Oracle 10g PL/SQL Programming

Student Workbook

ORACLE 10G PL/SQL PROGRAMMING

Julie Johnson and Robert Seitz

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112.

Contributing Authors: Denise Geller, Danielle Hopkins, and Rob Roselius.

Editor: Jan Waleri

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Roger Jones, Jim McNally, and Kevin Smith, and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

Chapter 1 - Course Introduction 7

Course Objectives 8
Course Overview 10
Using the Workbook 11
Suggested References 12

Chapter 2 - Triggers 15

Beyond Declarative Integrity 16
Triggers 18
Types of Triggers 20
Row-Level Triggers 22
Cascading Triggers and Mutating Tables 24
Generating an Error 26
Triggers on Views 28
System Triggers 30
Maintaining Triggers 32
Labs 34
Labs (cont'd) 36

Chapter 3 - PL/SQL Syntax and Logic 39

PL/SQL Blocks and Programs 40
Declaring Variables 42
Datatypes 44
Subtypes 46
Character Data 48
Dates and Timestamps 50
Date Intervals 52
Anchored Types 54
Assignment and Conversions 56
Selecting into a Variable 58
Conditional Statements 60
Comments and Labels 62
Loops 64
WHILE and FOR Loops 66
Labs 68

Chapter 4 - Stored Procedures and Functions 71

Stored Subprograms 72
Procedures and Functions 74
Creating a Stored Procedure 76
Calling a Stored Procedure 78
Passing Parameters and Default Arguments 80
Parameter Modes 82
Creating a Stored Function 84
Calling a Stored Function 86
Stored Functions and SQL 88
Local Procedures and Functions 90
Labs 92

Chapter 5 - Exception Handling 95

SQLCODE and SQLERRM 96
Exception Handlers 98
Nesting Blocks 100
Scope and Name Resolution 102
User-Defined Exceptions 104
compile-Time Warnings 106
Labs 108

Chapter 6 - Records and Collections 111

Record Variables 112
Using the %ROWTYPE Attribute 114
VARRAY and Nested TABLE Collections 116
Using Nested TABLEs 118
Using VARRAYs 120
Associative Array Collections 122
Collection Methods 124
Iterating Through Collections 126
FORALL Statement 128
BULK COLLECT Clause 130
Labs 132

Chapter 7 - Cursors 135

Multi-Row Queries 136
Declaring and Opening Cursors 138

Fetching Rows 140
Closing Cursors 142
The Cursor FOR Loop 144
FOR UPDATE Cursors 146
Cursor Parameters 148
The Implicit (SQL) Cursor 150
Labs 152

Chapter 8 - Using Packages 155

Packages 156
Oracle-Supplied Packages 158
The DBMS_OUTPUT Package 160
The DBMS_UTILITY Package 162
The UTL_FILE Package 164
The DBMS_LOB Package 166
The DBMS_METADATA Package 168
XML Packages 170
Networking Packages 172
Other Supplied Packages 174
Labs 176

Chapter 9 - Creating Packages 179

Structure of a Package 180
The Package Interface and Implementation 182
Package Variables 184
Overloading Package Functions and Procedures 186
Named Parameter Notation 188
REF CURSOR Variables 190
Labs 192

Chapter 10 - Maintaining PL/SQL Code 195

Priveleges for Stored Programs 196
PL/SQL Stored Program Compilation 198
The PL/SQL Execution Environment 200
Dependecies and Validation 202
Maintaining Stored Programs 204
Labs 206

Appendix A - Dynamic SQL 209
Generating SQL at Runtime 210
Native Dynamic SQL vs. DBMS_SQL Package 212
The EXECUTE IMMEDIATE Statement 214
Using Bind Variables 216
Multi-row Dynamic Queries 218
Bulk Operations with Dynamic SQL 220
Using DBMS_SQL 222
DBMS_SQL Subprograms 224

Appendix B - PL/SQL Versions, Datatypes and Language Limits 227

Appendix C - Oracle10g Supplied Packages 237

Solutions 245

Index 273

CHAPTER 5 - EXCEPTION HANDLING

OBJECTIVES

- Use **SQLCODE** and **SQLERRM** in your programs to check for errors.
- Create handlers for exceptions.
- Use nested blocks to catch exceptions.
- Control the scope of program variables.
- Declare and raise user-defined exceptions.
- Set up exception propagation for arbitrary Oracle errors.

SQLCODE AND SQLERRM

- Any expression or statement might result in some error; all errors are numbered:

```
0      No error; normal, successful completion.
100    No data found.
<0     Actual errors have negative error codes.
```

  - The **SQLCODE** function returns the current Oracle error number.

```
errnum := SQLCODE;
```

- For each error, Oracle provides a brief message describing what kind of error it is.

  - The **SQLERRM** function returns the current Oracle error message.

```
errmess := 'Error in program: ' || SQLERRM;
```

  - To retrieve the message for any arbitrary Oracle error number, just pass the number to **SQLERRM**:

```
errinfo := 'Error -2714 is: ' || SQLERRM(-2714);
```

- You can't use the values of **SQLCODE** or **SQLERRM** in SQL statements; you'd be trying to use the values while Oracle is setting them.

  - Save them in program variables, before the SQL statement.

```
code := SQLCODE;
INSERT INTO mylogtable VALUES (SYSDATE, code);
```

- **SQLCODE** and **SQLERRM** are useful in exception handlers.

The Oracle server maintains a full list of error messages in all supported languages. The messages are kept in files under the Oracle installation directory, not in the system catalog. The **SQLERRM()** function allows you to look up arbitrary messages by number.

EXCEPTION HANDLERS

- Certain runtime operations can cause an *exception*: A named error.

```
SELECT min(balance) INTO d FROM account;
a := t / d; -- What if min(balance) is zero?
```

- The system *raises* the exception, causing execution of the current block to stop.
  - Execution transfers to the current block's **EXCEPTION** section, at the end of the block.

- The system will look in the **EXCEPTION** section for a *handler* for the specific exception.

```
EXCEPTION
  WHEN zero_divide THEN
    a := 0;
END;
```

- You can write several exception handlers for a block.

```
EXCEPTION
  WHEN no_data_found THEN
    -- These statements make up the
    -- no_data_found handler.
  WHEN zero_divide THEN
    -- Now we're in the zero_divide handler.
END;
```

  - The optional **WHEN OTHERS** exception handler will catch any exceptions for which you haven't written a specific handler.

```
WHEN OTHERS THEN
  -- Could be anything; handle generically.
END;
```


Pre-defined Oracle exceptions:

Exception Name	Oracle Error	SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51

NESTING BLOCKS

- An exception halts execution of the block in which it was raised.
  - You can use the handler to deal with the error, but the remainder of the block itself is abandoned.
- By nesting blocks inside other blocks, you can trap exceptions.
  - The inner block's **EXCEPTION** section handles its exceptions, allowing the outer block to continue executing.
  - Note that each handler must contain at least one statement; to catch and ignore an exception, use the statement **NULL;**.
  - In an exception handler, the **RAISE** statement will re-raise the current exception to the enclosing block.

```
EXCEPTION
  WHEN no_data_found THEN
    ROLLBACK;-- All prior DML
    RAISE;   -- Re-raise
END;
```

- If an inner block has no handler for it, an exception will *propagate* to the next higher block.
  - If no handler is present, exception propagation continues through enclosing blocks, all the way to the host environment.

add_stock.sql

```
CREATE OR REPLACE PROCEDURE add_stock (
      snum  inventory.store_number%TYPE,
      pid   inventory.product_id%TYPE,
      new_stock  inventory.quantity_on_hand%TYPE )
AS
   qtyoh inventory.quantity_on_hand%TYPE := 0;
BEGIN

   BEGIN
      SELECT  quantity_on_hand INTO qtyoh
        FROM  inventory
       WHERE  store_number = snum AND product_id = pid;
   EXCEPTION
      WHEN no_data_found THEN
         BEGIN
            INSERT INTO inventory (store_number, product_id)
                   VALUES (snum, pid);
         EXCEPTION
            WHEN OTHERS THEN
               IF SQLCODE = -2291 THEN
                  raise_application_error(-20005,
                     'Invalid store or product ID.', false);
               ELSE
                  RAISE;
               END IF;
         END;
   END;

   qtyoh := qtyoh + new_stock;

   UPDATE  inventory SET quantity_on_hand = qtyoh
    WHERE  store_number = snum AND product_id = pid;

END;
/
```

SCOPE AND NAME RESOLUTION

- When you nest blocks inside other blocks (for example, to trap exceptions), *scope* determines where variables can be seen.
  - A variable declared in an outer block can be used in inner blocks.
  - A variable declared in an inner block is local to that block and cannot be used in outer blocks.
  - A variable declared in an inner block hides a variable of the same name declared in an outer block.
- You can label a block of code.
  - You can then qualify an identifier with the label, using dot notation.

```
<<outer>>
DECLARE
  num1 NUMBER := 1;
BEGIN
  <<inner>>
  DECLARE
    num1 NUMBER(3,2) := 2.14;
  BEGIN
    outer.num1 := num1 + outer.num1;
  END;
  num1 := num1 * 10;
END;
```


An identifier can refer to a variable, package, table, procedure, or other database object. Identifiers often use dot notation to disambiguate objects. For example, **s1.x** could refer to table **x** in **s1**'s schema, variable **x** in package **s1**, or column **x** in table **s1**.

Every time you compile a PL/SQL block, Oracle associates an identifier with its appropriate object. When compiling a SQL statement, Oracle first checks to see if there is an object in the current schema by that name. It then checks packages, types, tables, and views.

Compiling a PL/SQL statement uses a different search order. It looks for packages, types, tables, and views in the named schema, then for objects within that schema.

USER-DEFINED EXCEPTIONS

- There are three basic types of exceptions:
  - Pre-defined Oracle exceptions
  - User-defined exceptions
  - User-named Oracle exceptions
- Exceptions like **NO_DATA_FOUND** are pre-defined by Oracle.
- You may declare your own exceptions in the **DECLARE** section.

```
exception_name EXCEPTION;
```

  - You must explicitly raise user-defined exceptions with the **RAISE *exception_name*** statement.
- You can associate a name with an Oracle error number.

```
exception_name EXCEPTION;
PRAGMA EXCEPTION_INIT(exception_name, err_number);
```

  - The exception will be raised by the system when the error number occurs.
  - Naming an Oracle error allows you to create a handler that is specific to that exception, instead of using the **WHEN OTHERS** handler.

add_stock2.sql

```
CREATE OR REPLACE PROCEDURE add_stock (
      snum inventory.store_number%TYPE,
      pid   inventory.product_id%TYPE,
      new_stock inventory.quantity_on_hand%TYPE )
AS
   qtyoh inventory.quantity_on_hand%TYPE := 0;
   no_parent_record EXCEPTION;
   PRAGMA EXCEPTION_INIT(no_parent_record, -2291);
BEGIN

   BEGIN
      SELECT quantity_on_hand INTO qtyoh
        FROM inventory
       WHERE store_number = snum AND product_id = pid;
   EXCEPTION
      WHEN no_data_found THEN
         BEGIN
            INSERT INTO inventory (store_number, product_id)
                   VALUES (snum, pid);
         EXCEPTION
            WHEN no_parent_record THEN
                  raise_application_error(-20005,
                     'Invalid store or product ID.', false);
         END;
   END;

   qtyoh := qtyoh + new_stock;

   UPDATE inventory SET quantity_on_hand = qtyoh
    WHERE store_number = snum AND product_id = pid;

END;
/
```

COMPILE-TIME WARNINGS

- Compile-time warnings let you know if code might have a performance flaw or possible runtime error.

- There are three types of warnings:

  - **SEVERE** — Alerts for conditions that might produce wrong results
  - **PERFORMANCE** — Alerts when a statement might affect code performance
  - **INFORMATIONAL** — Neither severe nor performance warning, such as code that may never be executed

- You may enable and disable **plsql_warnings** at the database, session, or object level:

```
ALTER DATABASE SET plsql_warnings = 'enable:all';
                                 --enable all three types
ALTER SESSION SET plsql_warnings =
  'enable:severe','enable:performance';
ALTER PROCEDURE emp_raise SET plsql_warnings =
  'enable:all';
```

  - Oracle disables warnings by default.

  - You cannot turn on warnings for anonymous blocks.

- To display warnings, use the **SHOW ERRORS** SQL*Plus command or query the **USER_ERRORS** Data Dictionary view.

- Query the **USER_PLSQL_OBJECT_SETTINGS** Data Dictionary view to find the warning settings for the objects you own:

```
SELECT name, plsql_warnings
  FROM user_plsql_object_settings;
```


Use the **DBMS_WARNING** package to change warning settings programmatically from a PL/SQL block or other language (such as Java, C, etc...).

```
DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:ALL' ,'SESSION');
DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:PERFORMANCE' ,'SYSTEM');
```

Labs

❶ If you have not yet written a trigger on the employee table to enforce limits on raises (which limits raises for salaried employees to $10,000), do so now. This trigger should use exception number -20001 as the error number.
(Solution: *maxraise.sql*)

❷ Write a stored procedure named **raise_mgr_pay** that will loop through all of the stores and give each store manager a 20% raise. Once the procedure is created, write a test program to call the procedure. You should see an exception if any manager's raise is above the allowed limit.
(Solutions: *raise_mgr_pay1.sql*, *test_raise.sql*)

❸ Modify **raise_mgr_pay** so that it handles the exception and continues on with the other managers' raises. Since the exception is not one of the standard named exceptions, use a **WHEN OTHERS** handler to catch the error. Retest your procedure. Some managers should have received raises this time. Hint: To handle the exception without actually doing anything about the error, place a **NULL;** statement in the handler.
(Solution: *raise_mgr_pay2.sql*)

❹ Modify the **raise_mgr_pay** stored procedure again. This time, declare an exception named **OVER_MAX_RAISE** and use **PRAGMA EXCEPTION_INIT** to associate error number -20001 with **OVER_MAX_RAISE**. Modify your handler to look for just this exception.
(Solution: *raise_mgr_pay3.sql*)

