINUE or CONTINUE WHEN statement.

   ANSWER: Recall script ch06_4a.sql:

```
SET SERVEROUTPUT ON
DECLARE
   v_factorial NUMBER := 1;
BEGIN
   FOR v_counter IN 1..10 LOOP
```

```
      v_factorial := v_factorial * v_counter;
   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Factorial of ten is: '||v_factorial);
END;
```

Next, consider a new version of the script that uses a CONTINUE WHEN statement. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_factorial NUMBER := 1;
BEGIN
   FOR v_counter IN 1..10 LOOP
      CONTINUE WHEN MOD(v_counter, 2) != 0;
      v_factorial := v_factorial * v_counter;
   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE
      ('Factorial of even numbers between 1 and 10 is: '||
       v_factorial);
END;
```

In this version of the script, you add a CONTINUE WHEN statement that passes control to the top of the loop if the current value of `v_counter` is not an even number. The rest of the script remains unchanged. Note that you could specify the CONTINUE condition using an IF-THEN statement as well:

```
IF MOD(v_counter, 2) != 0 THEN
   CONTINUE;
END IF;
```

When run, this example shows the following output:

```
Factorial of even numbers between 1 and 10 is: 3840

PL/SQL procedure successfully completed.
```

2) Rewrite script ch07_3a.sql using a simple loop instead of the outer FOR loop, and a WHILE loop for the inner FOR loop. Make sure that the output produced by this script does not differ from the output produced by the original script.

ANSWER: Consider the original version of the script:

```
SET SERVEROUTPUT ON
DECLARE
   v_test NUMBER := 0;
BEGIN
   <<outer_loop>>
   FOR i IN 1..3 LOOP
      DBMS_OUTPUT.PUT_LINE('Outer Loop');
      DBMS_OUTPUT.PUT_LINE('i = '||i);
      DBMS_OUTPUT.PUT_LINE('v_test = '||v_test);
      v_test := v_test + 1;
```

```
      <<inner_loop>>
      FOR j IN 1..2 LOOP
         DBMS_OUTPUT.PUT_LINE('Inner Loop');
         DBMS_OUTPUT.PUT_LINE('j = '||j);
         DBMS_OUTPUT.PUT_LINE('i = '||i);
         DBMS_OUTPUT.PUT_LINE('v_test = '||v_test);
      END LOOP inner_loop;
   END LOOP outer_loop;
END;
```

Next, consider a modified version of the script that uses simple and WHILE loops. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   i INTEGER := 1;
   j INTEGER := 1;
   v_test NUMBER := 0;
BEGIN
   <<outer_loop>>
    LOOP
      DBMS_OUTPUT.PUT_LINE ('Outer Loop');
      DBMS_OUTPUT.PUT_LINE ('i = '||i);
      DBMS_OUTPUT.PUT_LINE ('v_test = '||v_test);
      v_test := v_test + 1;

      -- reset inner loop counter
      j := 1;

      <<inner_loop>>
      WHILE j <= 2 LOOP
         DBMS_OUTPUT.PUT_LINE ('Inner Loop');
         DBMS_OUTPUT.PUT_LINE ('j = '||j);
         DBMS_OUTPUT.PUT_LINE ('i = '||i);
         DBMS_OUTPUT.PUT_LINE ('v_test = '||v_test);
         j := j + 1;
      END LOOP inner_loop;

      i := i + 1;
      -- EXIT condition of the outer loop
      EXIT WHEN i > 3;
   END LOOP outer_loop;
END;
```

Note that this version of the script contains changes that are important due to the nature of the loops that are used.

First, both counters, for outer and inner loops, must be declared and initialized. Moreover, the counter for the inner loop must be initialized to 1 before the inner loop is executed, not in the declaration section of this script. In other words, the inner loop executes three times. It is important not to confuse the phrase *execution of the loop* with the term *iteration*. *Each execution of the*

WHILE loop causes the statements inside this loop to iterate twice. Before each execution, the loop counter `j` must be reset to 1 again. This step is necessary because the WHILE loop does not initialize its counter implicitly like a numeric FOR loop. As a result, after the first execution of the WHILE loop is complete, the value of counter `j` is equal to 3. If this value is not reset to 1 again, the loop does not execute a second time.

Second, both loop counters must be incremented. Third, the EXIT condition must be specified for the outer loop, and the test condition must be specified for the inner loop.

When run, the exercise produces the following output:

```
Outer Loop
i = 1
v_test = 0
Inner Loop
j = 1
i = 1
v_test = 1
Inner Loop
j = 2
i = 1
v_test = 1
Outer Loop
i = 2
v_test = 1
Inner Loop
j = 1
i = 2
v_test = 2
Inner Loop
j = 2
i = 2
v_test = 2
Outer Loop
i = 3
v_test = 2
Inner Loop
j = 1
i = 3
v_test = 3
Inner Loop
j = 2
i = 3
v_test = 3

PL/SQL procedure successfully completed.
```

Chapter 8, "Error Handling and Built-In Exceptions"

1) Create the following script: Check to see whether there is a record in the STUDENT table for a given student ID. If there is not, insert a record into the STUDENT table for the given student ID.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   v_student_id NUMBER       := &sv_student_id;
   v_first_name VARCHAR2(30) := '&sv_first_name';
   v_last_name  VARCHAR2(30) := '&sv_last_name';
   v_zip        CHAR(5)      := '&sv_zip';
   v_name       VARCHAR2(50);
BEGIN
   SELECT first_name||' '||last_name
     INTO v_name
     FROM student
    WHERE student_id = v_student_id;

   DBMS_OUTPUT.PUT_LINE ('Student '||v_name||' is a valid student');
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE
         ('This student does not exist, and will be '||
          'added to the STUDENT table');

      INSERT INTO student
         (student_id, first_name, last_name, zip, registration_date,
          created_by, created_date, modified_by, modified_date)
      VALUES
         (v_student_id, v_first_name, v_last_name, v_zip, SYSDATE,
          USER, SYSDATE, USER, SYSDATE);
      COMMIT;
END;
```

This script accepts a value for student's ID from a user. For a given student ID, it determines the student's name using the SELECT INTO statement and displays it on the screen. If the value provided by the user is not a valid student ID, control of execution is passed to the exception-handling section of the block, where the NO_DATA_FOUND exception is raised. As a result, the message `This student does not exist` ... is displayed on the screen, and a new record is inserted into the STUDENT table.

To test this script fully, consider running it for two values of student ID. Only one value should correspond to an existing student ID. It is important to note that a valid zip code must be provided for both runs. Why do you think this is necessary?

When 319 is provided for the student ID (it is a valid student ID), this exercise produces the following output:

```
Enter value for sv_student_id: 319
old   2:    v_student_id NUMBER := &sv_student_id;
new   2:    v_student_id NUMBER := 319;
Enter value for sv_first_name: John
old   3:    v_first_name VARCHAR2(30) := '&sv_first_name';
new   3:    v_first_name VARCHAR2(30) := 'John';
Enter value for sv_last_name: Smith
old   4:    v_last_name VARCHAR2(30) := '&sv_last_name';
new   4:    v_last_name VARCHAR2(30) := 'Smith';
```

```
Enter value for sv_zip: 07421
old   5:    v_zip CHAR(5) := '&sv_zip';
new   5:    v_zip CHAR(5) := '07421';
Student George Eakheit is a valid student

PLSQL procedure successfully completed.
```

Notice that the name displayed by the script does not correspond to the name entered at runtime. Why do you think this is?

When 555 is provided for the student ID (it is not a valid student ID), this exercise produces the following output:

```
Enter value for sv_student_id: 555
old   2:    v_student_id NUMBER := &sv_student_id;
new   2:    v_student_id NUMBER := 555;
Enter value for sv_first_name: John
old   3:    v_first_name VARCHAR2(30) := '&sv_first_name';
new   3:    v_first_name VARCHAR2(30) := 'John';
Enter value for sv_last_name: Smith
old   4:    v_last_name VARCHAR2(30) := '&sv_last_name';
new   4:    v_last_name VARCHAR2(30) := 'Smith';
Enter value for sv_zip: 07421
old   5:    v_zip CHAR(5) := '&sv_zip';
new   5:    v_zip CHAR(5) := '07421';
This student does not exist, and will be added to the STUDENT table

PLSQL procedure successfully completed.
```

Next, you can select this new record from the STUDENT table as follows:

```
SELECT student_id, first_name, last_name
  FROM student
 WHERE student_id = 555;

STUDENT_ID FIRST_NAME                LAST_NAME
---------- ------------------------- ----------------
       555 John                      Smith
```

2) Create the following script: For a given instructor ID, check to see whether it is assigned to a valid instructor. Then check to see how many sections this instructor teaches, and display this information on the screen.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   v_instructor_id NUMBER := &sv_instructor_id;
   v_name VARCHAR2(50);
   v_total NUMBER;
 BEGIN
   SELECT first_name||' '||last_name
     INTO v_name
     FROM instructor
    WHERE instructor_id = v_instructor_id;
```

```
   -- check how many sections are taught by this instructor
   SELECT COUNT(*)
     INTO v_total
     FROM section
    WHERE instructor_id = v_instructor_id;

   DBMS_OUTPUT.PUT_LINE ('Instructor, '||v_name||
      ', teaches '||v_total||' section(s)');
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('This is not a valid instructor');
END;
```

This script accepts a value for the instructor's ID from a user. For a given instructor ID, it determines the instructor's name using the SELECT INTO statement. This SELECT INTO statement checks to see if the ID provided by the user is a valid instructor ID. If this value is not valid, control of execution is passed to the exception-handling section of the block, where the NO_DATA_FOUND exception is raised. As a result, the message `This is not a valid instructor` is displayed on the screen. On the other hand, if the value provided by the user is a valid instructor ID, the second SELECT INTO statement calculates how many sections are taught by this instructor.

To test this script fully, consider running it for two values of instructor ID. When 105 is provided for the instructor ID (it is a valid instructor ID), this exercise produces the following output:

```
Enter value for sv_instructor_id: 105
old   2:    v_instructor_id NUMBER := &sv_instructor_id;
new   2:    v_instructor_id NUMBER := 105;
Instructor, Anita Morris, teaches 10 section(s)

PLSQL procedure successfully completed.
```

When 123 is provided for the instructor ID (it is not a valid student ID), this exercise produces the following output:

```
Enter value for sv_instructor_id: 123
old   2:    v_instructor_id NUMBER := &sv_instructor_id;
new   2:    v_instructor_id NUMBER := 123;
This is not a valid instructor

PLSQL procedure successfully completed.
```

Chapter 9, "Exceptions"

1) Create the following script: For a course section provided at runtime, determine the number of students registered. If this number is equal to or greater than 10, raise the user-defined exception `e_too_many_students` and display an error message. Otherwise, display how many students are in a section.

   ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   v_section_id        NUMBER := &sv_section_id;
   v_total_students    NUMBER;
   e_too_many_students EXCEPTION;
```

```
BEGIN
   -- Calculate number of students enrolled
   SELECT COUNT(*)
     INTO v_total_students
     FROM enrollment
    WHERE section_id = v_section_id;

   IF v_total_students >= 10 THEN
      RAISE e_too_many_students;
   ELSE
      DBMS_OUTPUT.PUT_LINE ('There are '||v_total_students||
         ' students for section ID: '||v_section_id);
   END IF;
EXCEPTION
   WHEN e_too_many_students THEN
      DBMS_OUTPUT.PUT_LINE ('There are too many '||
         'students for section '||v_section_id);
END;
```

In this script, you declare two variables, `v_section_id` and `v_total_students`, to store the section ID provided by the user and the total number of students in that section ID, respectively. You also declare a user-defined exception `e_too_many_students`. You raise this exception using the IF-THEN statement if the value returned by the COUNT function exceeds 10. Otherwise, you display the message specifying how many students are enrolled in a given section.

To test this script fully, consider running it for two values of section ID. When 101 is provided for the section ID (this section has more than ten students), this script produces the following output:

```
Enter value for sv_section_id: 101
old   2:    v_section_id        NUMBER := &sv_section_id;
new   2:    v_section_id        NUMBER := 101;
There are too many students for section 101

PL/SQL procedure successfully completed.
```

When 116 is provided for the section ID (this section has fewer than ten students), this script produces different output:

```
Enter value for sv_section_id: 116
old   2:    v_section_id        NUMBER := &sv_section_id;
new   2:    v_section_id        NUMBER := 116;
There are 8 students for section ID: 116

PL/SQL procedure successfully completed.
```

Next, consider running this script for a nonexistent section ID:

```
Enter value for sv_section_id: 999
old   2:    v_section_id        NUMBER := &sv_section_id;
new   2:    v_section_id        NUMBER := 999;
There are 0 students for section ID: 999

PL/SQL procedure successfully completed.
```

Note that the script does not produce any errors. Instead, it states that section 999 has 0 students. How would you modify this script to ensure that when there is no corresponding section ID in the ENROLLMENT table, the message `This section does not exist` is displayed on the screen?

2) Modify the script you just created. After the exception `e_too_many_students` has been raised in the inner block, reraise it in the outer block.

ANSWER: The new version of the script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_section_id        NUMBER := &sv_section_id;
   v_total_students    NUMBER;
   e_too_many_students EXCEPTION;
BEGIN
   -- Add inner block
   BEGIN
      -- Calculate number of students enrolled
      SELECT COUNT(*)
        INTO v_total_students
        FROM enrollment
       WHERE section_id = v_section_id;

      IF v_total_students >= 10 THEN
         RAISE e_too_many_students;
      ELSE
         DBMS_OUTPUT.PUT_LINE ('There are '||v_total_students||
            ' students for section ID: '||v_section_id);
      END IF;
   -- Re-raise exception
   EXCEPTION
      WHEN e_too_many_students THEN
         RAISE;
   END;
EXCEPTION
   WHEN e_too_many_students THEN
      DBMS_OUTPUT.PUT_LINE ('There are too many '||
         'students for section '||v_section_id);
END;
```

In this version of the script, you introduce an inner block where the `e_too_many_students` exception is raised first and then propagated to the outer block. This version of the script produces output identical to the original script.

Next, consider a different version in which the original PL/SQL block (the PL/SQL block from the original script) has been enclosed in another block:

```
SET SERVEROUTPUT ON
-- Outer PL/SQL block
BEGIN
   -- This block became inner PL/SQL block
```

```
   DECLARE
      v_section_id        NUMBER := &sv_section_id;
      v_total_students    NUMBER;
      e_too_many_students EXCEPTION;
   BEGIN
      -- Calculate number of students enrolled
      SELECT COUNT(*)
        INTO v_total_students
        FROM enrollment
       WHERE section_id = v_section_id;

      IF v_total_students >= 10 THEN
         RAISE e_too_many_students;
      ELSE
         DBMS_OUTPUT.PUT_LINE ('There are '||v_total_students||
            ' students for section ID: '||v_section_id);
      END IF;
   EXCEPTION
      WHEN e_too_many_students THEN
         RAISE;
   END;

EXCEPTION
   WHEN e_too_many_students THEN
      DBMS_OUTPUT.PUT_LINE ('There are too many '||
         'students for section '||v_section_id);
END;
```

This version of the script causes the following error message:

```
Enter value for sv_section_id: 101
old   4:       v_section_id        NUMBER := &sv_section_id;
new   4:       v_section_id        NUMBER := 101;
   WHEN e_too_many_students THEN
        *
ERROR at line 26:
ORA-06550: line 26, column 9:
PLS-00201: identifier 'E_TOO_MANY_STUDENTS' must be declared
ORA-06550: line 0, column 0:
PL/SQL: Compilation unit analysis terminated
```

This occurs because the `e_too_many_students` exception is declared in the inner block and, as a result, is not visible to the outer block. In addition, the `v_section_id` variable used by the exception-handling section of the outer block is declared in the inner block as well, and, as a result, is not accessible in the outer block.

To correct these errors, the previous version of the script can be modified as follows:

```
SET SERVEROUTPUT ON
-- Outer PL/SQL block
DECLARE
   v_section_id        NUMBER := &sv_section_id;
   e_too_many_students EXCEPTION;
```

```
BEGIN
   -- This block became inner PL/SQL block
   DECLARE
      v_total_students NUMBER;
   BEGIN
      -- Calculate number of students enrolled
      SELECT COUNT(*)
        INTO v_total_students
        FROM enrollment
       WHERE section_id = v_section_id;

      IF v_total_students >= 10 THEN
         RAISE e_too_many_students;
      ELSE
         DBMS_OUTPUT.PUT_LINE ('There are '||v_total_students||
            ' students for section ID: '||v_section_id);
      END IF;
   EXCEPTION
      WHEN e_too_many_students THEN
         RAISE;
   END;

EXCEPTION
   WHEN e_too_many_students THEN
      DBMS_OUTPUT.PUT_LINE ('There are too many '||
         'students for section '||v_section_id);
END;
```

Chapter 10, "Exceptions: Advanced Concepts"

1) Modify the script you created in project 1 of the "Try It Yourself" section in Chapter 9. Raise a user-defined exception with the RAISE_APPLICATION_ERROR statement. Otherwise, display how many students are in a section. Make sure your program can process all sections.

   ANSWER: The script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_section_id        NUMBER := &sv_section_id;
   v_total_students    NUMBER;
BEGIN
   -- Calculate number of students enrolled
   SELECT COUNT(*)
     INTO v_total_students
     FROM enrollment
    WHERE section_id = v_section_id;

   IF v_total_students >= 10 THEN
      RAISE_APPLICATION_ERROR
         (-20000, 'There are too many students for '||
          'section '||v_section_id);
```

```
   ELSE
      DBMS_OUTPUT.PUT_LINE ('There are '||v_total_students||
         ' students for section ID: '||v_section_id);
   END IF;
END;
```

In this version of the script, you use the RAISE_APPLICATION_ERROR statement to handle the following error condition: If the number of students enrolled in a particular section is equal to or greater than ten, an error is raised. It is important to remember that the RAISE_APPLICATION_ERROR statement works with the unnamed user-defined exceptions. Therefore, notice that there is no reference to the exception `e_too_many_students` anywhere in this script. On the other hand, an error number has been associated with the error message.

When run, this exercise produces the following output (the same section IDs are used for this script as well: 101, 116, and 999):

```
Enter value for sv_section_id: 101
old   2:    v_section_id        NUMBER := &sv_section_id;
new   2:    v_section_id        NUMBER := 101;
DECLARE
*
ERROR at line 1:
ORA-20000: There are too many students for section 101
ORA-06512: at line 12


Enter value for sv_section_id: 116
old   2:    v_section_id        NUMBER := &sv_section_id;
new   2:    v_section_id        NUMBER := 116;
There are 8 students for section ID: 116

PL/SQL procedure successfully completed.


Enter value for sv_section_id: 999
old   2:    v_section_id        NUMBER := &sv_section_id;
new   2:    v_section_id        NUMBER := 999;
There are 0 students for section ID: 999

PL/SQL procedure successfully completed.
```

2) Create the following script: Try to add a record to the INSTRUCTOR table without providing values for the columns CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE. Define an exception and associate it with the Oracle error number so that the error generated by the INSERT statement is handled.

ANSWER: Consider the following script. Notice that it has no exception handlers:

```
DECLARE
   v_first_name instructor.first_name%type := '&sv_first_name';
   v_last_name  instructor.last_name%type  := '&sv_last_name';
BEGIN
   INSERT INTO instructor
      (instructor_id, first_name, last_name)
```

```
   VALUES
      (INSTRUCTOR_ID_SEQ.NEXTVAL, v_first_name, v_last_name);
   COMMIT;
END;
```

In this version of the script, you are trying to add a new record to the INSTRUCTOR table. The INSERT statement has only three columns: INSTRUCTOR_ID, FIRST_NAME, and LAST_NAME. The value for the column INSTRUCTOR_ID is determined from the sequence INSTRUCTOR_ID_SEQ, and the user provides the values for the columns FIRST_NAME and LAST_NAME.

When run, this script produces the following error message:

```
Enter value for sv_first_name: John
old   2:       '&sv_first_name';
new   2:       'John';
Enter value for sv_last_name: Smith
old   3:       '&sv_last_name';
new   3:       'Smith';
DECLARE
*
ERROR at line 1:
ORA-01400: cannot insert NULL into
  ("STUDENT"."INSTRUCTOR"."CREATED_BY")
ORA-06512: at line 5
```

This error message states that a NULL value cannot be inserted into the column CREATED_BY of the INSTRUCTOR table. Therefore, you need to add an exception handler to the script, as follows. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_first_name instructor.first_name%type := '&sv_first_name';
   v_last_name  instructor.last_name%type  := '&sv_last_name';

   e_non_null_value EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_non_null_value, -1400);
BEGIN
   INSERT INTO INSTRUCTOR
      (instructor_id, first_name, last_name)
   VALUES
      (INSTRUCTOR_ID_SEQ.NEXTVAL, v_first_name, v_last_name);
   COMMIT;
EXCEPTION
   WHEN e_non_null_value THEN
      DBMS_OUTPUT.PUT_LINE ('A NULL value cannot be '||
         'inserted. Check constraints on the INSTRUCTOR table.');
END;
```

In this version of the script, you declare a new exception called `e_non_null_value`. Next, you associate an Oracle error number with this exception. As a result, you can add an exception-handling section to trap the error generated by Oracle.

When run, the new version produces the following output:

```
Enter value for sv_first_name: John
old   2:       '&sv_first_name';
new   2:       'John';
Enter value for sv_last_name: Smith
old   3:       '&sv_last_name';
new   3:       'Smith';
A NULL value cannot be inserted. Check constraints on the
INSTRUCTOR table.

PL/SQL procedure successfully completed.
```

3) Modify the script you just created. Instead of declaring a user-defined exception, add the OTHERS exception handler to the exception-handling section of the block. Then display the error number and the error message on the screen.

ANSWER: The script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   v_first_name instructor.first_name%type := '&sv_first_name';
   v_last_name  instructor.last_name%type  := '&sv_last_name';
BEGIN
   INSERT INTO INSTRUCTOR
      (instructor_id, first_name, last_name)
   VALUES
      (INSTRUCTOR_ID_SEQ.NEXTVAL, v_first_name, v_last_name);
   COMMIT;
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ('Error code: '||SQLCODE);
      DBMS_OUTPUT.PUT_LINE ('Error message: '||
         SUBSTR(SQLERRM, 1, 200));
END;
```

Notice that as long as the OTHERS exception handler is used, there is no need to associate an Oracle error number with a user-defined exception. When run, this exercise produces the following output:

```
Enter value for sv_first_name: John
old   2:       '&sv_first_name';
new   2:       'John';
Enter value for sv_last_name: Smith
old   3:       '&sv_last_name';
new   3:       'Smith';
Error code: -1400
Error message: ORA-01400: cannot insert NULL into
("STUDENT"."INSTRUCTOR"."CREATED_BY")

PL/SQL procedure successfully completed.
```

Chapter 11, "Introduction to Cursors"

1) Write a nested cursor in which the parent cursor SELECTs information about each section of a course. The child cursor counts the enrollment. The only output is one line for each course, with the course name, section number, and total enrollment.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR c_course IS
      SELECT course_no, description
        FROM course
       WHERE course_no < 120;

   CURSOR c_enrollment(p_course_no IN course.course_no%TYPE)
   IS
      SELECT s.section_no section_no, count(*) count
        FROM section s, enrollment e
       WHERE s.course_no = p_course_no
         AND s.section_id = e.section_id
       GROUP BY s.section_no;
BEGIN
   FOR r_course IN c_course LOOP
      DBMS_OUTPUT.PUT_LINE
         (r_course.course_no||' '|| r_course.description);

      FOR r_enroll IN c_enrollment(r_course.course_no) LOOP
         DBMS_OUTPUT.PUT_LINE
           (Chr(9)||'Section:  '||r_enroll.section_no||
            ' has an enrollment of: '||r_enroll.count);
      END LOOP;

   END LOOP;
END;
```

2) Write an anonymous PL/SQL block that finds all the courses that have at least one section that is at its maximum enrollment. If no courses meet that criterion, pick two courses and create that situation for each.

   A) For each of those courses, add another section. The instructor for the new section should be taken from the existing records in the instructor table. Use the instructor who is signed up to teach the fewest courses. Handle the fact that, during the execution of your program, the instructor teaching the most courses may change.

   B) Use any exception-handling techniques you think are useful to capture error conditions.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   v_instid_min     instructor.instructor_id%TYPE;
   v_section_id_new section.section_id%TYPE;
   v_snumber_recent section.section_no%TYPE := 0;
```

```
   -- This cursor determines the courses that have at least
   -- one section filled to capacity.
   CURSOR c_filled IS
      SELECT DISTINCT s.course_no
        FROM section s
       WHERE s.capacity = (SELECT COUNT(section_id)
                             FROM enrollment e
                            WHERE e.section_id = s.section_id);
BEGIN
   FOR r_filled IN c_filled LOOP
      -- For each course in this list, add another section.
      -- First, determine the instructor who is teaching
      -- the fewest courses. If more than one instructor
      -- is teaching the same number of minimum courses
      -- (e.g. if there are three instructors teaching one
      -- course) use any of those instructors.
      SELECT instructor_id
        INTO v_instid_min
        FROM instructor
       WHERE EXISTS (SELECT NULL
                       FROM section
                      WHERE section.instructor_id =
                            instructor.instructor_id
                     GROUP BY instructor_id
                     HAVING COUNT(*) =
                       (SELECT MIN(COUNT(*))
                          FROM section
                         WHERE instructor_id IS NOT NULL
                         GROUP BY instructor_id)
                    )
         AND ROWNUM = 1;

      -- Determine the section_id for the new section.
      -- Note that this method would not work in a multiuser
      -- environment. A sequence should be used instead.
      SELECT MAX(section_id) + 1
        INTO v_section_id_new
        FROM section;

      -- Determine the section number for the new section.
      -- This only needs to be done in the real world if
      -- the system specification calls for a sequence in
      -- a parent. The sequence in parent here refers to
      -- the section_no incrementing within the course_no,
      -- and not the section_no incrementing within the
      -- section_id.
      DECLARE
         CURSOR c_snumber_in_parent IS
            SELECT section_no
              FROM section
```

```
            WHERE course_no = r_filled.course_no
          ORDER BY section_no;
      BEGIN
         -- Go from the lowest to the highest section_no
         -- and find any gaps. If there are no gaps make
         -- the new section_no equal to the highest
         -- current section_no + 1.

         FOR r_snumber_in_parent IN c_snumber_in_parent LOOP
            EXIT WHEN
               r_snumber_in_parent.section_no > v_snumber_recent
                 + 1;
               v_snumber_recent := r_snumber_in_parent.section_no
                 + 1;
         END LOOP;

         -- At this point, v_snumber_recent will be equal
         -- either to the value preceeding the gap or to
         -- the highest section_no for that course.
      END;
      -- Do the insert.
      INSERT INTO section
        (section_id, course_no, section_no, instructor_id)
      VALUES
        (v_section_id_new, r_filled.course_no, v_snumber_recent,
         v_instid_min);
      COMMIT;
   END LOOP;
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

Chapter 12, "Advanced Cursors"

This chapter has no "Try It Yourself" section.

Chapter 13, "Triggers"

1) Create or modify a trigger on the ENROLLMENT table that fires before an INSERT statement. Make sure that all columns that have NOT NULL and foreign key constraints defined on them are populated with their proper values.

   ANSWER: The trigger should look similar to the following:

```
CREATE OR REPLACE TRIGGER enrollment_bi
BEFORE INSERT ON ENROLLMENT
FOR EACH ROW
DECLARE
   v_valid NUMBER := 0;
```

```
BEGIN
   SELECT COUNT(*)
     INTO v_valid
     FROM student
    WHERE student_id = :NEW.STUDENT_ID;

   IF v_valid = 0 THEN
      RAISE_APPLICATION_ERROR (-20000,
         'This is not a valid student');
   END IF;

   SELECT COUNT(*)
     INTO v_valid
     FROM section
    WHERE section_id = :NEW.SECTION_ID;

   IF v_valid = 0 THEN
      RAISE_APPLICATION_ERROR (-20001,
         'This is not a valid section');
   END IF;

   :NEW.ENROLL_DATE   := SYSDATE;
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;
END;
```

Consider this trigger. It fires before the INSERT statement on the ENROLLMENT table. First, you validate new values for student ID and section ID. If one of the IDs is invalid, the exception is raised, and the trigger is terminated. As a result, the INSERT statement causes an error. If both student and section IDs are found in the STUDENT and SECTION tables, respectively, ENROLL_DATE, CREATED_DATE, and MODIFIED_DATE are populated with the current date, and the columns CREATED_BY and MODIFIED_BY are populated with the current user name.

Consider the following INSERT statement:

```
INSERT INTO enrollment (student_id, section_id)
VALUES (777, 123);
```

The value 777 in this INSERT statement does not exist in the STUDENT table and therefore is invalid. As a result, this INSERT statement causes the following error:

```
INSERT INTO enrollment (student_id, section_id)
*
ERROR at line 1:
ORA-20000: This is not a valid student
ORA-06512: at "STUDENT.ENROLLMENT_BI", line 10
ORA-04088: error during execution of trigger 'STUDENT.ENROLLMENT_BI'
```

2) Create or modify a trigger on the SECTION table that fires before an UPDATE statement. Make sure that the trigger validates incoming values so that there are no constraint violation errors.

ANSWER: The trigger should look similar to the following:

```
CREATE OR REPLACE TRIGGER section_bu
BEFORE UPDATE ON SECTION
FOR EACH ROW
DECLARE
   v_valid NUMBER := 0;
BEGIN
   IF :NEW.INSTRUCTOR_ID IS NOT NULL THEN
      SELECT COUNT(*)
        INTO v_valid
        FROM instructor
       WHERE instructor_id = :NEW.instructor_ID;

      IF v_valid = 0 THEN
         RAISE_APPLICATION_ERROR (-20000,
            'This is not a valid instructor');
      END IF;
   END IF;

   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;
END;
```

This trigger fires before the UPDATE statement on the SECTION table. First, you check to see if there is a new value for an instructor ID with the help of an IF-THEN statement. If the IF-THEN statement evaluates to TRUE, the instructor's ID is checked against the INSTRUCTOR table. If a new instructor ID does not exist in the INSTRUCTOR table, the exception is raised, and the trigger is terminated. Otherwise, all columns with NOT NULL constraints are populated with their respective values.

Note that this trigger does not populate the CREATED_BY and CREATED_DATE columns with the new values. This is because when the record is updated, the values for these columns do not change, because they reflect when this record was added to the SECTION table.

Consider the following UPDATE statement:

```
UPDATE section
   SET instructor_id = 220
 WHERE section_id = 79;
```

The value 220 in this UPDATE statement does not exist in the INSTRUCTOR table and therefore is invalid. As a result, this UPDATE statement when run causes an error:

```
UPDATE section
*
ERROR at line 1:
ORA-20000: This is not a valid instructor
ORA-06512: at "STUDENT.SECTION_BU", line 11
ORA-04088: error during execution of trigger 'STUDENT.SECTION_BU'
```

Next, consider an UPDATE statement that does not cause any errors:

```
UPDATE section
     SET instructor_id = 105
   WHERE section_id = 79;

1 row updated.

rollback;

Rollback complete.
```

Chapter 14, "Compound Triggers"

1) Create a compound trigger on the INSTRUCTOR table that fires on the INSERT and UPDATE statements. The trigger should not allow an insert or update on the INSTRUCTOR table during off hours. Off hours are weekends and times of day outside the 9 a.m. to 5 p.m. window. The trigger should also populate the INSTRUCTOR_ID, CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE columns with their default values.

   ANSWER: The trigger should look similar to the following:

```
CREATE OR REPLACE TRIGGER instructor_compound
FOR INSERT OR UPDATE ON instructor
COMPOUND TRIGGER

   v_date DATE;
   v_user VARCHAR2(30);

BEFORE STATEMENT IS
BEGIN
   IF RTRIM(TO_CHAR(SYSDATE, 'DAY')) NOT LIKE 'S%' AND
      RTRIM(TO_CHAR(SYSDATE, 'HH24:MI')) BETWEEN '09:00' AND '17:00'
   THEN
      v_date := SYSDATE;
      v_user := USER;
   ELSE
      RAISE_APPLICATION_ERROR
         (-20000, 'A table cannot be modified during off hours');
   END IF;

END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
   IF INSERTING THEN
      :NEW.instructor_id := INSTRUCTOR_ID_SEQ.NEXTVAL;
      :NEW.created_by    := v_user;
      :NEW.created_date  := v_date;

   ELSIF UPDATING THEN
      :NEW.created_by    := :OLD.created_by;
```

```
      :NEW.created_date  := :OLD.created_date;
   END IF;

   :NEW.modified_by   := v_user;
   :NEW.modified_date := v_date;

END BEFORE EACH ROW;

END instructor_compound;
```

This compound trigger has two executable sections, BEFORE STATEMENT and BEFORE EACH ROW. The BEFORE STATEMENT portion prevents any updates to the INSTRUCTOR table during off hours. In addition, it populates the `v_date` and `v_user` variables that are used to populate the CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE columns. The BEFORE EACH ROW section populates these columns. In addition, it assigns a value to the INSTRUCTOR_ID column from INSTRUCTOR_ID_SEQ.

Note the use of the INSERTING and UPDATING functions in the BEFORE EACH ROW section. The INSERTING function is used because the INSTRUCTOR_ID, CREATED_BY, and CREATED_DATE columns are populated with new values only if a record is being inserted in the INSTRUCTOR table. This is not so when a record is being updated. In this case, the CREATED_BY and CREATED_DATE columns are populated with the values copied from the OLD pseudorecord. However, the MODIFIED_BY and MODIFIED_DATE columns need to be populated with the new values regardless of the INSERT or UPDATE operation.

The newly created trigger may be tested as follows:

```
SET SERVEROUTPUT ON
DECLARE
   v_date VARCHAR2(20);
BEGIN
   v_date := TO_CHAR(SYSDATE, 'DD/MM/YYYY HH24:MI');
   DBMS_OUTPUT.PUT_LINE ('Date: '||v_date);

   INSERT INTO instructor
      (salutation, first_name, last_name, street_address, zip, phone)
   VALUES
      ('Mr.', 'Test', 'Instructor', '123 Main Street', '07112',
       '2125555555');

   ROLLBACK;
END;
/
```

The output is as follows:

```
Date: 25/04/2008 15:47

PL/SQL procedure successfully completed.
```

Here's the second test:

```
SET SERVEROUTPUT ON
DECLARE
   v_date VARCHAR2(20);
```

```
BEGIN
   v_date := TO_CHAR(SYSDATE, 'DD/MM/YYYY HH24:MI');
   DBMS_OUTPUT.PUT_LINE ('Date: '||v_date);

   UPDATE instructor
      SET phone = '2125555555'
    WHERE instructor_id = 101;

   ROLLBACK;
END;
/
```

The output is as follows:

```
Date: 26/04/2008 19:50
DECLARE
*
ERROR at line 1:
ORA-20000: A table cannot be modified during off hours
ORA-06512: at "STUDENT.INSTRUCTOR_COMPOUND", line 15
ORA-04088: error during execution of trigger 'STUDENT.INSTRUCTOR_COM-
POUND'
ORA-06512: at line 7
```

2) Create a compound trigger on the ZIPCODE table that fires on the INSERT and UPDATE statements. The trigger should populate the CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE columns with their default values. In addition, it should record in the STATISTICS table the type of the transaction, the name of the user who issued the transaction, and the date of the transaction. Assume that the STATISTICS table has the following structure:

```
Name                           Null?    Type
------------------------------ -------- ----
TABLE_NAME                              VARCHAR2(30)
TRANSACTION_NAME                        VARCHAR2(10)
TRANSACTION_USER                        VARCHAR2(30)
TRANSACTION_DATE                        DATE
```

ANSWER: The trigger should look similar to the following:

```
CREATE OR REPLACE TRIGGER zipcode_compound
FOR INSERT OR UPDATE ON zipcode
COMPOUND TRIGGER

   v_date DATE;
   v_user VARCHAR2(30);
   v_type VARCHAR2(10);

BEFORE STATEMENT IS
BEGIN
   v_date := SYSDATE;
   v_user := USER;
END BEFORE STATEMENT;
```

```
BEFORE EACH ROW IS
BEGIN
   IF INSERTING THEN
      :NEW.created_by   := v_user;
      :NEW.created_date := v_date;

   ELSIF UPDATING THEN
      :NEW.created_by   := :OLD.created_by;
      :NEW.created_date := :OLD.created_date;
   END IF;

   :NEW.modified_by   := v_user;
   :NEW.modified_date := v_date;

END BEFORE EACH ROW;

AFTER STATEMENT IS
BEGIN
   IF INSERTING THEN
      v_type := 'INSERT';

   ELSIF UPDATING THEN
      v_type := 'UPDATE';
   END IF;

   INSERT INTO statistics
      (table_name, transaction_name, transaction_user,
       transaction_date)
   VALUES ('ZIPCODE', v_type, v_user, v_date);

END AFTER STATEMENT;

END zipcode_compound;
```

```
UPDATE zipcode
   SET city = 'Test City'
 WHERE zip  = '01247';
```

1 row updated.

```
SELECT *
  FROM statistics
 WHERE transaction_date >= TRUNC(sysdate);
```

```
TABLE_NAME TRANSACTION_NAME TRANSACTION_USER TRANSACTION_DATE
---------- --------------- --------------- ---------------
ZIPCODE    UPDATE          STUDENT         24-APR-08
```

```
ROLLBACK;
```

Rollback complete.

Chapter 15, "Collections"

1) Create the following script: Create an associative array (index-by table), and populate it with the instructor's full name. In other words, each row of the associative array should contain the first name, middle initial, and last name. Display this information on the screen.

   ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR name_cur IS
      SELECT first_name||' '||last_name name
        FROM instructor;

   TYPE name_type IS TABLE OF VARCHAR2(50)
      INDEX BY BINARY_INTEGER;
   name_tab name_type;

   v_counter INTEGER := 0;
BEGIN
   FOR name_rec IN name_cur LOOP
      v_counter := v_counter + 1;
      name_tab(v_counter) := name_rec.name;

      DBMS_OUTPUT.PUT_LINE ('name('||v_counter||'): '||
         name_tab(v_counter));
   END LOOP;
END;
```

   In the preceding example, the associative array `name_tab` is populated with instructors' full names. Notice that the variable `v_counter` is used as a subscript to reference individual array elements. This example produces the following output:

```
name(1): Fernand Hanks
name(2): Tom Wojick
name(3): Nina Schorin
name(4): Gary Pertez
name(5): Anita Morris
name(6): Todd Smythe
name(7): Marilyn Frantzen
name(8): Charles Lowry
name(9): Rick Chow

PL/SQL procedure successfully completed.
```

2) Modify the script you just created. Instead of using an associative array, use a varray.

   ANSWER: The script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR name_cur IS
      SELECT first_name||' '||last_name name
        FROM instructor;
```

```
   TYPE name_type IS VARRAY(15) OF VARCHAR2(50);
   name_varray name_type := name_type();

   v_counter INTEGER := 0;
BEGIN
   FOR name_rec IN name_cur LOOP
      v_counter := v_counter + 1;
      name_varray.EXTEND;
      name_varray(v_counter) := name_rec.name;

      DBMS_OUTPUT.PUT_LINE ('name('||v_counter||'): '||
         name_varray(v_counter));
   END LOOP;
END;
```

In this version of the script, you define a varray of 15 elements. It is important to remember to initialize the array before referencing its individual elements. In addition, the array must be extended before new elements are added to it.

3) Modify the script you just created. Create an additional varray, and populate it with unique course numbers for the courses that each instructor teaches. Display the instructor's name and the list of courses he or she teaches.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR instructor_cur IS
      SELECT instructor_id, first_name||' '||last_name name
        FROM instructor;

   CURSOR course_cur (p_instructor_id NUMBER) IS
      SELECT unique course_no course
        FROM section
       WHERE instructor_id = p_instructor_id;

   TYPE name_type IS VARRAY(15) OF VARCHAR2(50);
   name_varray name_type := name_type();

   TYPE course_type IS VARRAY(10) OF NUMBER;
   course_varray course_type;

   v_counter1 INTEGER := 0;
   v_counter2 INTEGER;
BEGIN
   FOR instructor_rec IN instructor_cur LOOP
      v_counter1 := v_counter1 + 1;
      name_varray.EXTEND;
      name_varray(v_counter1) := instructor_rec.name;

      DBMS_OUTPUT.PUT_LINE ('name('||v_counter1||'): '||
         name_varray(v_counter1));
```

```
      -- Initialize and populate course_varray
      v_counter2 := 0;
      course_varray := course_type();
      FOR course_rec in course_cur (instructor_rec.instructor_id)
      LOOP
         v_counter2 := v_counter2 + 1;
         course_varray.EXTEND;
         course_varray(v_counter2) := course_rec.course;

         DBMS_OUTPUT.PUT_LINE ('course('||v_counter2||'): '||
            course_varray(v_counter2));
      END LOOP;
      DBMS_OUTPUT.PUT_LINE ('============================');
   END LOOP;
END;
```

Consider the script you just created. First, you declare two cursors, INSTRUCTOR_CUR and COURSE_CUR. COURSE_CUR accepts a parameter because it returns a list of courses taught by a particular instructor. Notice that the SELECT statement uses the function UNIQUE to retrieve distinct course numbers. Second, you declare two varray types and variables, `name_varray` and `course_varray`. Notice that you do not initialize the second varray at the time of declaration. Next, you declare two counters and initialize the first counter only.

In the body of the block, you open INSTRUCTOR_CUR and populate `name_varray` with its first element. Next, you initialize the second counter and `course_varray`. This step is necessary because you need to repopulate `course_varray` for the next instructor. Next, you open COURSE_CUR to retrieve corresponding courses and display them on the screen.

When run, the script produces the following output:

```
name(1): Fernand Hanks
course(1): 25
course(2): 120
course(3): 122
course(4): 125
course(5): 134
course(6): 140
course(7): 146
course(8): 240
course(9): 450
============================
name(2): Tom Wojick
course(1): 10
course(2): 25
course(3): 100
course(4): 120
course(5): 124
course(6): 125
course(7): 134
course(8): 140
course(9): 146
course(10): 240
============================
```

```
name(3): Nina Schorin
course(1): 20
course(2): 25
course(3): 100
course(4): 120
course(5): 124
course(6): 130
course(7): 134
course(8): 142
course(9): 147
course(10): 310
============================
name(4): Gary Pertez
course(1): 20
course(2): 25
course(3): 100
course(4): 120
course(5): 124
course(6): 130
course(7): 135
course(8): 142
course(9): 204
course(10): 330
============================
name(5): Anita Morris
course(1): 20
course(2): 25
course(3): 100
course(4): 122
course(5): 124
course(6): 130
course(7): 135
course(8): 142
course(9): 210
course(10): 350
============================
name(6): Todd Smythe
course(1): 20
course(2): 25
course(3): 100
course(4): 122
course(5): 125
course(6): 130
course(7): 135
course(8): 144
course(9): 220
course(10): 350
============================
```

```
name(7): Marilyn Frantzen
course(1): 25
course(2): 120
course(3): 122
course(4): 125
course(5): 132
course(6): 135
course(7): 145
course(8): 230
course(9): 350
============================
name(8): Charles Lowry
course(1): 25
course(2): 120
course(3): 122
course(4): 125
course(5): 132
course(6): 140
course(7): 145
course(8): 230
course(9): 420
============================
name(9): Rick Chow
============================
name(10): Irene Willig
============================

PL/SQL procedure successfully completed.
```

As mentioned, it is important to reinitialize the variable `v_counter2` that is used to reference individual elements of `course_varray`. When this step is omitted and the variable is initialized only once, at the time of declaration, the script generates the following runtime error:

```
name(1): Fernand Hanks
course(1): 25
course(2): 120
course(3): 122
course(4): 125
course(5): 134
course(6): 140
course(7): 146
course(8): 240
course(9): 450
name(2): Tom Wojick
DECLARE
*
ERROR at line 1:
ORA-06533: Subscript beyond count
ORA-06512: at line 33
```

Why do you think this error occurs?

4) Find and explain the errors in the following script:

```
DECLARE
   TYPE varray_type1 IS VARRAY(7) OF INTEGER;
   TYPE table_type2  IS TABLE OF varray_type1 INDEX BY
      BINARY_INTEGER;

   varray1 varray_type1 := varray_type1(1, 2, 3);
   table2  table_type2  := table_type2(varray1,
                                       varray_type1(8, 9, 0));

BEGIN
   DBMS_OUTPUT.PUT_LINE ('table2(1)(2): '||table2(1)(2));

   FOR i IN 1..10 LOOP
      varray1.EXTEND;
      varray1(i) := i;
      DBMS_OUTPUT.PUT_LINE ('varray1('||i||'): '||varray1(i));
   END LOOP;
END;
```

ANSWER: This script generates the following errors:

```
   table2 table_type2 := table_type2(varray1, varray_type1(8, 9, 0));
                         *
ERROR at line 6:
ORA-06550: line 6, column 26:
PLS-00222: no function with name 'TABLE_TYPE2' exists in this scope
ORA-06550: line 6, column 11:
PL/SQL: Item ignored
ORA-06550: line 9, column 44:
PLS-00320: the declaration of the type of this expression is
incomplete or malformed
ORA-06550: line 9, column 4:
PL/SQL: Statement ignored
```

Notice that this error refers to the initialization of `table2`, which has been declared as an associative array of varrays. Recall that associative arrays are not initialized prior to their use. As a result, the declaration of `table2` must be modified. Furthermore, an additional assignment statement must be added to the executable portion of the block:

```
DECLARE
   TYPE varray_type1 IS VARRAY(7) OF INTEGER;
   TYPE table_type2  IS TABLE OF varray_type1 INDEX BY
     BINARY_INTEGER;

   varray1 varray_type1 := varray_type1(1, 2, 3);
   table2  table_type2;
BEGIN
   -- These statements populate associative array
   table2(1) := varray1;
   table2(2) := varray_type1(8, 9, 0);
```

```
   DBMS_OUTPUT.PUT_LINE ('table2(1)(2): '||table2(1)(2));

   FOR i IN 1..10 LOOP
      varray1.EXTEND;
      varray1(i) := i;
      DBMS_OUTPUT.PUT_LINE ('varray1('||i||'): '||varray1(i));
   END LOOP;
END;
```

When run, this version produces a different error:

```
table2(1)(2): 2
varray1(1): 1
varray1(2): 2
varray1(3): 3
varray1(4): 4
DECLARE
*
ERROR at line 1:
ORA-06532: Subscript outside of limit
ORA-06512: at line 15
```

Notice that this is a runtime error that refers to `varray1`. This error occurs because you are trying to extend the varray beyond its limit. `varray1` can contain up to seven integers. After initialization, it contains three integers. As a result, it can be populated with no more than four additional integers. So the fifth iteration of the loop tries to extend the varray to eight elements, which in turn causes a `Subscript outside of limit` error.

It is important to note that there is no correlation between the loop counter and the EXTEND method. Every time the EXTEND method is called, it increases the size of the varray by one element. Because the varray has been initialized to three elements, the EXTEND method adds a fourth element to the array for the first iteration of the loop. At the same time, the first element of the varray is assigned a value of 1 through the loop counter. For the second iteration of the loop, the EXTEND method adds a fifth element to the varray while the second element is assigned a value of 2, and so forth.

Finally, consider the error-free version of the script:

```
DECLARE
   TYPE varray_type1 IS VARRAY(7) OF INTEGER;
   TYPE table_type2  IS TABLE OF varray_type1 INDEX BY
     BINARY_INTEGER;

   varray1 varray_type1 := varray_type1(1, 2, 3);
   table2  table_type2;
BEGIN
   -- These statements populate associative array
   table2(1) := varray1;
   table2(2) := varray_type1(8, 9, 0);

   DBMS_OUTPUT.PUT_LINE ('table2(1)(2): '||table2(1)(2));
```

```
   FOR i IN 4..7 LOOP
      varray1.EXTEND;
      varray1(i) := i;
   END LOOP;

   -- Display elements of the varray
   FOR i IN 1..7 LOOP
      DBMS_OUTPUT.PUT_LINE ('varray1('||i||'): '||varray1(i));
   END LOOP;
END;
```

The output is as follows:

```
table2(1)(2): 2
varray1(1): 1
varray1(2): 2
varray1(3): 3
varray1(4): 4
varray1(5): 5
varray1(6): 6
varray1(7): 7

PL/SQL procedure successfully completed.
```

Chapter 16, "Records"

1) Create an associative array with the element type of a user-defined record. This record should contain the first name, last name, and total number of courses that a particular instructor teaches. Display the records of the associative array on the screen.

   ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR instructor_cur IS
      SELECT first_name, last_name,
             COUNT(UNIQUE s.course_no) courses
        FROM instructor i
        LEFT OUTER JOIN section s
          ON (s.instructor_id = i.instructor_id)
      GROUP BY first_name, last_name;

   TYPE rec_type IS RECORD
      (first_name     instructor.first_name%type,
       last_name      instructor.last_name%type,
       courses_taught NUMBER);

   TYPE instructor_type IS TABLE OF REC_TYPE
INDEX BY BINARY_INTEGER;
```

```
   instructor_tab instructor_type;

   v_counter INTEGER := 0;
BEGIN
   FOR instructor_rec IN instructor_cur LOOP
      v_counter := v_counter + 1;

      -- Populate associative array of records
      instructor_tab(v_counter).first_name :=
         instructor_rec.first_name;
      instructor_tab(v_counter).last_name :=
         instructor_rec.last_name;
      instructor_tab(v_counter).courses_taught :=
         instructor_rec.courses;

      DBMS_OUTPUT.PUT_LINE ('Instructor, '||
         instructor_tab(v_counter).first_name||' '||
         instructor_tab(v_counter).last_name||', teaches '||
         instructor_tab(v_counter).courses_taught||' courses.');
   END LOOP;
END;
```

Consider the SELECT statement used in this script. It returns the instructor's name and the total number of courses he or she teaches. The statement uses an outer join so that if a particular instructor is not teaching any courses, he or she will be included in the results of the SELECT statement. Note that the SELECT statement uses the ANSI 1999 SQL standard.

BY THE WAY

You will find detailed explanations and examples of the statements using the new ANSI 1999 SQL standard in Appendix C and in the Oracle help. Throughout this book we have tried to provide examples illustrating both standards; however, our main focus is on PL/SQL features rather than SQL.

In this script, you define a cursor against the INSTRUCTOR and SECTION tables that is used to populate the associative array of records, `instructor_tab`. Each row of this table is a user-defined record of three elements. You populate the associative array using the cursor FOR loop. Consider the notation used to reference each record element of the associative array:

```
instructor_tab(v_counter).first_name
instructor_tab(v_counter).last_name
instructor_tab(v_counter).courses_taught
```

To reference each row of the associative array, you use the counter variable. However, because each row of this table is a record, you must also reference individual fields of the underlying record. When run, this script produces the following output:

```
Instructor, Anita Morris, teaches 10 courses.
Instructor, Charles Lowry, teaches 9 courses.
Instructor, Fernand Hanks, teaches 9 courses.
Instructor, Gary Pertez, teaches 10 courses.
Instructor, Marilyn Frantzen, teaches 9 courses.
Instructor, Nina Schorin, teaches 10 courses.
Instructor, Rick Chow, teaches 1 courses.
```

```
Instructor, Todd Smythe, teaches 10 courses.
Instructor, Tom Wojick, teaches 9 courses.

PL/SQL procedure successfully completed.
```

2) Modify the script you just created. Instead of using an associative array, use a nested table.

ANSWER: The script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR instructor_cur IS
      SELECT first_name, last_name,
             COUNT(UNIQUE s.course_no) courses
        FROM instructor i
        LEFT OUTER JOIN section s
          ON (s.instructor_id = i.instructor_id)
      GROUP BY first_name, last_name;

   TYPE rec_type IS RECORD
      (first_name     instructor.first_name%type,
       last_name      instructor.last_name%type,
       courses_taught NUMBER);

   TYPE instructor_type IS TABLE OF REC_TYPE;
   instructor_tab instructor_type := instructor_type();

   v_counter INTEGER := 0;
BEGIN
   FOR instructor_rec IN instructor_cur LOOP
      v_counter := v_counter + 1;
      instructor_tab.EXTEND;

      -- Populate associative array of records
      instructor_tab(v_counter).first_name :=
         instructor_rec.first_name;
      instructor_tab(v_counter).last_name :=
         instructor_rec.last_name;
      instructor_tab(v_counter).courses_taught :=
         instructor_rec.courses;

      DBMS_OUTPUT.PUT_LINE ('Instructor, '||
         instructor_tab(v_counter).first_name||' '||
         instructor_tab(v_counter).last_name||', teaches '||
         instructor_tab(v_counter).courses_taught||' courses.');
   END LOOP;
END;
```

Notice that the `instructor_tab` must be initialized and extended before its individual elements can be referenced.

3) Modify the script you just created. Instead of using a nested table, use a varray.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR instructor_cur IS
      SELECT first_name, last_name,
             COUNT(UNIQUE s.course_no) courses
        FROM instructor i
        LEFT OUTER JOIN section s
          ON (s.instructor_id = i.instructor_id)
      GROUP BY first_name, last_name;

   TYPE rec_type IS RECORD
      (first_name     instructor.first_name%type,
       last_name      instructor.last_name%type,
       courses_taught NUMBER);

   TYPE instructor_type IS VARRAY(10) OF REC_TYPE;
   instructor_tab instructor_type := instructor_type();

   v_counter INTEGER := 0;
BEGIN
   FOR instructor_rec IN instructor_cur LOOP
      v_counter := v_counter + 1;
      instructor_tab.EXTEND;

      -- Populate associative array of records
      instructor_tab(v_counter).first_name :=
         instructor_rec.first_name;
      instructor_tab(v_counter).last_name :=
         instructor_rec.last_name;
      instructor_tab(v_counter).courses_taught :=
         instructor_rec.courses;

      DBMS_OUTPUT.PUT_LINE ('Instructor, '||
         instructor_tab(v_counter).first_name||' '||
         instructor_tab(v_counter).last_name||', teaches '||
         instructor_tab(v_counter).courses_taught||' courses.');
   END LOOP;
END;
```

This version of the script is almost identical to the previous version. Instead of using a nested table, you are using a varray of 15 elements.

4) Create a user-defined record with four fields: `course_no`, `description`, `cost`, and `prerequisite_rec`. The last field, `prerequisite_rec`, should be a user-defined record with three fields: `prereq_no`, `prereq_desc`, and `prereq_cost`. For any ten courses that have a prerequisite course, populate the user-defined record with all the corresponding data, and display its information on the screen.

ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR c_cur IS
      SELECT course_no, description, cost, prerequisite
        FROM course
       WHERE prerequisite IS NOT NULL
         AND rownum <= 10;

   TYPE prerequisite_type IS RECORD
      (prereq_no   NUMBER,
       prereq_desc VARCHAR(50),
       prereq_cost NUMBER);

   TYPE course_type IS RECORD
      (course_no        NUMBER,
       description      VARCHAR2(50),
       cost             NUMBER,
       prerequisite_rec PREREQUISITE_TYPE);

   course_rec COURSE_TYPE;
BEGIN
   FOR c_rec in c_cur LOOP
      course_rec.course_no := c_rec.course_no;
      course_rec.description := c_rec.description;
      course_rec.cost := c_rec.cost;

      SELECT course_no, description, cost
        INTO course_rec.prerequisite_rec.prereq_no,
             course_rec.prerequisite_rec.prereq_desc,
             course_rec.prerequisite_rec.prereq_cost
        FROM course
       WHERE course_no = c_rec.prerequisite;

      DBMS_OUTPUT.PUT_LINE ('Course: '||
         course_rec.course_no||' - '||
         course_rec.description);
      DBMS_OUTPUT.PUT_LINE ('Cost: '|| course_rec.cost);
      DBMS_OUTPUT.PUT_LINE ('Prerequisite: '||
         course_rec.prerequisite_rec. prereq_no||' - '||
         course_rec.prerequisite_rec.prereq_desc);
      DBMS_OUTPUT.PUT_LINE ('Prerequisite Cost: '||
         course_rec.prerequisite_rec.prereq_cost);
      DBMS_OUTPUT.PUT_LINE
         ('========================================');
   END LOOP;
END;
```

In the declaration portion of the script, you define a cursor against the COURSE table; two user-defined record types, `prerequisite_type` and `course_type`; and user-defined record, `course_rec`. It is important to note the order in which the record types are declared. The `prerequsite_type` must be declared first because one of the `course_type` elements is of the `prerequisite_type`.

In the executable portion of the script, you populate `course_rec` using the cursor FOR loop. First, you assign values to `course_rec.course_no`, `course_rec.description`, and `course_rec.cost`. Next, you populate the nested record, `prerequsite_rec`, using the SELECT INTO statement against the COURSE table. Consider the notation used to reference individual elements of the nested record:

```
course_rec.prerequisite_rec.prereq_no,
course_rec.prerequisite_rec.prereq_desc,
course_rec.prerequisite_rec.prereq_cost
```

You specify the name of the outer record followed by the name of the inner (nested) record, followed by the name of the element. Finally, you display record information on the screen.

Note that this script does not contain a NO_DATA_FOUND exception handler even though there is a SELECT INTO statement. Why do you think this is the case?

When run, the script produces the following output:

```
Course: 230 - Intro to the Internet
Cost: 1095
Prerequisite: 10 - Technology Concepts
Prerequisite Cost: 1195
=======================================
Course: 100 - Hands-On Windows
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
=======================================
Course: 140 - Systems Analysis
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
=======================================
Course: 142 - Project Management
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
=======================================
Course: 147 - GUI Design Lab
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
=======================================
Course: 204 - Intro to SQL
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
=======================================
```

```
Course: 240 - Intro to the BASIC Language
Cost: 1095
Prerequisite: 25 - Intro to Programming
Prerequisite Cost: 1195
======================================
Course: 420 - Database System Principles
Cost: 1195
Prerequisite: 25 - Intro to Programming
Prerequisite Cost: 1195
======================================
Course: 120 - Intro to Java Programming
Cost: 1195
Prerequisite: 80 - Programming Techniques
Prerequisite Cost: 1595
======================================
Course: 220 - PL/SQL Programming
Cost: 1195
Prerequisite: 80 - Programming Techniques
Prerequisite Cost: 1595
======================================

PL/SQL procedure successfully completed.
```

Chapter 17, "Native Dynamic SQL"

This chapter has no "Try It Yourself" section.

Chapter 18, "Bulk SQL"

Before beginning these exercises, create the MY_SECTION table based on the SECTION table. This table should be created empty.

The MY_SECTION table can be created as follows:

```
CREATE TABLE my_section AS
SELECT *
  FROM section
 WHERE 1 = 2;

Table created.
```

Specifying this criterion guarantees the creation of an empty table.

1) Create the following script: Populate the MY_SECTION table using the FORALL statement with the SAVE EXCEPTIONS clause. After MY_SECTION is populated, display how many records were inserted.

   ANSWER: The script should look similar to the following:

```
SET SERVEROUTPUT ON
DECLARE
   -- Declare collection types
```

```
   TYPE number_type IS TABLE of NUMBER INDEX BY PLS_INTEGER;
   TYPE string_type IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
   TYPE date_type   IS TABLE OF DATE INDEX BY PLS_INTEGER;

   -- Declare collection variables to be used by the FORALL statement
   section_id_tab      number_type;
   course_no_tab       number_type;
   section_no_tab      number_type;
   start_date_time_tab date_type;
   location_tab        string_type;
   instructor_id_tab   number_type;
   capacity_tab        number_type;
   cr_by_tab           string_type;
   cr_date_tab         date_type;
   mod_by_tab          string_type;
   mod_date_tab        date_type;

   v_counter PLS_INTEGER := 0;
   v_total   INTEGER := 0;

   -- Define user-defined exception and associated Oracle
   -- error number with it
   errors EXCEPTION;
   PRAGMA EXCEPTION_INIT(errors, -24381);

BEGIN
   -- Populate individual collections
   FOR rec IN (SELECT *
                 FROM section)
   LOOP
      v_counter := v_counter + 1;
      section_id_tab(v_counter)      := rec.section_id;
      course_no_tab(v_counter)       := rec.course_no;
      section_no_tab(v_counter)      := rec.section_no;
      start_date_time_tab(v_counter) := rec.start_date_time;
      location_tab(v_counter)        := rec.location;
      instructor_id_tab(v_counter)   := rec.instructor_id;
      capacity_tab(v_counter)        := rec.capacity;
      cr_by_tab(v_counter)           := rec.created_by;
      cr_date_tab(v_counter)         := rec.created_date;
      mod_by_tab(v_counter)          := rec.modified_by;
      mod_date_tab(v_counter)        := rec.modified_date;
   END LOOP;

   -- Populate MY_SECTION table
   FORALL i in 1..section_id_tab.COUNT SAVE EXCEPTIONS
      INSERT INTO my_section
         (section_id, course_no, section_no, start_date_time,
          location, instructor_id, capacity, created_by,
          created_date, modified_by, modified_date)
```

```
      VALUES
         (section_id_tab(i), course_no_tab(i), section_no_tab(i),
          start_date_time_tab(i), location_tab(i),
          instructor_id_tab(i), capacity_tab(i), cr_by_tab(i),
          cr_date_tab(i), mod_by_tab(i), mod_date_tab(i));
   COMMIT;

   -- Check how many records were added to MY_SECTION table
   SELECT COUNT(*)
     INTO v_total
     FROM my_section;

   DBMS_OUTPUT.PUT_LINE
      (v_total||' records were added to MY_SECTION table');

EXCEPTION
   WHEN errors THEN
      -- Display total number of exceptions encountered
      DBMS_OUTPUT.PUT_LINE
         ('There were '||SQL%BULK_EXCEPTIONS.COUNT||' exceptions');

      -- Display detailed exception information
      FOR i in 1.. SQL%BULK_EXCEPTIONS.COUNT LOOP
         DBMS_OUTPUT.PUT_LINE ('Record '||
            SQL%BULK_EXCEPTIONS(i).error_index||' caused error '||i||
            ': '||SQL%BULK_EXCEPTIONS(i).ERROR_CODE||' '||
            SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
      END LOOP;

      -- Commit records if any that were inserted successfully
      COMMIT;
END;
```

This script populates the MY_SECTION table with records selected from the SECTION table. To enable use of the FORALL statement, it employs 11 collections. Note that only three collection types are associated with these collections. This is because the individual collections store only three datatypes—NUMBER, VARCHAR2, and DATE.

The script uses a cursor FOR loop to populate the individual collections and then uses them with the FORALL statement with the SAVE EXCEPTIONS option to populate the MY_SECTION table. To enable the SAVE EXCEPTIONS options, this script declares a user-defined exception and associates an Oracle error number with it. This script also contains an exception-handling section where a user-defined exception is processed. This section displays how many exceptions were encountered in the FORALL statement as well as detailed exception information. Note the COMMIT statement in the exception-handling section. This statement is added so that records that are inserted successfully by the FORALL statement are committed when control of the execution is passed to the exception-handling section of the block.

When run, this script produces the following output:

```
78 records were added to MY_SECTION table

PL/SQL procedure successfully completed.
```

2) Modify the script you just created. In addition to displaying the total number of records inserted in the MY_SECTION table, display how many records were inserted for each course. Use the BULK COLLECT statement to accomplish this step. Note that you should delete all the rows from the MY_SECTION table before executing this version of the script.

ANSWER: The new version of the script should look similar to the following. Changes are shown in bold.

```
SET SERVEROUTPUT ON
DECLARE
   -- Declare collection types
   TYPE number_type IS TABLE of NUMBER INDEX BY PLS_INTEGER;
   TYPE string_type IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
   TYPE date_type   IS TABLE OF DATE INDEX BY PLS_INTEGER;

   -- Declare collection variables to be used by the FORALL statement
   section_id_tab      number_type;
   course_no_tab       number_type;
   section_no_tab      number_type;
   start_date_time_tab date_type;
   location_tab        string_type;
   instructor_id_tab   number_type;
   capacity_tab        number_type;
   cr_by_tab           string_type;
   cr_date_tab         date_type;
   mod_by_tab          string_type;
   mod_date_tab        date_type;
   total_recs_tab      number_type;

   v_counter PLS_INTEGER := 0;
   v_total   INTEGER := 0;

   -- Define user-defined exception and associated Oracle
   -- error number with it
   errors EXCEPTION;
   PRAGMA EXCEPTION_INIT(errors, -24381);

BEGIN
   -- Populate individual collections
   FOR rec IN (SELECT *
               FROM section)
   LOOP
      v_counter := v_counter + 1;
      section_id_tab(v_counter)      := rec.section_id;
      course_no_tab(v_counter)       := rec.course_no;
      section_no_tab(v_counter)      := rec.section_no;
      start_date_time_tab(v_counter) := rec.start_date_time;
      location_tab(v_counter)        := rec.location;
      instructor_id_tab(v_counter)   := rec.instructor_id;
      capacity_tab(v_counter)        := rec.capacity;
      cr_by_tab(v_counter)           := rec.created_by;
      cr_date_tab(v_counter)         := rec.created_date;
```

```
      mod_by_tab(v_counter)          := rec.modified_by;
      mod_date_tab(v_counter)        := rec.modified_date;
   END LOOP;

   -- Populate MY_SECTION table
   FORALL i in 1..section_id_tab.COUNT SAVE EXCEPTIONS
      INSERT INTO my_section
         (section_id, course_no, section_no, start_date_time,
          location, instructor_id, capacity, created_by,
          created_date, modified_by, modified_date)
      VALUES
         (section_id_tab(i), course_no_tab(i), section_no_tab(i),
          start_date_time_tab(i), location_tab(i),
          instructor_id_tab(i), capacity_tab(i), cr_by_tab(i),
          cr_date_tab(i), mod_by_tab(i), mod_date_tab(i));
   COMMIT;

   -- Check how many records were added to MY_SECTION table
   SELECT COUNT(*)
     INTO v_total
     FROM my_section;

   DBMS_OUTPUT.PUT_LINE
      (v_total||' records were added to MY_SECTION table');

   -- Check how many records were inserted for each course
   -- and display this information
   -- Fetch data from MY_SECTION table via BULK COLLECT clause
   SELECT course_no, COUNT(*)
     BULK COLLECT INTO course_no_tab, total_recs_tab
     FROM my_section
   GROUP BY course_no;

   IF course_no_tab.COUNT > 0 THEN
      FOR i IN course_no_tab.FIRST..course_no_tab.LAST
      LOOP
         DBMS_OUTPUT.PUT_LINE
         ('course_no: '||course_no_tab(i)||
          ', total sections: '||total_recs_tab(i));
      END LOOP;
   END IF;

EXCEPTION
   WHEN errors THEN
      -- Display total number of exceptions encountered
      DBMS_OUTPUT.PUT_LINE
         ('There were '||SQL%BULK_EXCEPTIONS.COUNT||' exceptions');

      -- Display detailed exception information
      FOR i in 1.. SQL%BULK_EXCEPTIONS.COUNT LOOP
```

```
         DBMS_OUTPUT.PUT_LINE ('Record '||
            SQL%BULK_EXCEPTIONS(i).error_index||' caused error '||i||
            ': '||SQL%BULK_EXCEPTIONS(i).ERROR_CODE||' '||
            SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
      END LOOP;

      -- Commit records if any that were inserted successfully
      COMMIT;
END;
```

In this version of the script, you define one more collection, `total_recs_tab`, in the declaration portion of the PL/SQL block. This collection is used to store the total number of sections for each course. In the executable portion of the PL/SQL block, you add a SELECT statement with a BULK COLLECT clause that repopulates `course_no_tab` and initializes `total_recs_tab`. Next, if the `course_no_tab` collection contains data, you display course numbers and the total number of sections for each course on the screen.

When run, this version of the script produces the following output:

```
78 records were added to MY_SECTION table
course_no: 10, total sections: 1
course_no: 20, total sections: 4
course_no: 25, total sections: 9
course_no: 100, total sections: 5
course_no: 120, total sections: 6
course_no: 122, total sections: 5
course_no: 124, total sections: 4
course_no: 125, total sections: 5
course_no: 130, total sections: 4
course_no: 132, total sections: 2
course_no: 134, total sections: 3
course_no: 135, total sections: 4
course_no: 140, total sections: 3
course_no: 142, total sections: 3
course_no: 144, total sections: 1
course_no: 145, total sections: 2
course_no: 146, total sections: 2
course_no: 147, total sections: 1
course_no: 204, total sections: 1
course_no: 210, total sections: 1
course_no: 220, total sections: 1
course_no: 230, total sections: 2
course_no: 240, total sections: 2
course_no: 310, total sections: 1
course_no: 330, total sections: 1
course_no: 350, total sections: 3
course_no: 420, total sections: 1
course_no: 450, total sections: 1

PL/SQL procedure successfully completed.
```

3) Create the following script: Delete all the records from the MY_SECTION table, and display how many records were deleted for each course as well as individual section IDs deleted for each course. Use BULK COLLECT with the RETURNING option.

ANSWER: This script should look similar to the following:

```
SET SERVEROUTPUT ON;
DECLARE
   -- Define collection types and variables to be used by the
   -- BULK COLLECT clause
   TYPE section_id_type IS TABLE OF my_section.section_id%TYPE;

   section_id_tab section_id_type;

BEGIN
   FOR rec IN (SELECT UNIQUE course_no
                 FROM my_section)
   LOOP
      DELETE FROM MY_SECTION
       WHERE course_no = rec.course_no
      RETURNING section_id
      BULK COLLECT INTO section_id_tab;

      DBMS_OUTPUT.PUT_LINE ('Deleted '||SQL%ROWCOUNT||
         ' rows for course '||rec.course_no);

      IF section_id_tab.COUNT > 0 THEN
         FOR i IN section_id_tab.FIRST..section_id_tab.LAST
         LOOP
            DBMS_OUTPUT.PUT_LINE
               ('section_id: '||section_id_tab(i));
         END LOOP;
         DBMS_OUTPUT.PUT_LINE ('==============================');
      END IF;
      COMMIT;
   END LOOP;
END;
```

In this script you declare a single collection, `section_id_tab`. Note that there is no need to declare a collection to store course numbers. This is because the records from the MY_SECTION table are deleted for each course number instead of all at once. To accomplish this, you introduce a cursor FOR loop that selects unique course numbers from the MY_SECTION table. Next, for each course number, you DELETE records from the MY_SECTION table, returning the corresponding section IDs and collecting them in `section_id_tab`. Next, you display how many records were deleted for a given course number, along with individual section IDs for this course.

Note that even though the collection `section_id_tab` is repopulated for each iteration of the cursor loop, there is no need to reinitialize it (in other words, empty it). This is because the DELETE statement does this implicitly.

Consider the partial output produced by this script:

```
Deleted 1 rows for course 10
section_id: 80
================================
Deleted 4 rows for course 20
section_id: 81
section_id: 82
section_id: 83
section_id: 84
================================
Deleted 9 rows for course 25
section_id: 85
section_id: 86
section_id: 87
section_id: 88
section_id: 89
section_id: 90
section_id: 91
section_id: 92
section_id: 93
================================
Deleted 5 rows for course 100
section_id: 141
section_id: 142
section_id: 143
section_id: 144
section_id: 145
================================
Deleted 6 rows for course 120
section_id: 146
section_id: 147
section_id: 148
section_id: 149
section_id: 150
section_id: 151
================================
Deleted 5 rows for course 122
section_id: 152
section_id: 153
section_id: 154
section_id: 155
section_id: 156
================================
...

PL/SQL procedure successfully completed.
```

Chapter 19, "Procedures"

PART 1

1) Write a procedure with no parameters. The procedure should say whether the current day is a weekend or weekday. Additionally, it should tell you the user's name and the current time. It also should specify how many valid and invalid procedures are in the database.

ANSWER: The procedure should look similar to the following:

```
CREATE OR REPLACE PROCEDURE current_status
AS
   v_day_type CHAR(1);
   v_user     VARCHAR2(30);
   v_valid    NUMBER;
   v_invalid  NUMBER;
BEGIN
   SELECT SUBSTR(TO_CHAR(sysdate, 'DAY'), 0, 1)
     INTO v_day_type
     FROM dual;

   IF v_day_type = 'S' THEN
      DBMS_OUTPUT.PUT_LINE ('Today is a weekend.');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('Today is a weekday.');
   END IF;
   --
   DBMS_OUTPUT.PUT_LINE('The time is: '||
      TO_CHAR(sysdate, 'HH:MI AM'));
   --
   SELECT user
     INTO v_user
     FROM dual;
   DBMS_OUTPUT.PUT_LINE ('The current user is '||v_user);
   --
   SELECT NVL(COUNT(*), 0)
     INTO v_valid
     FROM user_objects
    WHERE status = 'VALID'
      AND object_type = 'PROCEDURE';
   DBMS_OUTPUT.PUT_LINE
      ('There are '||v_valid||' valid procedures.');
   --
   SELECT NVL(COUNT(*), 0)
     INTO v_invalid
     FROM user_objects
    WHERE status = 'INVALID'
      AND object_type = 'PROCEDURE';
```

```
   DBMS_OUTPUT.PUT_LINE
      ('There are '||v_invalid||' invalid procedures.');
END;

SET SERVEROUTPUT ON
EXEC current_status;
```

2) Write a procedure that takes in a zip code, city, and state and inserts the values into the zip code table. It should check to see if the zip code is already in the database. If it is, an exception should be raised, and an error message should be displayed. Write an anonymous block that uses the procedure and inserts your zip code.

ANSWER: The script should look similar to the following:

```
CREATE OR REPLACE PROCEDURE insert_zip
  (I_ZIPCODE IN zipcode.zip%TYPE,
   I_CITY    IN zipcode.city%TYPE,
   I_STATE   IN zipcode.state%TYPE)
AS
   v_zipcode zipcode.zip%TYPE;
   v_city    zipcode.city%TYPE;
   v_state   zipcode.state%TYPE;
   v_dummy   zipcode.zip%TYPE;
BEGIN
   v_zipcode := i_zipcode;
   v_city    := i_city;
   v_state   := i_state;
--
   SELECT zip
     INTO v_dummy
     FROM zipcode
    WHERE zip = v_zipcode;
--
   DBMS_OUTPUT.PUT_LINE('The zipcode '||v_zipcode||
      ' is already in the database and cannot be'||
      ' reinserted.');
--
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO ZIPCODE
      VALUES (v_zipcode, v_city, v_state, user, sysdate,
              user, sysdate);
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ('There was an unknown error '||
         'in insert_zip.');
END;

SET SERVEROUTPUT ON
BEGIN
  insert_zip (10035, 'No Where', 'ZZ');
END;
```

```
BEGIN
  insert_zip (99999, 'No Where', 'ZZ');
END;

ROLLBACK;
```

PART 2

1) Create a stored procedure based on the script ch17_1c.sql, version 3.0, created in Lab 17.1 of Chapter 17. The procedure should accept two parameters to hold a table name and an ID and should return six parameters with first name, last name, street, city, state, and zip code information.

ANSWER: The procedure should look similar to the following. Changes are shown in bold.

```
CREATE OR REPLACE PROCEDURE get_name_address
   (table_name_in   IN VARCHAR2
   ,id_in           IN NUMBER
   ,first_name_out OUT VARCHAR2
   ,last_name_out  OUT VARCHAR2
   ,street_out     OUT VARCHAR2
   ,city_out       OUT VARCHAR2
   ,state_out      OUT VARCHAR2
   ,zip_out        OUT VARCHAR2)
AS
   sql_stmt VARCHAR2(200);
BEGIN
   sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address'||
               '      ,b.city, b.state, b.zip'                    ||
               '  FROM '||table_name_in||' a, zipcode b'          ||
               ' WHERE a.zip = b.zip'                             ||
               '   AND '||table_name_in||'_id = :1';
   EXECUTE IMMEDIATE sql_stmt
   INTO first_name_out, last_name_out, street_out, city_out,
        state_out, zip_out
   USING id_in;
END get_name_address;
```

This procedure contains two IN parameters whose values are used by the dynamic SQL statement and six OUT parameters that hold data returned by the SELECT statement. After it is created, this procedure can be tested with the following PL/SQL block:

```
SET SERVEROUTPUT ON
DECLARE
   v_table_name VARCHAR2(20) := '&sv_table_name';
   v_id         NUMBER := &sv_id;
   v_first_name VARCHAR2(25);
   v_last_name  VARCHAR2(25);
   v_street     VARCHAR2(50);
   v_city       VARCHAR2(25);
   v_state      VARCHAR2(2);
   v_zip        VARCHAR2(5);
```

```
BEGIN
   get_name_address (v_table_name, v_id, v_first_name, v_last_name,
                     v_street, v_city, v_state, v_zip);

   DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
   DBMS_OUTPUT.PUT_LINE ('Last Name:  '||v_last_name);
   DBMS_OUTPUT.PUT_LINE ('Street:     '||v_street);
   DBMS_OUTPUT.PUT_LINE ('City:       '||v_city);
   DBMS_OUTPUT.PUT_LINE ('State:      '||v_state);
   DBMS_OUTPUT.PUT_LINE ('Zip Code:   '||v_zip);
END;
```

When run, this script produces the following output. The first run is against the STUDENT table, and the second run is against the INSTRUCTOR table.

```
Enter value for sv_table_name: student
old   2:    v_table_name VARCHAR2(20) := '&sv_table_name';
new   2:    v_table_name VARCHAR2(20) := 'student';
Enter value for sv_id: 105
old   3:    v_id NUMBER := &sv_id;
new   3:    v_id NUMBER := 105;
First Name: Angel
Last Name:  Moskowitz
Street:     320 John St.
City:       Ft. Lee
State:      NJ
Zip Code:   07024

PL/SQL procedure successfully completed.

Enter value for sv_table_name: instructor
old   2:    v_table_name VARCHAR2(20) := '&sv_table_name';
new   2:    v_table_name VARCHAR2(20) := 'instructor';
Enter value for sv_id: 105
old   3:    v_id NUMBER := &sv_id;
new   3:    v_id NUMBER := 105;
First Name: Anita
Last Name:  Morris
Street:     34 Maiden Lane
City:       New York
State:      NY
Zip Code:   10015

PL/SQL procedure successfully completed.
```

2) Modify the procedure you just created. Instead of using six parameters to hold name and address information, the procedure should return a user-defined record that contains six fields that hold name and address information. Note: You may want to create a package in which you define a record type. This record may be used later, such as when the procedure is invoked in a PL/SQL block.

ANSWER: The package should look similar to the following. Changes are shown in bold.

```
CREATE OR REPLACE PACKAGE dynamic_sql_pkg
AS
   -- Create user-defined record type
   TYPE name_addr_rec_type IS RECORD
      (first_name VARCHAR2(25),
       last_name  VARCHAR2(25),
       street     VARCHAR2(50),
       city       VARCHAR2(25),
       state      VARCHAR2(2),
       zip        VARCHAR2(5));

   PROCEDURE get_name_address (table_name_in  IN VARCHAR2
                              ,id_in          IN NUMBER
                              ,name_addr_rec OUT name_addr_rec_type);
END dynamic_sql_pkg;
/

CREATE OR REPLACE PACKAGE BODY dynamic_sql_pkg AS

PROCEDURE get_name_address (table_name_in  IN VARCHAR2
                           ,id_in          IN NUMBER
                           ,name_addr_rec OUT name_addr_rec_type)
IS
   sql_stmt VARCHAR2(200);
BEGIN
   sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address'||
               '      ,b.city, b.state, b.zip'                     ||
               '  FROM '||table_name_in||' a, zipcode b'           ||
               ' WHERE a.zip = b.zip'                              ||
               '   AND '||table_name_in||'_id = :1';
   EXECUTE IMMEDIATE sql_stmt
   INTO name_addr_rec
   USING id_in;
END get_name_address;

END dynamic_sql_pkg;
/
```

In this package specification, you declare a user-defined record type. The procedure uses this record type for its OUT parameter, `name_addr_rec`. After the package is created, its procedure can be tested with the following PL/SQL block (changes are shown in bold):

```
SET SERVEROUTPUT ON
DECLARE
   v_table_name VARCHAR2(20) := '&sv_table_name';
   v_id NUMBER := &sv_id;
   name_addr_rec DYNAMIC_SQL_PKG.NAME_ADDR_REC_TYPE;
```

```
BEGIN
   dynamic_sql_pkg.get_name_address (v_table_name, v_id,
                                     name_addr_rec);

   DBMS_OUTPUT.PUT_LINE ('First Name: '||name_addr_rec.first_name);
   DBMS_OUTPUT.PUT_LINE ('Last Name:  '||name_addr_rec.last_name);
   DBMS_OUTPUT.PUT_LINE ('Street:     '||name_addr_rec.street);
   DBMS_OUTPUT.PUT_LINE ('City:       '||name_addr_rec.city);
   DBMS_OUTPUT.PUT_LINE ('State:      '||name_addr_rec.state);
   DBMS_OUTPUT.PUT_LINE ('Zip Code:   '||name_addr_rec.zip);

END;
```

Notice that instead of declaring six variables, you declare one variable of the user-defined record type, `name_addr_rec_type`. Because this record type is defined in the package DYNAMIC_SQL_PKG, the name of the record type is prefixed with the name of the package. Similarly, the name of the package is added to the procedure call statement.

When run, this script produces the following output. The first output is against the STUDENT table, and the second output is against the INSTRUCTOR table.

```
Enter value for sv_table_name: student
old   2:    v_table_name VARCHAR2(20) := '&sv_table_name';
new   2:    v_table_name VARCHAR2(20) := 'student';
Enter value for sv_id: 105
old   3:    v_id NUMBER := &sv_id;
new   3:    v_id NUMBER := 105;
First Name: Angel
Last Name:  Moskowitz
Street:     320 John St.
City:       Ft. Lee
State:      NJ
Zip Code:   07024

PL/SQL procedure successfully completed.

Enter value for sv_table_name: instructor
old   2:    v_table_name VARCHAR2(20) := '&sv_table_name';
new   2:    v_table_name VARCHAR2(20) := 'instructor';
Enter value for sv_id: 105
old   3:    v_id NUMBER := &sv_id;
new   3:    v_id NUMBER := 105;
First Name: Anita
Last Name:  Morris
Street:     34 Maiden Lane
City:       New York
State:      NY
Zip Code:   10015

PL/SQL procedure successfully completed.
```

Chapter 20, "Functions"

1) Write a stored function called `new_student_id` that takes in no parameters and returns a `student.student_id%TYPE`. The value returned will be used when inserting a new student into the CTA application. It will be derived by using the formula `student_id_seq.NEXTVAL`.

   ANSWER: The function should look similar to the following:

```
CREATE OR REPLACE FUNCTION new_student_id
RETURN student.student_id%TYPE
AS
   v_student_id student.student_id%TYPE;
BEGIN
   SELECT student_id_seq.NEXTVAL
     INTO v_student_id
     FROM dual;
   RETURN(v_student_id);
END;
```

2) Write a stored function called `zip_does_not_exist` that takes in a `zipcode.zip%TYPE` and returns a Boolean. The function will return TRUE if the zip code passed into it does not exist. It will return a FALSE if the zip code does exist. Hint: Here's an example of how this might be used:

```
DECLARE
   cons_zip CONSTANT zipcode.zip%TYPE := '&sv_zipcode';
   e_zipcode_is_not_valid EXCEPTION;
BEGIN
   IF zipcode_does_not_exist(cons_zip)
   THEN
      RAISE e_zipcode_is_not_valid;
   ELSE
      -- An insert of an instructor's record which
      -- makes use of the checked zipcode might go here.
      NULL;
   END IF;
EXCEPTION
   WHEN e_zipcode_is_not_valid THEN
      RAISE_APPLICATION_ERROR
         (-20003, 'Could not find zipcode '||cons_zip||'.');
END;
```

   ANSWER: The function should look similar to the following:

```
CREATE OR REPLACE FUNCTION zipcode_does_not_exist
   (i_zipcode IN zipcode.zip%TYPE)
RETURN BOOLEAN
AS
   v_dummy char(1);
BEGIN
   SELECT NULL
     INTO v_dummy
```

```
      FROM zipcode
     WHERE zip = i_zipcode;

   -- Meaning the zipcode does exit
   RETURN FALSE;
EXCEPTION
   WHEN OTHERS THEN
      -- The select statement above will cause an exception
      -- to be raised if the zipcode is not in the database.
      RETURN TRUE;
END zipcode_does_not_exist;
```

3) Create a new function. For a given instructor, determine how many sections he or she is teaching. If the number is greater than or equal to 3, return a message saying that the instructor needs a vacation. Otherwise, return a message saying how many sections this instructor is teaching.

ANSWER: The function should look similar to the following:

```
CREATE OR REPLACE FUNCTION instructor_status
   (i_first_name IN instructor.first_name%TYPE,
    i_last_name  IN instructor.last_name%TYPE)
RETURN VARCHAR2
AS
   v_instructor_id instructor.instructor_id%TYPE;
   v_section_count NUMBER;
   v_status        VARCHAR2(100);
BEGIN
   SELECT instructor_id
     INTO v_instructor_id
     FROM instructor
    WHERE first_name = i_first_name
      AND last_name = i_last_name;

   SELECT COUNT(*)
     INTO v_section_count
     FROM section
    WHERE instructor_id = v_instructor_id;

   IF v_section_count >= 3 THEN
      v_status :=
         'The instructor '||i_first_name||'  '||
          i_last_name||' is teaching '||v_section_count||
         ' and needs a vaction.';
   ELSE
      v_status :=
         'The instructor '||i_first_name||'  '||
         i_last_name||' is teaching '||v_section_count||
         ' courses.';
   END IF;
   RETURN v_status;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
```

```
      -- Note that either of the SELECT statements can raise
      -- this exception
      v_status :=
         'The instructor '||i_first_name||'  '||
          i_last_name||' is not shown to be teaching'||
         ' any courses.';
      RETURN v_status;
   WHEN OTHERS THEN
      v_status :=
         'There has been in an error in the function.';
      RETURN v_status;
END;
```

Test the function as follows:

```
SELECT instructor_status(first_name, last_name)
  FROM instructor;
/
```

Chapter 21, "Packages"

1) Add a procedure to the `student_api` package called `remove_student`. This procedure accepts a `student_id` and returns nothing. Based on the student ID passed in, it removes the student from the database. If the student does not exist or if a problem occurs while removing the student (such as a foreign key constraint violation), let the calling program handle it.

   ANSWER: The package should be similar to the following:

```
CREATE OR REPLACE PACKAGE student_api AS
   v_current_date DATE;

   PROCEDURE discount;

   FUNCTION new_instructor_id
   RETURN instructor.instructor_id%TYPE;

   FUNCTION total_cost_for_student
      (p_student_id IN student.student_id%TYPE)
   RETURN course.cost%TYPE;
   PRAGMA RESTRICT_REFERENCES
      (total_cost_for_student, WNDS, WNPS, RNPS);

   PROCEDURE get_student_info
      (p_student_id   IN student.student_id%TYPE,
       p_last_name   OUT student.last_name%TYPE,
       p_first_name  OUT student.first_name%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE get_student_info
      (p_last_name    IN student.last_name%TYPE,
       p_first_name   IN student.first_name%TYPE,
```

```
       p_student_id  OUT student.student_id%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE remove_student
      (p_studid IN student.student_id%TYPE);
END student_api;
/
CREATE OR REPLACE PACKAGE BODY student_api AS

PROCEDURE discount
IS
   CURSOR c_group_discount IS
      SELECT distinct s.course_no, c.description
        FROM section s, enrollment e, course c
       WHERE s.section_id = e.section_id
      GROUP BY s.course_no, c.description,
               e.section_id, s.section_id
      HAVING COUNT(*) >=8;
BEGIN
   FOR r_group_discount IN c_group_discount LOOP
      UPDATE course
         SET cost = cost * .95
       WHERE course_no = r_group_discount.course_no;

      DBMS_OUTPUT.PUT_LINE
         ('A 5% discount has been given to'||
          r_group_discount.course_no||' '||
          r_group_discount.description);
   END LOOP;
END discount;

FUNCTION new_instructor_id
RETURN instructor.instructor_id%TYPE
IS
   v_new_instid instructor.instructor_id%TYPE;
BEGIN
   SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
     INTO v_new_instid
     FROM dual;
   RETURN v_new_instid;
EXCEPTION
   WHEN OTHERS THEN
      DECLARE
         v_sqlerrm VARCHAR2(250) := SUBSTR(SQLERRM,1,250);
      BEGIN
         RAISE_APPLICATION_ERROR
            (-20003, 'Error in instructor_id: '||v_sqlerrm);
      END;
END new_instructor_id;
```

```
FUNCTION get_course_descript_private
   (p_course_no  course.course_no%TYPE)
RETURN course.description%TYPE
IS
   v_course_descript course.description%TYPE;
BEGIN
   SELECT description
     INTO v_course_descript
     FROM course
    WHERE course_no = p_course_no;
   RETURN v_course_descript;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
END get_course_descript_private;

FUNCTION total_cost_for_student
   (p_student_id IN student.student_id%TYPE)
RETURN course.cost%TYPE
IS
   v_cost course.cost%TYPE;
BEGIN
   SELECT sum(cost)
     INTO v_cost
     FROM course c, section s, enrollment e
    WHERE c.course_no = c.course_no
      AND e.section_id = s.section_id
      AND e.student_id = p_student_id;
   RETURN v_cost;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
END total_cost_for_student;

PROCEDURE get_student_info
   (p_student_id   IN  student.student_id%TYPE,
    p_last_name   OUT student.last_name%TYPE,
    p_first_name  OUT student.first_name%TYPE,
    p_zip         OUT student.zip%TYPE,
    p_return_code OUT NUMBER)
IS
BEGIN
   SELECT last_name, first_name, zip
     INTO p_last_name, p_first_name, p_zip
     FROM student
    WHERE student.student_id = p_student_id;
   p_return_code := 0;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Student ID is not valid.');
```

```
      p_return_code := -100;
      p_last_name := NULL;
      p_first_name := NULL;
      p_zip := NULL;
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
         ('Error in procedure get_student_info');
END get_student_info;

PROCEDURE get_student_info
   (p_last_name    IN student.last_name%TYPE,
    p_first_name   IN student.first_name%TYPE,
    p_student_id  OUT student.student_id%TYPE,
    p_zip         OUT student.zip%TYPE,
    p_return_code OUT NUMBER)
IS
BEGIN
   SELECT student_id, zip
     INTO p_student_id, p_zip
     FROM student
    WHERE UPPER(last_name) = UPPER(p_last_name)
      AND UPPER(first_name) = UPPER(p_first_name);
   p_return_code := 0;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Student name is not valid.');
      p_return_code := -100;
      p_student_id := NULL;
      p_zip := NULL;
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
        ('Error in procedure get_student_info');
END get_student_info;

PROCEDURE remove_student
   (p_studid IN student.student_id%TYPE)
IS
BEGIN
   DELETE
     FROM STUDENT
    WHERE student_id = p_studid;
END;

BEGIN
   SELECT trunc(sysdate, 'DD')
     INTO v_current_date
     FROM dual;
END student_api;
/
```

2) Alter `remove_student` in the `student_api` package body to accept an additional parameter. This new parameter should be a VARCHAR2 and called `p_ri`. Make `p_ri` default to R. The new parameter may contain a value of R or C. If R is received, it represents DELETE RESTRICT, and the procedure acts as it does now. If there are enrollments for the student, the delete is disallowed. If a C is received, it represents DELETE CASCADE. This functionally means that the `remove_student` procedure locates all records for the student in all the tables. It removes them from the database before attempting to remove the student from the student table. Decide how to handle the situation when the user passes in a code other than C or R.

ANSWER: The package should look similar to the following:

```
CREATE OR REPLACE PACKAGE student_api AS
   v_current_date DATE;

   PROCEDURE discount;

   FUNCTION new_instructor_id
      RETURN instructor.instructor_id%TYPE;

   FUNCTION total_cost_for_student
      (p_student_id IN student.student_id%TYPE)
   RETURN course.cost%TYPE;
   PRAGMA RESTRICT_REFERENCES
      (total_cost_for_student, WNDS, WNPS, RNPS);

   PROCEDURE get_student_info
      (p_student_id   IN  student.student_id%TYPE,
       p_last_name   OUT student.last_name%TYPE,
       p_first_name  OUT student.first_name%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE get_student_info
      (p_last_name    IN student.last_name%TYPE,
       p_first_name   IN student.first_name%TYPE,
       p_student_id  OUT student.student_id%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE remove_student
      (p_studid IN student.student_id%TYPE,
       p_ri     IN VARCHAR2 DEFAULT 'R');
END student_api;
/

CREATE OR REPLACE PACKAGE BODY student_api AS

PROCEDURE discount
IS
   CURSOR c_group_discount IS
      SELECT distinct s.course_no, c.description
        FROM section s, enrollment e, course c
```

```
       WHERE s.section_id = e.section_id
      GROUP BY s.course_no, c.description,
               e.section_id, s.section_id
      HAVING COUNT(*) >=8;
BEGIN
   FOR r_group_discount IN c_group_discount LOOP
      UPDATE course
         SET cost = cost * .95
       WHERE course_no = r_group_discount.course_no;

      DBMS_OUTPUT.PUT_LINE
         ('A 5% discount has been given to'||
          r_group_discount.course_no||' '||
          r_group_discount.description);
   END LOOP;
END discount;

FUNCTION new_instructor_id
RETURN instructor.instructor_id%TYPE
IS
   v_new_instid instructor.instructor_id%TYPE;
BEGIN
   SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
     INTO v_new_instid
     FROM dual;
   RETURN v_new_instid;
EXCEPTION
   WHEN OTHERS THEN
      DECLARE
         v_sqlerrm VARCHAR2(250) := SUBSTR(SQLERRM,1,250);
      BEGIN
         RAISE_APPLICATION_ERROR
            (-20003, 'Error in instructor_id: '||v_sqlerrm);
      END;
END new_instructor_id;

FUNCTION get_course_descript_private
   (p_course_no course.course_no%TYPE)
RETURN course.description%TYPE
IS
   v_course_descript course.description%TYPE;
BEGIN
   SELECT description
     INTO v_course_descript
     FROM course
    WHERE course_no = p_course_no;
   RETURN v_course_descript;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
```

```
END get_course_descript_private;

FUNCTION total_cost_for_student
   (p_student_id IN student.student_id%TYPE)
RETURN course.cost%TYPE
IS
   v_cost course.cost%TYPE;
BEGIN
   SELECT sum(cost)
     INTO v_cost
     FROM course c, section s, enrollment e
    WHERE c.course_no = c.course_no
      AND e.section_id = s.section_id
      AND e.student_id = p_student_id;
   RETURN v_cost;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
END total_cost_for_student;

PROCEDURE get_student_info
   (p_student_id   IN  student.student_id%TYPE,
    p_last_name   OUT student.last_name%TYPE,
    p_first_name  OUT student.first_name%TYPE,
    p_zip         OUT student.zip%TYPE,
    p_return_code OUT NUMBER)
IS
BEGIN
   SELECT last_name, first_name, zip
     INTO p_last_name, p_first_name, p_zip
     FROM student
    WHERE student.student_id = p_student_id;
   p_return_code := 0;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Student ID is not valid.');
      p_return_code := -100;
      p_last_name := NULL;
      p_first_name := NULL;
      p_zip := NULL;
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
         ('Error in procedure get_student_info');
END get_student_info;

PROCEDURE get_student_info
   (p_last_name    IN student.last_name%TYPE,
    p_first_name   IN student.first_name%TYPE,
    p_student_id  OUT student.student_id%TYPE,
    p_zip         OUT student.zip%TYPE,
    p_return_code OUT NUMBER)
```

```
IS
BEGIN
   SELECT student_id, zip
     INTO p_student_id, p_zip
     FROM student
    WHERE UPPER(last_name) = UPPER(p_last_name)
      AND UPPER(first_name) = UPPER(p_first_name);
   p_return_code := 0;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE
         ('Student name is not valid.');
      p_return_code := -100;
      p_student_id := NULL;
      p_zip := NULL;
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
         ('Error in procedure get_student_info');
END get_student_info;

PROCEDURE remove_student
   -- The parameters student_id and p_ri give the user an
   -- option of cascade delete or restrict delete for
   -- the given student's records
   (p_studid IN student.student_id%TYPE,
    p_ri     IN VARCHAR2 DEFAULT 'R')
IS
   -- Declare exceptions for use in procedure
   enrollment_present EXCEPTION;
   bad_pri EXCEPTION;
BEGIN
   -- R value is for restrict delete option
   IF p_ri = 'R' THEN
      DECLARE
         -- A variable is needed to test if the student
         -- is in the enrollment table
         v_dummy CHAR(1);
      BEGIN
         -- This is a standard existence check.
         -- If v_dummy is assigned a value via the
         -- SELECT INTO, the exception
         -- enrollment_present will be raised.
         -- If the v_dummy is not assigned a value, the
         -- exception no_data_found will be raised.
         SELECT NULL
           INTO v_dummy
           FROM enrollment e
          WHERE e.student_id = p_studid
            AND ROWNUM = 1;
```

```
         -- The rownum set to 1 prevents the SELECT
         -- INTO statement raise to_many_rows
         -- exception.
         -- If there is at least one row in the enrollment
         -- table with a corresponding student_id, the
         -- restrict delete parameter will disallow the
         -- deletion of the student by raising
         -- the enrollment_present exception.
         RAISE enrollment_present;
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
            -- The no_data_found exception is raised
            -- when there are no students found in the
            -- enrollment table. Since the p_ri indicates
            -- a restrict delete user choice the delete
            -- operation is permitted.
            DELETE FROM student
             WHERE student_id = p_studid;
      END;
   -- When the user enters "C" for the p_ri
   -- he/she indicates a cascade delete choice
   ELSIF p_ri = 'C' THEN
      -- Delete the student from the enrollment and
      -- grade tables
      DELETE FROM enrollment
       WHERE student_id = p_studid;

      DELETE FROM grade
       WHERE student_id = p_studid;

      -- Delete from student table only after corresponding
      -- records have been removed from the other tables
      -- because the student table is the parent table
      DELETE FROM student
       WHERE student_id = p_studid;
   ELSE
      RAISE bad_pri;
   END IF;
EXCEPTION
   WHEN bad_pri THEN
      RAISE_APPLICATION_ERROR
         (-20231, 'An incorrect p_ri value was '||
          'entered. The remove_student procedure can '||
          'only accept a C or R for the p_ri parameter.');

   WHEN enrollment_present THEN
      RAISE_APPLICATION_ERROR
         (-20239, 'The student with ID'||p_studid||
          ' exists in the enrollment table thus records'||
          ' will not be removed.');
```

```
END remove_student;

BEGIN
   SELECT trunc(sysdate, 'DD')
     INTO v_current_date
     FROM dual;
END student_api;
```

Chapter 22, "Stored Code"

1) Add a function to the `student_api` package specification called `get_course_descript`. The caller takes a `course.cnumber%TYPE` parameter, and it returns a `course.description%TYPE`.

   ANSWER: The package should look similar to the following:

```
CREATE OR REPLACE PACKAGE student_api AS
   v_current_date DATE;

   PROCEDURE discount;

   FUNCTION new_instructor_id
   RETURN instructor.instructor_id%TYPE;

   FUNCTION total_cost_for_student
      (p_student_id IN student.student_id%TYPE)
   RETURN course.cost%TYPE;
   PRAGMA RESTRICT_REFERENCES
      (total_cost_for_student, WNDS, WNPS, RNPS);

   PROCEDURE get_student_info
      (p_student_id   IN student.student_id%TYPE,
       p_last_name   OUT student.last_name%TYPE,
       p_first_name  OUT student.first_name%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE get_student_info
      (p_last_name    IN student.last_name%TYPE,
       p_first_name   IN student.first_name%TYPE,
       p_student_id  OUT student.student_id%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE remove_student
      (p_studid IN student.student_id%TYPE,
       p_ri     IN VARCHAR2 DEFAULT 'R');

   FUNCTION get_course_descript
      (p_cnumber course.course_no%TYPE)
   RETURN course.description%TYPE;
END student_api;
```

2) Create a function in the `student_api` package body called `get_course_description`. A caller passes in a course number, and it returns the course description. Instead of searching for the description itself, it makes a call to `get_course_descript_private`. It passes its course number to `get_course_descript_private`. It passes back to the caller the description it gets back from `get_course_descript_private`.

ANSWER: The package body should look similar to the following:

```
CREATE OR REPLACE PACKAGE BODY student_api AS

PROCEDURE discount
IS
   CURSOR c_group_discount IS
      SELECT distinct s.course_no, c.description
        FROM section s, enrollment e, course c
       WHERE s.section_id = e.section_id
      GROUP BY s.course_no, c.description,
               e.section_id, s.section_id
      HAVING COUNT(*) >=8;
BEGIN
   FOR r_group_discount IN c_group_discount LOOP
      UPDATE course
         SET cost = cost * .95
       WHERE course_no = r_group_discount.course_no;

      DBMS_OUTPUT.PUT_LINE
         ('A 5% discount has been given to'||
          r_group_discount.course_no||' '||
          r_group_discount.description);
   END LOOP;
END discount;

FUNCTION new_instructor_id
RETURN instructor.instructor_id%TYPE
IS
   v_new_instid instructor.instructor_id%TYPE;
BEGIN
   SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
     INTO v_new_instid
     FROM dual;
   RETURN v_new_instid;
EXCEPTION
   WHEN OTHERS THEN
      DECLARE
         v_sqlerrm VARCHAR2(250) := SUBSTR(SQLERRM,1,250);
      BEGIN
         RAISE_APPLICATION_ERROR
            (-20003, 'Error in instructor_id: '||v_sqlerrm);
      END;
END new_instructor_id;
```

```
FUNCTION get_course_descript_private
   (p_course_no  course.course_no%TYPE)
RETURN course.description%TYPE
IS
   v_course_descript course.description%TYPE;
BEGIN
   SELECT description
     INTO v_course_descript
     FROM course
    WHERE course_no = p_course_no;
   RETURN v_course_descript;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
END get_course_descript_private;

FUNCTION total_cost_for_student
   (p_student_id IN student.student_id%TYPE)
RETURN course.cost%TYPE
IS
   v_cost course.cost%TYPE;
BEGIN
   SELECT sum(cost)
     INTO v_cost
     FROM course c, section s, enrollment e
    WHERE c.course_no = c.course_no
      AND e.section_id = s.section_id
      AND e.student_id = p_student_id;
   RETURN v_cost;
EXCEPTION
   WHEN OTHERS THEN
      RETURN NULL;
END total_cost_for_student;

PROCEDURE get_student_info
   (p_student_id   IN student.student_id%TYPE,
    p_last_name   OUT student.last_name%TYPE,
    p_first_name  OUT student.first_name%TYPE,
    p_zip         OUT student.zip%TYPE,
    p_return_code OUT NUMBER)
IS
BEGIN
   SELECT last_name, first_name, zip
     INTO p_last_name, p_first_name, p_zip
     FROM student
    WHERE student.student_id = p_student_id;
   p_return_code := 0;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Student ID is not valid.');
```

```
      p_return_code := -100;
      p_last_name   := NULL;
      p_first_name  := NULL;
      p_zip         := NULL;

   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
         ('Error in procedure get_student_info');
END get_student_info;

PROCEDURE get_student_info
   (p_last_name    IN student.last_name%TYPE,
    p_first_name   IN student.first_name%TYPE,
    p_student_id  OUT student.student_id%TYPE,
    p_zip         OUT student.zip%TYPE,
    p_return_code OUT NUMBER)
IS
BEGIN
   SELECT student_id, zip
     INTO p_student_id, p_zip
     FROM student
    WHERE UPPER(last_name)  = UPPER(p_last_name)
      AND UPPER(first_name) = UPPER(p_first_name);
   p_return_code := 0;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Student name is not valid.');
      p_return_code := -100;
      p_student_id  := NULL;
      p_zip         := NULL;

   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
         ('Error in procedure get_student_info');
END get_student_info;

PROCEDURE remove_student
   -- The parameters student_id and p_ri give the user an
   -- option of cascade delete or restrict delete for
   -- the given student's records
   (p_studid IN student.student_id%TYPE,
    p_ri     IN VARCHAR2 DEFAULT 'R')
IS
   -- Declare exceptions for use in procedure
   enrollment_present EXCEPTION;
   bad_pri EXCEPTION;
BEGIN
   -- The R value is for restrict delete option
   IF p_ri = 'R' THEN
      DECLARE
```

```
      -- A variable is needed to test if the student
      -- is in the enrollment table
      v_dummy CHAR(1);
   BEGIN
      -- This is a standard existence check.
      -- If v_dummy is assigned a value via the
      -- SELECT INTO, the exception
      -- enrollment_present will be raised.
      -- If the v_dummy is not assigned a value, the
      -- exception no_data_found will be raised.
      SELECT NULL
        INTO v_dummy
        FROM enrollment e
       WHERE e.student_id = p_studid
         AND ROWNUM = 1;

      -- The rownum set to 1 prevents the SELECT
      -- INTO statement raise to_many_rows exception.
      -- If there is at least one row in the enrollment
      -- table with a corresponding student_id, the
      -- restrict delete parameter will disallow
      -- the deletion of the student by raising
      -- the enrollment_present exception.
      RAISE enrollment_present;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         -- The no_data_found exception is raised
         -- when no students are found in the
         -- enrollment table.
         -- Since the p_ri indicates a restrict
         -- delete user choice, the delete operation
         -- is permitted.
         DELETE FROM student
          WHERE student_id = p_studid;
   END;
-- When the user enters "C" for the p_ri
-- he/she indicates a cascade delete choice
ELSIF p_ri = 'C' THEN
   -- Delete the student from the enrollment and
   -- grade tables
   DELETE FROM enrollment
    WHERE student_id = p_studid;

   DELETE FROM grade
    WHERE student_id = p_studid;

   -- Delete from student table only after
   -- corresponding records have been removed from
   -- the other tables because the student table is
   -- the parent table
```

```
      DELETE
        FROM student
       WHERE student_id = p_studid;
   ELSE
      RAISE bad_pri;
   END IF;
EXCEPTION
   WHEN bad_pri THEN
      RAISE_APPLICATION_ERROR
         (-20231, 'An incorrect p_ri value was '||
          'entered. The remove_student procedure can '||
          'only accept a C or R for the p_ri parameter.');

   WHEN enrollment_present THEN
      RAISE_APPLICATION_ERROR
         (-20239, 'The student with ID'||p_studid||
         ' exists in the enrollment table thus records'||
         ' will not be removed.');
END remove_student;

FUNCTION get_course_descript
   (p_cnumber course.course_no%TYPE)
RETURN course.description%TYPE
IS
BEGIN
   RETURN get_course_descript_private(p_cnumber);
END get_course_descript;

BEGIN
   SELECT trunc(sysdate, 'DD')
     INTO v_current_date
     FROM dual;
END student_api;
```

3) Add a PRAGMA RESTRICT_REFERENCES to `student_api` for `get_course_description` specifying the following: It writes no database state, it writes no package state, and it reads no package state.

ANSWER: The package should look similar to the following:

```
CREATE OR REPLACE PACKAGE student_api AS
   v_current_date DATE;

   PROCEDURE discount;

   FUNCTION new_instructor_id
   RETURN instructor.instructor_id%TYPE;

   FUNCTION total_cost_for_student
      (p_student_id IN student.student_id%TYPE)
      RETURN course.cost%TYPE;
```

```
   PRAGMA RESTRICT_REFERENCES
      (total_cost_for_student, WNDS, WNPS, RNPS);

   PROCEDURE get_student_info
      (p_student_id   IN student.student_id%TYPE,
       p_last_name   OUT student.last_name%TYPE,
       p_first_name  OUT student.first_name%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE get_student_info
      (p_last_name    IN student.last_name%TYPE,
       p_first_name   IN student.first_name%TYPE,
       p_student_id  OUT student.student_id%TYPE,
       p_zip         OUT student.zip%TYPE,
       p_return_code OUT NUMBER);

   PROCEDURE remove_student
      (p_studid IN student.student_id%TYPE,
       p_ri     IN VARCHAR2 DEFAULT 'R');

   FUNCTION get_course_descript
      (p_cnumber course.course_no%TYPE)
   RETURN course.description%TYPE;
   PRAGMA RESTRICT_REFERENCES
      (get_course_descript,WNDS, WNPS, RNPS);
END student_api;
/
```

Chapter 23, "Object Types in Oracle"

1) Create the object type `student_obj_type` with attributes derived from the STUDENT table.

   ANSWER: The object type should look similar to the following:

```
CREATE OR REPLACE TYPE student_obj_type AS OBJECT
   (student_id        NUMBER(8),
    salutation        VARCHAR2(5),
    first_name        VARCHAR2(25),
    last_name         VARCHAR2(25),
    street_address    VARCHAR2(50),
    zip               VARCHAR2(5),
    phone             VARCHAR2(15),
    employer          VARCHAR2(50),
    registration_date DATE,
    created_by        VARCHAR2(30),
    created_date      DATE,
    modified_by       VARCHAR2(30),
    modified_date     DATE);
/
```

After this object type is created, it can be used as follows:

```
SET SERVEROUTPUT ON
DECLARE
   v_student_obj student_obj_type;
BEGIN
   -- Use default contructor method to initialize student object
   SELECT student_obj_type(student_id, salutation, first_name,
             last_name, street_address, zip, phone, employer,
             registration_date, null, null, null, null)
     INTO v_student_obj
     FROM student
    WHERE student_id = 103;

   DBMS_OUTPUT.PUT_LINE ('Student ID: '||v_student_obj.student_id);
   DBMS_OUTPUT.PUT_LINE ('Salutation: '||v_student_obj.salutation);
   DBMS_OUTPUT.PUT_LINE ('First Name: '||v_student_obj.first_name);
   DBMS_OUTPUT.PUT_LINE ('Last Name: ' ||v_student_obj.last_name);
   DBMS_OUTPUT.PUT_LINE
      ('Street Address: '||v_student_obj.street_address);
   DBMS_OUTPUT.PUT_LINE ('Zip: '     ||v_student_obj. zip);
   DBMS_OUTPUT.PUT_LINE ('Phone: '   ||v_student_obj.phone);
   DBMS_OUTPUT.PUT_LINE ('Employer: '||v_student_obj.employer);
   DBMS_OUTPUT.PUT_LINE
      ('Registration Date: '||v_student_obj.registration_date);
END;
/
```

The output is as follows:

```
Student ID: 103
Salutation: Ms.
First Name: J.
Last Name: Landry
Street Address: 7435 Boulevard East #45
Zip: 07047
Phone: 201-555-5555
Employer: Albert Hildegard Co.
Registration Date: 22-JAN-03

PL/SQL procedure successfully completed.
```

2) Add user-defined constructor function, member procedure, static procedure, and order function methods. You should determine on your own how these methods should be structured.

ANSWER: The newly modified student object should be similar to the following:

```
CREATE OR REPLACE TYPE student_obj_type AS OBJECT
  (student_id        NUMBER(8),
   salutation        VARCHAR2(5),
   first_name        VARCHAR2(25),
   last_name         VARCHAR2(25),
   street_address    VARCHAR2(50),
   zip               VARCHAR2(5),
```

```
   phone              VARCHAR2(15),
   employer           VARCHAR2(50),
   registration_date  DATE,
   created_by         VARCHAR2(30),
   created_date       DATE,
   modified_by        VARCHAR2(30),
   modified_date      DATE,

   CONSTRUCTOR FUNCTION student_obj_type
      (SELF IN OUT NOCOPY STUDENT_OBJ_TYPE,
       in_student_id  IN NUMBER,   in_salutation IN VARCHAR2,
       in_first_name  IN VARCHAR2, in_last_name  IN VARCHAR2,
       in_street_addr IN VARCHAR2, in_zip        IN VARCHAR2,
       in_phone       IN VARCHAR2, in_employer   IN VARCHAR2,
       in_reg_date    IN DATE,     in_cr_by      IN VARCHAR2,
       in_cr_date     IN DATE,     in_mod_by     IN VARCHAR2,
       in_mod_date    IN DATE)
   RETURN SELF AS RESULT,

   CONSTRUCTOR FUNCTION student_obj_type
      (SELF IN OUT NOCOPY STUDENT_OBJ_TYPE,
       in_student_id IN NUMBER)
   RETURN SELF AS RESULT,

   MEMBER PROCEDURE get_student_info
      (student_id  OUT NUMBER,   salutation OUT VARCHAR2,
       first_name  OUT VARCHAR2, last_name  OUT VARCHAR2,
       street_addr OUT VARCHAR2, zip        OUT VARCHAR2,
       phone       OUT VARCHAR2, employer   OUT VARCHAR2,
       reg_date    OUT DATE,     cr_by      OUT VARCHAR2,
       cr_date     OUT DATE,     mod_by     OUT VARCHAR2,
       mod_date    OUT DATE),

   STATIC PROCEDURE display_student_info
      (student_obj IN STUDENT_OBJ_TYPE),

   ORDER MEMBER FUNCTION student
      (student_obj STUDENT_OBJ_TYPE)
   RETURN INTEGER);
/

CREATE OR REPLACE TYPE BODY student_obj_type AS

CONSTRUCTOR FUNCTION student_obj_type
   (SELF IN OUT NOCOPY STUDENT_OBJ_TYPE,
    in_student_id  IN NUMBER,   in_salutation IN VARCHAR2,
    in_first_name  IN VARCHAR2, in_last_name  IN VARCHAR2,
    in_street_addr IN VARCHAR2, in_zip        IN VARCHAR2,
    in_phone       IN VARCHAR2, in_employer   IN VARCHAR2,
    in_reg_date    IN DATE,     in_cr_by      IN VARCHAR2,
```

```
    in_cr_date     IN DATE,     in_mod_by     IN VARCHAR2,
    in_mod_date    IN DATE)
RETURN SELF AS RESULT
IS
BEGIN
   -- Validate incoming value of zip
   SELECT zip
     INTO SELF.zip
     FROM zipcode
    WHERE zip = in_zip;

   -- Check incoming value of student ID
   -- If it is not populated, get it from the sequence
   IF in_student_id IS NULL THEN
      student_id := STUDENT_ID_SEQ. NEXTVAL;
   ELSE
      student_id := in_student_id;
   END IF;

   salutation        := in_salutation;
   first_name        := in_first_name;
   last_name         := in_last_name;
   street_address    := in_street_addr;
   phone             := in_phone;
   employer          := in_employer;
   registration_date := in_reg_date;

   IF in_cr_by IS NULL THEN created_by := USER;
   ELSE                     created_by := in_cr_by;
   END IF;

   IF in_cr_date IS NULL THEN created_date := SYSDATE;
   ELSE                       created_date := in_cr_date;
   END IF;

   IF in_mod_by IS NULL THEN modified_by := USER;
   ELSE                      modified_by := in_mod_by;
   END IF;

   IF in_mod_date IS NULL THEN modified_date := SYSDATE;
   ELSE                        modified_date := in_mod_date;
   END IF;

   RETURN;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN;
END;
```

```
CONSTRUCTOR FUNCTION student_obj_type
   (SELF IN OUT NOCOPY STUDENT_OBJ_TYPE,
    in_student_id IN NUMBER)
RETURN SELF AS RESULT
IS
BEGIN
   SELECT student_id, salutation, first_name, last_name,
          street_address, zip, phone, employer,
          registration_date, created_by, created_date,
          modified_by, modified_date
     INTO SELF.student_id, SELF.salutation, SELF.first_name,
          SELF.last_name, SELF.street_address, SELF.zip,
          SELF.phone, SELF.employer, SELF.registration_date,
          SELF.created_by, SELF.created_date,
          SELF.modified_by, SELF.modified_date
     FROM student
    WHERE student_id = in_student_id;

   RETURN;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN;
END;

MEMBER PROCEDURE get_student_info
   (student_id  OUT NUMBER,   salutation OUT VARCHAR2,
    first_name  OUT VARCHAR2, last_name  OUT VARCHAR2,
    street_addr OUT VARCHAR2, zip        OUT VARCHAR2,
    phone       OUT VARCHAR2, employer   OUT VARCHAR2,
    reg_date    OUT DATE,     cr_by      OUT VARCHAR2,
    cr_date     OUT DATE,     mod_by     OUT VARCHAR2,
    mod_date    OUT DATE)IS
BEGIN
   student_id  := SELF.student_id;
   salutation  := SELF.salutation;
   first_name  := SELF.first_name;
   last_name   := SELF.last_name;
   street_addr := SELF.street_address;
   zip         := SELF.zip;
   phone       := SELF.phone;
   employer    := SELF.employer;
   reg_date    := SELF.registration_date;
   cr_by       := SELF.created_by;
   cr_date     := SELF.created_date;
   mod_by      := SELF.modified_by;
   mod_date    := SELF.modified_date;
END;
```

```
STATIC PROCEDURE display_student_info
   (student_obj IN STUDENT_OBJ_TYPE)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE ('Student ID: '||student_obj.student_id);
   DBMS_OUTPUT.PUT_LINE ('Salutation: '||student_obj.salutation);
   DBMS_OUTPUT.PUT_LINE ('First Name: '||student_obj.first_name);
   DBMS_OUTPUT.PUT_LINE ('Last Name: ' ||student_obj.last_name);
   DBMS_OUTPUT.PUT_LINE
      ('Street Address: '||student_obj.street_address);
   DBMS_OUTPUT.PUT_LINE ('Zip: '     ||student_obj.zip);
   DBMS_OUTPUT.PUT_LINE ('Phone: '   ||student_obj.phone);
   DBMS_OUTPUT.PUT_LINE ('Employer: '||student_obj.employer);
   DBMS_OUTPUT.PUT_LINE
      ('Registration Date: '||student_obj.registration_date);
END;

ORDER MEMBER FUNCTION student (student_obj STUDENT_OBJ_TYPE)
RETURN INTEGER
IS
BEGIN
   IF    student_id < student_obj.student_id THEN RETURN -1;
   ELSIF student_id = student_obj.student_id THEN RETURN  0;
   ELSIF student_id > student_obj.student_id THEN RETURN  1;
   END IF;
END;

END;
/
```

This student object type has two overloaded constructor functions, member procedure, static procedure, and order function methods.

Both constructor functions have the same name as the object type. The first constructor function evaluates incoming values of student ID, zip code, created and modified users, and dates. Specifically, it checks to see if the incoming student ID is null and then populates it from STUDENT_ID_SEQ. Take a closer look at the statement that assigns a sequence value to the STUDENT_ID attribute. The ability to access a sequence via a PL/SQL expression is a new feature in Oracle 11g. Previously, sequences could be accessed only by queries. It also validates that the incoming value of zip exists in the ZIPCODE table. Finally, it checks to see if incoming values of the created and modified user and date are null. If any of these incoming values are null, the constructor function populates the corresponding attributes with the default values based on the system functions USER and SYSDATE. The second constructor function initializes the object instance based on the incoming value of student ID using the SELECT INTO statement.

The member procedure GET_STUDENT_INFO populates out parameters with corresponding values of object attributes. The static procedure DISPLAY_STUDENT_INFO displays values of the incoming student object on the screen. Recall that static methods do not have access to the data associated with a particular object type instance. As a result, they may not reference the default parameter SELF. The order member function compares two instances of the student object type based on values of the `student_id` attribute.

The newly created object type may be tested as follows:

```
DECLARE
   v_student_obj1 student_obj_type;
   v_student_obj2 student_obj_type;

   v_result INTEGER;
BEGIN
   -- Populate student objects via user-defined constructor method
   v_student_obj1 :=
      student_obj_type (in_student_id  => NULL,
                        in_salutation  => 'Mr.',
                        in_first_name  => 'John',
                        in_last_name   => 'Smith',
                        in_street_addr => '123 Main Street',
                        in_zip         => '00914',
                        in_phone       => '555-555-5555',
                        in_employer    => 'ABC Company',
                        in_reg_date    => TRUNC(sysdate),
                        in_cr_by       => NULL,
                        in_cr_date     => NULL,
                        in_mod_by      => NULL,
                        in_mod_date    => NULL);

   v_student_obj2 := student_obj_type(103);

   -- Display student information for both objects
   student_obj_type.display_student_info (v_student_obj1);
   DBMS_OUTPUT.PUT_LINE ('================================');
   student_obj_type.display_student_info (v_student_obj2);
   DBMS_OUTPUT.PUT_LINE ('================================');

   -- Compare student objects
   v_result := v_student_obj1.student(v_student_obj2);
   DBMS_OUTPUT.PUT_LINE ('The result of comparison is '||v_result);

   IF v_result = 1 THEN
      DBMS_OUTPUT.PUT_LINE
         ('v_student_obj1 is greater than v_student_obj2');

   ELSIF v_result = 0 THEN
      DBMS_OUTPUT.PUT_LINE
         ('v_student_obj1 is equal to v_student_obj2');

   ELSIF v_result = -1 THEN
      DBMS_OUTPUT.PUT_LINE
         ('v_student_obj1 is less than v_student_obj2');
   END IF;

END;
/
```

The output is as follows:

```
Student ID: 403
Salutation: Mr.
First Name: John
Last Name: Smith
Street Address: 123 Main Street
Zip: 00914
Phone: 555-555-5555
Employer: ABC Company
Registration Date: 24-APR-08
================================
Student ID: 103
Salutation: Ms.
First Name: J.
Last Name: Landry
Street Address: 7435 Boulevard East #45
Zip: 07047
Phone: 201-555-5555
Employer: Albert Hildegard Co.
Registration Date: 22-JAN-03
================================
The result of comparison is 1
v_student_obj1 is greater than v_student_obj2

PL/SQL procedure successfully completed.
```

Chapter 24, "Oracle Supplied Packages"

This chapter has no "Try It Yourself" section.