BENJAMIN ROSENZWEIG AND ELENA RAKHIMOV

Oracle® PL/SQL™ by Example

Fourth Edition

- Updated for Oracle 11g
- Hundreds of examples, questions, and answers
- Real-life labs
- No Oracle PL/SQL experience necessary!
- Build PL/SQL Applications-NOW!

THE PRENTICE HALL PTR ORACLE SERIES

THE INDEPENDENT VOICE ON ORACLE

Library of Congress Cataloging-in-Publication Data

Rosenzweig, Benjamin.
Oracle PL/SQL by example / Benjamin Rosenzweig, Elena Silvestrova Rakhimov.
p. cm.
ISBN 0-13-714422-9 (pbk. : alk. paper) 1. PL/SQL (Computer program language) 2. Oracle (Computer file) 3. Relational databases. I. Rakhimov, Elena Silvestrova. II. Title.
QA76.73.P258R68 2008
005.75'6—dc22

2008022398

ISBN-13: 978-0-137-14422-8
ISBN-10: 0-137-14422-9

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan

First printing August 2008

Editor-in-Chief: Mark Taub
Acquisitions Editor: Trina MacDonald
Development Editor: Songlin Qiu
Managing Editor: Kristy Hart
Project Editor: Todd Taber
Copy Editor: Gayle Johnson
Indexer: Erika Millen
Proofreader: Debbie Williams
Technical Reviewers: Oleg Voskoboynikov,
Shahdad Moradi
Publishing Coordinator: Olivia Basegio
Cover Designer: Chuti Prasertsith
Composition: Nonie Ratcliff

INTRODUCTION

PL/SQL New Features in Oracle 11g

Oracle 11g has introduced a number of new features and improvements for PL/SQL. This introduction briefly describes features not covered in this book and points you to specific chapters for features that *are* within scope of this book. The list of features described here is also available in the "What's New in PL/SQL?" section of the PL/SQL Language Reference manual offered as part of Oracle help available online.

The new PL/SQL features and enhancements are as follows:

- Enhancements to regular expression built-in SQL functions
- SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE datatypes
- CONTINUE statement
- Sequences in PL/SQL expressions
- Dynamic SQL enhancements
- Named and mixed notation in PL/SQL subprogram invocations
- Cross-session PL/SQL function result cache
- More control over triggers
- Compound triggers
- Database resident connection pool
- Automatic subprogram inlining
- PL/Scope
- PL/SQL hierarchical profiler
- PL/SQL native compiler generates native code directly

Enhancements to Regular Expression Built-In SQL Functions

In this release Oracle has introduced a new regular expression built-in function, REGEXP_COUNT. It returns the number of times a specified search pattern appears in a source string.

FOR EXAMPLE

```
SELECT
   REGEXP_COUNT ('Oracle PL/SQL By Example Updated for Oracle 11g',
                 'ora', 1, 'i')
  FROM dual;

REGEXP_COUNT('ORACLEPL/SQLBYEXAMPLEUPDATEDFORORACLE11G','ORA',1,'I')
--------------------------------------------------------------------
                                                                   2
```

The REGEXP_COUNT function returns how many times the search pattern `'ora'` appears in the source string `'Oracle PL/SQL...'` `1` indicates the position of the source string where the search begins, and `'i'` indicates case-insensitive matching.

The existing regular expression built-in functions, REGEXP_INSTR and REGEXP_SUBSTR, have a new parameter called SUBEXPR. This parameter represents a subexpression in a search pattern. Essentially it is a portion of a search pattern enclosed in parentheses that restricts pattern matching, as illustrated in the following example.

FOR EXAMPLE

```
SELECT
   REGEXP_INSTR ('Oracle PL/SQL By Example Updated for Oracle 11g',
                 '((ora)(cle))', 1, 2, 0, 'i')
  FROM dual;

REGEXP_INSTR('ORACLEPL/SQLBYEXAMPLEUPDATEDFORORACLE11G',...)
------------------------------------------------------------
                                                          38
```

The REGEXP_INSTR function returns the position of the first character in the source string `'Oracle PL/SQL...'` corresponding to the second occurrence of the first subexpression `'ora'` in the seach pattern `(ora)(cle)`. `1` indicates the position of the source string where the search begins, `2` indicates the occurrence of the subexpression in the source string, `0` indicates that the position returned corresponds to the position of the first character where the match occurs, and `'i'` indicates case-insensitive matching and REGEXP_SUBSTR.

SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE Datatypes

These datatypes are predefined subtypes of the PLS_INTEGER, BINARY_FLOAT, and BINARY_DOUBLE, respectively. As such, they have the same range as their respective base types. In addition, these subtypes have NOT NULL constraints.

These subtypes provide significant performance improvements over their respective base types when the PLSQL_CODE_TYPE parameter is set to NATIVE. This is because arithmetic operations for these subtypes are done directly in the hardware layer. Note that when PLSQL_CODE_TYPE is set to INTERPRETED (the default value), the performance gains are significantly smaller. This is illustrated by the following example.

FOR EXAMPLE

```
SET SERVEROUTPUT ON
DECLARE
   v_pls_value1    PLS_INTEGER := 0;
   v_pls_value2    PLS_INTEGER := 1;

   v_simple_value1 SIMPLE_INTEGER := 0;
   v_simple_value2 SIMPLE_INTEGER := 1;

   -- Following are used for elapsed time calculation
   -- The time is calculated in 100th of a second
   v_start_time    NUMBER;
   v_end_time      NUMBER;

BEGIN
   -- Perform calculations with PLS_INTEGER
   v_start_time := DBMS_UTILITY.GET_TIME;

   FOR i in 1..50000000 LOOP
      v_pls_value1 := v_pls_value1 + v_pls_value2;
   END LOOP;

   v_end_time := DBMS_UTILITY.GET_TIME;
   DBMS_OUTPUT.PUT_LINE ('Elapsed time for PLS_INTEGER: '||
      (v_end_time - v_start_time));

   -- Perform the same calculations with SIMPLE_INTEGER
   v_start_time := DBMS_UTILITY.GET_TIME;

   FOR i in 1..50000000 LOOP
      v_simple_value1 := v_simple_value1 + v_simple_value2;
   END LOOP;

   v_end_time := DBMS_UTILITY.GET_TIME;
   DBMS_OUTPUT.PUT_LINE ('Elapsed time for SIMPLE_INTEGER: '||
      (v_end_time - v_start_time));
END;
```

This script compares the performance of the PLS_INTEGER datatype with its subtype SIMPLE_INTEGER via a numeric FOR loop. Note that for this run the PLSQL_CODE_TYPE parameter is set to its default value, INTERPRETED.

```
Elapsed time for PLS_INTEGER: 147
Elapsed time for SIMPLE_INTEGER: 115

PL/SQL procedure successfully completed.
```

CONTINUE Statement

Similar to the EXIT statement, the CONTINUE statement controls loop iteration. Whereas the EXIT statement causes a loop to terminate and passes control of the execution outside the loop, the CONTINUE statement causes a loop to terminate its current iteration and passes control to the next iteration of the loop. The CONTINUE statement is covered in detail in Chapter 7, "Iterative Control—Part 2."

Sequences in PL/SQL Expressions

Prior to Oracle 11g, the sequence pseudocolumns CURRVAL and NEXTVAL could be accessed in PL/SQL only through queries. Starting with Oracle 11g, these pseudocolumns can be accessed via expressions. This change not only improves PL/SQL source code, it also improves runtime performance and scalability.

FOR EXAMPLE

```
CREATE SEQUENCE test_seq START WITH 1 INCREMENT BY 1;

Sequence created.

SET SERVEROUTPUT ON
DECLARE
   v_seq_value NUMBER;
BEGIN
   v_seq_value := test_seq.NEXTVAL;
   DBMS_OUTPUT.PUT_LINE ('v_seq_value: '||v_seq_value);
END;
```

This script causes an error when executed in Oracle 10g:

```
      v_seq_value := test_seq.NEXTVAL;
                             *
ERROR at line 4:
ORA-06550: line 4, column 28:
PLS-00357: Table,View Or Sequence reference 'TEST_SEQ.NEXTVAL' not
allowed in this context
ORA-06550: line 4, column 4:
PL/SQL: Statement ignored
```

and it completes successfully when executed in Oracle 11g:

```
v_seq_value: 1

PL/SQL procedure successfully completed.
```

Consider another example that illustrates performance improvement when the PL/SQL expression is used to manipulate sequences:

FOR EXAMPLE

```
SET SERVEROUTPUT ON
DECLARE
   v_seq_value  NUMBER;

   -- Following are used for elapsed time calculation
   v_start_time NUMBER;
   v_end_time   NUMBER;

BEGIN
   -- Retrieve sequence via SELECT INTO statement
   v_start_time := DBMS_UTILITY.GET_TIME;

   FOR i in 1..10000 LOOP
      SELECT test_seq.NEXTVAL
        INTO v_seq_value
        FROM dual;
   END LOOP;

   v_end_time := DBMS_UTILITY.GET_TIME;
   DBMS_OUTPUT.PUT_LINE
      ('Elapsed time to retrieve sequence via SELECT INTO: '||
       (v_end_time-v_start_time));

   -- Retrieve sequence via PL/SQL expression
   v_start_time := DBMS_UTILITY.GET_TIME;

   FOR i in 1..10000 LOOP
      v_seq_value := test_seq.NEXTVAL;
   END LOOP;

   v_end_time := DBMS_UTILITY.GET_TIME;
   DBMS_OUTPUT.PUT_LINE
      ('Elapsed time to retrieve sequence via PL/SQL expression: '||
       (v_end_time-v_start_time));
END;

Elapsed time to retrieve sequence via SELECT INTO: 52
Elapsed time to retrieve sequence via PL/SQL expression: 43

PL/SQL procedure successfully completed.
```

Dynamic SQL Enhancements

In this version, Oracle has introduced a number of enhancements to the native dynamic SQL and DBMS_SQL package.

Native dynamic SQL enables you to generate dynamic SQL statements larger than 32K. In other words, it supports the CLOB datatype. Native dynamic SQL is covered in detail in Chapter 17, "Native Dynamic SQL."

The DBMS_SQL package now supports all datatypes that native dynamic SQL supports. This includes the CLOB datatype. In addition, two new functions, DBMS_SQL.TO_REFCURSOR and DBMS_SQL.TO_CURSOR_NUMBER, enable you to switch between the native dynamic SQL and DBMS_SQL package.

Named and Mixed Notation in PL/SQL Subprogram Invocations

Prior to Oracle 11g, a SQL statement invoking a function had to specify the parameters in positional notation. In this release, mixed and named notations are allowed as well. Examples of positional, named, and mixed notations can be found in Chapter 21, "Packages," and Chapter 23, "Object Types in Oracle."

Consider the following example:

FOR EXAMPLE

```
CREATE OR REPLACE FUNCTION test_function
   (in_val1 IN NUMBER, in_val2 IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
   RETURN (in_val1||' - '||in_val2);
END;

Function created.

SELECT
   test_function(1, 'Positional Notation') col1,
   test_function(in_val1 => 2, in_val2 => 'Named Notation') col2,
   test_function(3, in_val2 => 'Mixed Notation') col3
  FROM dual;

COL1                    COL2                COL3
----------------------  ------------------  ------------------
1 - Positional Notation  2 - Named Notation  3 - Mixed Notation
```

Note that mixed notation has a restriction: positional notation may not follow named notation. This is illustrated by the following SELECT:

```
SELECT
   test_function(1, 'Positional Notation') col1,
   test_function(in_val1 => 2, in_val2 => 'Named Notation') col2,
   test_function(in_val1 => 3, 'Mixed Notation') col3
  FROM dual;

   test_function(in_val1 => 3, 'Mixed Notation') col3
   *
ERROR at line 4:
ORA-06553: PLS-312: a positional parameter association may not follow
a named association
```

Cross-Session PL/SQL Function Result Cache

A result-cached function is a function whose parameter values and result are stored in the cache. This means that when such a function is invoked with the same parameter values, its result is retrieved from the cache instead of being computed again. This caching mechanism is known as single-session caching because each session requires its own copy of the cache where function parameters and its results are stored.

Starting with Oracle 11, the caching mechanism for result-cached functions has been expanded to cross-session caching. In other words, the parameter values and results of the result-cached function are now stored in the shared global area (SGA) and are available to any session. Note that when an application is converted from single-session caching to cross-session caching, it requires more SGA but considerably less total system memory.

Consider the following example, which illustrates how a result-cached function may be created:

FOR EXAMPLE

```
-- Package specification
CREATE OR REPLACE PACKAGE test_pkg AS

   -- User-defined record type
   TYPE zip_record IS RECORD
      (zip   VARCHAR2(5),
       city  VARCHAR2(25),
       state VARCHAR2(2));

   -- Result-cached function
   FUNCTION get_zip_info (in_zip NUMBER) RETURN zip_record
   RESULT_CACHE;

END test_pkg;
/

-- Package body
CREATE OR REPLACE PACKAGE BODY test_pkg AS

-- Result-cached function
```

```
FUNCTION get_zip_info (in_zip NUMBER) RETURN zip_record
RESULT_CACHE
RELIES_ON (ZIPCODE)
IS
   rec zip_record;
BEGIN
   SELECT zip, city, state
     INTO rec
     FROM zipcode
    WHERE zip = in_zip;
   RETURN rec;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN null;
END get_zip_info;

END test_pkg;
/
```

Note the use of the RESULT_CACHE and RELIES_ON clauses. RESULT_CACHE specifies that the function is a result-cached function, and RELIES_ON specifies any tables and/or views that the function results depend on.

More Control over Triggers

Starting with Oracle 11g, the CREATE OR REPLACE TRIGGER clause may include ENABLE, DISABLE, and FOLLOWS options. The ENABLE and DISABLE options allow you to create a trigger in the enabled or disabled state, respectively. The FOLLOWS option allows you to specify the order in which triggers fire. Note that the FOLLOWS option applies to triggers that are defined on the same table and fire at the same timing point. Triggers are covered in detail in Chapter 13, "Triggers."

Compound Triggers

A compound trigger is a new type of trigger that allows you to combine different types of triggers into one trigger. Specifically, you can combine the following:

- A statement trigger that fires before the firing statement
- A row trigger that fires before each row that the firing statement affects
- A row trigger that fires after each row that the firing statement affects
- A statement trigger that fires after the firing statement

This means that a single trigger may fire at different times when a transaction occurs. Compound triggers are covered in detail in Chapter 14, "Compound Triggers."

Database Resident Connection Pool

Database Resident Connection Pool (DRCP) provides a connection pool that is shared by various middle-tier processes. The new package, DBMS_CONNECTION_POOL, enables database administrators to start and stop DRCP and configure its parameters.

Automatic Subprogram Inlining

The PL/SQL compiler translates PL/SQL code into machine code. Starting with Oracle 10g, the PL/SQL compiler can use the performance optimizer when compiling PL/SQL code. The performance optimizer enables the PL/SQL compiler to rearrange PL/SQL code to enhance performance. The optimization level used by the PL/SQL compiler is controlled by the PLSQL_OPTIMIZE_LEVEL parameter. Its values range from 0 to 2, where 2 is the default value. This means that the PL/SQL compiler performs optimization by default.

Starting with Oracle 11g, the PL/SQL compiler can perform subprogram inlining. Subprogram inlining substitutes a subprogram invocation with an actual copy of the called subprogram. This is achieved by specifying PRAGMA INLINE or setting the PLSQL_OPTIMIZE_LEVEL parameter to a new value, 3. When PLSQL_OPTIMIZE_LEVEL is set to 3, the PL/SQL compiler performs automatic subprogram inlining where appropriate. However, in some instances, the PL/SQL compiler may choose not to perform subprogram inlining because it believes it is undesirable.

The use of PRAGMA INLINE is illustrated in the following example. Note that in this example, PLSQL_OPTIMIZE_LEVEL has been set to its default value, 2.

FOR EXAMPLE

```
SET SERVEROUTPUT ON
DECLARE
   v_num    PLS_INTEGER := 1;
   v_result PLS_INTEGER;

   -- Following are used for elapsed time calculation
   v_start_time NUMBER;
   v_end_time   NUMBER;

   -- Define function to test PRAGMA INLINE
   FUNCTION test_inline_pragma
      (in_num1 IN PLS_INTEGER, in_num2 IN PLS_INTEGER)
   RETURN PLS_INTEGER
   IS
   BEGIN
      RETURN (in_num1 + in_num2);
   END test_inline_pragma;

BEGIN
   -- Test function with INLINE PRAGMA enabled
   v_start_time := DBMS_UTILITY.GET_TIME;

   FOR i in 1..10000000 LOOP
```

```
      PRAGMA INLINE (test_inline_pragma, 'YES');
      v_result := test_inline_pragma (1, i);
   END LOOP;

   v_end_time := DBMS_UTILITY.GET_TIME;
   DBMS_OUTPUT.PUT_LINE
      ('Elapsed time when PRAGMA INLINE enabled: '||
       (v_end_time-v_start_time));

   -- Test function with PRAGMA INLINE disabled
   v_start_time := DBMS_UTILITY.GET_TIME;

   FOR i in 1..10000000 LOOP
      PRAGMA INLINE (test_inline_pragma, 'NO');
      v_result := test_inline_pragma (1, i);
   END LOOP;

   v_end_time := DBMS_UTILITY.GET_TIME;
   DBMS_OUTPUT.PUT_LINE
      ('Elapsed time when INLINE PRAGMA disabled: '||
       (v_end_time-v_start_time));
END;

Elapsed time when PRAGMA INLINE enabled: 59
Elapsed time when PRAGMA INLINE disabled: 220

PL/SQL procedure successfully completed.
```

Note that PRAGMA INLINE affects every call to the specified subprogram when PRAGMA INLINE is placed immediately before one of the following:

- Assignment
- Call
- Conditional
- CASE
- CONTINUE-WHEN
- EXECUTE IMMEDIATE
- EXIT-WHEN
- LOOP
- RETURN

PL/Scope

PL/Scope gathers and organizes data about user-defined identifiers used in PL/SQL code. This tool is used primarily in interactive development environments such as SQL Developer or Jdeveloper rather than directly in PL/SQL.

PL/SQL Hierarchical Profiler

PL/SQL hierarchical profiler enables you to profile PL/SQL applications. In other words, it gathers statistical information about the application such as execution times for SQL and PL/SQL, the number of calls to a particular subprogram made by the application, and the amount of time spent in the subprogram itself.

The hierarchical profiler is implemented via the Oracle-supplied package DBMS_HPROF, which is covered in Chapter 24, "Oracle Supplied Packages."

PL/SQL Native Compiler Generates Native Code Directly

In this version of Oracle, the PL/SQL native compiler can generate native code directly. Previously, PL/SQL code was translated into C code, which then was translated by the C compiler into the native code. In some cases, this improves performance significantly. The PL/SQL compiler type is controlled via the PLSQL_CODE_TYPE parameter, which can be set to either INTERPRETED (the default value) or NATIVE.

CHAPTER 4

Conditional Control: IF Statements

CHAPTER OBJECTIVES

In this chapter, you will learn about

- IF statements
- ELSIF statements
- Nested IF statements

In almost every program you write, you need to make decisions. For example, if it is the end of the fiscal year, bonuses must be distributed to the employees based on their salaries. To compute employee bonuses, a program needs a conditional control. In other words, it needs to employ a selection structure.

Conditional control allows you to control the program's flow of the execution based on a condition. In programming terms, this means that the statements in the program are not executed sequentially. Rather, one group of statements or another is executed, depending on how the condition is evaluated.

PL/SQL has three types of conditional control: IF, ELSIF, and CASE statements. This chapter explores the first two types and shows you how they can be nested inside one another. CASE statements are discussed in the next chapter.

LAB 4.1
IF Statements

LAB OBJECTIVES

After completing this lab, you will be able to

- Use the IF-THEN statement
- Use the IF-THEN-ELSE statement

An IF statement has two forms: IF-THEN and IF-THEN-ELSE. An IF-THEN statement allows you to specify only one group of actions to take. In other words, this group of actions is taken only when a condition evaluates to TRUE. An IF-THEN-ELSE statement allows you to specify two groups of actions. The second group of actions is taken when a condition evaluates to FALSE or NULL.

IF-THEN STATEMENTS

An IF-THEN statement is the most basic kind of a conditional control; it has the following structure:

```
IF CONDITION THEN
   STATEMENT 1;
   ...
   STATEMENT N;
END IF;
```

The reserved word IF marks the beginning of the IF statement. Statements 1 through N are a sequence of executable statements that consist of one or more standard programming structures. The word *CONDITION* between the keywords IF and THEN determines whether these statements are executed. END IF is a reserved phrase that indicates the end of the IF-THEN construct.

Figure 4.1 shows this flow of logic.

When an IF-THEN statement is executed, a condition is evaluated to either TRUE or FALSE. If the condition evaluates to TRUE, control is passed to the first executable statement of the IF-THEN construct. If the condition evaluates to FALSE, control is passed to the first executable statement after the END IF statement.

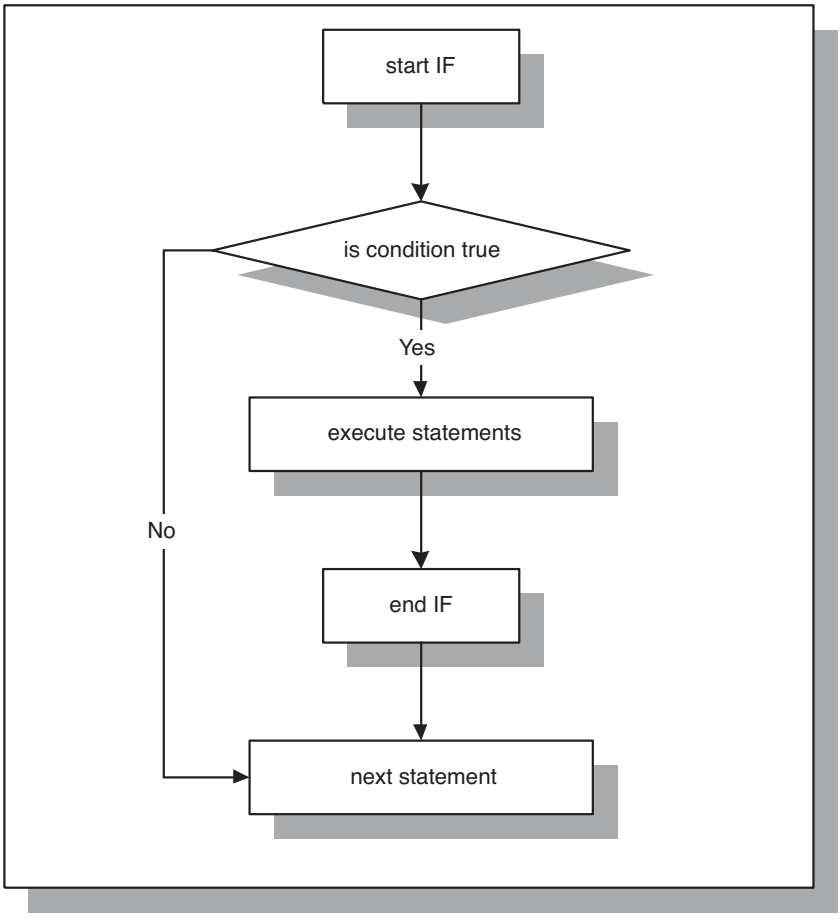

FIGURE 4.1
IF-THEN statement

Consider the following example. Two numeric values are stored in the variables v_num1 and v_num2. You need to arrange their values so that the smaller value is always stored in v_num1 and the larger value is always stored in v_num2.

FOR EXAMPLE

```
DECLARE
   v_num1 NUMBER := 5;
   v_num2 NUMBER := 3;
   v_temp NUMBER;
BEGIN
   -- if v_num1 is greater than v_num2 rearrange their values
   IF v_num1 > v_num2 THEN
      v_temp := v_num1;
```

```
      v_num1 := v_num2;
      v_num2 := v_temp;
   END IF;

   -- display the values of v_num1 and v_num2
   DBMS_OUTPUT.PUT_LINE ('v_num1 = '||v_num1);
   DBMS_OUTPUT.PUT_LINE ('v_num2 = '||v_num2);
END;
```

In this example, condition `v_num1 > v_num2` evaluates to TRUE because 5 is greater than 3. Next, the values are rearranged so that 3 is assigned to `v_num1` and 5 is assigned to `v_num2`. This is done with the help of the third variable, `v_temp`, which is used for temporary storage.

This example produces the following output:

```
v_num1 = 3
v_num2 = 5

PL/SQL procedure successfully completed.
```

IF-THEN-ELSE STATEMENT

An IF-THEN statement specifies the sequence of statements to execute only if the condition evaluates to TRUE. When this condition evaluates to FALSE, there is no special action to take, except to proceed with execution of the program.

An IF-THEN-ELSE statement enables you to specify two groups of statements. One group of statements is executed when the condition evaluates to TRUE. Another group of statements is executed when the condition evaluates to FALSE. This is indicated as follows:

```
IF CONDITION THEN
   STATEMENT 1;
ELSE
   STATEMENT 2;
END IF;
STATEMENT 3;
```

When *CONDITION* evaluates to TRUE, control is passed to *STATEMENT 1*; when *CONDITION* evaluates to FALSE, control is passed to *STATEMENT 2*. After the IF-THEN-ELSE construct has completed, *STATEMENT 3* is executed. Figure 4.2 illustrates this flow of logic.

DID YOU KNOW?

You should use the IF-THEN-ELSE construct when trying to choose between two mutually exclusive actions. Consider the following example:

```
DECLARE
   v_num NUMBER := &sv_user_num;
BEGIN
   -- test if the number provided by the user is even
   IF MOD(v_num,2) = 0  THEN
      DBMS_OUTPUT.PUT_LINE (v_num||
```

```
        ' is even number');
   ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
   END IF;
   DBMS_OUTPUT.PUT_LINE ('Done');
END;
```

For any given number, only one of the DBMS_ OUTPUT.PUT_LINE statements is executed. Hence, the IF-THEN-ELSE construct enables you to specify two and only two mutually exclusive actions.

When run, this example produces the following output:

```
Enter value for v_user_num: 24
old   2:    v_num  NUMBER := &v_user_num;
new   2:    v_num  NUMBER := 24;
24 is even number
Done

PL/SQL procedure successfully completed.
```

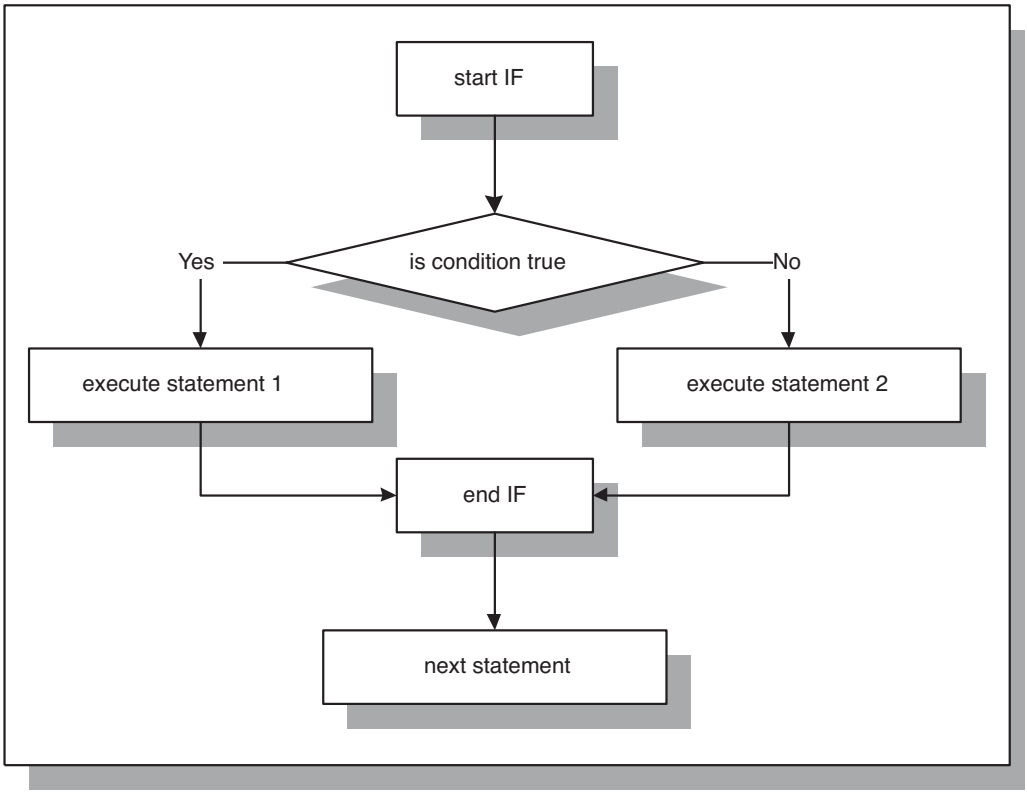

FIGURE 4.2
IF-THEN-ELSE statement

NULL CONDITION

In some cases, a condition used in an IF statement can be evaluated to NULL instead of TRUE or FALSE. For the IF-THEN construct, the statements are not executed if an associated condition evaluates to NULL. Next, control is passed to the first executable statement after END IF. For the IF-THEN-ELSE construct, the statements specified after the keyword ELSE are executed if an associated condition evaluates to NULL.

FOR EXAMPLE

```
DECLARE
   v_num1 NUMBER := 0;
   v_num2 NUMBER;
BEGIN
   IF v_num1 = v_num2 THEN
      DBMS_OUTPUT.PUT_LINE ('v_num1 = v_num2');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('v_num1 != v_num2');
  END IF;
END;
```

This example produces the following output:

```
v_num1 != v_num2

PL/SQL procedure successfully completed.
```

The condition

```
v_num1 = v_num2
```

is evaluated to NULL because a value is not assigned to the variable `v_num2`. Therefore, variable `v_num2` is NULL. Notice that the IF-THEN-ELSE construct is behaving as if the condition evaluated to FALSE, and the second DBMS_ OUTPUT.PUT_LINE statement is executed.

▼ LAB 4.1 EXERCISES

This section provides exercises and suggested answers, with discussion related to how those answers resulted. The most important thing to realize is whether your answer works. You should figure out the implications of the answers and what the effects are of any different answers you may come up with.

4.1.1 Use the IF-THEN Statement

In this exercise, you use the IF-THEN statement to test whether the date provided by the user falls on the weekend (in other words, if the day is Saturday or Sunday).

Create the following PL/SQL script:

```
-- ch04_1a.sql, version 1.0
SET SERVEROUTPUT ON
DECLARE
   v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
```

```
   v_day  VARCHAR2(15);
BEGIN
   v_day := RTRIM(TO_CHAR(v_date, 'DAY'));

   IF v_day IN ('SATURDAY', 'SUNDAY') THEN
      DBMS_OUTPUT.PUT_LINE (v_date||' falls on weekend');
   END IF;

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

To test this script fully, execute it twice. For the first run, enter `09-JAN-2008`, and for the second run, enter `13-JAN-2008`. Execute the script, and then answer the following questions:

A) What output is printed on the screen (for both dates)?

ANSWER: The first output produced for the date is 09-JAN-2008. The second output produced for the date is 13-JAN-2008.

```
Enter value for sv_user_date: 09-JAN-2008
old   2:    v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
new   2:    v_date DATE := TO_DATE('09-JAN-2008', 'DD-MON-YYYY');
Done...

PL/SQL procedure successfully completed.
```

When the value of 09-JAN-2008 is entered for `v_date`, the day of the week is determined for the variable `v_day` with the help of the functions TO_CHAR and RTRIM. Next, the following condition is evaluated:

```
v_day IN ('SATURDAY', 'SUNDAY')
```

Because the value of `v_day` is 'WEDNESDAY', the condition evaluates to FALSE. Then, control is passed to the first executable statement after END IF. As a result, `Done...` is displayed on the screen:

```
Enter value for sv_user_date: 13-JAN-2008
old   2:    v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
new   2:    v_date DATE := TO_DATE('13-JAN-2008', 'DD-MON-YYYY');
13-JAN-08 falls on weekend
Done...

PL/SQL procedure successfully completed.
```

As in the previous run, the value of `v_day` is derived from the value of `v_date`. Next, the condition of the IF-THEN statement is evaluated. Because it evaluates to TRUE, the statement after the keyword THEN is executed. Therefore, `13-JAN-2008 falls on weekend` is displayed on the screen. Next, control is passed to the last DBMS_OUTPUT.PUT_LINE statement, and `Done...` is displayed on the screen.

B) Explain why the output produced for the two dates is different.

ANSWER: The first date, 09-JAN-2008, is a Wednesday. As a result, the condition `v_day IN ('SATURDAY,' 'SUNDAY')` does not evaluate to TRUE. Therefore, control is transferred to the statement after END IF, and `Done...` is displayed on the screen.

The second date, 13-JAN-2008, is a Sunday. Because Sunday falls on a weekend, the condition evaluates to TRUE, and the message `13-JAN-2008 falls on weekend` is displayed on the screen. Next, the last DBMS_OUTPUT.PUT_LINE statement is executed, and `Done...` is displayed on the screen.

Remove the RTRIM function from the assignment statement for `v_day` as follows:

```
v_day := TO_CHAR(v_date, 'DAY');
```

Run the script again, entering `13-JAN-2008` for `v_date`.

C) What output is printed on the screen? Why?

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

```
-- ch04_1b.sql, version 2.0
SET SERVEROUTPUT ON
DECLARE
   v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
   v_day  VARCHAR2(15);
BEGIN
   v_day := TO_CHAR(v_date, 'DAY');

   IF v_day IN ('SATURDAY', 'SUNDAY') THEN
      DBMS_OUTPUT.PUT_LINE (v_date||' falls on weekend');
   END IF;

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

This script produces the following output:

```
Enter value for sv_user_date: 13-JAN-2008
old   2:    v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
new   2:    v_date DATE := TO_DATE('13-JAN-2008', 'DD-MON-YYYY');
Done...

PL/SQL procedure successfully completed.
```

In the original example, the variable `v_day` is calculated with the help of the statement `RTRIM(TO_CHAR(v_date, 'DAY'))`. First, the function TO_CHAR returns the day of the week, padded with blanks. The size of the value retrieved by the function TO_CHAR is always 9 bytes. Next, the RTRIM function removes trailing spaces.

In the statement

```
v_day := TO_CHAR(v_date, 'DAY')
```

the TO_CHAR function is used without the RTRIM function. Therefore, trailing blanks are not removed after the day of the week has been derived. As a result, the condition of the IF-THEN statement evaluates to FALSE even though the given date falls on the weekend, and control is passed to the last DBMS_ OUTPUT.PUT_LINE statement.

D) Rewrite this script using the LIKE operator instead of the IN operator so that it produces the same results for the dates specified earlier.

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

```
-- ch04_1c.sql, version 3.0
SET SERVEROUTPUT ON
DECLARE
   v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
   v_day  VARCHAR2(15);
BEGIN
   v_day := RTRIM(TO_CHAR(v_date, 'DAY'));

   IF v_day LIKE 'S%' THEN
      DBMS_OUTPUT.PUT_LINE (v_date||' falls on weekend');
   END IF;

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

Saturday and Sunday are the only days of the week that start with S. As a result, there is no need to spell out the names of the days or specify any additional letters for the LIKE operator.

E) Rewrite this script using the IF-THEN-ELSE construct. If the date specified does not fall on the weekend, display a message to the user saying so.

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

```
-- ch04_1d.sql, version 4.0
SET SERVEROUTPUT ON
DECLARE
   v_date DATE := TO_DATE('&sv_user_date', 'DD-MON-YYYY');
   v_day  VARCHAR2(15);
BEGIN
   v_day := RTRIM(TO_CHAR(v_date, 'DAY'));

   IF v_day IN ('SATURDAY', 'SUNDAY') THEN
      DBMS_OUTPUT.PUT_LINE (v_date||' falls on weekend');
   ELSE
      DBMS_OUTPUT.PUT_LINE
         (v_date||' does not fall on the weekend');
   END IF;

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

To modify the script, the ELSE part was added to the IF statement. The rest of the script has not been changed.

4.1.2 Use the IF-THEN-ELSE Statement

In this exercise, you use the IF-THEN-ELSE statement to check how many students are enrolled in course number 25, section 1. If 15 or more students are enrolled, section 1 of course number 25 is full. Otherwise, section 1 of course number 25 is not full, and more students can register for it. In both cases, a message should be displayed to the user, indicating whether section 1 is full. Try to answer the questions before you run the script. After you have answered the questions, run the script and check your answers. Note that the SELECT INTO statement uses the ANSI 1999 SQL standard.

Create the following PL/SQL script:

```
-- ch04_2a.sql, version 1.0
SET SERVEROUTPUT ON
DECLARE
   v_total NUMBER;
BEGIN
   SELECT COUNT(*)
     INTO v_total
     FROM enrollment e
     JOIN section s USING (section_id)
    WHERE s.course_no = 25
      AND s.section_no = 1;

   -- check if section 1 of course 25 is full
   IF v_total >= 15 THEN
      DBMS_OUTPUT.PUT_LINE
         ('Section 1 of course 25 is full');
   ELSE
      DBMS_OUTPUT.PUT_LINE
         ('Section 1 of course 25 is not full');
   END IF;
   -- control resumes here
END;
```

Notice that the SELECT INTO statement uses an equijoin. The join condition is listed in the JOIN clause, indicating columns that are part of the primary key and foreign key constraints. In this example, column SECTION_ID of the ENROLLMENT table has a foreign key constraint defined on it. This constraint references column SECTION_ID of the SECTION table, which, in turn, has a primary key constraint defined on it.

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 try to provide examples illustrating both standards; however, our main focus is PL/SQL features rather than SQL.

Try to answer the following questions, and then execute the script:

A) What DBMS_OUTPUT.PUT_LINE statement is displayed if 15 students are enrolled in section 1 of course number 25?

ANSWER: If 15 or more students are enrolled in section 1 of course number 25, the first DBMS_OUTPUT.PUT_LINE statement is displayed on the screen.

The condition

```
v_total >= 15
```

evaluates to TRUE, and as a result, the statement

```
DBMS_OUTPUT.PUT_LINE ('Section 1 of course 25 is full');
```

is executed.

B) What DBMS_OUTPUT.PUT_LINE statement is displayed if three students are enrolled in section 1 of course number 25?

ANSWER: If three students are enrolled in section 1 of course number 25, the second DBMS_OUTPUT.PUT_LINE statement is displayed on the screen.

The condition

```
v_total >= 15
```

evaluates to FALSE, and the ELSE part of the IF-THEN-ELSE statement is executed. As a result, the statement

```
DBMS_OUTPUT.PUT_LINE ('Section 1 of course 25 is not full');
```

is executed.

C) What DBMS_OUTPUT.PUT_LINE statement is displayed if there is no section 1 for course number 25?

ANSWER: If there is no section 1 for course number 25, the ELSE part of the IF-THEN-ELSE statement is executed. So the second DBMS_OUTPUT.PUT_LINE statement is displayed on the screen.

The COUNT function used in the SELECT statement:

```
SELECT COUNT(*)
  INTO v_total
  FROM enrollment e
  JOIN section s USING (section_id)
 WHERE s.course_no = 25
   AND s.section_no = 1;
```

returns 0. The condition of the IF-THEN-ELSE statement evaluates to FALSE. Therefore, the ELSE part of the IF-THEN-ELSE statement is executed, and the second DBMS_OUTPUT.PUT_LINE statement is displayed on the screen.

D) How would you change this script so that the user provides both course and section numbers?

ANSWER: Two additional variables must be declared and initialized with the help of the substitution variables as follows. The script should look similar to the following. Changes are shown in bold.

```
-- ch04_2b.sql, version 2.0
SET SERVEROUTPUT ON
DECLARE
   v_total      NUMBER;
   v_course_no  CHAR(6) := '&sv_course_no';
   v_section_no NUMBER  := &sv_section_no;
BEGIN
   SELECT COUNT(*)
     INTO v_total
     FROM enrollment e
```

```
      JOIN section s USING (section_id)
   WHERE s.course_no  = v_course_no
     AND s.section_no = v_section_no;

   -- check if a specific section of a course is full
   IF v_total >= 15 THEN
      DBMS_OUTPUT.PUT_LINE
         ('Section 1 of course 25 is full');
   ELSE
      DBMS_OUTPUT.PUT_LINE
         ('Section 1 of course 25 is not full');
   END IF;
   -- control resumes here
END;
```

E) How would you change this script so that if fewer than 15 students are enrolled in section 1 of course number 25, a message appears indicating how many students can still enroll?

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

```
-- ch04_2c.sql, version 3.0
SET SERVEROUTPUT ON
DECLARE
   v_total    NUMBER;
   v_students NUMBER;
BEGIN
   SELECT COUNT(*)
     INTO v_total
     FROM enrollment e
     JOIN section s USING (section_id)
    WHERE s.course_no = 25
      AND s.section_no = 1;

   -- check if section 1 of course 25 is full
   IF v_total >= 15 THEN
      DBMS_OUTPUT.PUT_LINE
         ('Section 1 of course 25 is full');
   ELSE
      v_students := 15 - v_total;
      DBMS_OUTPUT.PUT_LINE (v_students||
         ' students can still enroll into section 1'||
         ' of course 25');
   END IF;
   -- control resumes here
END;
```

Notice that if the IF-THEN-ELSE statement evaluates to FALSE, the statements associated with the ELSE part are executed. In this case, the value of the variable `v_total` is subtracted from 15. The result of this operation indicates how many more students can enroll in section 1 of course number 25.

LAB 4.2
ELSIF Statements

LAB OBJECTIVES

After completing this lab, you will be able to

- Use the ELSIF statement

An ELSIF statement has the following structure:

```
IF CONDITION 1 THEN
   STATEMENT 1;
ELSIF CONDITION 2 THEN
   STATEMENT 2;
ELSIF CONDITION 3 THEN
   STATEMENT 3;
...
ELSE
   STATEMENT N;
END IF;
```

The reserved word IF marks the beginning of an ELSIF construct. *CONDITION 1* through *CONDITION N* are a sequence of the conditions that evaluate to TRUE or FALSE. These conditions are mutually exclusive. In other words, if *CONDITION 1* evaluates to TRUE, *STATEMENT 1* is executed, and control is passed to the first executable statement after the reserved phrase END IF. The rest of the ELSIF construct is ignored. When *CONDITION 1* evaluates to FALSE, control is passed to the ELSIF part and *CONDITION 2* is evaluated, and so forth. If none of the specified conditions yields TRUE, control is passed to the ELSE part of the ELSIF construct. An ELSIF statement can contain any number of ELSIF clauses. Figure 4.3 shows this flow of logic.

Figure 4.3 shows that if condition 1 evaluates to TRUE, statement 1 is executed, and control is passed to the first statement after END IF. If condition 1 evaluates to FALSE, control is passed to condition 2. If condition 2 yields TRUE, statement 2 is executed. Otherwise, control is passed to the statement following END IF, and so forth. Consider the following example.

FOR EXAMPLE

```
DECLARE
   v_num NUMBER := &sv_num;
BEGIN
   IF v_num < 0 THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
```

```
   ELSIF v_num = 0 THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is equal to zero');
   ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
  END IF;
END;
```

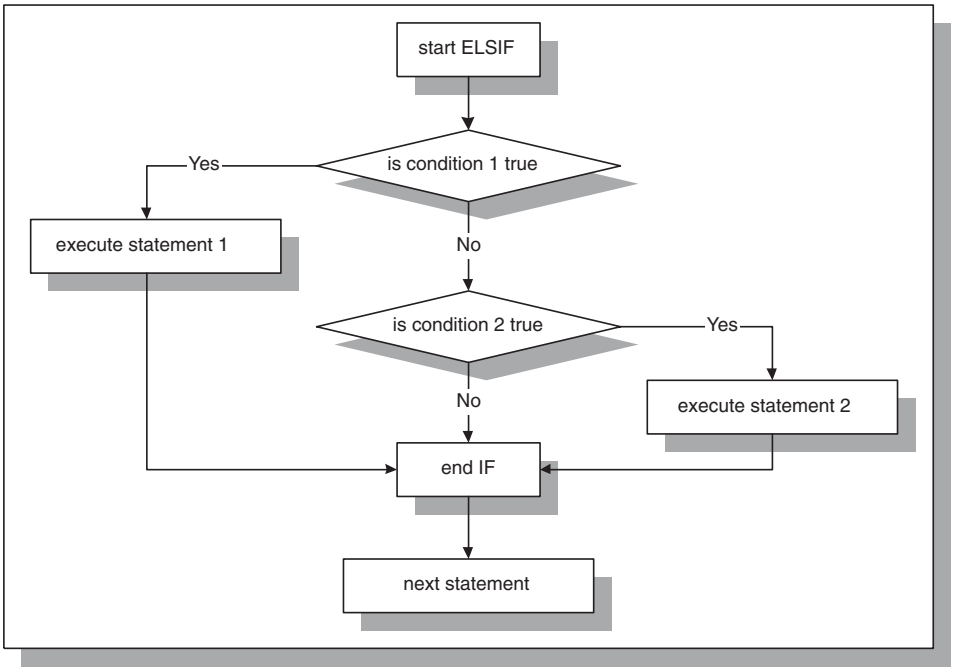

FIGURE 4.3
ELSIF statement

The value of v_num is provided at runtime and is evaluated with the help of the ELSIF statement. If the value of v_num is less than 0, the first DBMS_ OUTPUT.PUT_LINE statement executes, and the ELSIF construct terminates. If the value of v_num is greater than 0, both conditions

```
v_num < 0
```

and

```
v_num = 0
```

evaluate to FALSE, and the ELSE part of the ELSIF construct executes.

Assume that the value of v_num equals 5 at runtime. This example produces the following output:

```
Enter value for sv_num: 5
old   2:    v_num  NUMBER := &sv_num;
new   2:    v_num  NUMBER := 5;
5 is a positive number

PL/SQL procedure successfully completed.
```

DID YOU KNOW?

Consider the following information about an ELSIF statement:

- Always match an IF with an END IF.
- There must be a space between END and IF. If the space is omitted, the compiler produces the following error:

  ```
  ERROR at line 22:
  ORA-06550: line 22, column 4:
  PLS-00103: Encountered the symbol ";" when
  expecting one of the following: if
  ```

  As you can see, this error message is not very clear, and it can take you some time to correct it, especially if you have not encountered it before.
- There is no second E in ELSIF.
- Conditions of an ELSIF statement must be mutually exclusive. These conditions are evaluated in sequential order, from the first to the last. When a condition evaluates to TRUE, the remaining conditions of the ELSIF statement are not evaluated. Consider this example of an ELSIF construct:

  ```
  IF v_num >= 0 THEN
     DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');
  ELSIF v_num =< 10 THEN
     DBMS_OUTPUT.PUT_LINE ('v_num is less than 10');
  ELSE
     DBMS_OUTPUT.PUT_LINE
           ('v_num is less than ? or greater than ?');
  END IF;
  ```

Assume that the value of `v_num` is equal to 5. Both conditions of the ELSIF statement can evaluate to TRUE because 5 is greater than 0, and 5 is less than 10. However, when the first condition, `v_num >= 0`, evaluates to TRUE, the rest of the ELSIF construct is ignored.

For any value of `v_num` that is greater than or equal to 0 and less than or equal to 10, these conditions are not mutually exclusive. Therefore, the DBMS_OUTPUT.PUT_LINE statement associated with the ELSIF clause does not execute for any such value of `v_num`. For the second condition, `v_num <= 10`, to yield TRUE, the value of `v_num` must be less than 0.

How would you rewrite this ELSIF construct to capture any value of `v_num` between 0 and 10 and display it on the screen with a single condition?

When using an ELSIF construct, it is not necessary to specify what action should be taken if none of the conditions evaluates to TRUE. In other words, an ELSE clause is not required in the ELSIF construct. Consider the following example:

FOR EXAMPLE

```
DECLARE
   v_num NUMBER := &sv_num;
BEGIN
   IF v_num < 0 THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
   ELSIF v_num > 0 THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
   END IF;
   DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

As you can see, no action is specified when v_num is equal to 0. If the value of v_num is equal to 0, both conditions evaluate to FALSE, and the ELSIF statement does not execute. When a value of 0 is specified for v_num, this example produces the following output:

```
Enter value for sv_num: 0
old   2:    v_num  NUMBER := &sv_num;
new   2:    v_num  NUMBER := 0;
Done...

PL/SQL procedure successfully completed.
```

DID YOU KNOW?

You probably noticed that for all IF statement examples, the reserved words IF, ELSIF, ELSE, and END IF are entered on a separate line and are aligned with the word IF. In addition, all executable statements in the IF construct are indented. The format of the IF construct makes no difference to the compiler; however, the meaning of the formatted IF construct becomes obvious to us.

This IF-THEN-ELSE statement:

```
IF x = y THEN v_txt := 'YES'; ELSE v_txt :=
'NO'; END IF;
```

is equivalent to

```
IF x = y THEN
   v_txt := 'YES';
ELSE
   v_txt := 'NO';
END IF;
```

The formatted version of the IF construct is easier to read and understand.

LAB 4.2 EXERCISES

This section provides exercises and suggested answers, with discussion related to how those answers resulted. The most important thing to realize is whether your answer works. You should figure out the implications of the answers and what the effects are of any different answers you may come up with.

4.2.1 Use the ELSIF Statement

In this exercise, you use an ELSIF statement to display a letter grade for a student registered for a specific section of course number 25.

Create the following PL/SQL script:

```
-- ch04_3a.sql, version 1.0
SET SERVEROUTPUT ON
DECLARE
   v_student_id   NUMBER := 102;
   v_section_id   NUMBER := 89;
   v_final_grade  NUMBER;
   v_letter_grade CHAR(1);
BEGIN
   SELECT final_grade
     INTO v_final_grade
     FROM enrollment
    WHERE student_id = v_student_id
      AND section_id = v_section_id;

   IF v_final_grade BETWEEN 90 AND 100 THEN
      v_letter_grade := 'A';
   ELSIF v_final_grade BETWEEN 80 AND 89 THEN
      v_letter_grade := 'B';
   ELSIF v_final_grade BETWEEN 70 AND 79 THEN
      v_letter_grade := 'C';
   ELSIF v_final_grade BETWEEN 60 AND 69 THEN
      v_letter_grade := 'D';
   ELSE
      v_letter_grade := 'F';
   END IF;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Letter grade is: '||
      v_letter_grade);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('There is no such student or section');
END;
```

Note that you may need to change the values for the variables `v_student_id` and `v_section_id` as you see fit to test some of your answers.

Try to answer the following questions, and then execute the script:

A) What letter grade is displayed on the screen:

- **i)** if the value of `v_final_grade` is equal to 85?
- **ii)** if the value of `v_final_grade` is NULL?
- **iii)** if the value of `v_final_grade` is greater than 100?

ANSWER:

i) If the value of `v_final_grade` is equal to 85, the value "B" of the letter grade is displayed on the screen.

The conditions of the ELSIF statement are evaluated in sequential order. The first condition:

```
v_final_grade BETWEEN 90 AND 100
```

evaluates to FALSE, and control is passed to the first ELSIF part of the ELSIF statement. Then, the second condition:

```
v_final_grade BETWEEN 80 AND 89
```

evaluates to TRUE, and the letter "B" is assigned to the variable `v_letter_grade`. Control is then passed to the first executable statement after END IF, and the message

```
Letter grade is: B
```

is displayed on the screen.

ii) If the value of `v_final_grade` is NULL, value "F" of the letter grade is displayed on the screen.

If the value of the `v_final_grade` is undefined or NULL, all conditions of the ELSIF statement evaluate to NULL (notice that they do not evaluate to FALSE). As a result, the ELSE part of the ELSIF statement is executed, and letter "F" is assigned to the `v_letter_grade`.

iii) If the value of `v_final_grade` is greater than 100, value "F" of the letter grade is displayed on the screen.

The conditions specified for the ELSIF statement cannot handle a value of `v_final_grade` greater than 100. Therefore, any student who should receive a letter grade of A+ will instead receive a letter grade of "F." After the ELSIF statement has terminated, `The letter grade is: F` is displayed on the screen.

B) How would you change this script so that the message `v_final_grade is null` is displayed on the screen if `v_final_grade` is NULL?

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

```
-- ch04_3b.sql, version 2.0
SET SERVEROUTPUT ON
DECLARE
   v_student_id   NUMBER := 102;
   v_section_id   NUMBER := 89;
   v_final_grade  NUMBER;
   v_letter_grade CHAR(1);
BEGIN
   SELECT final_grade
     INTO v_final_grade
     FROM enrollment
    WHERE student_id = v_student_id
      AND section_id = v_section_id;

   IF v_final_grade IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('v_final_grade is null');
   ELSIF v_final_grade BETWEEN 90 AND 100 THEN
      v_letter_grade := 'A';
```

```
   ELSIF v_final_grade BETWEEN 80 AND 89 THEN
      v_letter_grade := 'B';
   ELSIF v_final_grade BETWEEN 70 AND 79 THEN
      v_letter_grade := 'C';
   ELSIF v_final_grade BETWEEN 60 AND 69 THEN
      v_letter_grade := 'D';
   ELSE
      v_letter_grade := 'F';
   END IF;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Letter grade is: '||
      v_letter_grade);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('There is no such student or section');
END;
```

One more condition has been added to the ELSIF statement. The condition

```
v_final_grade BETWEEN 90 AND 100
```

becomes the first ELSIF condition. Now, if the value of `v_final_grade` is NULL, the message `v_final_grade is null` is displayed on the screen. However, no value is assigned to the variable `v_letter_grade`. The message `Letter grade is:` is displayed on the screen as well.

C) How would you change this script so that the user provides the student ID and section ID?

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

```
-- ch04_3c.sql, version 3.0
SET SERVEROUTPUT ON
DECLARE
   v_student_id   NUMBER := &sv_student_id;
   v_section_id   NUMBER := &sv_section_id;
   v_final_grade  NUMBER;
   v_letter_grade CHAR(1);
BEGIN
   SELECT final_grade
     INTO v_final_grade
     FROM enrollment
    WHERE student_id = v_student_id
      AND section_id = v_section_id;

   IF v_final_grade BETWEEN 90 AND 100 THEN
      v_letter_grade := 'A';
   ELSIF v_final_grade BETWEEN 80 AND 89 THEN
      v_letter_grade := 'B';
   ELSIF v_final_grade BETWEEN 70 AND 79 THEN
      v_letter_grade := 'C';
   ELSIF v_final_grade BETWEEN 60 AND 69 THEN
      v_letter_grade := 'D';
```

```
   ELSE
      v_letter_grade := 'F';
   END IF;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Letter grade is: '||
      v_letter_grade);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('There is no such student or section');
END;
```

D) How would you change the script to define a letter grade without specifying the upper limit of the final grade? In the statement `v_final_grade BETWEEN 90 and 100`, number 100 is the upper limit.

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

```
-- ch04_3d.sql, version 4.0
SET SERVEROUTPUT ON
DECLARE
   v_student_id   NUMBER := 102;
   v_section_id   NUMBER := 89;
   v_final_grade  NUMBER;
   v_letter_grade CHAR(1);
BEGIN
   SELECT final_grade
     INTO v_final_grade
     FROM enrollment
    WHERE student_id = v_student_id
      AND section_id = v_section_id;

   IF v_final_grade >= 90 THEN
      v_letter_grade := 'A';
   ELSIF v_final_grade >= 80 THEN
      v_letter_grade := 'B';
   ELSIF v_final_grade >= 70 THEN
      v_letter_grade := 'C';
   ELSIF v_final_grade >= 60 THEN
      v_letter_grade := 'D';
   ELSE
      v_letter_grade := 'F';
   END IF;

   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Letter grade is: '||
      v_letter_grade);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('There is no such student or section');
END;
```

In this example, no upper limit is specified for the variable `v_final_grade` because the BETWEEN operator has been replaced with the >= operator. Thus, this script can handle a value of `v_final_grade` that is greater than 100. Instead of assigning letter "F" to `v_letter_grade` (in version 1.0 of the script), the letter "A" is assigned to the variable `v_letter_grade`. As a result, this script produces more accurate results.

LAB 4.3
Nested IF Statements

LAB OBJECTIVES

After completing this lab, you will be able to

- Use nested IF statements

You have encountered different types of conditional controls: the IF-THEN statement, the IF-THEN-ELSE statement, and the ELSIF statement. These types of conditional controls can be nested inside one another. For example, an IF statement can be nested inside an ELSIF, and vice versa. Consider the following:

FOR EXAMPLE

```
DECLARE
   v_num1  NUMBER := &sv_num1;
   v_num2  NUMBER := &sv_num2;
   v_total NUMBER;
BEGIN
   IF v_num1 > v_num2 THEN
      DBMS_OUTPUT.PUT_LINE ('IF part of the outer IF');
      v_total := v_num1 - v_num2;
   ELSE
      DBMS_OUTPUT.PUT_LINE ('ELSE part of the outer IF');
      v_total := v_num1 + v_num2;

      IF v_total < 0 THEN
         DBMS_OUTPUT.PUT_LINE ('Inner IF');
         v_total := v_total * (-1);
      END IF;

   END IF;
   DBMS_OUTPUT.PUT_LINE ('v_total = '||v_total);
END;
```

The IF-THEN-ELSE statement is called an outer IF statement because it encompasses the IF-THEN statement (shown in bold). The IF-THEN statement is called an inner IF statement because it is enclosed by the body of the IF-THEN-ELSE statement.

Assume that the values for v_num1 and v_num2 are –4 and 3, respectively. First, the condition

```
v_num1 > v_num2
```

of the outer IF statement is evaluated. Because –4 is not greater than 3, the ELSE part of the outer IF statement is executed. As a result, the message

```
ELSE part of the outer IF
```

is displayed, and the value of v_total is calculated. Next, the condition

```
v_total < 0
```

of the inner IF statement is evaluated. Because that value of v_total is equal to –l, the condition yields TRUE, and the message

```
Inner IF
```

is displayed. Next, the value of v_total is calculated again. This logic is demonstrated by the output that the example produces:

```
Enter value for sv_num1: -4
old   2:    v_num1  NUMBER := &sv_num1;
new   2:    v_num1  NUMBER := -4;
Enter value for sv_num2: 3
old   3:    v_num2  NUMBER := &sv_num2;
new   3:    v_num2  NUMBER := 3;
ELSE part of the outer IF
Inner IF
v_total = 1

PL/SQL procedure successfully completed.
```

LOGICAL OPERATORS

So far in this chapter, you have seen examples of different IF statements. All of these examples used test operators, such as >, <, and =, to test a condition. Logical operators can be used to evaluate a condition as well. In addition, they allow a programmer to combine multiple conditions into a single condition if such a need exists.

FOR EXAMPLE

```
DECLARE
   v_letter CHAR(1) := '&sv_letter';
BEGIN
   IF (v_letter >= 'A' AND v_letter <= 'Z') OR
      (v_letter >= 'a' AND v_letter <= 'z')
   THEN
      DBMS_OUTPUT.PUT_LINE ('This is a letter');
   ELSE
      DBMS_OUTPUT.PUT_LINE ('This is not a letter');

      IF v_letter BETWEEN '0' and '9' THEN
```

```
         DBMS_OUTPUT.PUT_LINE ('This is a number');
      ELSE
         DBMS_OUTPUT.PUT_LINE ('This is not a number');
      END IF;

   END IF;
END;
```

In this example, the condition

```
(v_letter >= 'A' AND v_letter <= 'Z') OR
(v_letter >= 'a' AND v_letter <= 'z')
```

uses the logical operators AND and OR. Two conditions:

```
(v_letter >= 'A' AND v_letter <= 'Z')
```

and

```
(v_letter >= 'a' AND v_letter <= 'z')
```

are combined into one with the help of the OR operator. It is also important to understand the purpose of the parentheses. In this example, they are only used to improve readability, because the AND operator takes precedence over the OR operator.

When the symbol `?` is entered at runtime, this example produces the following output:

```
Enter value for sv_letter: ?
old   2:    v_letter CHAR(1) := '&sv_letter';
new   2:    v_letter CHAR(1) := '?';
This is not a letter
This is not a number

PL/SQL procedure successfully completed.
```

LAB 4.3 EXERCISES

This section provides exercises and suggested answers, with discussion related to how those answers resulted. The most important thing to realize is whether your answer works. You should figure out the implications of the answers and what the effects are of any different answers you may come up with.

4.3.1 Use Nested IF Statements

In this exercise, you use nested IF statements. This script converts the value of a temperature from one system to another. If the temperature is supplied in Fahrenheit, it is converted to Celsius, and vice versa.

Create the following PL/SQL script:

```
-- ch04_4a.sql, version 1.0
SET SERVEROUTPUT ON
DECLARE
   v_temp_in   NUMBER := &sv_temp_in;
   v_scale_in  CHAR   := '&sv_scale_in';
```

```
   v_temp_out  NUMBER;
   v_scale_out CHAR;
BEGIN
   IF v_scale_in != 'C' AND v_scale_in != 'F' THEN
      DBMS_OUTPUT.PUT_LINE ('This is not a valid scale');
   ELSE
      IF v_scale_in = 'C' THEN
         v_temp_out  := ( (9 * v_temp_in) / 5 ) + 32;
         v_scale_out := 'F';
      ELSE
         v_temp_out  := ( (v_temp_in - 32) * 5 ) / 9;
         v_scale_out := 'C';
      END IF;
      DBMS_OUTPUT.PUT_LINE ('New scale is: '||v_scale_out);
      DBMS_OUTPUT.PUT_LINE ('New temperature is: '||v_temp_out);
   END IF;
END;
```

Execute the script, and then answer the following questions:

A) What output is printed on the screen if the value of 100 is entered for the temperature, and the letter "C" is entered for the scale?

ANSWER: The output should look like the following:

```
Enter value for sv_temp_in: 100
old   2:    v_temp_in    NUMBER := &sv_temp_in;
new   2:    v_temp_in    NUMBER := 100;
Enter value for sv_scale_in: C
old   3:    v_scale_in   CHAR := '&sv_scale_in';
new   3:    v_scale_in   CHAR := 'C';
New scale is: F
New temperature is: 212

PL/SQL procedure successfully completed.
```

After the values for `v_temp_in` and `v_scale_in` have been entered, the condition

```
v_scale_in != 'C' AND v_scale_in != 'F'
```

of the outer IF statement evaluates to FALSE, and control is passed to the ELSE part of the outer IF statement. Next, the condition

```
v_scale_in = 'C'
```

of the inner IF statement evaluates to TRUE, and the values of the variables `v_temp_out` and `v_scale_out` are calculated. Control is then passed back to the outer IF statement, and the new value for the temperature and the scale are displayed on the screen.

B) Try to run this script without providing a value for the temperature. What message is displayed on the screen? Why?

ANSWER: If the value for the temperature is not entered, the script does not compile.

The compiler tries to assign a value to `v_temp_in` with the help of the substitution variable. Because the value for `v_temp_in` has not been entered, the assignment statement fails, and the following error message is displayed:

```
Enter value for sv_temp_in:
old   2:    v_temp_in    NUMBER := &sv_temp_in;
new   2:    v_temp_in    NUMBER := ;
Enter value for sv_scale_in: C
old   3:    v_scale_in   CHAR := '&sv_scale_in';
new   3:    v_scale_in   CHAR := 'C';
   v_temp_in    NUMBER := ;
                          *
ERROR at line 2:
ORA-06550: line 2, column 27:
PLS-00103: Encountered the symbol ";" when expecting one of the
following:
( - + mod not null <an identifier>
<a double-quoted delimited-identifier> <a bind variable> avg
count current exists max min prior sql stddev sum variance
cast <a string literal with character set specification>
<a number> <a single-quoted SQL string>
The symbol "null" was substituted for ";" to continue.
```

You have probably noticed that even though the mistake seems small and insignificant, the error message is fairly long and confusing.

C) Try to run this script providing an invalid letter for the temperature scale, such as "V." What message is displayed on the screen, and why?

ANSWER: If an invalid letter is entered for the scale, the message `This is not a valid scale` is displayed on the screen.

The condition of the outer IF statement evaluates to TRUE. As a result, the inner IF statement is not executed, and the message `This is not a valid scale` is displayed on the screen.

Assume that letter "V" was typed by mistake. This example produces the following output:

```
Enter value for sv_temp_in: 45
old   2:    v_temp_in    NUMBER := &sv_temp_in;
new   2:    v_temp_in    NUMBER := 45;
Enter value for sv_scale_in: V
old   3:    v_scale_in   CHAR := '&sv_scale_in';
new   3:    v_scale_in   CHAR := 'V';
This is not a valid scale

PL/SQL procedure successfully completed.
```

D) Rewrite this script so that if an invalid letter is entered for the scale, `v_temp_out` is initialized to 0 and `v_scale_out` is initialized to C.

ANSWER: The script should look similar to the following. Changes are shown in bold. Notice that the two final DBMS_OUTPUT.PUT_LINE statements have been moved from the body of the outer IF statement.

```
-- ch04_4b.sql, version 2.0
DECLARE
   v_temp_in   NUMBER := &sv_temp_in;
   v_scale_in  CHAR   := '&sv_scale_in';
   v_temp_out  NUMBER;
   v_scale_out CHAR;
```

```
BEGIN
   IF v_scale_in != 'C' AND v_scale_in != 'F' THEN
      DBMS_OUTPUT.PUT_LINE ('This is not a valid scale');
      v_temp_out  := 0;
      v_scale_out := 'C';
   ELSE
      IF v_scale_in = 'C' THEN
         v_temp_out  := ( (9 * v_temp_in) / 5 ) + 32;
         v_scale_out := 'F';
      ELSE
         v_temp_out  := ( (v_temp_in - 32) * 5 ) / 9;
         v_scale_out := 'C';
      END IF;
   END IF;
   DBMS_OUTPUT.PUT_LINE ('New scale is: '||v_scale_out);
   DBMS_OUTPUT.PUT_LINE ('New temperature is: '||v_temp_out);
END;
```

The preceding script produces the following output:

```
Enter value for sv_temp_in: 100
old   2:    v_temp_in    NUMBER := &sv_temp_in;
new   2:    v_temp_in    NUMBER := 100;
Enter value for sv_scale_in: V
old   3:    v_scale_in   CHAR := '&sv_scale_in';
new   3:    v_scale_in   CHAR := 'V';
This is not a valid scale.
New scale is: C
New temperature is: 0

PL/SQL procedure successfully completed.
```

TRY IT YOURSELF

In this chapter you've learned about different types of IF statements. You've also learned that all these different IF statements can be nested inside one another. Here are some exercises that will help you test the depth of your understanding:

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.
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.
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;
```

The projects in this section are meant to have you use all the skills you have acquired throughout this chapter. The answers to these projects can be found in Appendix D and on this book's companion Web site. Visit the Web site periodically to share and discuss your answers.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z