



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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
800-382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: www.informit.com/ph

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

Copyright © 2009 Pearson Education, Inc. All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise.

For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671 3447

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')
```

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',...)
```

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 search 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`.

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.

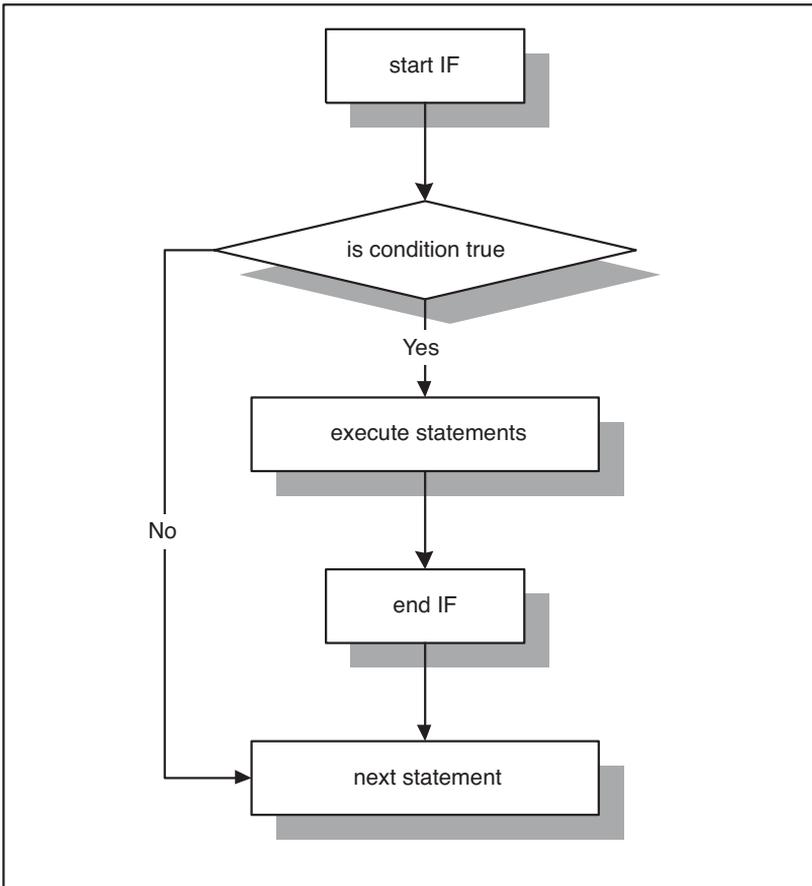


FIGURE 4.1
IF-THEN statement

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.

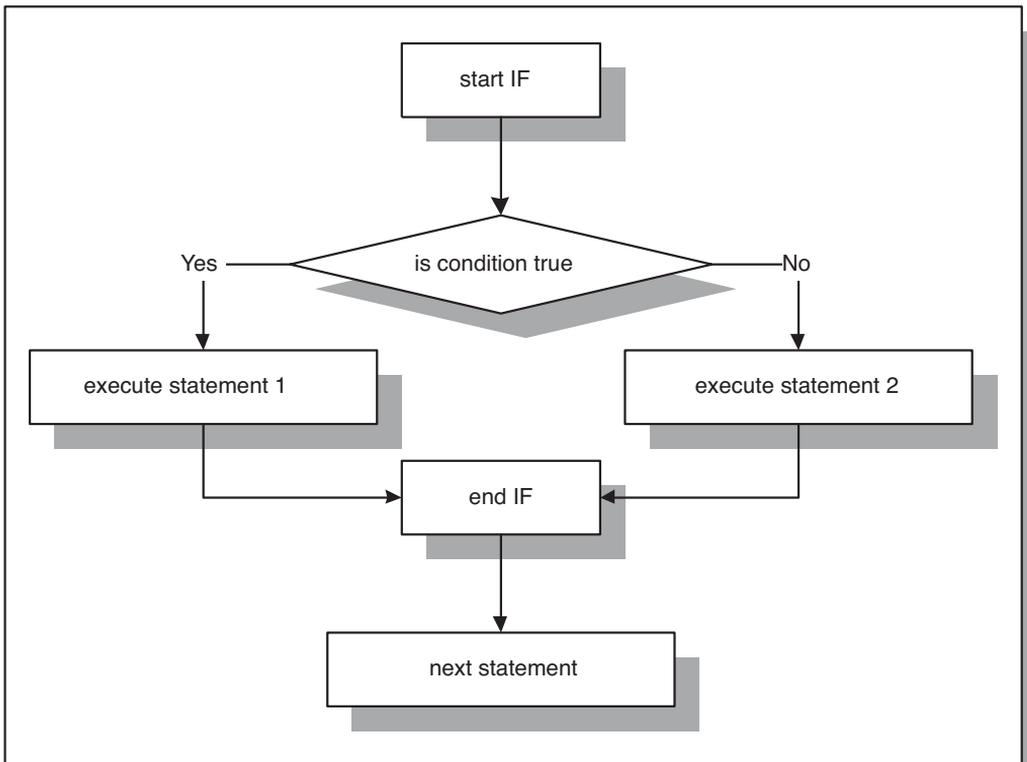


FIGURE 4.2
IF-THEN-ELSE statement

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;
```

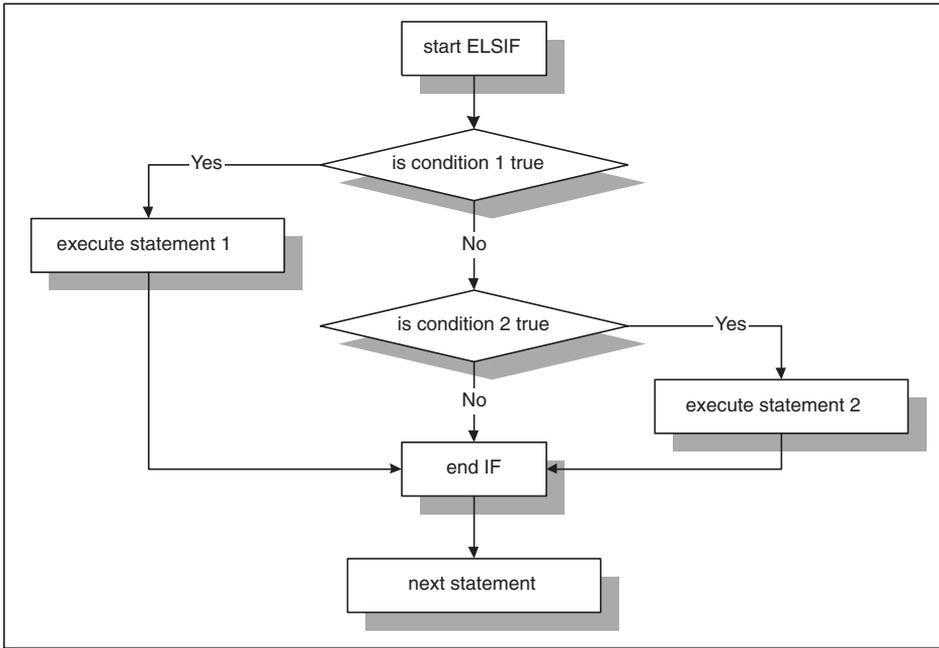


FIGURE 4.3
ELSIF statement

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 `-1`, 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

& (ampersand), 13-15, 25

+ (plus sign), 24

A

accessing sequences, 43

actual parameters
(procedures), 444-445

AFTER triggers, 269-270,
274-276

aliases for columns, 245

ALTER SYSTEM command, 564

ALTER TRIGGER statement,
265-266

American National
Standards Institute. *See*
ANSI SQL standards

ampersand (&), 13-15, 25

anchored datatypes, 28-29

anonymous blocks, 4, 8-10, 440

ANSI SQL standards

joins

CROSS JOIN syntax,
607-608

EQUI JOIN syntax,
608-609

NATURAL JOIN
syntax, 609-610

OUTER JOIN syntax,
610-611

overview, 607

scalar subqueries, 611

Application Server 10g. *See*

Oracle Application Server 10g

application server tier (Oracle
Application Server 10g), 580

architecture, client/server

anonymous blocks,
4, 8-10

declaration section
(blocks), 5

exception-handling
section (blocks), 6-7

executable section
(blocks), 5-6

executing blocks,
7, 10-11

named blocks, 4

overview, 2-4

area of circle, calculating,
19, 613

arrays

associative arrays

declaring, 317

examples, 317-319

name_tab, 652

sample script, 326-330

varrays

city_varray, 338-341

collection methods,
336-338

course_varray,
653-657

creating, 334-335

- definition, 334
- name_varray, 652-653
- NULL varrays, 335-336
- runtime errors, 656-659
- sample script, 338-341

assignment operator, 31

associative arrays

- declaring, 317
- examples, 317-319
- name_tab, 652
- sample script, 326-330

attributes, cursor, 240-242

automatic subprogram inlining, xxv-xxvi

AUTONOMOUS_TRANSACTION pragma, 270-272

autonomous transactions, 270-272

B

BEFORE triggers, 267-269, 274-276

BEGIN keyword, 6

binary files, reading, 559

BINARY_INTEGER datatype, 30

bind arguments, 380

blocks

- anonymous blocks, 4, 8-10, 440
- block structure, 440
- declaration section, 5
- DML in, 42-43
- exception-handling section, 6-7
- executable section, 5-6
- executing, 7, 10-11

- labels, 35-36
- multiple transactions in, 50
- named blocks, 4
- nested blocks, 35-36
- overview, 4
- sequences in, 44
- try-it-yourself projects, 37, 614-616
- writing, 37, 614-616

body of packages

- creating, 462-464
- rules for, 460
- syntax, 459-460

BOOLEAN datatype, 30

Boolean expressions, terminating loops with, 126

BROKEN procedure, 564

built-in exceptions. See exceptions

BULK COLLECT statement

- fetching data into collections, 425-426
- fetching records with, 422-423
- with FORALL statement, 427-428
- LIMIT option, 423-425
- RETURNING option, 426-427
- sample scripts, 428-436
- structure, 422

bulk SQL

BULK COLLECT statement

- fetching data into collections, 425-426
- fetching records with, 422-423

- with FORALL statement, 427-428
- LIMIT option, 423-425
- RETURNING option, 426-427
- sample scripts, 428-436
- structure, 422

FORALL statement

- BULK COLLECT clause, 427-428
- INDICES OF option, 410
- sample script, 413-421
- SAVE EXCEPTIONS option, 408-409
- simple examples, 405-408
- structure, 404-405
- VALUES OF option, 411-412
- overview, 403
- try-it-yourself projects, 437, 665-672

C

c_course cursor, 257

c_grade cursor, 257

c_grades cursor, 481

c_grade_type cursor, 480

c_student cursor, 257

c_zip cursor, 254

calculating

- area of circle, 19, 613
- factorial of 10, 137-138
- sum of integers between 1 and 10, 128-131

- calling stored packages, 464-465
- CAPACITY column (SECTION table), 602
- case, formatting, 597
- CASE expressions
 - differences between CASE statement and CASE expression, 97-99
 - displaying letter grades for students, 100-102
 - example, 96-97
 - overview, 96
- CASE statement
 - displaying name of day, 89-91
 - examples, 83-84
 - overview, 82
 - searched CASE statements
 - differences between CASE and searched CASE, 86-89
 - differences between CASE statement and CASE expression, 97-99
 - displaying letter grade for student, 91-95
 - example, 86
 - syntax, 84-85
 - try-it-yourself projects, 112, 622-626
- Celsius/Fahrenheit conversion script, 76-79
- CHANGE procedure, 564
- CHAP4 table, creating, 616
- CHAP4_SEQ sequence, creating, 616
- CHAR datatype, 29
- character types, 22-23
- circles, calculating area of, 19, 613-614
- CITY column (ZIPCODE table), 603
- city_varray, 338-341
- clauses. *See* statements
- client-side HTML image maps, 592-593
- client/server architecture, PL/SQL in
 - blocks
 - anonymous blocks, 4, 8-10
 - declaration section, 5
 - exception-handling section, 6-7
 - executable section, 5-6
 - executing, 7, 10-11
 - named blocks, 4
 - overview, 4
 - overview, 2-4
 - try-it-yourself projects, 19, 613-614
- client tier (Oracle Application Server 10g), 580
- CLOSE statement
 - sample script, 395-400
 - syntax, 394-395
- closing cursors, 237-240, 394-395
- COALESCE function, 26, 105-106, 626
 - evaluating list of numbers, 109-111
 - example, 106-107
 - syntax, 105
- collection methods
 - applying to PL/SQL tables, 322-325
 - applying to varrays, 336-338
 - COUNT, 322
 - DELETE, 323, 337
 - EXISTS, 322
 - EXTEND, 323
 - FIRST, 323
 - LAST, 323
 - NEXT, 323
 - PRIOR, 323
 - TRIM, 323
- collections
 - collection methods, 322-325
 - COUNT, 322
 - DELETE, 323, 337
 - example, 323-325
 - EXISTS, 322
 - EXTEND, 323
 - FIRST, 323
 - LAST, 323
 - NEXT, 323
 - PRIOR, 323
 - TRIM, 323
 - definition, 315
 - empty collections, 322
 - fetching data into, 425-426
 - multilevel collections, 342-347
 - NULL collections, 322
 - of object types, 520-522, 526-530

- PL/SQL tables
 - associative arrays, 317-319, 326-330
 - definition, 316-317
 - nested tables, 319-321, 330-333
- of records, 373-377
- try-it-yourself projects, 348, 652-659
- varrays
 - city_varray, 338-341
 - collection methods, 336-338
 - creating, 334-335
 - definition, 334
 - NULL varrays, 335-336
 - sample script, 338-341
- columns**
 - aliases, 245
 - COURSE table columns, 601
 - ENROLLMENT table columns, 602
 - GRADE table columns, 605
 - GRADE_CONVERSION table columns, 605
 - GRADE_TYPE table columns, 604
 - GRADE_TYPE_WEIGHT table columns, 604
 - INSTRUCTOR table columns, 603
 - SECTION table columns, 601-602
 - STUDENT table columns, 602
 - ZIPCODE table columns, 603
- commands. See statements**
- comments**
 - definition, 23
 - formatting guidelines, 598-599
- COMMENTS column (GRADE table), 605**
- COMMIT statement, 46-47**
- committing transactions, 46-47**
- comparing**
 - expressions with NULLIF function, 103-104, 107-109
 - objects
 - map methods, 538-541
 - order methods, 541-544
- compatibility of records, 355-357**
- compiler, xxvii**
- complex functions, 454**
- compound triggers, xxiv**
 - capabilities, 300
 - examples, 302-306
 - modifying, 306-312
 - restrictions, 301
 - structure, 300-301
 - try-it-yourself projects, 313, 648-651
- conditional control**
 - CASE expressions
 - differences between CASE statement and CASE expression, 97-99
 - displaying letter grades for students, 100-102
 - example, 96-97
 - overview, 96
 - CASE statement
 - differences between CASE statement and CASE expression, 97-99
 - displaying name of day, 89-91
 - examples, 83-84
 - overview, 82
 - searched CASE statements, 84-95
 - syntax, 82
 - try-it-yourself projects, 112, 622-626
 - COALESCE function, 105-106
 - evaluating list of numbers, 109-111
 - example, 106-107
 - syntax, 105
 - ELSEIF statement
 - conditions, 67
 - displaying letter grade for student, 69-73
 - example, 66
 - examples, 65-68
 - syntax, 65

- IF statements
 - IF-THEN, 54-61
 - IF-THEN-ELSE, 54-64
 - nested IF statements, 74-79
 - try-it-yourself projects, 80, 619-622
- NULLIF function, 104-105
 - displaying letter grades for students, 107-109
 - example, 103
 - restrictions, 104
 - syntax, 103
- overview, 53
- CONSTANT keyword, 5
- constraints (datatype), passing with parameter values, 445
- constructor methods, 531-534
- CONTINUE statement, xx
 - compared to EXIT statement, 148
 - overview, 144-145
 - sample script, 146-151
- CONTINUE WHEN statement
 - overview, 145
 - sample script, 152-153
 - try-it-yourself projects, 629-630
- converting Celsius/Fahrenheit, 76-79
- COST column (COURSE table), 601
- COUNT() function, 183, 322
- COURSE table, 601
- courses, checking number of students enrolled in, 62-64
- COURSE_NO column
 - COURSE table, 601
 - SECTION table, 601
- course_rec record, 350
- course_varray varray, 653-657
- CREATE FUNCTION statement, 450-451
- CREATE OR REPLACE TRIGGER clause, xxiv
- CREATE OR REPLACE TYPE clause, 514
- CREATE TRIGGER statement, 265
- CREATED_BY column
 - COURSE table, 601
 - ENROLLMENT table, 603
 - GRADE table, 605
 - GRADE_CONVERSION table, 605
 - GRADE_TYPE table, 604
 - INSTRUCTOR table, 603
 - SECTION table, 602
 - STUDENT table, 602
 - ZIPCODE table, 603
- CREATED_DATE column
 - COURSE table, 601
 - ENROLLMENT table, 603
 - GRADE table, 605
 - GRADE_CONVERSION table, 605
 - GRADE_TYPE table, 604
 - INSTRUCTOR table, 603
 - SECTION table, 602
 - STUDENT table, 602
 - ZIPCODE table, 603
- creating
 - cursor variables, 472
 - explain plans, 570-577
 - object types, 514-516
 - package bodies, 462-464
 - package variables, 469-470
 - private objects, 465-469
 - procedures, 441-442
 - triggers, 264-265, 272-273, 645-648
- CROSS JOINS, 607-608
- cross-section function result cache, xxiii-xxiv
- current_status procedure, 673-674
- cursor variables
 - capabilities, 471-472
 - compared to cursors, 472
 - creating, 472
 - handling, 473
 - rules for use, 479
 - sample cursor variable in package, 473-475
 - sample script, 475-479
 - strong cursor variables, 472
 - weak cursor variables, 472
- cursor-based records, 233-235
 - student_rec example, 350-353
 - zip_rec example, 358-362
- cursors
 - c_course cursor, 257
 - c_grade cursor, 257

c_grades, 481
 c_grade_type, 480
 c_student cursor, 257
 c_zip cursor, 254
 closing, 394-395
 column aliases, 245
 compared to cursor variables, 472
 creating, 469-470
 cursor-based records
 student_rec example, 350-353
 zip_rec example, 358-362
 declaring, 232-233
 definition, 229-230
 explicit cursors
 attributes, 240-242
 closing, 237-240
 declaring, 232
 definition, 230
 example, 242-245
 fetching rows in, 236-237
 opening, 236
 fetching, 393-394
 FOR loops, 246-247
 FOR UPDATE clause, 258-260
 implicit cursors
 capabilities, 231
 definition, 230
 example, 231-232
 nested cursors, 247-251, 255-257
 opening, 392-393
 parameters, 254-255

record types
 cursor-based records, 233-235
 table-based records, 233-234
 scope, 245
 SELECT list, 245
 try-it-yourself projects, 252, 643-645
 WHERE CURRENT clause, 261
 zip_cur, 371

D

Data Definition Language (DDL), 41
data dictionary, 443
data directory, querying for stored code information, 496-500
Database Resident Connection Pool (DRCP), xxv
database tier (Oracle Application Server 10g), 580
databases
 records. *See* records
 student database. *See* student database
 triggers
 AFTER triggers, 269-270, 274-276
 autonomous transactions, 270-272
 BEFORE triggers, 267-269, 274-276
 creating, 264-265, 272-273
 definition, 264

 disabling, 265-266
 INSTEAD OF triggers, 278-289
 mutating table issues, 292-299
 restrictions, 266-267
 row triggers, 277, 283-285
 statement triggers, 277-278, 283-285
 triggering events, 264
 try-it-yourself projects, 290, 645-648

datatypes

 anchored datatypes, 28-29
 BINARY_INTEGER, 30
 BOOLEAN, 30
 CHAR, 29
 DATE, 30
 FILE_TYPE, 560
 LOB (large object), 31
 LONG, 31
 LONG RAW, 31
 NUMBER, 29
 ROWID, 31
 SIMPLE_DOUBLE, xviii-xx
 SIMPLE_FLOAT, xviii-xx
 SIMPLE_INTEGER, xviii-xx
 TIMESTAMP, 30
 VARCHAR2, 29
DATE datatype, 30
dates
 DATE datatype, 30
 displaying name of day, 89-91

testing whether date falls on weekend, 58-61

DBMS_HPROF package, 556-559

DBMS_JOB package, 563-567

DBMS_LOB package, 559

DBMS_OUTPUT.PUT_LINE statement, 16-18

DBMS_PROFILER package, 556

DBMS_XPLAN package, 568-570

- generating explain plans, 570-577
- PLAN_TABLE, 568-569

DDL (Data Definition Language), 41

declaration section (blocks), 5

DECLARE keyword, 5

declaring

- associative arrays, 317
- cursors, 232-233
- user-defined exceptions, 188
- variables, 31-34

DELETE collection method, 323, 337

delimiters, 23-24

DEPTREE utility, 500

DESCRIPTION column (COURSE table), 601

DISABLE option (CREATE OR REPLACE TRIGGER clause), xxiv

disabling triggers, 265-266

display_student_count procedure, 466

DML

- Oracle sequences
 - accessing, 43
 - definition, 43
 - drawing numbers from, 43-44
 - in PL/SQL blocks, 44
 - incrementing values, 43
 - in PL/SQL blocks, 42-43
 - variable initialization with SELECT INTO, 40-42
- double ampersand (&&), 13-15
- DRCP (Database Resident Connection Pool), xxv
- DROP_LOWEST column (GRADE_TYPE table), 604
- drop_lowest flag, 483
- DUP_VALUE_ON_INDEX exception, 171
- dynamic SQL. *See* native dynamic SQL
- dynamic_sql_pkg, 677

E

e_exception1 exception, 201

e_exception2 exception, 201

e_invalid_id exception, 188-191

e_my_exception exception, 192

e_non_null_value exception, 641-642

e_no_sections exception, 206-208

e_Show_Exception_Scope variable, 36

e_too_many_sections exception, 193-196

e_too_many_students exception, 635-639

editing compound triggers, 306-312

elements of packages, referencing, 460

ELSEIF statement

- conditions, 67
- displaying letter grade for student, 69-73
- example, 66
- examples, 65-68
- syntax, 65

EMPLOYER column (STUDENT table), 602

empty collections, 322

ENABLE option (CREATE OR REPLACE TRIGGER clause), xxiv

END IF statement, 6, 54

END LOOP statement, 114

enforcing stored code purity level with RESTRICT_REFERENCES pragma, 500-506

ENROLLMENT table, 602

ENROLLMENT_OBJ_TYPE, 522

ENROLL_DATE column (ENROLLMENT table), 602

EQUI JOINS, 608-609

error handling. *See also* exceptions

- error messages, creating, 212-216
- mutating table errors, 292-299

- overview, 163
- RAISE_APPLICATION_ ERROR statement, 639-640
- runtime errors, 11, 164-167
- syntax errors, 11
- try-it-yourself projects, 178, 632-635
 - e_non_null_value exception, 641-642
 - e_too_many_students exception, 635-639
 - invalid instructor IDs, handling, 634-635
 - invalid student IDs, handling, 632-634
- evaluating expressions
 - with COALESCE function, 105-106, 109-111
- events, triggering, 264
- EXCEPTION keyword, 6, 188
- exception-handling section (blocks), 6-7
- exceptions
 - DUP_VALUE_ON_INDEX, 171
 - error messages
 - creating, 212-216
 - returning with SQLERRM function, 222-226
 - error number, returning
 - with SQLCODE function, 222-226
 - example, 169-170
 - EXCEPTION_INIT pragma, 217-221
 - handling multiple exceptions, 171-173
 - INTERNAL_ERROR, 561
 - INVALID_ FILEHANDLE, 561
 - INVALID_MODE, 561
 - INVALID_NUMBER, 185
 - INVALID_ OPERATION, 561
 - INVALID_PATH, 561
 - LOGIN_DENIED, 171
 - NO_DATA_FOUND, 170, 180-182
 - OTHERS exception handler, 173-174
 - overview, 163
 - PROGRAM_ERROR, 171
 - propagating, 197-206
 - RAISE_APPLICATION_ ERROR, 212-216
 - READ_ERROR, 561
 - reraising, 201-202, 206-208
 - sample exception-handling script, 174-177
 - scope
 - examples, 180-183
 - sample script, 183-187
 - TOO_MANY_ROWS, 170, 189
 - try-it-yourself projects, 178, 209, 227, 632-642
 - e_non_null_value exception, 641-642
 - e_too_many_students exception, 635-639
 - invalid instructor IDs, handling, 634-635
 - invalid student IDs, handling, 632-634
 - RAISE_ APPLICATION_ ERROR statement, 639-640
 - user-defined exceptions
 - declaring, 188
 - e_exception1, 201
 - e_exception2, 201
 - e_invalid_id, 188-191
 - e_my_exception, 192
 - e_no_sections, 206-208
 - e_non_null_value, 641-642
 - e_too_many_sections, 193-196
 - e_too_many_students, 635-639
 - raising, 189-191
 - sample script, 193-196
 - scope, 191-192
 - VALUE_ERROR, 167-168, 171, 185
 - WRITE_ERROR, 561
 - ZERO_DIVIDE, 170
- EXCEPTION_INIT pragma, 217-221
- executable section (blocks), 5-6
- EXECUTE
 - IMMEDIATE statement
 - common errors, 383-386
 - overview, 380
 - passing NULL values, 386-387
 - sample script, 387-391
 - structure, 381-382
- executing PL/SQL blocks, 7, 10-11

EXISTS collection method, 322

EXIT statement
 compared to **CONTINUE** statement, 148
 sample script, 118-120
 syntax, 114-115

EXIT WHEN statement
 sample script, 120-123
 syntax, 116-117

explain plans
 generating, 570-577
PLAN_TABLE, 568-569

explicit cursors
 attributes, 240-242
 closing, 237-240
 column aliases, 245
 declaring, 232
 definition, 230
 example, 242-245
 fetching rows in, 236-237
 opening, 236
 scope, 245
SELECT list, 245

expressions
CASE
 differences between
 CASE statement and
 CASE expression,
 97-99
 displaying letter grades
 for students, 100-102
 example, 96-97
 overview, 96
 comparing with
 NULLIF function,
 103-104, 107-109

evaluating with
 COALESCE function,
 105-106, 109-111
REGEXP_COUNT
 function, xvii-xviii
REGEXP_INSTR
 function, xviii
REGEXP_SUBSTR
 function, xviii
 sequences in, xx-xxi

EXTEND collection
 method, 323

extending packages, 480-492

F

factorial of 10, calculating,
 137-138

Fahrenheit/Celsius conversion
 script, 76-79

FCLOSE procedure, 560

FCLOSE_ALL procedure, 560

FETCH statement
 sample script, 395-400
 syntax, 393-394

fetching
 cursors, 393-394
 rows in cursors, 236-237

FFLUSH procedure, 561

files
 binary files, reading, 559
 operating system files,
 reading from/writing to,
 559-562

FILE_TYPE datatype, 560

FINAL_GRADE column
 (**ENROLLMENT** table), 603

final_grade procedure,
 481, 484-487

find_sname procedure,
 446, 461

FIRST collection method, 323

FIRST_NAME column
 INSTRUCTOR table, 603
 STUDENT table, 602

FOLLOWS option (**CREATE
 OR REPLACE TRIGGER**
 clause), xxiv

FOPEN function, 560

FOR loops
 calculating factorial of
 10, 137-138
 cursor **FOR** loops,
 246-247
 example, 133-135
 flow of logic, 133
IN option, 132, 137-138
 premature termination
 of, 136-137
REVERSE option,
 135-136, 139-141
 syntax, 132
 try-it-yourself projects,
 627-628

FOR UPDATE clause (cursors),
 258-260

FORALL statement
BULK COLLECT
 clause, 427-428
INDICES OF option, 410
 sample script, 413-421
SAVE EXCEPTIONS
 option, 408-409
 simple examples,
 405-408
 structure, 404-405
VALUES OF option,
 411-412

form procedures, 587-588

formal parameters
(procedures), 444-445

formatting guidelines
case, 597
comments, 598-599
example, 599-600
naming conventions, 598
white space, 597

forms, 588-589

%FOUND attribute
(cursors), 240

FRAMESET procedures, 587

functions. *See also* methods
COALESCE, 26,
105-106, 626
evaluating list of
numbers, 109-111
example, 106-107
syntax, 105
complex functions,
writing, 454
COUNT(), 183, 322
creating, 450-451
cross-section function
result cache, xxiii-xxiv
definition, 450
FOPEN, 560
get_course_descript,
512, 690
get_course_description,
512, 691-696
get_course_descript_
private, 467
get_student_info, 509
HTF functions, 586
id_is_good, 452-453, 461

INSERTING, 649
instructor_status,
680-681
invoking in SQL
statements, 453-454
IS_OPEN, 560
new_instructor_id,
454, 461
new_student_id,
455, 679
NULLIF, 104-105
displaying letter grades
for students, 107-109
example, 103
restrictions, 104
syntax, 103
REGEXP_COUNT,
xvii-xviii
REGEXP_INSTR, xviii
REGEXP_SUBSTR, xviii
RTRIM, 60
scode_at_line, 497
show_description,
451-453
SQLCODE, 222-226
SQLERRM, 222-226
stored functions
creating, 451-452
enforcing purity
level with
RESTRICT_
REFERENCES
pragma, 500-506
overloaded modules,
506-511
overview, 495
projects, 512, 690-696

querying data
directory about,
496-500
running, 452-453
stored function
requirements in
SQL, 503
student_count_priv, 466
syntax, 450-451
SYSDATE, 619-620
TO_CHAR, 60
try-it-yourself projects,
455, 679-681
instructor_status
function, 680-681
new_student_id
function, 679
zip_does_not_exist
function, 679-680
USER, 121
zip_does_not_exist,
455, 679-680

G

get_course_descript
function, 512, 690
get_course_description
function, 512, 691-696
get_course_descript_private
function, 467
GET_LINE procedure, 561
get_name_address
procedure, 675-676
get_student_info function,
477, 509
GRADE table, 605
grades. *See* letter grades

GRADE_CODE_OCCURRENCE
column (GRADE table), 605

GRADE_CONVERSION
table, 605

GRADE_POINT column
(GRADE_CONVERSION
table), 605

GRADE_TYPE table, 604

GRADE_TYPE_CODE column
GRADE table, 605
GRADE_TYPE table, 604

GRADE_TYPE_WEIGHT
table, 604

H

handling
cursor variables, 473
errors. *See* error handling

hierarchical profiler, xxvii,
557-559

HTF functions, 586

HTML forms, 588-589

HTP procedures, 581-582, 586

I

identifiers (variables), 31. *See*
also specific variables

anchored datatypes,
28-29

cursor variables
capabilities, 471-472
compared to
cursors, 472
creating, 472
handling, 473
rules for use, 479

sample cursor variable
in package, 473-475

sample script, 475-479

strong cursor
variables, 472

weak cursor
variables, 472

declaring and
initializing, 31-34

definition, 23-24

examples, 27-28

illegal identifiers, 24-26

initializing with SELECT
INTO, 40-42

naming conventions, 24

package variables,
469-470

scope, 34

substitution variables,
13-17

syntax, 24

id_is_good function, 461

IF statements

IF-THEN

example, 54-56

overview, 54

syntax, 54

testing whether
date falls on
weekend, 58-61

IF-THEN-ELSE

checking number of
students enrolled in
course, 62-64

NULL condition, 58

overview, 54

syntax, 56

when to use, 56

nested IF statements
Celsius/Fahrenheit
conversion script,
76-79

example, 74-75

logical operators,
75-76

try-it-yourself projects,
80, 619-622

IF-THEN statement

example, 54-56

overview, 54

syntax, 54

testing whether date falls
on weekend, 58-61

IF-THEN-ELSE statement

checking number of
students enrolled in
course, 62-64

NULL condition, 58

overview, 54

syntax, 56

when to use, 56

illegal identifiers, 24-26

image maps

client-side HTML image
maps, 592-593

creating, 592-593

in PL/SQL, 593

server-side HTML image
maps, 589-592

image procedures, 589

implicit cursors

capabilities, 231

definition, 230

example, 231-232

IN option (FOR loops), 132, 137-138

- IN OUT parameters (procedures), 445
- IN parameters (procedures), 445-446
- incrementing sequence values, 43
- index-by tables. *See* associative arrays
- INDICES OF option (FORALL statement), 410
- initializing variables, 31-34, 40-42
- inner joins, 608-609
- INSERTING function, 649
- insert_zip procedure, 674-675
- INSTEAD OF triggers, 278-289
- INSTRUCTOR table, 603
- instructors, determining number of sections taught by, 620-621
- INSTRUCTOR_ID column
 - INSTRUCTOR table, 603
 - SECTION table, 601
- instructor_status function, 680-681
- INTERNAL_ERROR exception, 561
- INTERVAL procedure, 564
- INVALID_FILEHANDLE exception, 561
- invalid instructor IDs, handling, 634-635
- INVALID_MODE exception, 561
- INVALID_NUMBER exception, 185
- INVALID_OPERATION exception, 561
- INVALID_PATH exception, 561
- invalid student IDs, handling, 632-634
- invoking functions
 - in SQL statements, 453-454
 - stored functions, 452-453
- IS_OPEN function, 560
- %ISOPEN attribute (cursors), 240
- iterative control
 - CONTINUE statement
 - compared to EXIT statement, 148
 - overview, 144-145
 - sample script, 146-151
 - CONTINUE WHEN statement
 - overview, 145
 - sample script, 152-153
 - EXIT statement, 148
 - nested loops
 - example, 154-155
 - loop labels, 155-157
 - sample exercise, 157-160
 - numeric FOR loops, 627-628
 - calculating factorial of 10, 137-138
 - example, 133-135
 - flow of logic, 133
 - IN option, 132, 137-138
 - premature termination of, 136-137
 - REVERSE option, 135-136, 139-141
 - syntax, 132
 - simple loops, 628-629
 - EXIT condition, 114-115, 118-120
 - EXIT WHEN condition, 116-117, 120-123
 - structure, 114
 - WHILE loops, 626-627
 - calculating sum of integers between 1 and 10, 128-131
 - example, 124-125
 - flow of logic, 124-125
 - infinite WHILE loops, 125-126
 - premature termination of, 126-128
 - syntax, 124

J-K

- jobs, scheduling, 563-567
- joins, 607
 - CROSS JOINS, 607-608
 - EQUI JOINS, 608-609
 - NATURAL JOINS, 609-610
 - OUTER JOINS, 610-611
- keywords. *See* reserved words

L

- labels
 - adding to blocks, 35-36
 - loop labels, 155-157
- LAST collection method, 323
- LAST_NAME column
 - INSTRUCTOR table, 603
 - STUDENT table, 602

- letter grades, displaying for students, 69-73, 91
 - CASE expression, 100-102
 - CASE statement, 91-95
 - NULLIF function, 107-109
- LETTER_GRADE column (GRADE_CONVERSION table), 605
- lexical units
 - comments, 23
 - delimiters, 23-24
 - identifiers
 - anchored datatypes, 28-29
 - declaring and initializing, 31-34
 - definition, 23-24
 - examples, 27-28
 - illegal identifiers, 24-26
 - initializing with SELECT INTO, 40-42
 - naming conventions, 24
 - scope, 34
 - syntax, 24
 - literals, 23
 - reserved words, 23, 26-27
- LIMIT option (BULK COLLECT statement), 423, 425
- literals, 23
- LOB (large object) datatype, 31
- LOCATION column (SECTION table), 601
- logical operators, 75-76
- LOGIN_DENIED exception, 171
- LONG datatype, 31
- LONG RAW datatype, 31
- LOOP keyword, 114
- loops
 - CONTINUE condition
 - compared to EXIT condition, 148
 - overview, 144-145
 - sample script, 146-151
 - CONTINUE WHEN condition, 629-630
 - overview, 145
 - sample script, 152-153
 - EXIT statement, 148
 - FOR loops, 627-628
 - cursor FOR loops, 246-247
 - try-it-yourself projects, 627-628
 - nested loops
 - example, 154-155
 - loop labels, 155-157
 - sample exercise, 157-160
 - numeric FOR loops
 - calculating factorial of 10, 137-138
 - example, 133-135
 - flow of logic, 133
 - IN option, 132, 137-138
 - premature termination of, 136-137
 - REVERSE option, 135-136, 139-141
 - syntax, 132
 - simple loops
 - EXIT condition, 114-115, 118-120
 - EXIT WHEN condition, 116-117, 120-123
 - structure, 114
 - try-it-yourself projects, 628-629
 - terminating
 - with Boolean expressions, 126
 - with EXIT condition, 114-115, 118-120
 - with EXIT WHEN condition, 116-117, 120-123
 - premature termination, 126-128, 136-137
 - WHILE loops
 - calculating sum of integers between 1 and 10, 128-131
 - example, 124-125
 - flow of logic, 124-125
 - infinite WHILE loops, 125-126
 - premature termination of, 126-128
 - syntax, 124
 - try-it-yourself projects, 626-627
- lowercase, 597

M

manage_grades package, 480-492

manage_students package, 461, 465-469

map methods, 538-541

MAX_GRADE column (GRADE_CONVERSION table), 605

median_grade procedure, 487-488

member methods, 534-536

methods. *See also* functions

- collection methods, 322-325
 - applying to varrays, 336-338
- COUNT, 322
- DELETE, 323, 337
- example, 323-325
- EXISTS, 322
- EXTEND, 323
- FIRST, 323
- LAST, 323
- NEXT, 323
- PRIOR, 323
- TRIM, 323

object type methods

- constructor methods, 531-534
- map methods, 538-541
- member methods, 534-536
- order methods, 541-544
- overview, 531

- sample object type methods, 544-553
- static methods, 536-538

MIN_GRADE column (GRADE_CONVERSION table), 605

mixed notation, xxii-xxiii

modes, parameter, 444

MODIFIED_BY column

- COURSE table, 601
- ENROLLMENT table, 603
- GRADE table, 605
- GRADE_CONVERSION table, 605
- GRADE_TYPE table, 604
- INSTRUCTOR table, 603
- SECTION table, 602
- STUDENT table, 602
- ZIPCODE table, 603

MODIFIED_DATE column

- COURSE table, 601
- ENROLLMENT table, 603
- GRADE table, 605
- GRADE_CONVERSION table, 605
- GRADE_TYPE table, 604
- INSTRUCTOR table, 603
- SECTION table, 602
- STUDENT table, 602
- ZIPCODE table, 603

modular code

- anonymous blocks, 440
- benefits of, 439
- block structure, 440

modules, overloading, 506-511

multilevel collections, 342-347

multiline comments, 599

multiple exceptions, handling, 171-173

multiple transactions in blocks, 50

mutating tables, 292-299

MY_SECTION table, 665-672

N

name_rec record, 367

name_tab associative array, 652

name_tab table, 375-377

name_varray varray, 652-653

named blocks, 4

named notation, xxii-xxiii

naming conventions, 24, 598

native compiler, xxvii

native dynamic SQL

- CLOSE statement
 - sample script, 395-400
 - syntax, 394-395
- dynamic SQL enhancements, xxii
- EXECUTE IMMEDIATE statement
 - common errors, 383-386
 - overview, 380
 - passing NULL values, 386-387
 - sample script, 387-391
 - structure, 381-382

- FETCH statement
 - sample script, 395-400
 - syntax, 393-394
- OPEN-FOR statement
 - sample script, 395-400
 - syntax, 392-393
- overview, 379
- NATURAL JOINS, 609-610**
- nesting**
 - blocks, 35-36
 - cursors, 247-251, 255-257
 - IF statements
 - Celsius/Fahrenheit conversion script, 76-79
 - example, 74-75
 - logical operators, 75-76
 - loops
 - example, 154-155
 - loop labels, 155-157
 - sample exercise, 157-160
 - PL/SQL tables, 319-321, 330-333
 - records, 367-372
- new features**
 - automatic subprogram inlining, xxv-xxvi
 - compound triggers, xxiv
 - CONTINUE statement, xx
 - cross-section function result cache, xxiii-xxiv
 - DRCP (Database Resident Connection Pool), xxv
 - dynamic SQL enhancements, xxii
 - hierarchical profiler, xxvii
 - named and mixed notation, xxii-xxiii
 - overview, xvii
 - PL/Scope, xxvii
 - PL/SQL native compiler, xxvii
 - REGEXP_COUNT function, xvii-xviii
 - REGEXP_INSTR function, xviii
 - REGEXP_SUBSTR function, xviii
 - sequences in expressions, xx-xxi
 - SIMPLE_DOUBLE datatype, xviii-xx
 - SIMPLE_FLOAT datatype, xviii-xx
 - SIMPLE_INTEGER datatype, xviii-xx
 - trigger control, xxiv
- new_instructor_id** function, 454, 461
- NEW_LINE** procedure, 561
- new_student_id** function, 455, 679
- NEXT** collection method, 323
- NEXT_DATE** procedure, 564
- NO_DATA_FOUND** exception, 170, 180-182
- NOT NULL** constraint, 354-355
- %NOTFOUND** attribute (cursors), 240
- NULL values, 58**
 - NULL collections, 322
 - NULL varrays, 335-336
 - passing, 386-387
- NULLIF** function, 104-105
 - displaying letter grades for students, 107-109
 - example, 103
 - restrictions, 104
 - syntax, 103
- NUMBER** datatype, 29
- numbers**
 - drawing from sequences, 43-44
 - error numbers, returning with SQLCODE function, 222-226
- NUMBER_PER_SECTION** column (GRADE_TYPE table), 604
- numeric FOR** loops
 - calculating factorial of 10, 137-138
 - example, 133-135
 - flow of logic, 133
 - IN option, 132, 137-138
 - premature termination of, 136-137
 - REVERSE option, 135-136, 139-141
 - syntax, 132
- NUMERIC_GRADE** column (GRADE table), 605

O

OAS (Oracle Application Server), 578

objects

comparing

map methods,
538-541

order methods,
541-544

object types

collections of,
520-522, 526-530

constructor methods,
531-534

creating, 514-516

map methods,
538-541

member methods,
534-536

object type
specifications, 516

order methods,
541-544

overview, 513-514

sample object type
methods, 544-553

sample script, 522-526

static methods,
536-538

student_obj_type,
696-703

try-it-yourself projects,
554, 696-703

type

specifications, 516

uninitialized objects,
517-518

zipcode_obj_type
example, 517

private objects, 465-469
uninitialized objects,
517-518

OPEN-FOR statement

sample script, 395-400
syntax, 392-393

opening cursors, 236, 392-393

**operating system files, reading
from/writing to, 559-562**

operators

assignment operator, 31
logical operators, 75-76
overview, 34

Oracle Application Server 10g

application server
tier, 580

client tier, 580

database tier, 580

Oracle HTTP server
modules, 580

overview, 578-579

Web Toolkit

client-side HTML
image maps, 592-593
form procedures,
587-588

FRAMESET
procedures, 587

generating Web pages,
582-586, 594-596

HTF functions, 586

HTML forms, 588-589

HTP procedures,
581-582

image procedures, 589
server-side HTML

image maps, 589-592

table of packages, 581

**Oracle Application Server
11g, 578**

**Oracle HTTP server
modules, 580**

Oracle sequences

accessing, 43

definition, 43

drawing numbers
from, 43-44

incrementing values, 43

in PL/SQL blocks, 44

student_id_seq, 44

*Oracle SQL by Example, Third
Edition, 568*

*Oracle Web Application
Programming for PL/SQL
Developers, 580*

Oracle Web Toolkit

client-side HTML image
maps, 592-593

form procedures,
587-588

FRAMESET procedures,
587

generating Web pages,
582-586, 594-596

HTF functions, 586

HTML forms, 588-589

HTP procedures, 581-582

image procedures, 589

server-side HTML image
maps, 589-592

table of packages, 581

order methods, 541-544

**OTHERS exception
handler, 173-174**

OUT parameters (procedures),
445-446
OUTER JOINS, 610-611
overloading modules, 506-511

P

p-code, 10

packages

- benefits of, 458-459
- cursor variables
 - capabilities, 471-472
 - compared to cursors, 472
 - creating, 472
 - handling, 473
 - rules for use, 479
 - sample cursor variable in package, 473-475
 - sample script, 475-479
 - strong cursor variables, 472
 - weak cursor variables, 472
- cursors, 469-470
- DBMS_HPROF, 556-559
- DBMS_JOB, 563-567
- DBMS_LOB, 559
- DBMS_PROFILER, 556
- DBMS_XPLAN
 - generating explain plans, 570-577
 - PLAN_TABLE, 568-569
- definition, 457
- dynamic_sql_pkg, 677

- elements,
 - referencing, 460
- extending, 480-492
- manage_grades, 480-492
- manage_students, 461, 465-469
- overview, 555
- package body
 - creating, 462-464
 - rules for, 460
 - syntax, 459-460
- package specifications, 459-461
- private objects, 465-469
- projects, 493, 681-690
- school_api, 461-463, 467
- stored packages, 464-465
- student_api
 - get_course_descript function, 690
 - get_course_description function, 691-696
 - remove_student procedure, 681-690
- student_info_pkg, 477
- try-it-yourself projects, 554, 696-703
- UTL_FILE
 - example, 561-562
 - exceptions, 561
 - functions, procedures, and datatypes, 560-561
 - overview, 559
 - sample script, 563
- variables, 469-470

parameters

- cursor parameters, 254-255
- parameter modes, 444
- procedure parameters
 - actual parameters, 444-445
 - datatype constraints, 445
 - formal parameters, 444-445
- IN parameters, 445-446
- modes, 444
- OUT parameters, 445-446
- try-it-yourself projects, 447, 673-678
- passing NULL values, 386-387
- PERCENT_OF_FINAL_GRADE column (GRADE_TYPE table), 604
- person_rec record, 367-368
- PHONE column
 - INSTRUCTOR table, 603
 - STUDENT table, 602
- PL/Scope, xxvii
- plans, explain plans
 - generating, 570-577
 - PLAN_TABLE, 568-569
- PLAN_TABLE table, 568-569
- plus sign (+), 24
- pragmas
 - AUTONOMOUS_TRANSACTION, 270-272
 - definition, 217-221

- EXCEPTION_INIT, 217-221
- restrictions, 504
- RESTRICT_
- REFERENCES, 500-506
- predefined exceptions.**
See exceptions
- premature termination**
 - of FOR loops, 136-137
 - of WHILE loops, 126-128
- PREREQUISITE column**
(COURSE table), 601
- PRIOR collection method,** 323
- private objects,** 465-469
- procedures.** *See also*
functions; methods
 - anonymous blocks, 440
 - benefits of, 439
 - block structure, 440
 - BROKEN, 564
 - CHANGE, 564
 - creating, 441-442
 - current_status, 673-674
 - display_student_count, 466
 - dynamic_sql_pkg, 677
 - FCLOSE, 560
 - FCLOSE_ALL, 560
 - FFLUSH, 561
 - final_grade, 481, 484-487
 - find_sname, 446, 461
 - form procedures, 587-588
 - FRAMESET
 - procedures, 587
 - GET_LINE, 561
 - get_name_address, 675-676
 - get_student_info, 477
 - HTP procedures, 581-582, 586
 - image procedures, 589
 - insert_zip, 674-675
 - INTERVAL, 564
 - median_grade, 487-488
 - NEW_LINE, 561
 - NEXT_DATE, 564
 - overview, 439
 - parameters
 - actual parameters, 444-445
 - datatype constraints, 445
 - formal parameters, 444-445
 - IN parameters, 445-446
 - modes, 444
 - OUT parameters, 445-446
 - PUT, 561
 - PUTF, 561
 - PUT_LINE, 561
 - querying data dictionary for information on, 443
 - REMOVE, 564
 - remove_student, 681-690
 - RUN, 564
 - SUBMIT, 564-565
- profiling PL/SQL,** 556-559
- PROGRAM_ERROR exception,** 171
- projects (try-it-yourself)**
 - block projects, 37, 614-616
 - bulk SQL projects, 437, 665-672
 - CASE statement projects, 112, 622-626
 - collection projects, 348, 652-659
 - compound trigger projects, 313, 648-651
 - cursor projects, 252, 643-645
 - error handling projects, 178, 632-635
 - e_non_null_value exception, 641-642
 - e_too_many_students exception, 635-639
 - invalid instructor IDs, handling, 634-635
 - invalid student IDs, handling, 632-634
 - exception projects, 209, 227, 635-642
 - function projects, 455, 679-681
 - instructor_status function, 680-681
 - new_student_id function, 679
 - zip_does_not_exist function, 679-680
 - IF statement projects, 80, 619-622
 - iterative control projects, 142, 161, 627-632
 - CONTINUE WHEN statement, 629-630
 - FOR loops, 627-628
 - simple loops, 628-629
 - WHILE loops, 626-627
 - object type projects, 554, 696-703

- package projects, 493, 681-690
 - PL/SQL in client/server architecture, 19, 613-614
 - PL/SQL in SQL*Plus, 19, 613-614
 - procedure projects, 447, 673-678
 - current_status procedure, 673-674
 - dynamic_sql_pkg procedure, 677
 - get_name_address procedure, 675-676
 - insert_zip procedure, 674-675
 - record projects, 378, 659-665
 - stored code projects, 512
 - get_course_descript function, 690
 - get_course_description function, 691-696
 - transaction projects, 51
 - CHAP4 table, creating, 616
 - CHAP4_SEQ sequence, creating, 616
 - PL/SQL block script, 617-619
 - trigger projects, 290
 - compound triggers, 648-651
 - creating triggers, 645-648
 - propagating exceptions, 197-206
 - PUT procedure, 561
 - PUTF procedure, 561
 - PUT_LINE procedure, 561
-
- Q**
-
- QL*Plus, PL/SQL in
 - DBMS_OUTPUT.PUT_LINE statement, 16-18
 - overview, 12-13
 - substitution variables, 13-17
 - try-it-yourself projects, 19, 613-614
 - querying
 - data dictionary for procedure information, 443
 - data dictionary for stored code information, 496-500
 - scalar subqueries, 611
-
- R**
-
- RAISE_APPLICATION_ERROR statement, 212-216, 639-640
 - RAISE statement, 191
 - raising user-defined exceptions, 189-191
 - reading
 - binary files, 559
 - operating system files, 559-562
 - READ_ERROR exception, 561
 - records
 - collections of records, 373-377
 - compatibility, 355-357
 - cursor-based records, 233-235
 - student_rec example, 350-353
 - zip_rec example, 358-362
 - definition, 233, 349
 - fetching with BULK COLLECT statement, 422-423
 - nesting, 367-372
 - %ROWTYPE attribute, 350
 - table-based records, 233-234
 - course_rec example, 350-353
 - zip_rec example, 358-362
 - try-it-yourself projects, 378, 659-665
 - user-defined records
 - creating, 353, 659-665
 - NOT NULL constraint, 354-355
 - time_rec_type example, 353-355
 - zip_info_rec example, 362-366
 - REF CURSOR keywords, 472
 - referencing package elements, 460
 - REGEXP_COUNT function, xvii-xviii
 - REGEXP_INSTR function, xviii
 - REGEXP_SUBSTR function, xviii
 - REGISTRATION_DATE column (STUDENT table), 602

regular expressions.
See expressions

regular joins, 608-609

REMOVE procedure, 564

remove_student procedure, 681-690

reraising exceptions, 201-202, 206-208

reserved words. *See also* statements

- BEGIN, 6
- CONSTANT, 5
- DECLARE, 5
- definition, 23, 26
- END, 6
- EXCEPTION, 6, 188
- IN, 132
- invalid use of, 26-27
- LOOP, 114
- REF CURSOR, 472
- REVERSE, 132, 135-136

RESTRICT_REFERENCES pragma, 500-506

RETURN statement, 115, 450

RETURNING option (BULK COLLECT statement), 426-427

REVERSE option (FOR loops), 135-136, 139-141

Rischert, Alice, 568

RNDS pragma restriction, 504

RNPS pragma restriction, 504

ROLLBACK statement, 47

rolling back transactions, 47-49

row triggers, 277, 283-285

%ROWCOUNT attribute (cursors), 240

ROWID datatype, 31

rows, fetching in cursors, 236-237

%ROWTYPE attribute, 350

RTRIM function, 60

rules for package bodies, 460

RUN procedure, 564

runtime errors, 11, 164-167

S

SALUTATION column

- INSTRUCTOR table, 603
- STUDENT table, 602

SAVE EXCEPTIONS option (FORALL statement), 408-409

SAVEPOINT statement, 47-49

scalar subqueries, 611

scheduling jobs, 563-567

school_api package, 461-463, 467

scode_at_line function, 497

scope

- of cursors, 245
- of exceptions
 - examples, 180-183
 - sample script, 183-187
- PL/Scope, xxvii
- of user-defined exceptions, 191-192
- of variables, 34

searched CASE statements

- differences between CASE and searched CASE, 86-89
- displaying letter grade for student, 91-95
- example, 86
- syntax, 84-85

SECTION table, 601-602

SECTION_ID column

- ENROLLMENT table, 602
- GRADE table, 605
- GRADE_TYPE table, 604
- SECTION table, 601

SECTION_NO column (SECTION table), 601

SELECT INTO statement, 40-42

SELECT list cursors, 245

sequences

- accessing, 43
- CHAP4_SEQ sequence, creating, 616
- definition, 43
- drawing numbers from, 43-44
- in expressions, xx-xxi
- in PL/SQL blocks, 44
- incrementing values, 43
- student_id_seq, 44

server-side HTML image maps, 589-592

SET statement, 15

show error command, 443

show_description function, 451-453

simple loops

- EXIT condition
 - sample script, 118-120
 - syntax, 114-115
- EXIT WHEN condition
 - sample script, 120-123
 - syntax, 116-117
- structure, 114
- try-it-yourself projects, 628-629

- SIMPLE_DOUBLE datatype, xviii, xx
- SIMPLE_FLOAT datatype, xviii, xx
- SIMPLE_INTEGER datatype, xviii, xx
- single-line comments, 599
- specifications
 - object type specifications, 516
 - package specifications, 459-461
- SQL standards
 - joins
 - CROSS JOIN syntax, 607-608
 - EQUI JOIN syntax, 608-609
 - NATURAL JOIN syntax, 609-610
 - OUTER JOIN syntax, 610-611
 - overview, 607
 - scalar subqueries, 611
- SQL statements. *See statements*
- SQLCODE function, 222-226
- SQLERRM function, 222-226
- START_TIME_DATE column (SECTION table), 601
- STATE column (ZIPCODE table), 603
- statement triggers, 277-278, 283-285
- statements
 - ALTER SYSTEM, 564
 - ALTER TRIGGER, 265-266
- BULK COLLECT
 - fetching data into collections, 425-426
 - fetching records with, 422-423
 - with FORALL statement, 427-428
 - LIMIT option, 423-425
 - RETURNING option, 426-427
 - sample scripts, 428-436
 - structure, 422
- CASE
 - differences between CASE statement and CASE expression, 97-99
 - displaying name of day, 89-91
 - examples, 83-84
 - overview, 82
 - searched CASE statements, 84-95
 - syntax, 82
 - try-it-yourself projects, 112
- CLOSE
 - sample script, 395-400
 - syntax, 394-395
- COMMIT, 46-47
- CONTINUE, xx
 - compared to EXIT statement, 148
 - overview, 144-145
 - sample script, 146-151
- CONTINUE WHEN
 - overview, 145
 - sample script, 152-153
 - try-it-yourself projects, 629-630
- CREATE FUNCTION, 450-451
- CREATE OR REPLACE TRIGGER, xxiv
- CREATE OR REPLACE TYPE, 514
- CREATE TRIGGER, 265
- DBMS_OUTPUT.PUT_LINE, 16-18
- ELSEIF
 - conditions, 67
 - displaying letter grade for student, 69-73
 - example, 66
 - examples, 65-68
 - syntax, 65
- END IF, 54
- END LOOP, 114
- EXECUTE IMMEDIATE
 - common errors, 383-386
 - overview, 380
 - passing NULL values, 386-387
 - sample script, 387-391
 - structure, 381-382
- EXIT
 - compared to CONTINUE statement, 148
 - sample script, 118-120
 - syntax, 114-115

- EXIT WHEN
 - sample script, 120-123
 - syntax, 116-117
- FETCH
 - sample script, 395-400
 - syntax, 393-394
- FOR UPDATE, 258-260
- FORALL
 - BULK COLLECT
 - clause, 427-428
 - INDICES OF
 - option, 410
 - sample script, 413-421
 - SAVE EXCEPTIONS
 - option, 408-409
 - simple examples, 405-408
 - structure, 404-405
 - VALUES OF option, 411-412
- IF-THEN
 - example, 54-56
 - overview, 54
 - syntax, 54
 - testing whether date falls on weekend, 58-61
- IF-THEN-ELSE
 - checking number of students enrolled in course, 62-64
 - NULL condition, 58
 - overview, 54
 - syntax, 56
 - when to use, 56
- invoking functions
 - in, 453-454
- nested IF statements
 - Celsius/Fahrenheit conversion
 - script, 76-79
 - example, 74-75
 - logical operators, 75-76
- OPEN-FOR
 - sample script, 395-400
 - syntax, 392-393
- RAISE, 191
- RAISE_APPLICATION_ERROR, 639-640
- RETURN, 115, 450
- ROLLBACK, 47
- SAVEPOINT, 47-49
- SELECT INTO, 40-42
- SET, 15
- show error, 443
- TYPE, 335
- WHERE CURRENT, 261
- static methods, 536-538**
- stored code**
 - creating, 451-452
 - enforcing purity level with RESTRICT_REFERENCES pragma, 500-506
 - overloaded modules, 506-511
 - overview, 495
 - querying data directory
 - about, 496-500
 - running, 452-453
- stored function
 - requirements in SQL, 503
- try-it-yourself projects, 512
 - get_course_descript function, 690
 - get_course_description function, 691-696
- stored packages, calling, 464-465**
- STREET_ADDRESS column**
 - INSTRUCTOR table, 603
 - STUDENT table, 602
- strong cursor variables, 472**
- student database**
 - COURSE table, 601
 - ENROLLMENT table, 602
 - GRADE table, 605
 - GRADE_CONVERSION table, 605
 - GRADE_TYPE table, 604
 - GRADE_TYPE_WEIGHT table, 604
 - INSTRUCTOR table, 603
 - SECTION table, 601-602
 - STUDENT table, 602
 - ZIPCODE table, 603
- student IDs**
 - instructor student IDs, handling, 634-635
 - invalid student IDs, handling, 632-634
- STUDENT table, 602**

students

- checking number of students enrolled in course, 62-64
- displaying letter grades for, 69-73
 - CASE expression, 100-102
 - CASE statement, 91-95
 - NULLIF function, 107-109
- displaying number of students for given zip code, 183-187

student_api package

- get_course_descript function, 690
- get_course_description function, 691-696
- remove_student procedure, 681-690

student_count_priv function, 466**STUDENT_ID column**

- ENROLLMENT table, 602
- GRADE table, 605
- STUDENT table, 602

student_id_seq sequence, 44**student_info_pkg package, 477****student_obj_type, 696-703****student_rec record, 351****SUBEXPR parameter (REGEXP_INSTR/REGEXP_SUBSTR functions), xviii****SUBMIT procedure, 564-565****submitting jobs to queue, 564-567****substitution variables, 13-17****syntax errors, 10-11****SYSDATE function, 619-620**

T

table-based records, 233-234

- course_rec example, 350-353
- zip_rec example, 358-362

tables

- CHAP4 table, creating, 616
- COURSE, 601
- ENROLLMENT, 602
- GRADE, 605
- GRADE_
 - CONVERSION, 605
- GRADE_TYPE, 604
- GRADE_TYPE_
 - WEIGHT, 604
- INSTRUCTOR, 603
- mutating tables, 292-299
- MY_SECTION, 665-672
- name_tab, 375-377
- PL/SQL tables
 - associative arrays, 317-319, 326-330
 - definition, 316-317
 - nested tables, 319-321, 330-333
- PLAN_TABLE, 568-577
- SECTION, 601-602

STUDENT, 602**table-based records**

- course_rec example, 350-353
- zip_rec example, 358-362

ZIPCODE, 603**terminating loops**

- with Boolean expressions, 126
- with EXIT condition, 114-115, 118-120
- with EXIT WHEN condition, 116-117, 120-123
- TIMESTAMP datatype, 30
- premature termination, 126-128, 136-137

time_rec_type record, 353-355**TO_CHAR function, 60****TOO_MANY_ROWS exception, 170, 189****transactions**

- autonomous transactions, 270-272
- committing, 46-47
- definition, 39
- multiple transactions in blocks, 50
- overview, 45-46
- rolling back, 47-49
- try-it-yourself projects, 51
 - CHAP4 table, creating, 616

- CHAP4_SEQ
 - sequence,
 - creating, 616
 - PL/SQL block script,
 - 617-619
- triggering events, 264
- triggers
 - AFTER triggers, 269-270,
 - 274-276
 - autonomous transactions,
 - 270-272
 - BEFORE triggers,
 - 267-269, 274-276
 - compound triggers, xxiv
 - capabilities, 300
 - examples, 302-306
 - modifying, 306-312
 - restrictions, 301
 - structure, 300-301
 - try-it-yourself projects,
 - 313, 648-651
 - controlling, xxiv
 - creating, 264-265,
 - 272-273, 645-648
 - definition, 264
 - disabling, 265-266
 - INSTEAD OF triggers,
 - 278-289
 - mutating table issues,
 - 292-299
 - restrictions, 266-267
 - row triggers, 277,
 - 283-285
 - statement triggers,
 - 277-278, 283-285
 - triggering events, 264
 - try-it-yourself
 - projects, 290
 - compound triggers,
 - 648-651
 - creating triggers,
 - 645-648
 - TRIM collection method, 323
 - try-it-yourself projects
 - block projects, 37,
 - 614-616
 - bulk SQL projects,
 - 437, 665-672
 - CASE statement projects,
 - 112, 622-626
 - collection projects,
 - 348, 652-659
 - compound trigger
 - projects, 313, 648-651
 - cursor projects, 252,
 - 643-645
 - error handling projects,
 - 178, 632-635
 - e_non_null_value
 - exception, 641-642
 - e_too_many_students
 - exception, 635-639
 - invalid instructor IDs,
 - handling, 634-635
 - invalid student IDs,
 - handling, 632-634
 - exception projects, 209,
 - 227, 635-642
 - function projects, 455,
 - 679-681
 - instructor_status
 - function, 680-681
 - new_student_id
 - function, 679
 - zip_does_not_exist
 - function, 679-680
 - IF statement projects,
 - 80, 619-622
 - iterative control projects,
 - 142, 161, 627-632
 - CONTINUE WHEN
 - statement, 629-630
 - FOR loops, 627-628
 - simple loops, 628-629
 - WHILE loops,
 - 626-627
 - object type projects,
 - 554, 696-703
 - package projects,
 - 493, 681-690
 - PL/SQL in client/server
 - architecture, 19,
 - 613-614
 - PL/SQL in SQL*Plus, 19,
 - 613-614
 - procedure projects, 447,
 - 673-678
 - current_status
 - procedure, 673-674
 - dynamic_sql_pkg
 - procedure, 677
 - get_name_address
 - procedure, 675-676
 - insert_zip procedure,
 - 674-675
 - record projects,
 - 378, 659-665
 - stored code projects, 512
 - get_course_descript
 - function, 690
 - get_course_description
 - function, 691-696

transaction projects, 51
 CHAP4 table,
 creating, 616
 CHAP4_SEQ
 sequence,
 creating, 616
 PL/SQL block script,
 617-619
 trigger projects, 290
 compound triggers,
 648-651
 creating triggers,
 645-648
TYPE statement, 335
types. *See* object types

U

uninitialized objects, 517-518
 uppercase, 597
USER function, 121
 user-defined exceptions
 declaring, 188
 e_exception1, 201
 e_exception2, 201
 e_invalid_id, 188-191
 e_my_exception, 192
 e_no_sections, 206-208
 e_too_many_sections,
 193-196
 raising, 189-191
 sample script, 193-196
 scope, 191-192
 user-defined records
 creating, 353, 659-665
 NOT NULL constraint,
 354-355

time_rec_type
 example, 353-355
 zip_info_rec
 example, 362-366
USER_DEPENDENCIES view,
 499-500
USER_ERRORS view, 498
USER_OBJECTS view, 496
UTL_FILE package
 example, 561-563
 exceptions, 561
 functions, procedures,
 and datatypes, 560-561
 overview, 559

V

v_area variable, 613
 v_average_cost variable, 41
 v_calories_per_cookie
 variable, 32
 v_cookies_amt variable, 32
 v_counter variable, 33,
 118-119, 129-130, 138-141,
 147-149, 152-153
 v_counter1 variable, 154
 v_counter2 variable, 154
 v_current_date variable, 469
 v_date variable, 89, 619-620
 v_day variable, 90-91
 v_err_code variable, 224
 v_err_msg variable, 224
 v_exists variable, 175
 v_factorial variable, 138
 v_final_grade variable, 92
 v_instructor_id variable,
 294-295
 v_instructor_name variable,
 294-295
 v_letter_grade variable, 100
 v_lname variable, 33
 v_new_cost variable, 33
 v_num1 variable, 164
 v_num2 variable, 164
 v_num_flag variable, 86-88
 v_pctincr variable, 33
 v_radius variable, 613
 v_regdate variable, 33
 v_result variable, 164
 v_student_id variable, 36
 v_student_name variable, 177
 v_sum variable, 130
 v_zip variable, 175-176
VALUES OF option (FORALL
 statement), 411-412
VALUE_ERROR exception,
 167-168, 171, 185
VARCHAR2 datatype, 29
 variable-size arrays. *See* varrays
 variables, 31. *See also specific
 variables*
 anchored datatypes,
 28-29
 cursor variables
 capabilities, 471-472
 compared to
 cursors, 472
 creating, 472
 handling, 473
 rules for use, 479
 sample cursor variable
 in package, 473-475

- sample script, 475-479
- strong cursor
 - variables, 472
- weak cursor
 - variables, 472
- declaring and initializing, 31-34
- definition, 23-24
- examples, 27-28
- illegal identifiers, 24-26
- initializing with SELECT INTO, 40-42
- naming conventions, 24
- package variables, 469-470
- scope, 34
- substitution variables, 13-17
- syntax, 24
- varrays**
 - city_varray, 338-341
 - collection methods, 336-338
 - course_varray, 653-657
 - creating, 334-335
 - definition, 334
 - name_varray, 652-653
 - NULL varrays, 335-336
 - runtime errors, 656-659
 - sample script, 338-341
- views**
 - USER_DEPENDENCIES, 499-500
 - USER_ERRORS, 498
 - USER_OBJECTS, 496
- vr_student variable, 234
- vr_zip variable, 234-235

W

- weak cursor variables, 472
- Web pages, generating with Oracle Web Toolkit, 582-596**
- Web Toolkit**
 - client-side HTML image maps, 592-593
 - form procedures, 587-588
 - FRAMESET procedures, 587
 - generating Web pages, 582-596
 - HTF functions, 586
 - HTML forms, 588-589
 - HTP procedures, 581-582
 - image procedures, 589
 - server-side HTML image maps, 589-592
 - table of packages, 581
- WHERE CURRENT clause, 261**
- WHILE loops**
 - calculating sum of integers between 1 and 10, 128-131
 - example, 124-125
 - flow of logic, 124-125
 - infinite WHILE loops, 125-126
 - premature termination of, 126-128
 - syntax, 124
 - try-it-yourself projects, 627
- white space, formatting, 597**
- WNDS pragma restriction, 504**
- WNPS pragma restriction, 504**

- WRITE_ERROR exception, 561**

writing

- blocks, 37, 614-616
- complex functions, 454
- to operating system files, 559-562

X-Y-Z

- ZERO_DIVIDE exception, 170**
- ZIP column**
 - INSTRUCTOR table, 603
 - STUDENT table, 602
 - ZIPCODE table, 603
- zip_cur cursor, 371**
- zip_does_not_exist function, 455, 679-680**
- zip_info_rec record, 362-366**
- zip_rec record, 358-362**
- zipcode_obj_type, 517**
- ZIPCODE table, 603**